INTRODUCTION À LA LOGIQUE INFORMATIQUE

SYLVAIN SCHMITZ

Université de Paris, France

Contenu des notes

Ce cours fournit une introduction à la logique informatique. L'accent est mis sur l'utilisation de la logique pour modéliser des problèmes, et les résoudre ensuite à l'aide de solveurs comme les solveurs SAT et les solveurs SMT, dont on voit aussi les principes de fonctionnement. Le cours revisite la logique propositionnelle vue en cours d'« outils logiques » (OL4), et introduit la logique du premier ordre. Le programme général est :

— *Logique propositionnelle : syntaxe et sémantique. Conséquences et équivalences logiques. Formes normales, forme clausale. Modélisation, solveurs SAT, recherche de modèle, algorithme DPLL. Recherche de preuve, calcul des séquents propositionnel.*

— *Logique du premier ordre : syntaxe et sémantique. Formes normales, skolémisation. Théories logiques, interprétations normales, élimination des quantificateurs. Modélisation, solveurs SMT. Recherche de preuve, calcul des séquents du premier ordre.*

Les (sous-)sections dont le titre est précédé d'une astérisque « * » et grisées dans le texte apportent des compléments qui ne seront pas traités en cours.

Partie 1. Introduction 5
1. Contexte et motivations 5
1.1. Logique philosophique 5
1.2. Logique mathématique 5
1.3. Logique informatique 6
1.3.1. Circuits logiques 6
1.3.2. Complexité algorithmique 6
1.3.3. Problèmes « combinatoires » 7
1.3.4. Programmation logique 7
1.3.5. Bases de données 7
1.3.6. Vérification de programmes 8
1.4. Des nombreuses logiques 8

Partie 2. Logique classique propositionnelle 9
2. Syntaxe 9
2.1. Arbres de syntaxe abstraite 9
2.1.1. Représentation en Java 10
2.1.2. Représentation en OCaml 11
2.2. Syntaxe concrète 12

© 2019–2021 Sylvain Schmitz
licensed under Creative Commons License CC-BY-SA

3. Sémantique 13
3.1. Valeurs de vérité 13
3.1.1. Interprétations 13
3.1.2. Sémantique 13
3.1.3. Implémentation de la sémantique en Java 15
3.1.4. Implémentation de la sémantique en OCaml 15
3.1.5. Tables de vérité 16
3.1.6. Étendre la syntaxe 17
3.1.7. Complétude fonctionnelle 17
3.2. Satisfiabilité et validité 18
4. Conséquences et équivalences logiques 20
4.1. Conséquences logiques 20
4.2. Équivalences logiques 21
4.3. Substitutions propositionnelles 22
4.3.1. Implémentation des substitutions propositionnelles en Java 22
4.3.2. Implémentation des substitutions propositionnelles en OCaml 23
4.3.3. Lemme de substitution propositionnelle 23
4.4. Équivalences usuelles 24
5. Formes normales 26
5.1. Forme normale négative 26
5.1.1. Implémentation des formes normales négatives en Java 27
5.2. Forme clausale 28
5.2.1. Forme clausale logiquement équivalente 28
5.2.2. Forme clausale équi-satisfiable 30
5.2.3. Format DIMACS 31
5.2.4. Implémentation de la forme clausale en Java 32
5.3. Forme normale disjonctive 34
6. Modélisation 36
6.1. Utilisation de solveurs SAT 36
6.1.1. Utilisation de MiniSAT 36
6.1.2. Utilisation de Sat4j dans un programme Java 36
6.2. Exemple de modélisation : coloration de graphe 37
6.3. Exemple de modélisation : dépendances logicielles 39
7. Satisfiabilité et recherche de modèle 44
7.1. Recherche de modèle par énumération 44
7.1.1. Implémentation de la recherche par énumération en Java 46
7.2. Recherche de modèle par simplification 47
7.2.1. Simplification de formes clausales 47
7.2.2. Recherche par simplification 48
7.2.3. Correction et complétude 49
7.2.4. Implémentation de la recherche par simplification en Java 50
7.3. Algorithme de Davis, Putnam, Logemann et Loveland 51
7.3.1. Correction et complétude 52
7.3.2. Algorithme DPLL 53
7.3.3. Implémentation d'un DPLL récursif en Java 54
8. Validité et recherche de preuve 56
8.1. Calcul des séquents propositionnel 56
8.2. Recherche de preuve 58
8.2.1. Inversibilité et algorithme de recherche de preuve 60
8.2.2. Implémentation de la recherche de preuve en Java 62
8.3. Correction et complétude 64

9. * Clauses de HORN 66
9.1. Modèle minimal 67
9.1.1. Les fonctions f_S 67
9.1.2. Itérations de la fonction f_S 68
9.2. HORNSAT 68
9.2.1. Conséquences logiques d'un ensemble de clauses de HORN 69
9.2.2. Algorithme naïf 69
9.2.3. Algorithme en temps linéaire 71
9.2.4. Implémentation de l'algorithme en temps linéaire en Java 72
9.2.5. Implémentation de l'algorithme en temps linéaire en OCaml 74
9.3. Exemple de modélisation : accessibilité dans un graphe orienté 75

Partie 3. Logique classique du premier ordre 78
10. Structures 78
10.1. Signatures 78
10.2. Interprétations 78
10.3. Représentation des interprétations en OCaml 80
11. Syntaxe 82
11.1. Formules 82
11.2. Variables libres et variables liées 83
11.3. Syntaxe en OCaml 83
11.3.1. Syntaxe abstraite en OCaml 83
11.3.2. Syntaxe concrète en OCaml 84
11.3.3. Variables libres et variables liées en OCaml 84
12. Sémantique 85
12.1. Sémantique 85
12.2. Modèles, satisfiabilité et validité 85
12.3. Implémentation de la sémantique en OCaml 86
12.4. Exemples de formules 87
13. Substitutions 91
13.1. Lemme de substitution 91
13.2. α-renommages 92
13.3. Implémentation des substitutions en OCaml 94
14. Formes normales 95
14.1. Forme normale négative 95
14.1.1. Implémentation de la forme normale négative en OCaml 96
14.2. Forme prénexe 97
14.2.1. Implémentation de la forme prénexe en OCaml 98
14.3. Skolémisation 99
14.3.1. Implémentation de la skolémisation en OCaml 101
14.4. * Forme clausale 101
14.4.1. Implémentation de la forme clausale en OCaml 101
14.5. * Modèles de HERBRAND 103
15. Théories et modèles 105
15.1. Théories logiques 105
15.1.1. Théories de structures 106
15.1.2. Théories axiomatiques 106
15.1.3. Interprétations normales 107
15.1.4. Cohérence, complétude et décidabilité 111
15.2. Élimination des quantificateurs et décidabilité 112
15.2.1. Implémentation en OCaml de l'élimination des quantificateurs 113

15.2.2. Théorie des ordres linéaires denses non bornés 114
15.2.3. Théorie de l'arithmétique linéaire rationnelle 117
15.3. * Indécidabilité 118
15.3.1. Indécidabilité de la théorie de l'arithmétique élémentaire 118
15.3.2. Indécidabilité des équations diophantiennes 119
15.3.3. Théorèmes d'incomplétude 120
16. Satisfiabilité modulo théorie 121
16.1. Utilisation de solveurs SMT 121
16.1.1. Principes de base des solveurs SMT 121
16.1.2. Élimination des quantificateurs 122
16.1.3. SMT-LIB 123
16.1.4. Théories usuelles 126
16.2. Exemple de modélisation : nombre de McNuggets 126
16.3. Exemple de modélisation : apprentissage d'automates séparateurs 127
16.4. Exemple de modélisation : synthèse d'invariant de programme 130
16.5. * Exemple de modélisation : pavage du plan 134
17. Calcul des séquents 139
17.1. Correction 141
17.2. * Règles admissibles 142
17.2.1. α-congruence syntaxique 142
17.2.2. Substitution syntaxique 143
17.2.3. Affaiblissement 144
17.2.4. Axiome étendu 145
17.2.5. Inversibilité syntaxique 145
17.2.6. Contraction 146
17.3. * Complétude 148
17.3.1. Lemme de Hintikka 148
17.3.2. Théorème de complétude 149
17.4. * Élimination des coupures 150

Références 154
Quelques ouvrages sur la logique 154
Quelques textes fondateurs 154
Autres ouvrages 156
Autres références 156

Ces notes de cours ne remplacent pas une lecture approfondie d'ouvrages. Je recommande particulièrement le livre de Jacques Duparc (2015) pour débuter. Le livre de John Harrisson (2009) se concentre plutôt sur le raisonnement automatique et illustre tous ses concepts par du code OCaml, ce qui correspond à l'approche quelque peu « utilitariste » que j'adopte dans ces notes. Pour aller plus loin, les livres de Jean Goubault-Larrecq et Ian Mackie (1997) et de René David, Karim Nour et Christophe Raffalli (2003) sont de bonnes références.

Une partie de ces notes est inspirée des notes de cours du « MOOC » *Introduction à la logique informatique* par David Baelde, Hubert Comon et Étienne Lozes [1], ainsi que des transparents du cours de *Logique* de Delia Kesner [2], des notes du cours *Outils logiques* de Ralf Treinen [3] et celles de Roberto Amadio [4].

1. Voir les pages `https://www.fun-mooc.fr/courses/ENSCachan/20004S02/session02/about` et `https://www.fun-mooc.fr/courses/ENSCachan/20009/session01/about`
2. `https://www.irif.fr/~kesner/enseignement/licence/logique/`
3. `https://www.irif.fr/~kesner/enseignement/ol3/poly.pdf`
4. `https://cel.archives-ouvertes.fr/cel-00163821`

Partie 1. Introduction

> La logique, du grec λογική / logikê, est un terme dérivé de λόγος / lógos — signifiant à la fois « raison », « langage » et « raisonnement » — est, dans une première approche, l'étude des règles formelles que doit respecter toute argumentation correcte.
>
> Elle est depuis l'Antiquité l'une des grandes disciplines de la philosophie [...]. En outre, on a assisté depuis le XIXe siècle au développement fulgurant d'une approche mathématique de la logique. Sa convergence opérée avec l'informatique depuis la fin du XXe siècle lui a donné un regain de vitalité. [5]

1. Contexte et motivations

1.1. Logique philosophique. La motivation des philosophes antiques comme Aristote est de déterminer si un raisonnement est concluant. Par exemple, le raisonnement suivant est concluant : « Tous les hommes sont mortels, or Socrate est un homme, donc Socrate est mortel » ; en effet,

(1) d'une part les deux prémisses « Tous les humains sont mortels » et « Socrate est humain » sont vraies, et

(2) d'autre part, l'inférence de la conclusion « Socrate est mortel » à partir des prémisses est valide.

Les deux ingrédients (1) et (2) ci-dessus d'un raisonnement concluant sont indépendants. Ainsi,
- « Toutes les souris sont vertes, or Yoda est une souris, donc Yoda est vert » est un raisonnement valide mais non concluant car au moins une prémisse est fausse, tandis que
- « Tous les humains sont mortels, or Socrate est mortel, donc Socrate est humain » n'est pas concluant car l'inférence n'est pas valide – on parle alors de raisonnement fallacieux ou de *non sequitur*.

Remarquons en passant que les conclusions de ces deux raisonnements non concluants sont bien vraies : nous examinons ici le raisonnement, et non sa conclusion.

La logique s'intéresse à l'ingrédient (2) ci-dessus, c'est-à-dire à formaliser ce qui constitue un raisonnement valide. En termes modernes, on écrirait de nos jours une *formule* logique dans un langage formel, par exemple en logique du premier ordre :

$$\big((\forall x\,.\,H(x) \Rightarrow M(x)) \land H(s)\big) \Rightarrow M(s) \tag{1}$$

où « $H(x)$ » et « $M(x)$ » dénotent respectivement que x est humain et que x est mortel, et « s » dénote Socrate ; cette formule est bien valide.

Ceci ne constitue qu'un minuscule aperçu de la logique en tant que discipline philosophique, qui est un sujet actif de recherche ; d'ailleurs, une excellente source d'information en logique est la *Stanford Encyclopedia of Philosophy*[6].

1.2. Logique mathématique. La nécessité d'employer un langage clair, à l'abri d'ambiguïtés, pour écrire et démontrer des énoncés mathématiques est reconnue depuis l'Antiquité et par exemple la géométrie d'Euclide. La logique en tant que discipline mathématique prend son essor au XIXe siècle grâce aux travaux de mathématiciens tels que Boole, de Morgan et Frege.

Cela amène à la « crise des fondements » de la fin du XIXe siècle, quand des paradoxes remettent en question l'utilisation naïve des ensembles. Une version vulgarisée d'un de ces paradoxes, due à Russell, est connue comme le *paradoxe du barbier* : imaginons une ville où un barbier rase tous les hommes qui ne se rasent pas eux-mêmes (et seulement ceux-là) ; est-ce que ce barbier se rase lui-même ? Si l'on suppose que ce barbier soit un homme, que la réponse soit

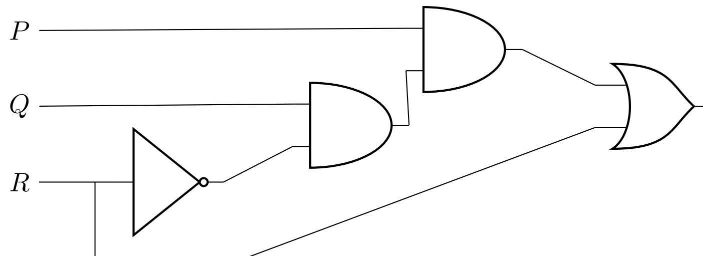
5. Article *Logique* de Wikipédia en français.
6. `https://plato.stanford.edu/`

oui ou non, on aboutit à une contradiction : ce barbier n'existe pas. La version ensembliste du paradoxe est la suivante : on définit $y \stackrel{\text{def}}{=} \{x \mid x \notin x\}$ la classe des ensembles qui ne se contiennent pas eux-mêmes ; est-ce que $y \in y$? Si l'on suppose que y est un ensemble, on aboutit à la contradiction $y \in y \Leftrightarrow y \notin y$. Le XX$^{\text{e}}$ siècle voit ainsi plusieurs tentatives pour formaliser la logique et les mathématiques, mais les limitations de ces approches apparaîtront bientôt grâce à Gödel, Turing et Church – vous en apprendrez plus en M1 en cours de « calculabilité et complexité ».

1.3. **Logique informatique.** Tout comme la logique permet de formaliser les énoncés mathématiques (en voyant ces énoncés comme des formules de la logique), on peut comprendre l'informatique via le prisme de la logique. Cette vision de l'informatique est incroyablement fructueuse. Un lien formel très fort existe en particulier entre programmes du côté informatique et preuves du côté logique – ceci est connu comme l'*isomorphisme de Curry-Howard* – qui se retrouve au cœur du fonctionnement d'assistants de preuve comme Coq ; vous pourrez en apprendre plus dans le cours de M1 de « preuves assistées par ordinateur » ou dans les cours sur la logique linéaire ou la théorie des types au MPRI.

Le point de vue de ce cours est cependant plutôt de voir la logique via le prisme de l'informatique. Les formules de la logique sont des objets très simples d'un point de vue informatique, à savoir des arbres. Les formules peuvent être manipulées par des programmes pour résoudre *automatiquement* quantité de problèmes.

1.3.1. *Circuits logiques.* Les circuits logiques qui composent les processeurs de nos ordinateurs sont des réalisations matérielles des formules de la logique propositionnelle. Par exemple, le circuit ci-dessous représente la formule propositionnelle $(P \wedge (Q \wedge \neg R)) \vee R$.



Raisonner sur les formules de la logique propositionnelle permet ainsi de raisonner sur les circuits : déterminer leur fonctionnalité, dire si deux circuits sont équivalents, minimiser la taille d'un circuit, etc. Ce sujet sera approfondi en M1 dans le cours de « circuits et architecture ».

1.3.2. *Complexité algorithmique.* En cours de M1 de « calculabilité et complexité », vous apprendrez que des problèmes logiques fournissent les exemples emblématiques de problèmes informatiques difficiles : SAT, la satisfiabilité des formules propositionnelles, est complet pour une classe de complexité appelée NP, et il en est de même pour d'autres classes de complexité (c.f. table 1).

Table 1. Complexité algorithmique de quelques problèmes de logique.

Problème	Complexité
QBF	PSPACE
SAT	NP
HornSAT	P
2SAT	NL

1.3.3. *Problèmes « combinatoires ».* Le point précédent signifie que quantité de problèmes informatiques peuvent être résolus en les *réduisant* à des problèmes logiques. L'intérêt est que nous disposons de logiciels extrêmement optimisés pour résoudre ces problèmes logiques, en particulier des *solveurs* SAT[7]. On parle même de « révolution SAT » car les performances actuelles de ces solveurs permettent de résoudre des problèmes qui paraissaient hors de portée.

En guise d'illustration, nous verrons entre autres comment résoudre des grilles de sudoku comme celle ci-dessous en faisant appel à un solveur SAT.[8]

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

9	8	7	6	5	4	3	2	1
2	4	6	1	7	3	9	8	5
3	5	1	9	2	8	7	4	6
1	2	8	5	3	7	6	9	4
6	3	4	8	9	2	1	5	7
7	9	5	4	6	1	8	3	2
5	1	9	2	8	6	4	7	3
4	7	2	3	1	9	5	6	8
8	6	3	7	4	5	2	1	9

1.3.4. *Programmation logique.* Dans la lignée du point précédent, le besoin en informatique de résoudre des problèmes qui peuvent s'exprimer sous la forme de contraintes (par exemple pour le sudoku, chaque ligne, colonne, et chacun des carrés 3×3 doit avoir exactement une occurrence de chaque nombre de 1 à 9) est tel que des langages de programmation spécialisés ont été développés. Dans un langage comme Prolog ou Mozart/Oz, le programmeur fournit les contraintes du problème, et laisse le système trouver les valeurs des inconnues qui répondent au problème. Ce paradigme de programmation, différent de la programmation impérative et de la programmation fonctionnelle, sera étudié dans le cours de M1 « programmation logique et par contraintes ».

1.3.5. *Bases de données.* En bases de données, le théorème de Codd relie les opérations que l'on peut effectuer sur une base de données relationnelle (l'algèbre relationnelle) aux requêtes que l'ont peut écrire (le calcul relationnel, c'est-à-dire la logique du premier ordre). Le langage SQL fournit une autre façon d'écrire des requêtes du calcul relationnel. Par exemple, la requête

```sql
SELECT Vols.depart
FROM Vols JOIN Aeroports ON Vols.arrivee = Aeroports.nom
WHERE Aeroports.pays = 'FR'
```

sur la base de données $\mathbb{D}$ ci-dessous retourne les codes des aéroports depuis lesquels on peut rejoindre la France, à savoir les aéroports 'ATL' et 'AMS'.

Aeroports	
nom	pays
'ATL'	'US'
'PEK'	'CN'
'LAX'	'US'
'CDG'	'FR'
'AMS'	'NL'
'FRA'	'DE'
'ORY'	'FR'

Vols	
depart	arrivee
'ATL'	'CDG'
'LAX'	'ATL'
'ORY'	'FRA'
'CDG'	'PEK'
'AMS'	'ORY'
'AMS'	'ATL'
'PEK'	'LAX'

7. Par exemple, MiniSat (`http://minisat.se/`), Glucose (`https://www.labri.fr/perso/lsimon/glucose/`) ou Sat4j (`http://www.sat4j.org/`); vous pouvez aussi tester logictools en ligne (`http://logictools.org/`).

8. Voir par exemple `https://blag.cedeela.fr/sudoku-msat/` pour une démonstration en ligne.

Cette requête peut s'écrire comme une formule $\varphi(\mathit{depart})$ en logique du premier ordre, définie comme :

$$\varphi(\mathit{depart}) \stackrel{\text{def}}{=} \exists \mathit{arrivee} \exists \mathit{nom} \exists \mathit{pays} \,.\, \mathit{Vols}(\mathit{depart}, \mathit{arrivee}) \land \mathit{Aeroports}(\mathit{nom}, \mathit{pays}) \\ \land\, \mathit{arrivee} = \mathit{nom} \land \mathit{pays} = \text{'FR'} \qquad (2)$$

Répondre à une telle requête SQL sur une base de données $\mathbb{D}$ comme celle ci-dessus revient à évaluer la formule logique sur la structure associée à $\mathbb{D}$. Optimiser les requêtes SQL revient trouver des formules équivalentes.

1.3.6. *Vérification de programmes.* Les programmes et systèmes informatiques sont sujets à quantité d'erreurs. Une façon de s'assurer qu'ils ont bien le fonctionnement voulu est d'écrire une *spécification* formelle du comportement attendu ; naturellement, ces spécifications sont écrites à l'aide de formules logiques. Par exemple, si on examine les événements successifs d'un système composé d'un client et d'un serveur, on pourrait vérifier qu'à chaque instant t où le client fait une requête ($\mathit{Requete}(t)$), il y a un instant t' plus tard où le serveur accède à la demande ($\mathit{Acces}(t')$) ; en logique du premier ordre,

$$\forall t \,.\, \mathit{Requete}(t) \Rightarrow \exists t' \,.\, t \leq t' \land \mathit{Acces}(t') \qquad (3)$$

Ce genre de spécifications sera étudié dans le cours de M2 « modélisation et spécification », où vous verrez aussi comment s'assurer automatiquement qu'un système satisfait sa spécification.

Une autre approche est plutôt basée sur les « preuves assistées par ordinateur ». Par exemple, on pourrait souhaiter démontrer que le code suivant calcule $r = n!$ pour un argument n :

```
int i = 0;
int r = 1;
while (i != n) {
  i++;
  r *= i;
}
```

Une approche bien connue pour cela est d'annoter le code avec des *pré-conditions* et des *post-relations* : chaque instruction `I` du code est annotée comme un *triplet de* HOARE $\{\varphi\}\ \texttt{I}\ \{\psi\}$ où φ et ψ sont des formules logiques portant sur les valeurs des variables du programme. Voici une preuve en *logique de* HOARE que le code ci-dessus est correct, c'est-à-dire que s'il termine alors $r = n!$.

$$\dfrac{\dfrac{}{\{\top\}\ \texttt{int i=0}\ \{i=0\}}\,\text{init} \quad \dfrac{\dfrac{\dfrac{}{\{i=0\}\ \texttt{int r=1}\ \{r=1\land i=0\}}\,\text{init}}{\{i=0\}\ \texttt{int r=1}\ \{r=i!\}}\,\text{csq} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{r(i+1)=(i+1)!\}\ \texttt{i++}\ \{ri=i!\}}\,\text{aff}}{\{r=i!\}\ \texttt{i++}\ \{ri=i!\}}\,\text{csq} \quad \dfrac{}{\{ri=i!\}\ \texttt{r*=i}\ \{r=i!\}}\,\text{aff}}{\{r=i!\}\ \texttt{i++;r*=i}\ \{r=i!\}}\,\text{seq}}{\{r=i!\land i\neq n\}\ \texttt{i++;r*=i}\ \{r=i!\}}\,\text{csq}}{\{r=i!\}\ \texttt{while(i!=n)\{i++;r*=i\}}\ \{r=i!\land i=n\}}\,\text{while}}{\{i=0\}\ \texttt{int r=1;while(i!=n)\{i++;r*=i\}}\ \{r=i!\land i=n\}}\,\text{seq}}{\{\top\}\ \texttt{int i=0;int r=1;while(i!=n)\{i++;r*=i\}}\ \{r=i!\land i=n\}}\,\text{seq}$$

1.4. **Des nombreuses logiques.** Afin de répondre à des besoins de modélisation formelle et de raisonnement formel, que ce soit en philosophie, mathématiques ou informatique, de nombreuses logiques ont été définies et de nouvelles sont constamment développées : logique modale, temporelle, de description, d'ordre supérieur, intuitioniste, linéaire, de séparation, hybride, infinitaire, etc.

Ce cours est l'occasion d'étudier deux logiques classiques, qui sont à la base de toutes les autres : la *logique propositionnelle*, que vous avez déjà rencontrée en cours d'« outils logiques » en L2, et la *logique du premier ordre*, que vous avez utilisée en mathématiques sans le savoir depuis des années.

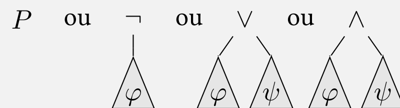
Partie 2. Logique classique propositionnelle

Il existe quantité de logiques employées en informatique, mathématiques et philosophie. La logique la plus simple, qui sert de base aux développement de la plupart des autres logiques, est la logique propositionnelle, aussi appelée « calcul des propositions ». Dans cette logique, une « proposition » dénote un énoncé qui peut être vrai ou faux, comme « il pleut aujourd'hui » ou « $1 + 1 = 2$ ». La logique propositionnelle permet de combiner de telles propositions, traitées comme des « variables propositionnelles », au moyen de connecteurs logiques pour construire des énoncés plus complexes appelés « formules ». Cette logique est aussi celle employée dans les solveurs SAT, qui permettent de résoudre en pratique des problèmes informatiques difficiles en les encodant comme des formules propositionnelles.

Commençons, afin de fixer les notations employées dans ces notes, par définir la syntaxe et la sémantique de la logique propositionnelle.

2. Syntaxe

Résumé. Étant donné un ensemble dénombrable $\mathcal{P}_0$ de *propositions*, les formules propositionnelles sont des arbres dont les feuilles sont des propositions et les nœuds internes des connecteurs logiques. Ainsi, une *formule propositionnelle* est un arbre de la forme



où $P \in \mathcal{P}_0$ et φ et ψ sont des formules propositionnelles.

2.1. **Arbres de syntaxe abstraite.** Soit $\mathcal{P}_0$ un ensemble infini dénombrable de symboles de propositions (aussi appelés « variables propositionnelles »). La syntaxe de la logique propositionnelle est définie par la syntaxe abstraite

(Duparc, 2015, sec. 1.2), (David, Nour et Raffalli, 2003, sec. 1.2.6), (Goubault-Larrecq et Mackie, 1997, sec. 2.1), (Harrisson, 2009, sec. 2.1)

$$\varphi ::= P \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \qquad \text{(formules propositionnelles)}$$

où $P \in \mathcal{P}_0$.

Concrètement, cela signifie qu'une formule propositionnelle est un *arbre* fini, dont les feuilles sont étiquetées par des propositions tirées de $\mathcal{P}_0$, et dont les nœuds internes sont étiquetés soit par $\neg$ (pour « non ») et ont exactement un nœud enfant, soit par $\vee$ (pour « ou ») ou $\wedge$ (pour « et ») et ont exactement deux nœuds enfants. Par exemple, la figure 1 décrit une formule propositionnelle φ_{ex} où P et Q sont des propositions de $\mathcal{P}_0$.

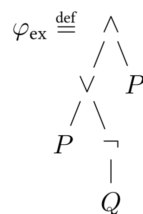


Figure 1. Une formule propositionnelle.

L'intérêt de travailler avec des arbres de syntaxe abstraite est que ceux-ci se prêtent très bien aux définitions et aux algorithmes par récurrence. Par exemple, on peut définir l'ensemble

$\text{fp}(\varphi) \subseteq \mathcal{P}_0$ des propositions qui apparaissent au moins une fois dans une formule propositionnelle φ :

$$\text{fp}(P) \stackrel{\text{def}}{=} \{P\}\ , \qquad \text{fp}(\neg\varphi) \stackrel{\text{def}}{=} \text{fp}(\varphi)\ ,$$

$$\text{fp}(\varphi \vee \psi) \stackrel{\text{def}}{=} \text{fp}(\varphi) \cup \text{fp}(\psi) \qquad \text{fp}(\varphi \wedge \psi) \stackrel{\text{def}}{=} \text{fp}(\varphi) \cup \text{fp}(\psi).$$

Dans l'exemple de la figure 1, $\text{fp}(\varphi_{\text{ex}}) = \{P, Q\}$.

2.1.1. *Représentation en Java.* On peut représenter les formules propositionnelles en Java de manière naturelle à l'aide d'une classe abstraite et de sous-classes pour chacun des types de nœuds comme ci-dessous. À noter que l'on utilise ici l'ensemble des objets de type `String` en guise d'ensemble de propositions $\mathcal{P}_0$.

```
Formule
public abstract class Formule {
    // sous-classes
    public static class Et extends Formule {
        protected Formule phi1; // sous-formule gauche
        protected Formule phi2; // sous-formule droite
        public Et (Formule phi1, Formule phi2) {
            this.phi1 = phi1;
            this.phi2 = phi2;
        }
        // méthodes pour formules ∧
        // ⋮
    }
    public static class Ou extends Formule {
        protected Formule phi1; // sous-formule gauche
        protected Formule phi2; // sous-formule droite
        public Ou (Formule phi1, Formule phi2) {
            this.phi1 = phi1;
            this.phi2 = phi2;
        }
        // méthodes pour formules ∨
        // ⋮
    }
    public static class Non extends Formule {
        protected Formule phi1; // sous-formule
        public Non (Formule phi1) {
            this.phi1 = phi1;
        }
        // méthodes pour formules ¬
        // ⋮
    }
    public static class Proposition extends Formule {
        protected String nom; // nom de la proposition
        public Proposition (String nom) {
            this.nom = nom;
        }
        // méthodes pour propositions
        // ⋮
    }
    // méthodes de la classe abstraite
    // ⋮
}
```

Avec ce code, on peut construire la formule propositionnelle de la figure 1 par

```
Formule phi = new Et
    (new Ou
         (new Proposition("P"),
          new Non(new Proposition("Q"))),
     new Proposition("Q"));
```

Les choses se gâtent quelque peu quand on souhaite implémenter des méthodes sur les formules propositionnelles. Par exemple, pour calculer l'ensemble $\mathrm{fp}(\varphi)$, l'idée naturelle est d'importer `java.util.*`, de déclarer une méthode abstraite de la classe `Formule`

```
Formule
public abstract Set<String> getPropositions();
```

et de l'implémenter dans chacune des sous-classes. Par exemple :

```
Formule.Et, Formule.Ou
public Set<String> getPropositions() {
    Set<String> propositions = phi1.getPropositions();
    propositions.addAll(phi2.getPropositions());
    return propositions;
}
Formule.Non
public Set<String> getPropositions() {
    return phi1.getPropositions();
}
Formule.Proposition
public Set<String> getPropositions() {
    Set<String> propositions = new TreeSet<String>();
    propositions.add(nom);
    return propositions;
}
```

On voit là un défaut de notre représentation Java, qui disperse le code des différents cas dans plusieurs locations des fichiers sources.

2.1.2. *Représentation en OCaml.* Un autre de langage de programmation que vous allez découvrir cette année dans le cours de « programmation fonctionnelle » est OCaml. Ce langage s'avère bien mieux adapté au style de programmes que l'ont souhaite écrire en logique. Par exemple, les formules propositionnelles peuvent utiliser le type abstrait ci-dessous (on utilise ici l'ensemble des chaînes de caractères comme ensemble de propositions $\mathcal{P}_0$).

```
type formule = Proposition of string
             | Non of formule
             | Et  of formule * formule
             | Ou  of formule * formule
```

La formule propositionnelle de la figure 1 se construit par

```
let phi = Et (Ou (Proposition "P", Non (Proposition "Q")),
              Proposition "P")
```

Écrire des programmes récursifs en OCaml sur une telle représentation est très aisé. Voici par exemple un programme qui calcule $\mathrm{fp}(\varphi)$ (en utilisant une fonction auxiliaire `enleve_duplicats` de type `'a list -> 'a list` qui retourne une liste sans duplicats) :

```
let rec liste_propositions = function
  | Proposition p   -> [p]
  | Non phi         -> liste_propositions phi
  | Et (phi1, phi2) -> liste_propositions phi1 @ liste_propositions phi2
  | Ou (phi1, phi2) -> liste_propositions phi1 @ liste_propositions phi2
let propositions phi = enleve_duplicats (liste_propositions phi)
```

(Duparc, 2015, sec. 1.4)

2.2. **Syntaxe concrète.** Cependant, dessiner des arbres à chaque fois que l'on veut écrire une formule propositionnelle est plutôt laborieux. On utilise plutôt une écriture « linéaire » en introduisant des parenthèses de manière judicieuse. Par exemple, la formule propositionnelle de la figure 1 s'écrit $\varphi_{\text{ex}} \stackrel{\text{def}}{=} ((P \vee \neg Q) \wedge P)$. On se permet généralement des facilités d'écritures, comme $(P \vee \neg Q) \wedge P$ pour φ_{ex} – où l'on a enlevé les parenthèses extérieures –, ou $P \wedge Q \wedge R$ pour $((P \wedge Q) \wedge R)$ ou $(P \wedge (Q \wedge R))$ – car les opérateurs $\wedge$ et $\vee$ sont associatifs (voir section 4).

La syntaxe concrète d'une formule propositionnelle s'implémente très aisément par un programme récursif ; par exemple, en OCaml :

```
let rec string_of_formule phi =
  match phi with
  | Proposition p   -> p
  | Non phi1        -> "¬" ^ string_of_formule phi1
  | Et (phi1, phi2) ->
    "(" ^ string_of_formule phi1 ^ " ⊠ " ^ string_of_formule phi2 ^ ")"
  | Ou (phi1, phi2) ->
    "(" ^ string_of_formule phi1 ^ " ⊠ " ^ string_of_formule phi2 ^ ")"
```

3. Sémantique

Résumé. Soit $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$ l'ensemble des *valeurs de vérité*, où 0 désigne « faux » et 1 désigne « vrai ». Étant donnée une *interprétation* $I: \mathcal{P}_0 \to \mathbb{B}$, la *sémantique* $[\![\varphi]\!]^I$ d'une formule propositionnelle est une valeur de vérité, qui ne dépend en réalité que des propositions qui apparaissent dans φ (propriété 3.3). On note « $I \vDash \varphi$ » si $[\![\varphi]\!]^I = 1$.

Ainsi, on peut aussi voir $[\![\varphi]\!]$ comme une *fonction booléenne* qui prend en argument les valeurs de vérité de ses propositions et retourne une valeur de vérité, et pour laquelle on peut écrire une *tables de vérité*. Inversement, pour toute fonction booléenne f, il existe une formule propositionnelle φ telle que $f = [\![\varphi]\!]$ (théorème 3.8 de complétude fonctionnelle).

Une formule propositionnelle φ est *satisfiable* s'il existe une interprétation I telle que $[\![\varphi]\!]^I = 1$. Elle est *valide* (noté « $\vDash \varphi$ ») si pour toute interprétation I, on a $[\![\varphi]\!]^I = 1$.

3.1. **Valeurs de vérité.** On note $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$ pour l'ensemble des *valeurs de vérité* : l'intuition est que 1 dénote la valeur « vrai » et 0 la valeur « faux ». On définit aussi trois opérations $\text{not}: \mathbb{B} \to \mathbb{B}$ et $\text{and}, \text{or}: \mathbb{B}^2 \to \mathbb{B}$ définies par $\text{not}\,1 = 0 \text{ or } 0 = 0 \text{ and } 0 = 1 \text{ and } 0 = 0 \text{ and } 1 = 0$ et $\text{not}\,0 = 1 \text{ or } 1 = 1 \text{ or } 0 = 0 \text{ or } 1 = 1 \text{ and } 1 = 1$ (voir la table 2). On appelle aussi $\mathbb{B}$ muni des trois opérations not , or et and « l'algèbre de Boole ».

Remarque 3.1. En français, le mot « ou » est ambigu, dans la mesure où il peut être compris de manière *inclusive* (A ou B ou les deux) ou *exclusive* (A ou B mais pas les deux). Dans l'usage courant, les deux cas A et B sont souvent implicitement exclusifs (« Je me lève à sept ou huit heures selon les jours »). Le « or » logique est en revanche inclusif.

(Duparc, 2015, sec. 2.3)

3.1.1. *Interprétations.* Une *interprétation* est une fonction (aussi appelée une « valuation propositionnelle ») $I: \mathcal{P}_0 \to \mathbb{B}$ qui associe une valeur de vérité $P^I \in \mathbb{B}$ pour chaque proposition $P \in \mathcal{P}_0$. Si I est une interprétation et P est une proposition, on écrit $I[1/P]$ (resp. $I[0/P]$) pour l'interprétation qui associe 1 à P (resp. 0) et Q^I à Q pour tout $Q \neq P$.

(Duparc, 2015, sec. 2.1)
On peut aussi voir une interprétation comme un sous-ensemble $I \stackrel{\text{def}}{=} \{P \in \mathcal{P}_0 \mid P^I = 1\}$ de propositions dans $2^{\mathcal{P}_0}$; les deux points de vue sont bien sûr équivalents.

3.1.2. *Sémantique.* La sémantique $[\![\varphi]\!]^I \in \mathbb{B}$ d'une formule propositionnelle φ dans une interprétation I est définie inductivement par

(Duparc, 2015, sec. 2.2), (Goubault-Larrecq et Mackie, 1997, def. 2.8), (Harrisson, 2009, sec. 2.2)

$$[\![P]\!]^I \stackrel{\text{def}}{=} P^I \;, \quad [\![\neg\varphi]\!]^I \stackrel{\text{def}}{=} \text{not}\,[\![\varphi]\!]^I \;, \quad [\![\varphi \vee \psi]\!]^I \stackrel{\text{def}}{=} [\![\varphi]\!]^I \text{ or } [\![\psi]\!]^I \;, \quad [\![\varphi \wedge \psi]\!]^I \stackrel{\text{def}}{=} [\![\varphi]\!]^I \text{ and } [\![\psi]\!]^I \;.$$

On dit que I *satisfait* φ (ou que I est un « modèle » de φ), noté $I \vDash \varphi$, si $[\![\varphi]\!]^I = 1$; cette écriture peut être définie de manière équivalente par

$$\begin{aligned} I &\vDash P && \text{si } P \in I \;, \\ I &\vDash \neg\varphi && \text{si } I \nvDash \varphi \;, \\ I &\vDash \varphi \vee \psi && \text{si } I \vDash \varphi \text{ ou } I \vDash \psi \;, \\ I &\vDash \varphi \wedge \psi && \text{si } I \vDash \varphi \text{ et } I \vDash \psi \;. \end{aligned}$$

On définit aussi $\text{Sat}(\varphi) \stackrel{\text{def}}{=} \{I \in \mathbb{B}^{\mathcal{P}_0} \mid I \vDash \varphi\}$ l'ensemble des interprétations qui satisfont φ.

On peut aussi définir la sémantique d'une formule propositionnelle via un jeu d'évaluation ; voir (Duparc, 2015, sec. 2.7).

Exemple 3.2. Considérons à nouveau la formule propositionnelle $\varphi_{\text{ex}} = (P \vee \neg Q) \wedge P$ de la figure 1, et l'interprétation I qui associe $P^I = 0$ à P, $Q^I = 0$ à Q, et $R^I = 1$ à toutes les

propositions $R \in \mathcal{P}_0$ différentes de P et Q. La sémantique $[\![\varphi_{\text{ex}}]\!]^I$ se calcule comme suit :

$$\begin{aligned}[\![(P \vee \neg Q) \wedge P]\!]^I &= [\![P \vee \neg Q]\!]^I \text{ and } [\![P]\!]^I \\ &= ([\![P]\!]^I \text{ or } [\![\neg Q]\!]^I) \text{ and } [\![P]\!]^I \\ &= (P^I \text{ or } [\![\neg Q]\!]^I) \text{ and } [\![P]\!]^I \\ &= (0 \text{ or } [\![\neg Q]\!]^I) \text{ and } [\![P]\!]^I \\ &= (0 \text{ or not } [\![Q]\!]^I) \text{ and } [\![P]\!]^I \\ &= (0 \text{ or not } Q^I) \text{ and } [\![P]\!]^I \\ &= (0 \text{ or not } 0) \text{ and } [\![P]\!]^I \\ &= (0 \text{ or } 1) \text{ and } [\![P]\!]^I \\ &= 1 \text{ and } [\![P]\!]^I \\ &= 1 \text{ and } P^I \\ &= 1 \text{ and } 0 \\ &= 0\ .\end{aligned}$$

À noter que l'on aurait pu atteindre ce résultat bien plus vite en écrivant

$$\begin{aligned}[\![(P \vee \neg Q) \wedge P]\!]^I &= [\![P \vee \neg Q]\!]^I \text{ and } [\![P]\!]^I \\ &= [\![P \vee \neg Q]\!]^I \text{ and } P^I \\ &= [\![P \vee \neg Q]\!]^I \text{ and } 0 \\ &= 0\ ,\end{aligned}$$

sans se préoccuper d'évaluer $[\![P \vee \neg Q]\!]^I$.

On peut observer que la valeur de vérité de φ ne dépend que de l'interprétation des propositions de $\text{fp}(\varphi)$: si $P \notin \text{fp}(\varphi)$, alors pour toute interprétation I, $[\![\varphi]\!]^{I[1/P]} = [\![\varphi]\!]^{I[0/P]}$. Cela se démontre par induction structurelle sur φ comme suit.

(Harrisson, 2009, thm. 2.2)

Propriété 3.3. *Pour toute formule propositionnelle φ et interprétations I et I', si $P^I = P^{I'}$ pour toute proposition $P \in \text{fp}(\varphi)$, alors $[\![\varphi]\!]^I = [\![\varphi]\!]^{I'}$.*

Démonstration. Par induction structurelle sur φ.

Pour le cas de base $\varphi = P$, on a $P \in \text{fp}(\varphi)$ donc $P^I = P^{I'}$ par hypothèse sur I et I', et on a bien

$$[\![P]\!]^I = P^I = P^{I'} = [\![P]\!]^{I'}\ .$$

Pour l'étape d'induction, si $\varphi = \neg\psi$, on a $\text{fp}(\varphi) = \text{fp}(\psi)$, donc $P^I = P^{I'}$ pour toute proposition $P \in \text{fp}(\psi)$ et on peut appliquer l'hypothèse d'induction à la sous-formule ψ : on a bien

$$[\![\neg\psi]\!]^I = \text{not } [\![\psi]\!]^I \overset{\text{i.h.}}{=} \text{not } [\![\psi]\!]^{I'} = [\![\neg\psi]\!]^{I'}\ .$$

Si $\varphi = \psi \vee \psi'$, on a $\text{fp}(\varphi) = \text{fp}(\psi) \cup \text{fp}(\psi')$, donc $P^I = P^{I'}$ pour toute proposition $P \in \text{fp}(\psi)$ ou $P \in \text{fp}(\psi')$ et on peut appliquer l'hypothèse d'induction aux deux sous-formules ψ et ψ' : on a bien

$$[\![\psi \vee \psi']\!]^I = [\![\psi]\!]^I \text{ or } [\![\psi']\!]^I \overset{\text{i.h.}}{=} [\![\psi]\!]^{I'} \text{ or } [\![\psi']\!]^{I'} = [\![\psi \vee \psi']\!]^{I'}\ .$$

Enfin, le cas où $\varphi = \psi \wedge \psi'$ est similaire au précédent. □

La propriété 3.3 signifie que, pour une interprétation I et une formule propositionnelle φ, seules les valeurs de vérité P^I pour $P \in \text{fp}(\varphi)$ influencent la sémantique $[\![\varphi]\!]^I$. On pourrait donc aussi employer des interprétations *partielles* $I\colon \mathcal{P}_0 \nrightarrow \mathbb{B}$ pour peu que leur domaine $\text{dom}(I)$ contienne les propositions de $\text{fp}(\varphi)$.

Introduisons quelques notations supplémentaires pour ces interprétations partielles. Pour des propositions distinctes $P_1, \ldots, P_n$ et des valeurs de vérité $b_1, \ldots, b_n$, on note $[b_1/P_1, \ldots, b_n/P_n]$ pour l'interprétation partielle de domaine fini $\{P_1, \ldots, P_n\}$ telle que $P_j^{[b_1/P_1,\ldots,b_n/P_n]} \stackrel{\text{def}}{=} b_j$ pour tout $1 \leq j \leq n$. Pour deux interprétations partielles I et I', on dit que I' *étend* I ou que I est la *restriction* de I' à $\text{dom}(I)$, et on écrit $I \sqsubseteq I'$, si $\text{dom}(I) \subseteq \text{dom}(I)'$ et pour toute proposition $P \in \text{dom}(I)$, $P^I = P^{I'}$.

Exemple 3.4. Comme vu dans l'exemple 3.2, on a $[\![\varphi_{\text{ex}}]\!]^I = 0$ pour toute interprétation I qui étend l'interprétation partielle $[0/P, 0/Q]$.

3.1.3. *Implémentation de la sémantique en Java.* Revenons à notre implémentation des formules propositionnelles en Java commencée en section 2.1.1. Une interprétation I pourrait naturellement être représentée comme un objet de type `Function<String,Boolean>` (du package `java.util.function`). Mais au vu de ce qui précède, nous pouvons nous contenter d'interprétations partielles de domaine fini, qui peuvent être représentées comme des objets de type `Map<String,Boolean>`. L'objectif est donc d'implémenter la méthode abstraite suivante de la classe `Formule` :

Formule
```
public abstract boolean evalue(Map<String,Boolean> interpretation);
```

Dans les différentes sous-classes, on pourrait écrire pour cela

⚠ *Avec ce code, il faut vérifier que* $\text{fp}(\varphi) \subseteq \text{dom}(I)$ *avant d'exécuter* `evalue`*, par exemple via une assertion* `interpretation.keySet().containsAll(getPropositions());`

Formule.Et
```
public boolean evalue(Map<String,Boolean> interpretation) {
    return phi1.evalue(interpretation)
        && phi2.evalue(interpretation);
}
```
Formule.Ou
```
public boolean evalue(Map<String,Boolean> interpretation) {
    return phi1.evalue(interpretation)
        || phi2.evalue(interpretation);
}
```
Formule.Non
```
public boolean evalue(Map<String,Boolean> interpretation) {
    return !phi1.evalue(interpretation);
}
```
Formule.Proposition
```
public boolean evalue(Map<String,Boolean> interpretation) {
    return interpretation.get(nom).booleanValue();
}
```

3.1.4. *Implémentation de la sémantique en OCaml.* Revenons maintenant à notre implémentation des formules propositionnelles en OCaml de la section 2.1.2. Dans ce cas, nous verrons une interprétation comme une fonction de type `string -> bool`. Une interprétation partielle de domaine fini peut s'implémenter en levant une exception ; voici un exemple de code pour $[0/P, 0/Q]$:

```
let i = function
    "P" -> false
  | "Q" -> false
  | p -> failwith ("Proposition "^p^" non interpretee")
```

L'évaluation d'une formule propositionnelle se fait récursivement :

```
let rec evalue phi interpretation =
  match phi with
    Et(phi1, phi2) -> (evalue phi1 interpretation)
      && (evalue phi2 interpretation)
  | Ou(phi1, phi2) -> (evalue phi1 interpretation)
      || (evalue phi2 interpretation)
  | Non phi1       -> not (evalue phi1 interpretation)
  | Proposition p  -> interpretation(p)
```

(Duparc, 2015, sec. 2.8)

3.1.5. *Tables de vérité.* Les opérateurs not, or et and sont des *fonctions booléennes*, c'est-à-dire des fonctions $f\colon \mathbb{B}^n \to \mathbb{B}$ pour un certain $n > 0$. Une façon de présenter de telles fonctions est sous la forme de *tables de vérité*, où chaque ligne indique les valeurs possibles des arguments $x_1, \dots, x_n$ de la fonction ainsi que la valeur de $f(x_1, \dots, x_n)$; voir la table 2 pour les tables de vérité de not, or et and.

Table 2. Les tables de vérité des fonctions not, or et and.

x_1	not x_1
1	0
0	1

x_1	x_2	x_1 or x_2
1	1	1
1	0	1
0	1	1
0	0	0

x_1	x_2	x_1 and x_2
1	1	1
1	0	0
0	1	0
0	0	0

Soit φ une formule propositionnelle. Par la propriété 3.3, la satisfaction de φ ne dépend que de l'interprétation des propositions de $\text{fp}(\varphi)$. On peut ainsi voir φ comme définissant une fonction des interprétations partielles $I \in \mathbb{B}^{\text{fp}(\varphi)}$ dans $\mathbb{B}$: $[\![\varphi]\!](I) \stackrel{\text{def}}{=} [\![\varphi]\!]^I$. C'est donc une fonction booléenne $\mathbb{B}^n \to \mathbb{B}$ à $n = |\text{fp}(\varphi)|$ variables, et on peut en donner la table de vérité. Cependant, plutôt que de seulement donner la table de vérité de $[\![\varphi]\!]$, il peut être pratique de donner par la même occasion la table de vérité de chacune de ses sous-formules.

Table 3. La table de vérité de la formule propositionnelle φ_{ex} de la figure 1.

P	Q	$\neg Q$	$P \vee \neg Q$	$(P \vee \neg Q) \wedge P$
1	1	0	1	1
1	0	1	1	1
0	1	0	0	0
0	0	1	1	0

Exemple 3.5. Pour la formule propositionnelle φ_{ex} de la figure 1, $[\![\varphi_{\text{ex}}]\!]$ est une fonction à deux variables qui représentent les valeurs de $[\![P]\!]^I$ et $[\![Q]\!]^I$. La formule propositionnelle φ_{ex} a pour sous-formules P, Q, $\neg Q$, $P \vee \neg Q$, ainsi que $(P \vee \neg Q) \wedge P$. La table de vérité correspondante est donnée dans la table 3.

Les deux colonnes « $\neg Q$ » et « $P \vee \neg Q$ » contiennent des calculs intermédiaires. La colonne « $\neg Q$ » est obtenue en appliquant not à la colonne Q ; la colonne « $P \vee \neg Q$ » l'est en appliquant or aux colonnes P et $\neg Q$; enfin, la colonne « $(P \vee \neg Q) \wedge P$ » est obtenue en appliquant and aux colonnes $P \vee \neg Q$ et P. À noter que la sémantique $[\![\varphi_{\text{ex}}]\!]^I = 0$ calculée dans l'exemple 3.2 pour I étendant $[0/P, 0/Q]$ correspond à la dernière ligne de la table.

On peut remarquer dans cette table que les colonnes « P » et « $(P \vee \neg Q) \wedge P$ » contiennent les mêmes valeurs de vérité : ces deux formules propositionnelles sont dites *fonctionnellement équivalentes*, car elles définissent les mêmes fonctions $[\![P]\!] = [\![(P \vee \neg Q) \wedge P]\!]$.

3.1.6. *Étendre la syntaxe.* D'autres opérations booléennes que not, or et and pourraient être utilisées dans la syntaxe des formules propositionnelles. Par exemple, il y a $2^{2^2} = 16$ fonctions booléennes à deux arguments, dont les fonctions indiquées dans la table 4.

(Duparc, 2015, sec. 2.11)

TABLE 4. Les tables de vérité des fonctions booléennes xor (« ou exclusif »), impl (« implique »), equiv (« si et seulement si ») et nand (« non et »).

x_1	x_2	x_1 xor x_2	x_1 impl x_2	x_1 equiv x_2	x_1 nand x_2
1	1	0	1	1	0
1	0	1	0	0	1
0	1	1	1	0	1
0	0	0	1	1	1

Pour chacune de ces fonctions, on pourrait étendre la syntaxe abstraite des formules propositionnelles pour s'autoriser à les utiliser :

$$\varphi ::= \cdots \mid \varphi \oplus \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi \mid \varphi \uparrow \varphi \,,$$

en étendant de même la sémantique par

$$[\![\varphi \oplus \psi]\!]^I \stackrel{\text{def}}{=} [\![\varphi]\!]^I \text{ xor } [\![\psi]\!]^I \,, \qquad [\![\varphi \Rightarrow \psi]\!]^I \stackrel{\text{def}}{=} [\![\varphi]\!]^I \text{ impl } [\![\psi]\!]^I \,,$$
$$[\![\varphi \Leftrightarrow \psi]\!]^I \stackrel{\text{def}}{=} [\![\varphi]\!]^I \text{ equiv } [\![\psi]\!]^I \,, \qquad [\![\varphi \uparrow \psi]\!] \stackrel{\text{def}}{=} [\![\varphi]\!]^I \text{ nand } [\![\psi]\!]^I \,.$$

Remarque 3.6. Comme mentionné dans la remarque 3.1, le « ou exclusif » xor correspond au sens souvent implicite du mot « ou » en français (A ou B mais pas les deux).

L'implication impl correspond approximativement à « si A alors B » ou « A implique B ». Mais en français usuel, ces locutions établissent souvent un lien de causalité implicite entre A et B, comme dans « S'il pleut, alors je prends mon parapluie. ». Cela rend aussi une phrase comme « S'il pleut et que je mets mon pull bleu, alors il pleut. » assez étrange, alors que $(P \land B) \Rightarrow P$ est une formule propositionnelle tout à fait raisonnable. L'implication en français courant peut aussi prendre un sens exclusif, comme dans « S'il reste ici, je m'en vais. », qu'on formaliserait en $R \oplus \neg V$ où R dénote « il reste ici » et V « je m'en vais ».

(Duparc, 2015, sec. 2.3)

3.1.7. *Complétude fonctionnelle.* Pour les applications de la logique propositionnelle, par exemple pour les expressions booléennes dans les langages de programmation ou de circuits logiques, il serait pour le moins souhaitable que toutes les fonctions booléennes soient exprimables comme la sémantique $[\![\varphi]\!]$ d'une formule propositionnelle φ. Une solution serait, pour chaque fonction booléenne $f\colon \mathbb{B}^n \to \mathbb{B}$, d'étendre la syntaxe abstraite des formules propositionnelles par

(Duparc, 2015, sec. 2.12)

$$\varphi ::= \cdots \mid f(\varphi, \ldots, \varphi)$$

et leur sémantique par

$$[\![f(\varphi_1, \ldots, \varphi_n)]\!]^I \stackrel{\text{def}}{=} f([\![\varphi_1]\!]^I, \ldots, [\![\varphi_n]\!]^I) \,.$$

Fort heureusement, il n'est pas nécessaire d'enrichir la syntaxe et la sémantique des formules propositionnelles par une infinité de cas – ce qui serait un obstacle pour leur implémentation ! En effet, toutes les fonctions booléennes peuvent être exprimées à l'aide des seules not, or et and (et des fonctions de projection) en les composant de manière appropriée.

Exemple 3.7. Les fonctions de la table 4 s'expriment comme suit :

$$x_1 \text{ xor } x_2 = (x_1 \text{ or } x_2) \text{ and not } (x_1 \text{ and } x_2) \,, \qquad x_1 \text{ impl } x_2 = \text{not } x_1 \text{ or } x_2 \,,$$
$$x_1 \text{ equiv } x_2 = (\text{not } x_1 \text{ or } x_2) \text{ and } (\text{not } x_2 \text{ or } x_1) \,, \qquad x_1 \text{ nand } x_2 = \text{not } (x_1 \text{ and } x_2) \,.$$

Intuitivement, les fonctions booléennes définissables comme des sémantiques $[\![\varphi]\!]$ de formules propositionnelles sont justement les fonctions exprimables à l'aide des seules not, or et and; par exemple, impl $= [\![\neg P_1 \vee P_2]\!]$. On obtient donc le résultat suivant.

Théorème 3.8 (complétude fonctionnelle)**.** *Pour tout $n > 0$ et toute fonction booléenne $f : \mathbb{B}^n \to \mathbb{B}$, il existe une formule propositionnelle φ sur n propositions $P_1, \ldots, P_n$ telle que $f = [\![\varphi]\!]$.*

Démonstration. Soit f une fonction booléenne $\mathbb{B}^n \to \mathbb{B}$ pour un certain $n > 0$. Si $n = 1$, alors il y a quatre fonctions booléennes de $\mathbb{B}$ dans $\mathbb{B}$, qui sont toutes exprimables à l'aide de formules propositionnelles :

- la fonction identité est $[\![P_1]\!]$,
- la fonction négation $[\![\neg P_1]\!]$,
- la fonction constante 1 est $[\![P_1 \vee \neg P_1]\!]$ et
- la fonction constante 0 est $[\![P_1 \wedge \neg P_1]\!]$.

Cette expression pour f est appelée sa forme normale disjonctive complète *; voir la section 5.3.*

Si $n > 1$, alors

$$f = [\![\bigvee_{(b_1,\ldots,b_n)\in\mathbb{B}^n : f(b_1,\ldots,b_n)=1} \varphi_{(b_1,\ldots,b_n)}]\!]$$

pourvu que chaque formule propositionnelle $\varphi_{(b_1,\ldots,b_n)}$ soit telle que $[\![\varphi_{(b_1,\ldots,b_n)}]\!]^I = 1$ si et seulement si pour tout $1 \leq i \leq n$, $P_i^I = b_i$. De telles formules propositionnelles $\varphi_{(b_1,\ldots,b_n)}$ s'écrivent comme des conjonctions

$$\varphi_{(b_1,\ldots,b_n)} \stackrel{\text{def}}{=} \bigwedge_{1\leq i\leq n} \ell_{(b_1,\ldots,b_n),i}$$

où les littéraux $\ell_{(b_1,\ldots,b_n),i}$ sont définis par

$$\ell_{(b_1,\ldots,b_n),i} \stackrel{\text{def}}{=} \begin{cases} P_i & \text{si } b_i = 1 \\ \neg P_i & \text{si } b_i = 0. \end{cases}$$ □

Exemple 3.9. Appliquons la preuve du théorème 3.8 de complétude fonctionnelle à la fonction equiv définie dans la table 4. Il y a deux valuations de (x_1, x_2) telles que x_1 equiv $x_2 = 1$, à savoir $(1, 1)$ et $(0, 0)$. Les deux formules propositionnelles associées sont $\varphi_{(1,1)} \stackrel{\text{def}}{=} P_1 \wedge P_2$ et $\varphi_{(0,0)} \stackrel{\text{def}}{=} \neg P_1 \wedge \neg P_2$, et on a bien equiv $= [\![(P_1 \wedge P_2) \vee (\neg P_1 \wedge \neg P_2)]\!]$.

(Duparc, 2015, sec. 2.9), (Harrisson, 2009, sec. 2.3)

3.2. **Satisfiabilité et validité.** Une formule propositionnelle φ est *satisfiable* s'il existe un modèle de φ, c'est-à-dire s'il existe une interprétation I telle que $I \vDash \varphi$. Elle est *valide*, noté $\vDash \varphi$, si pour toute interprétation I, $I \vDash \varphi$. Clairement, une formule propositionnelle valide est en particulier satisfiable, mais l'inverse n'est pas toujours vrai.

En termes des tables de vérité de la section 3.1.5, une formule propositionnelle est donc satisfiable si et seulement s'il existe au moins une entrée 1 dans sa colonne, et elle est valide si et seulement si sa colonne est entièrement constituée de 1. On voit dans la table 3 que la formule propositionnelle φ_{ex} de la figure 1 est satisfiable mais pas valide.

En termes d'ensemble de modèles, rappelons que $\text{Sat}(\varphi) \stackrel{\text{def}}{=} \{I \in \mathbb{B}^{\mathcal{P}_0} \mid I \vDash \varphi\}$ dénote l'ensemble des interprétations qui satisfont φ. Alors φ est satisfiable si $\text{Sat}(\varphi) \neq \emptyset$, tandis qu'elle est valide si $\text{Sat}(\varphi) = \mathbb{B}^{\mathcal{P}_0}$ est l'ensemble de toutes les interprétations possibles.

Exemple 3.10 (loi de Peirce)**.** La formule propositionnelle $\varphi \stackrel{\text{def}}{=} ((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ est une variante du tiers exclu, et est valide en logique classique propositionnelle. En effet, soit I une interprétation quelconque. Si $I \vDash P$, alors $I \vDash \varphi$. Sinon, $I \vDash P \Rightarrow Q$ donc $I \nvDash (P \Rightarrow Q) \Rightarrow P$ et donc $I \vDash \varphi$.

Propriété 3.11 (dualité entre satisfiabilité et validité)**.** *Une formule propositionnelle φ n'est pas satisfiable si et seulement si $\neg\varphi$ est valide ; elle n'est pas valide si et seulement si $\neg\varphi$ est satisfiable.*

Démonstration. Pour le premier énoncé, φ n'est pas satisfiable (aussi dite « contradictoire »)
- si et seulement si, pour toute interprétation I, $I \not\models \varphi$,
- si et seulement si, pour toute interprétation I, $I \models \neg\varphi$,
- si et seulement si, $\neg\varphi$ est valide.

Pour le second énoncé, φ n'est pas valide (aussi dite « falsifiable »)
- si et seulement si, il existe une interprétation I telle que $I \not\models \varphi$,
- si et seulement si, il existe une interprétation I telle que $I \models \neg\varphi$,
- si et seulement si, $\neg\varphi$ est satisfiable. □

Pour un ensemble de formules propositionnelles S et une interprétation I, on écrit $I \models S$ si $I \models \psi$ pour tout $\psi \in S$. Un ensemble S est *insatisfiable* s'il n'existe pas d'interprétation I telle que $I \models S$.

Exemple 3.12 (ensemble insatisfiable)**.** Soit l'ensemble F de formules propositionnelles suivant :

$$\{P \vee Q \vee \neg R,\ \ Q \vee R,\ \ \neg P \vee \neg Q \vee R,\ \ \neg P \vee \neg R,\ \ P \vee \neg Q\}\ .$$

Soit I une interprétation. Supposons tout d'abord que $I \models P$. Si $I \models R$, alors $I \not\models \neg P \vee \neg R$; sinon si $I \models Q$ alors $I \not\models \neg P \vee \neg Q \vee R$ et sinon $I \not\models Q \vee R$. Supposons maintenant $I \not\models P$. Si $I \models Q$, alors $I \not\models P \vee \neg Q$; sinon si $I \models R$ alors $I \not\models P \vee Q \vee \neg R$ et sinon $I \not\models Q \vee R$.

4. Conséquences et équivalences logiques

Résumé. Une formule propositionnelle φ est une *conséquence logique* d'une formule propositionnelle ψ (noté « $\psi \models \varphi$ ») si pour toute interprétation I, si $I \models \psi$ alors $I \models \varphi$. C'est le cas si et seulement si la formule propositionnelle $\psi \Rightarrow \varphi$ est valide, si et seulement si $[\![\psi]\!] \leq [\![\varphi]\!]$ (lemmes 4.2 et 4.3). Les formules propositionnelles φ et ψ sont *logiquement équivalentes* si $\psi \models \varphi$ et $\varphi \models \psi$; c'est le cas si et seulement si $\psi \Leftrightarrow \varphi$ est valide, si et seulement si $[\![\psi]\!] = [\![\varphi]\!]$.

Une *substitution propositionnelle* est une fonction τ de domaine fini qui associe à toute proposition $P \in \mathcal{P}_0$ une formule propositionnelle $\tau(P)$; par extension, $\varphi\tau$ est la formule propositionnelle dans laquelle toutes les occurrences de chaque proposition P ont été remplacées par $\tau(P)$.

On dénote par $I\tau$ l'interprétation qui associe pour toute proposition $P \in \mathcal{P}_0$ la valeur de vérité $P^{I\tau} \stackrel{\text{def}}{=} [\![\tau(P)]\!]^I$; alors le lemme 4.7 de substitution propositionnelle dit que $[\![\varphi\tau]\!]^I = [\![\varphi]\!]^{I\tau}$. Cela implique en particulier que si φ est valide, alors $\varphi\tau$ l'est aussi, et permet de démontrer de nombreuses *équivalences usuelles*.

Comme nous l'avons vu dans l'exemple 3.5, il peut y avoir plusieurs formules propositionnelles avec la même sémantique. Le but de cette section est de mieux comprendre ce phénomène.

(Duparc, 2015, sec. 2.5)

4.1. **Conséquences logiques.** Si S est un ensemble de formules propositionnelles et φ est une formule propositionnelle, on dit que φ est une *conséquence logique* de S et on écrit $S \models \varphi$ si pour toute interprétation I telle que $I \models S$ on a $I \models \varphi$ – autrement dit, si pour toute interprétation I, $I \models S$ implique $I \models \varphi$.

Propriété 4.1. *Soit φ une formule propositionnelle. Alors φ est valide si et seulement si $\emptyset \models \varphi$, c'est-à-dire φ est une conséquence logique de l'ensemble vide.*

Démonstration. On a $\emptyset \models \varphi$
- si et seulement si, pour toute interprétation I, si I satisfait toutes les formules de l'ensemble vide, alors $I \models \varphi$,
- si et seulement si, pour toute interprétation I, on a $I \models \varphi$,
- si et seulement si, φ est valide. □

Dans le cas d'un ensemble $S = \{\psi\}$ constitué d'une seule formule propositionnelle ψ, on notera plus simplement $\psi \models \varphi$ et on dira que φ est une *conséquence logique* de ψ. Si on écrit $\text{Sat}(\varphi) \stackrel{\text{def}}{=} \{I \in \mathbb{B}^{\mathcal{P}_0} \mid I \models \varphi\}$ pour l'ensemble des interprétations qui satisfont une formule propositionnelle φ, $\psi \models \varphi$ revient à $\text{Sat}(\psi) \subseteq \text{Sat}(\varphi)$. On peut relier cette notion à la validité d'une seule formule propositionnelle comme suit.

Lemme 4.2 (déduction). *Soit S un ensemble de formules propositionnelles, et φ et ψ deux formules propositionnelles. Alors $S \cup \{\psi\} \models \varphi$ si et seulement si $S \models \psi \Rightarrow \varphi$. En particulier quand $S = \emptyset$, $\psi \models \varphi$ si et seulement si $\psi \Rightarrow \varphi$ est valide.*

Démonstration. On a $S \cup \{\psi\} \models \varphi$
- si et seulement si, pour toute interprétation I, si I satisfait toutes les formules propositionnelles de $S \cup \{\psi\}$, alors $I \models \varphi$,
- si et seulement si, pour toute interprétation I, si I satisfait toutes les formules propositionnelles de S, et si de plus $I \models \psi$, alors $I \models \varphi$,
- si et seulement si, pour toute interprétation I, si I satisfait toutes les formules propositionnelles de S, alors $I \models \psi \Rightarrow \varphi$,
- si et seulement si, $S \models \psi \Rightarrow \varphi$.

Par suite, le cas où $S = \emptyset$ découle de la propriété 4.1. □

Une autre façon de comprendre les conséquences logiques est de définir un *pré-ordre* à l'aide des sémantiques fonctionnelles $[\![\varphi]\!]$. On considère pour cela l'ordre $0 < 1$ sur les valeurs de vérités, et on dit que ψ est *fonctionnellement plus petite* que φ, noté $[\![\psi]\!] \leq [\![\varphi]\!]$, si pour toute interprétation I, $[\![\psi]\!]^I \leq [\![\varphi]\!]^I$. D'après cette définition, deux formules propositionnelles φ et ψ sont *fonctionnellement équivalentes*, c'est-à-dire telles que $[\![\varphi]\!] = [\![\psi]\!]$, si et seulement si $[\![\varphi]\!] \leq [\![\psi]\!]$ et $[\![\psi]\!] \leq [\![\varphi]\!]$.

On peut visualiser le pré-ordre fonctionnel sous la forme d'un treillis comme celui de la figure 2. Dans cette figure, on donne pour chacun des 16 cas possibles sur deux propositions P et Q les valeurs de vérité des formules propositionnelles pour les interprétations partielles $[1/P, 1/Q]$, $[1/P, 0/Q]$, $[0/P, 1/Q]$ et $[0/P, 0/Q]$, dans cet ordre. Chaque élément du treillis est illustré par un exemple de formule propositionnelle avec cette sémantique ; il y en a bien sûr d'autres, comme $P \wedge \neg Q$ pour 0100 ou $\neg P \vee \neg Q$ pour 0111, ou comme vu dans l'exemple 3.5, $(P \vee \neg Q) \wedge P$ pour 1100. Dans ce treillis, $[\![\psi]\!] \leq [\![\varphi]\!]$ s'il existe un chemin pointillé qui monte de $[\![\psi]\!]$ à $[\![\varphi]\!]$.

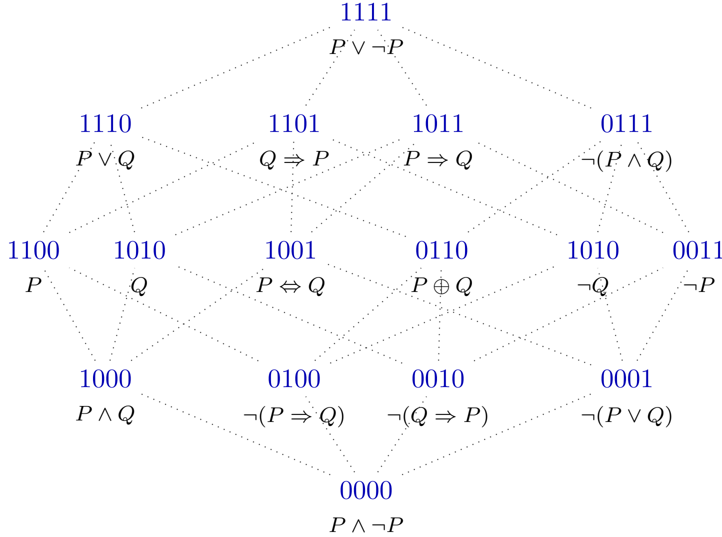


FIGURE 2. Le treillis du pré-ordre fonctionnel sur deux propositions P et Q.

Lemme 4.3 (pré-ordre fonctionnel)**.** *Soient φ et ψ deux formules propositionnelles. Alors $\psi \vDash \varphi$ si et seulement si $[\![\psi]\!] \leq [\![\varphi]\!]$.*

Démonstration. On a $\psi \vDash \varphi$

- si et seulement si, pour toute interprétation I, si $I \vDash \psi$ alors $I \vDash \varphi$,
- si et seulement si, pour toute interprétation I, si $[\![\psi]\!]^I = 1$ alors $[\![\varphi]\!]^I = 1$,
- si et seulement si, pour toute interprétation I, $[\![\psi]\!]^I \leq [\![\varphi]\!]^I$,
- si et seulement si, $[\![\psi]\!] \leq [\![\varphi]\!]$. □

4.2. **Équivalences logiques.** On dit que deux formules propositionnelles φ et ψ sont *logiquement équivalentes* si $\psi \vDash \varphi$ et $\varphi \vDash \psi$, c'est-à-dire si, pour toute interprétation I, $I \vDash \psi$ si et seulement si $I \vDash \varphi$. En terme d'ensemble de modèles, cela revient à demander $\text{Sat}(\psi) = \text{Sat}(\varphi)$. Par le lemme 4.2 de déduction, cela se produit si et seulement si $\psi \Leftrightarrow \varphi$ est une formule propositionnelle valide. Par le lemme 4.3 de pré-ordre fonctionnel, cela se produit si et seulement si φ et ψ sont fonctionnellement équivalentes, c'est-à-dire si et seulement si $[\![\varphi]\!] = [\![\psi]\!]$.

(Duparc, 2015, sec. 2.4)

Comme nous allons le voir, l'équivalence logique permet de remplacer une formule propositionnelle, qui représente par exemple une expression booléenne d'un programme ou un circuit logique, par une autre formule propositionnelle équivalente potentiellement plus efficace à évaluer ; ainsi, dans l'exemple 3.5, la formule propositionnelle P est plus facile à évaluer que $(P \vee \neg Q) \wedge P$.

Il est donc très utile de savoir dire si deux formules propositionnelles φ et ψ sont logiquement équivalentes ou non. Cela peut se faire à l'aide de tables de vérité

— d'après le lemme 4.2 de déduction, en vérifiant si $\varphi \Leftrightarrow \psi$ est valide, ou
— d'après le lemme 4.3 de pré-ordre fonctionnel, en vérifiant si $[\![\varphi]\!] = [\![\psi]\!]$.

Exemple 4.4. Appliquons les lemmes 4.2 et 4.3 à l'équivalence entre $\neg(P \vee Q)$ et $\neg P \wedge \neg Q$. La table de vérité correspondante est donnée dans la table 5 ci-dessous.
On vérifie dans cette table que la troisième colonne ($\neg(P \vee Q)$) et la cinquième colonne ($\neg P \wedge \neg Q$) sont identiques, donc ces deux formules propositionnelles sont logiquement équivalentes par le lemme 4.3 de pré-ordre fonctionnel. La sixième colonne ($\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$) ne contient que des 1, donc cette formule propositionnelle est valide, ce qui montre aussi que les deux formules propositionnelles sont logiquement équivalentes par le lemme 4.2 de déduction.

TABLE 5. Table de vérité de $\neg(P \vee Q)$, $\neg P \wedge \neg Q$ et $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$.

P	Q	$P \vee Q$	$\neg(P \vee Q)$	$\neg P$	$\neg Q$	$\neg P \wedge \neg Q$	$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
1	1	1	0	0	0	0	1
1	0	1	0	0	1	0	1
0	1	1	0	1	0	0	1
0	0	0	1	1	1	1	1

Cette approche via les tables de vérité a l'inconvénient de nécessiter de tester toutes les interprétations des propositions de $\text{fp}(\varphi) \cup \text{fp}(\psi)$, soit un nombre exponentiel de possibilités. Nous verrons des techniques qui permettent souvent de n'explorer qu'une toute petite partie de cet espace de recherche. En attendant, nous allons voir une approche qui s'applique bien à de petites formules propositionnelles et des raisonnements « à la main ».

(Duparc, 2015, sec. 2.6), (Goubault-Larrecq et Mackie, 1997, def. 2.4), (David, Nour et Raffalli, 2003, def. 4.6.4), (Harrisson, 2009, p. 41)

4.3. **Substitutions propositionnelles.** Une *substitution propositionnelle* (aussi appelée une « transduction propositionnelle ») est une fonction τ qui associe à chaque proposition $P \in \mathcal{P}_0$ une formule propositionnelle $\tau(P)$, de tel sorte que son *domaine* $\text{dom}(\tau) \stackrel{\text{def}}{=} \{P \in \mathcal{P}_0 \mid \tau(P) \neq P\}$ soit fini. On écrit $[\varphi_1/P_1, \ldots, \varphi_n/P_n]$ pour la substitution propositionnelle de domaine $\{P_1, \ldots, P_n\}$ où les P_i sont distinctes et qui associe φ_i à P_i. Toute substitution propositionnelle se relève en une fonction des formules propositionnelles dans les formules propositionnelles :

$$P\tau \stackrel{\text{def}}{=} \tau(P)\ , \quad (\neg\varphi)\tau \stackrel{\text{def}}{=} \neg(\varphi\tau)\ , \quad (\varphi \vee \psi)\tau \stackrel{\text{def}}{=} (\varphi\tau) \vee (\psi\tau)\ , \quad (\varphi \wedge \psi)\tau \stackrel{\text{def}}{=} (\varphi\tau) \wedge (\psi\tau)\ .$$

Exemple 4.5. Considérons la formule propositionnelle $\psi = (P \wedge Q)$, ainsi que la substitution propositionnelle $\tau = [(P \vee \neg Q)/P, P/Q]$. Alors $\psi\tau = (P \vee \neg Q) \wedge P$.

4.3.1. *Implémentation des substitutions propositionnelles en Java.* En Java, une substitution propositionnelle peut être représentée comme un objet de type `Map<String,Formule>`. Par exemple, la substitution propositionnelle τ de l'exemple 4.5 se code par

```
Map<String,Formule> tau = new HashMap<String,Formule>();
tau.put("P", new Proposition("Q"));
```

```
tau.put("Q", new Ou(new Proposition("P"),
                    new Non(new Proposition("Q"))));
```

Pour implémenter l'application d'une substitution propositionnelle, on déclare tout d'abord que `Formule` est `Cloneable`. L'application d'une substitution propositionnelle se fera en appelant la méthode `substitue()` suivante :

```
Formule
public abstract Formule substitue(Map<String,Formule> tau);
```

Son implémentation dans les sous-classes est très simple :

```
Formule.Et
public Formule substitue(Map<String,Formule> tau) {
    return new Et(phi1.substitue(tau), phi2.substitue(tau));
}
Formule.Ou
public Formule substitue(Map<String,Formule> tau) {
    return new Ou(phi1.substitue(tau), phi2.substitue(tau));
}
Formule.Non
public Formule substitue(Map<String,Formule> tau) {
    return new Non(phi1.substitue(tau));
}
```

```
Formule.Proposition
 public Formule
substitue(Map<String,Formule> tau) { return (tau.containsKey(nom))?
  tau.get(nom).clone(): new Proposition(nom); }
```

Si on ne faisait pas appel à `clone()` ici, on obtiendrait une formule propositionnelle qui n'est plus un arbre mais un graphe dirigé acyclique enraciné. Ça ne serait pas nécessairement un problème : pourquoi ?

4.3.2. *Implémentation des substitutions propositionnelles en OCaml.* Une substitution propositionnelle se représente en OCaml comme une fonction de type `string -> formule`. Par exemple, la substitution propositionnelle τ de l'exemple 4.5 se code par

```
let tau = function
    "P" -> Proposition "Q"
  | "Q" -> Ou (Proposition "P", Non (Proposition "Q"))
  | p   -> Proposition p
```

Voici un code qui implémente l'application d'une substitution propositionnelle :

```
let rec substitue phi tau =
  match phi with
    Et(phi1, phi2) -> Et(substitue phi1 tau, substitue phi2 tau)
  | Ou(phi1, phi2) -> Ou(substitue phi1 tau, substitue phi2 tau)
  | Non phi1       -> Non(substitue phi1 tau)
  | Proposition p  -> tau p
```

4.3.3. *Lemme de substitution propositionnelle.* Pour une interprétation I et une substitution propositionnelle τ, on définit l'interprétation $I\tau$ comme associant $P^{I\tau} \stackrel{\text{def}}{=} [\![\tau(P)]\!]^I$ à chaque proposition P de $\mathcal{P}_0$.

Exemple 4.6. Soit I une interprétation qui étend $[0/P, 0/Q]$ et $\tau = [(P \vee \neg Q)/P, P/Q]$. Alors $P^{I\tau} = [\![P \vee \neg Q]\!]^I = 1$ et $Q^{I\tau} = [\![P]\!]^I = 0$.

Lemme 4.7 (substitution propositionnelle)**.** *Pour toute formule propositionnelle φ, toute substitution propositionnelle τ et toute interprétation I, $[\![\varphi\tau]\!]^I = [\![\varphi]\!]^{I\tau}$.*

(Goubault-Larrecq et Mackie, 1997, thm. 2.10), (Harrisson, 2009, thm. 2.3)

Démonstration. Par induction structurelle sur φ. Pour le cas de base où $\varphi = P \in \mathcal{P}_0$,

$$[\![\varphi\tau]\!]^I = [\![P\tau]\!]^I = [\![\tau(P)]\!]^I = [\![P]\!]^{I\tau} = [\![\varphi]\!]^{I\tau} \;.$$

Pour l'étape d'induction où $\varphi = \neg\psi$,

$$[\![\varphi\tau]\!]^I = [\![(\neg\psi)\tau]\!]^I = [\![\neg(\psi\tau)]\!]^I = \text{not}\,[\![\psi\tau]\!]^I \overset{\text{h.i.}}{=} \text{not}\,[\![\psi]\!]^{I\tau} = [\![\neg\psi]\!]^{I\tau} = [\![\varphi]\!]^{I\tau} \;.$$

Pour l'étape d'induction où $\varphi = \varphi' \vee \psi$,

$$[\![\varphi\tau]\!]^I = [\![(\varphi'\vee\psi)\tau]\!]^I = [\![(\varphi'\tau)\vee(\psi\tau)]\!]^I = [\![\varphi'\tau]\!]^I \,\text{or}\, [\![\psi\tau]\!]^I \overset{\text{h.i.}}{=} [\![\varphi']\!]^{I\tau} \,\text{or}\, [\![\psi]\!]^{I\tau} = [\![\varphi'\vee\psi]\!]^{I\tau} = [\![\varphi]\!]^{I\tau}.$$

L'étape d'induction où $\varphi = \varphi' \wedge \psi$ est similaire. □

Exemple 4.8. Considérons comme dans les exemples 4.5 et 4.6 une interprétation I qui étend $[0/P, 0/Q]$, la substitution propositionnelle $\tau = [(P \vee \neg Q)/P, P/Q]$, et la formule propositionnelle $\psi = (P \wedge Q)$. Alors d'un côté $[\![\psi\tau]\!]^I = [\![(P \vee \neg Q) \wedge P]\!]^{[0/P,0/Q]} = 0$, et de l'autre $[\![\psi]\!]^{I\tau} = [\![P \wedge Q]\!]^{[1/P,0/Q]} = 0$.

(Harrisson, 2009, cor. 2.4)

Corollaire 4.9. *Soit φ une formule propositionnelle valide et τ une substitution propositionnelle. Alors $\varphi\tau$ est valide.*

Démonstration. Par le lemme 4.7 de substitution propositionnelle, pour toute interprétation I, $[\![\varphi\tau]\!]^I = [\![\varphi]\!]^{I\tau}$. Or, comme φ est valide, $[\![\varphi]\!]^{I\tau} = 1$. □

Exemple 4.10. Voyons tout de suite une application de la corollaire 4.9. Nous avons vu dans l'exemple 4.4 que $\neg(P \vee Q) \Leftrightarrow (\neg P \wedge \neg Q)$ est valide. Dès lors, pour toutes formules propositionnelles φ et ψ, on peut appliquer la substitution propositionnelle $[\varphi/P, \psi/Q]$ et déduire que $\neg(\varphi \vee \psi) \Leftrightarrow (\neg\varphi \wedge \neg\psi)$ est valide – autrement dit, que $\neg(\varphi \vee \psi)$ et $\neg\varphi \wedge \neg\psi$ sont logiquement équivalentes.

(Harrisson, 2009, pp. 44–46)

4.4. Équivalences usuelles.

En appliquant le même raisonnement que dans l'exemple 4.10, on a plus généralement les équivalences logiques suivantes.

Résumé. Pour toutes formules propositionnelles φ, ψ, et ψ',

$$(\varphi \vee \varphi) \Leftrightarrow \varphi \;, \qquad \text{(idempotence de } \vee)$$
$$(\varphi \vee \psi) \Leftrightarrow (\psi \vee \varphi) \;, \qquad \text{(commutativité de } \vee)$$
$$((\varphi \vee \psi) \vee \psi') \Leftrightarrow (\varphi \vee (\psi \vee \psi')) \qquad \text{(associativité de } \vee)$$
$$(\varphi \wedge \varphi) \Leftrightarrow \varphi \;, \qquad \text{(idempotence de } \wedge)$$
$$(\varphi \wedge \psi) \Leftrightarrow (\psi \wedge \varphi) \;, \qquad \text{(commutativité de } \wedge)$$
$$((\varphi \wedge \psi) \wedge \psi') \Leftrightarrow (\varphi \wedge (\psi \wedge \psi')) \qquad \text{(associativité de } \wedge)$$
$$\neg\neg\varphi \Leftrightarrow \varphi \;, \qquad \text{(double négation)}$$
$$(\varphi \wedge (\psi \vee \psi')) \Leftrightarrow ((\varphi \wedge \psi) \vee (\varphi \wedge \psi')) \;, \qquad \text{(distributivité de } \wedge \text{ sur } \vee)$$
$$(\varphi \vee (\psi \wedge \psi')) \Leftrightarrow ((\varphi \vee \psi) \wedge (\varphi \vee \psi')) \;, \qquad \text{(distributivité de } \vee \text{ sur } \wedge)$$
$$\neg(\varphi \vee \psi) \Leftrightarrow (\neg\varphi \wedge \neg\psi) \;, \qquad \text{(dualité de DE MORGAN pour } \vee)$$
$$\neg(\varphi \wedge \psi) \Leftrightarrow (\neg\varphi \vee \neg\psi) \;, \qquad \text{(dualité de DE MORGAN pour } \wedge)$$
$$(\varphi \Rightarrow \psi) \Leftrightarrow (\neg\varphi \vee \psi) \;, \qquad \text{(définition de l'implication)}$$
$$(\varphi \Rightarrow \psi) \Leftrightarrow (\neg\psi \Rightarrow \neg\varphi) \;, \qquad \text{(contraposition)}$$
$$((\varphi \wedge \psi) \Rightarrow \psi') \Leftrightarrow (\varphi \Rightarrow (\psi \Rightarrow \psi')) \;. \qquad \text{(curryfication)}$$

Ces équivalences logiques sont aussi très utiles pour simplifier des formules propositionnelles. On repose pour cela sur la propriété suivante.

Corollaire 4.11. *Soit φ une formule propositionnelle, et τ et τ' deux substitutions propositionnelles telles que, pour toute proposition $P \in \mathcal{P}_0$, $\tau(P)$ soit logiquement équivalente à $\tau'(P)$. Alors $\varphi\tau$ est logiquement équivalente à $\varphi\tau'$.*

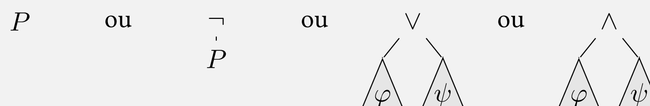
Démonstration. Pour montrer que $\varphi\tau$ et $\varphi\tau'$ sont logiquement équivalentes, on va montrer que pour toute interprétation $I \in \mathbb{B}^{\mathcal{P}_0}$, $[\![\varphi\tau]\!]^I = [\![\varphi\tau']\!]^I$.

Par hypothèse, pour toute proposition $P \in \mathcal{P}_0$, $\tau(P)$ et $\tau'(P)$ sont logiquement équivalentes. Cela signifie que pour toute interprétation I et toute proposition P, par définition de $I\tau$ et $I\tau'$, $P^{I\tau} = [\![\tau(P)]\!]^I = [\![\tau'(P)]\!]^I = P^{I\tau'}$, c'est-à-dire que pour toute interprétation I, $I\tau = I\tau'$ sont une seule et même interprétation. On en déduit par le lemme 4.7 de substitution propositionnelle que $[\![\varphi\tau]\!]^I = [\![\varphi]\!]^{I\tau} = [\![\varphi\tau']\!]^I = [\![\varphi]\!]^{I\tau'}$. □

Exemple 4.12. Voici une illustration : on souhaite montrer que $(P \Rightarrow Q) \Rightarrow P$ est logiquement équivalente à P. Les deux substitutions propositionnelles $[(P \Rightarrow Q)/P, P/Q]$ et $[(\neg P \vee Q)/P, P/Q]$ sont bien telles que $(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$ (par définition de l'implication) et $P \Leftrightarrow P$ est immédiat. En appliquant la corollaire 4.11 à la formule propositionnelle $\varphi = (P \Rightarrow Q)$, on en déduit que $(P \Rightarrow Q) \Rightarrow P$ est logiquement équivalente à $(\neg P \vee Q) \Rightarrow P$. Puis, par des raisonnements similaires, par définition de l'implication, elle est logiquement équivalente à $\neg(\neg P \vee Q) \vee P$, puis par dualité de de Morgan pour $\vee$, à $(\neg\neg P \wedge \neg Q) \vee P$, par double négation, à $(P \wedge \neg Q) \vee P$, par commutativité de $\vee$, à $P \vee (P \wedge \neg Q)$, par distributivité de $\vee$ sur $\wedge$, à $(P \vee P) \wedge (P \vee \neg Q)$, par idempotence de $\vee$, à $P \wedge (P \vee \neg Q)$, par commutativité de $\wedge$, à $(P \vee \neg Q) \wedge P$, qui comme nous l'avons vu dans l'exemple 3.5 est logiquement équivalente à P.

5. Formes normales

Résumé. On peut mettre n'importe quelle formule propositionnelle sous *forme normale négative* en « poussant » les négations vers les feuilles par application de la double négation et des dualités de de Morgan ; la formule propositionnelle obtenue est logiquement équivalente et de la forme



où $P \in \mathcal{P}_0$ et φ et ψ sont des formules propositionnelles sous forme normale négative. Les formules propositionnelles de la forme « P » ou « $\neg P$ » sont appelées des *littéraux*. On note « $\bar{\varphi}$ » pour la forme normale négative de $\neg\varphi$.

Une formule propositionnelle est sous *forme normale conjonctive* si elle s'écrit comme

$$\bigwedge_{1\leq i\leq m} \bigvee_{1\leq j\leq n_i} \ell_{i,j}$$

où les $\ell_{i,j}$ sont des littéraux. On peut mettre une formule propositionnelle sous forme normale conjonctive à partir d'une formule propositionnelle sous forme normale négative en « poussant » les disjonctions vers le bas par application de la distributivité de $\vee$ sur $\wedge$. Une formule propositionnelle est sous *forme normale disjonctive* si elle s'écrit comme

$$\bigvee_{1\leq i\leq m} \bigwedge_{1\leq j\leq n_i} \ell_{i,j}$$

où les $\ell_{i,j}$ sont des littéraux. On peut mettre une formule propositionnelle sous forme normale disjonctive à partir d'une formule propositionnelle sous forme normale négative en « poussant » les conjonctions vers le bas par application de la distributivité de $\wedge$ sur $\vee$. Les formules propositionnelles sous forme normale conjonctive ou disjonctive obtenues ainsi sont logiquement équivalentes à la formule d'origine mais potentiellement de taille exponentielle (exemple 5.5).

On préfère en pratique construire des formules propositionnelles sous forme normale conjonctive *équi-satisfiables* avec la formule d'origine (section 5.2.2) ; cette opération a un coût linéaire dans le pire des cas.

Il existe un format de fichier standard pour écrire des formules propositionnelles sous forme normale conjonctive : le *format DIMACS*.

Nous avons vu dans la section précédente qu'il existe de nombreuses manières d'écrire des formules propositionnelles logiquement équivalentes (c.f. section 4). Parmi toutes ces formules propositionnelles équivalentes, certaines seront plus faciles à traiter ; par exemple, la formule propositionnelle P est plus facile à évaluer que la formule propositionnelle $(P \vee \neg Q) \wedge P$. En particulier, les algorithmes de recherche de modèle ou de recherche de preuve que nous verrons par la suite travaillent sur des formules propositionnelles avec une syntaxe restreinte – on parle alors de *forme normale*.

(David, Nour et Raffalli, 2003, sec. 2.6), (Goubault-Larrecq et Mackie, 1997, def. 2.38), (Harrisson, 2009, sec. 2.5)

5.1. Forme normale négative.

Une formule propositionnelle est sous *forme normale négative* si elle respecte la syntaxe abstraite

$$\ell ::= P \mid \neg P \qquad \text{(littéraux)}$$

$$\varphi ::= \ell \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \qquad \text{(formules propositionnelles sous forme normale négative)}$$

où P est une proposition de $\mathcal{P}_0$. En d'autres termes, les négations ne peuvent apparaître que devant des formules atomiques. Par exemple, la formule propositionnelle $\varphi_{\text{ex}} = (P \vee \neg Q) \wedge P$ de la figure 1 est sous forme normale négative.

La mise sous forme normale négative procède en « poussant » les négations dans l'arbre de syntaxe abstraite de la formule propositionnelle vers les feuilles.

Définition 5.1 (forme normale négative)**.** Pour une formule propositionnelle φ, on notera $\text{nnf}(\varphi)$ sa forme normale négative obtenue inductivement par

$$\begin{aligned} \text{nnf}(P) &\stackrel{\text{def}}{=} P\,, & \text{nnf}(\neg P) &\stackrel{\text{def}}{=} \neg P\,,\\ \text{nnf}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \text{nnf}(\varphi) \vee \text{nnf}(\psi)\,, & \text{nnf}(\neg(\varphi \vee \psi)) &\stackrel{\text{def}}{=} \text{nnf}(\neg\varphi) \wedge \text{nnf}(\neg\psi)\,,\\ \text{nnf}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \text{nnf}(\varphi) \wedge \text{nnf}(\psi)\,, & \text{nnf}(\neg(\varphi \wedge \psi)) &\stackrel{\text{def}}{=} \text{nnf}(\neg\varphi) \vee \text{nnf}(\neg\psi)\,,\\ & & \text{nnf}(\neg\neg\varphi) &\stackrel{\text{def}}{=} \text{nnf}(\varphi)\,. \end{aligned}$$

On notera aussi en général

$$\overline{\varphi} \stackrel{\text{def}}{=} \text{nnf}(\neg\varphi)$$

pour la forme normale négative de la négation de φ, appelée la *formule duale* de φ.

À noter que les définitions dans la colonne de gauche de la définition 5.1 sont celles pour des formules propositionnelles qui n'ont pas de symbole « $\neg$ » à leur racine, tandis que celles de la colonne de droite s'occupent des différents cas de formules propositionnelles enracinées par « $\neg$ ». En termes algorithmiques, cette mise sous forme normale négative se fait en temps linéaire. Par les deux lois de dualité de de Morgan pour $\vee$ et pour $\wedge$ et par la loi de double négation, la mise sous forme normale négative préserve la sémantique des formules propositionnelles : φ et $\text{nnf}(\varphi)$ sont équivalentes.

Exemple 5.2. La loi de Peirce $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ s'écrit $\neg(\neg(\neg P \vee Q) \vee P) \vee P$ en syntaxe non étendue. Sa forme normale négative est

$$\begin{aligned} \text{nnf}(\neg(\neg(\neg P \vee Q) \vee P) \vee P) &= \text{nnf}(\neg(\neg(\neg P \vee Q) \vee P)) \vee P\\ &= (\text{nnf}(\neg\neg(\neg P \vee Q)) \wedge \neg P) \vee P\\ &= ((\neg P \vee Q) \wedge \neg P) \vee P\,. \end{aligned}$$

Sa formule duale est

$$\begin{aligned} \overline{\neg(\neg(\neg P \vee Q) \vee P)} &= \text{nnf}(\neg(\neg(\neg(\neg P \vee Q) \vee P) \vee P))\\ &= \text{nnf}(\neg\neg(\neg(\neg P \vee Q) \vee P)) \wedge \neg P\\ &= \text{nnf}(\neg(\neg P \vee Q) \vee P) \wedge \neg P\\ &= (\text{nnf}(\neg(\neg P \vee Q)) \vee P) \wedge \neg P\\ &= ((\text{nnf}(\neg\neg P) \wedge \neg Q) \vee P) \wedge \neg P\\ &= ((P \wedge \neg Q) \vee P) \wedge \neg P\,. \end{aligned}$$

5.1.1. *Implémentation des formes normales négatives en Java.* Comme d'habitude, nous déclarons des méthodes abstraites dans `Formule` :

L'implémentation des formes normales négatives en OCaml sera faite en TP du cours de « programmation fonctionnelle ».

Formule
```
public abstract Formule getNNF();
public abstract Formule getDualNNF();
```

La première méthode implémente la colonne de gauche de la définition 5.1, tandis que la seconde implémente la colonne de droite. Plus précisément, voici leurs implémentations dans les sous-classes de `Formule` :

```
Formule.Et
public Formule getNNF() {
    return new Et(phi1.getNNF(), phi2.getNNF());
}
public Formule getDualNNF() {
    return new Ou(phi1.getDualNNF(), phi2.getDualNNF());
}
Formule.Ou
public Formule getNNF() {
    return new Ou(phi1.getNNF(), phi2.getNNF());
}
public Formule getDualNNF() {
    return new Et(phi1.getDualNNF(), phi2.getDualNNF());
}
Formule.Non
public Formule getNNF() {
    return phi1.getDualNNF();
}
public Formule getDualNNF() {
    return phi1.getNNF();
}
Formule.Proposition
public Formule getNNF() {
    return new Proposition(nom);
}
public Formule getDualNNF() {
    return new Non (new Proposition(nom));
}
```

5.2. **Forme clausale.** Une formule propositionnelle en forme normale négative n'a que des opérateurs $\vee$ et $\wedge$ en guise de nœuds internes, sauf potentiellement des $\neg$ juste au-dessus des propositions. En utilisant les lois de distributivité, on peut encore normaliser ces formules propositionnelles pour imposer que tous les $\wedge$ soient au-dessus des $\vee$ (forme normale conjonctive) ou vice-versa (forme normale disjonctive). La forme clausale est ensuite simplement une écriture « ensembliste » d'une formule propositionnelle sous forme normale conjonctive, et est largement employée dans les algorithmes de recherche de modèle et de recherche de preuve.

Nous allons voir deux techniques pour mettre une formule propositionnelle sous forme normale conjonctive. La première est très simple et s'appuie sur la loi de distributivité de $\vee$ sur $\wedge$ et construit une formule propositionnelle logiquement équivalente. Cependant, elle peut avoir un coût prohibitif en pratique, et nous verrons ensuite une technique qui construit une formule propositionnelle *équi-satisfiable*, au sens suivant :

Définition 5.3 (équi-satisfiable)**.** Soit φ et ψ deux formules propositionnelles. Elles sont *équi-satisfiables* si φ est satisfiable si et seulement si ψ est satisfiable – autrement dit, $\exists I \,.\, I \vDash \varphi$ si et seulement si $\exists I' \,.\, I' \vDash \psi$.

(Duparc, 2015, sec. 2.10.2), (David, Nour et Raffalli, 2003, sec. 7.4.2)

5.2.1. *Forme clausale logiquement équivalente.* Soit φ une formule propositionnelle en forme normale négative. En utilisant de manière répétée la loi de distributivité de $\vee$ sur $\wedge$, on « pousse » les disjonctions vers le bas et on obtient une mise sous *forme normale conjonctive* $\mathrm{cnf}(\varphi)$ pour toute φ en forme normale négative. Le cas le plus important de cette transformation est le suivant :

$$\mathrm{cnf}(\varphi \vee (\psi \wedge \psi')) \stackrel{\text{def}}{=} \mathrm{cnf}((\psi \wedge \psi') \vee \varphi) \stackrel{\text{def}}{=} \mathrm{cnf}(\varphi \vee \psi) \wedge \mathrm{cnf}(\varphi \vee \psi') \ .$$

À noter que la formule φ est *dupliquée* lors de cette transformation. Cela explique que la mise sous forme normale conjonctive peut avoir un coût exponentiel (voir l'exemple 5.5 ci-dessous).

Une formule propositionnelle sous forme normale conjonctive s'écrit donc sous la forme

$$\bigwedge_{1\leq i\leq m} \bigvee_{1\leq j\leq n_i} \ell_{i,j}$$

où les $\ell_{i,j}$ sont des littéraux. Les sous-formules $C_i \stackrel{\text{def}}{=} \bigvee_{1\leq j\leq n_i} \ell_{i,j}$ sont appelées les *clauses* de la formule.

k-CNF. Quand les clauses sont des disjonctions d'au plus k littéraux, on dit que la formule est sous forme « k-CNF ». Par exemple, la formule propositionnelle $\varphi_{\text{ex}} = (P \vee \neg Q) \wedge P$ de la figure 1 est déjà sous forme normale conjonctive : cnf$(\varphi_{\text{ex}}) = \varphi_{\text{ex}}$. Elle est composée de deux clauses : $P \vee \neg Q$ et P ; comme ces deux clauses contiennent chacune au plus deux littéraux, c'est une formule en 2-CNF.

Forme clausale, forme k-clausale. Par les lois d'idempotence de $\vee$, de commutativité de $\vee$ et d'associativité de $\vee$, chaque clause peut-être vue comme un ensemble de littéraux, pour lequel les notations ensemblistes $\in$ et $\subseteq$ s'appliquent ; par exemple, la clause $P \vee Q \vee \neg R$ peut être vue comme l'ensemble de littéraux $\{P, Q, \neg R\}$. De même, par les lois d'idempotence de $\wedge$, de commutativité de $\wedge$ et d'associativité de $\wedge$, on peut voir une formule propositionnelle en forme normale conjonctive comme un ensemble de clauses.

Pour une formule propositionnelle φ donnée, on note Cl(φ) l'ensemble des clauses de cnf(φ) (où chaque clause est vue comme un ensemble) ; on appelle cela sa *forme clausale.* Pour la formule propositionnelle $\varphi_{\text{ex}} = (P \vee \neg Q) \wedge P$ de la figure 1, Cl$(\varphi_{\text{ex}}) = \{\{P, \neg Q\}, \{P\}\}$. L'ensemble de formules de l'exemple 3.12 peut aussi être vu comme une forme clausale

$$\{\{P, Q, \neg R\},\ \{Q, R\},\ \{\neg P, \neg Q, R\},\ \{\neg P, \neg R\},\ \{P, \neg Q\}\}\ .$$

Quand les clauses contiennent au plus k littéraux, on parle aussi de forme *k-clausale.*

Exemple 5.4. Comme vu dans l'exemple 5.2, la formule duale de la loi de Peirce $((P \Rightarrow Q) \Rightarrow P) \wedge \neg P$ s'écrit $((P \wedge \neg Q) \vee P) \wedge \neg P$. La mise sous forme conjonctive produit $(P \vee P) \wedge (\neg Q \vee P) \wedge \neg P$ et donc la forme clausale $\{\{P, P\}, \{\neg Q, P\}, \{\neg P\}\}$.

Exemple 5.5. La mise sous forme normale conjonctive peut avoir un coût exponentiel du fait des duplications de formules. Par exemple, la formule propositionnelle en forme normale disjonctive $(P_1 \wedge Q_1) \vee (P_2 \wedge Q_2) \vee (P_3 \wedge Q_3)$ a pour forme normale conjonctive

$$\begin{aligned}&(P_1 \vee P_2 \vee P_3) \wedge (P_1 \vee P_2 \vee Q_3) \wedge (P_1 \vee Q_2 \vee P_3) \wedge (P_1 \vee Q_2 \vee Q_3)\\ \wedge\ &(Q_1 \vee P_2 \vee P_3) \wedge (Q_1 \vee P_2 \vee Q_3) \wedge (Q_1 \vee Q_2 \vee P_3) \wedge (Q_1 \vee Q_2 \vee Q_3)\ .\end{aligned}$$

Cela correspond à la forme clausale

$$\begin{aligned}\{&\{P_1, P_2, P_3\}, \{P_1, P_2, Q_3\}, \{P_1, Q_2, P_3\}, \{P_1, Q_2, Q_3\},\\ &\{Q_1, P_2, P_3\}, \{Q_1, P_2, Q_3\}, \{Q_1, Q_2, P_3\}, \{Q_1, Q_2, Q_3\}\}\ .\end{aligned}$$

Sa généralisation $\bigvee_{1\leq i\leq n} P_i \wedge Q_i$ a pour forme normale conjonctive $\bigwedge_{J\subseteq\{1,\ldots,n\}} \bigvee_{1\leq i\leq n} \ell_{J,i}$ où $\ell_{J,i} = P_i$ si $i \in J$ et $\ell_{J,i} = Q_i$ sinon ; c'est une formule contenant 2^n clauses, chacune contenant n littéraux.

En conclusion, la mise sous forme normale conjonctive à l'aide de la distributivité de $\vee$ sur $\wedge$ produit une formule logiquement équivalente et est facile à utiliser sur de petits exemples « à la main », mais l'exemple 5.5 montre que cette transformation a un coût exponentiel, ce qui la rend inutilisable sur les formules propositionnelles que l'on souhaite manipuler en pratique. Ce problème a une solution : calculer une formule sous forme normale conjonctive *équi-satisfiable* à la place d'une formule logiquement équivalente ; c'est ce que nous allons voir maintenant.

(Perifel, 2014, prop. 3-Z), (Goubault-Larrecq et Mackie, 1997, exo. 2.23), (David, Nour et Raffalli, 2003, def. 7.4.15), (Carton, 2008, p. 191), (Harrisson, 2009, sec. 2.8)

5.2.2. *Forme clausale équi-satisfiable.* En général, étant donnée une formule propositionnelle φ en forme normale négative, on peut construire en temps linéaire une formule équi-satisfiable sous forme clausale.

Pour chaque sous-formule φ' de φ, on introduit pour cela une proposition fraîche $Q_{\varphi'} \notin \mathrm{fp}(\varphi)$ et on définit la formule propositionnelle

$$\psi_{\varphi'} \stackrel{\text{def}}{=} \begin{cases} P & \text{si } \varphi' = P\ , \\ \neg P & \text{si } \varphi' = \neg P\ , \\ Q_{\varphi_1} \vee Q_{\varphi_2} & \text{si } \varphi' = \varphi_1 \vee \varphi_2\ , \\ Q_{\varphi_1} \wedge Q_{\varphi_2} & \text{si } \varphi' = \varphi_1 \wedge \varphi_2\ . \end{cases}$$

Dans le cadre de la mise sous forme clausale, cette transformation est parfois appelée « transformation de Tseitin », qui historiquement ne suppose pas φ sous forme normale négative et utilise des équivalences $Q_{\varphi'} \Leftrightarrow \psi_{\varphi'}$ au lieu des implications $Q_{\varphi'} \Rightarrow \psi_{\varphi'}$.

La formule propositionnelle désirée est alors $\psi \stackrel{\text{def}}{=} Q_\varphi \wedge \bigwedge_{\varphi' \text{ sous-formule de } \varphi}(Q_{\varphi'} \Rightarrow \psi_{\varphi'})$. Cette formule propositionnelle a deux propriétés remarquables :

(1) elle est facile à mettre sous forme *3-CNF* : c'est en effet une conjonction où les implications peuvent être mises sous forme normale conjonctive en utilisant la définition de l'implication et au besoin la distributivité de $\vee$ sur $\wedge$: par exemple
$$(Q_{\varphi_1\wedge\varphi_2} \Rightarrow \psi_{\varphi_1\wedge\varphi_2}) \Leftrightarrow \big((\neg Q_{\varphi_1\wedge\varphi_2} \vee Q_{\varphi_1}) \wedge (\neg Q_{\varphi_1\wedge\varphi_2} \vee Q_{\varphi_2})\big)$$
et
$$(Q_{\varphi_1\vee\varphi_2} \Rightarrow \psi_{\varphi_1\vee\varphi_2}) \Leftrightarrow (\neg Q_{\varphi_1\vee\varphi_2} \vee Q_{\varphi_1} \vee Q_{\varphi_2})\ .$$

(2) sa représentation arborescente est de *taille linéaire* en la taille de la formule φ : il y a une implication $Q_{\varphi'} \Rightarrow \psi_{\varphi'}$ par sous-formule φ' de φ, et chacune de ces implications est de taille bornée par une constante.

Proposition 5.6. *Les formules propositionnelles φ et $\psi \stackrel{\text{def}}{=} Q_\varphi \wedge \bigwedge_{\varphi' \text{ sous-formule de } \varphi}(Q_{\varphi'} \Rightarrow \psi_{\varphi'})$ sont équi-satisfiables.*

Démonstration. Supposons φ satisfaite par une interprétation I. On étend cette interprétation en associant, pour chaque sous-formule φ', $[\![\varphi']\!]^I$ à la proposition $Q_{\varphi'}$: $I' \stackrel{\text{def}}{=} I[[\![\varphi']\!]^I/Q_{\varphi'}]_{\varphi'}$. Montrons que $I' \vDash \psi$ et donc que ψ est satisfiable. Il suffit de montrer que chacune des clauses de ψ est satisfaite par I'.

Tout d'abord, comme $I \vDash \varphi$, $I' \vDash Q_\varphi$. Puis on montre par analyse de cas que, pour toute sous-formule φ', on a $I' \vDash Q_{\varphi'} \Rightarrow \psi_{\varphi'}$.

- cas $\varphi' = P$: on veut montrer que $I' \vDash Q_P \Rightarrow P$. Supposons pour cela que $I' \vDash Q_P$. Alors par définition de I', $I \vDash P$. Toujours par définition de I', $I' \vDash P$ comme désiré.
- cas $\varphi' = \neg P$: on veut montrer que $I' \vDash Q_{\neg P} \Rightarrow \neg P$. Supposons pour cela que $I' \vDash Q_{\neg P}$. Alors par définition de I', $I \vDash \neg P$. Toujours par définition de I', $I' \vDash \neg P$ comme désiré.
- cas $\varphi' = \varphi_1 \wedge \varphi_2$: on veut montrer que $I' \vDash Q_{\varphi'} \Rightarrow Q_{\varphi_1} \wedge Q_{\varphi_2}$. Supposons pour cela que $I' \vDash Q_{\varphi'}$. Alors par définition de I', $I \vDash \varphi'$, donc $I \vDash \varphi_1$ et $I \vDash \varphi_2$. Toujours par définition de I', on a donc $I' \vDash Q_{\varphi_1}$ et $I' \vDash Q_{\varphi_2}$ comme désiré.
- cas $\varphi' = \varphi_1 \vee \varphi_2$: on fait une analyse similaire.

La preuve de la proposition 5.6 montre en fait que si ψ est satisfiable, alors elle l'est par une extension d'une interprétation qui satisfait φ.

Inversement, supposons ψ satisfaite par une interprétation I'. On montre par induction sur les sous-formules φ' de φ que $I' \vDash Q_{\varphi'}$ implique $I' \vDash \varphi'$; comme $I' \vDash Q_\varphi$ on aura bien $I' \vDash \varphi$ et donc φ satisfiable.

Pour les cas de base $\varphi' = P$ (resp. $\varphi' = \neg P$), on a par hypothèse $I' \vDash Q_P \Rightarrow P$ (resp. $I' \vDash Q_{\neg P} \Rightarrow \neg P$) et donc $I' \vDash Q_P$ implique $I \vDash P$ (resp. $I' \vDash \neg P$). Pour l'étape d'induction,

- si $\varphi' = \varphi_1 \wedge \varphi_2$, $I' \vDash Q_{\varphi'}$ implique $I' \vDash Q_{\varphi_1}$ et $I' \vDash Q_{\varphi_2}$ (car par hypothèse $I' \vDash Q_{\varphi'} \Rightarrow (Q_{\varphi_1} \wedge Q_{\varphi_2})$), qui implique $I' \vDash \varphi_1$ et $I' \vDash \varphi_2$ (par hypothèse d'induction), qui implique $I' \vDash \varphi'$;
- si $\varphi' = \varphi_1 \vee \varphi_2$, on fait une analyse similaire. □

Corollaire 5.7. *Pour toute formule propositionnelle, on peut construire en temps déterministe linéaire une formule équi-satisfiable sous forme 3-CNF.*

Exemple 5.8. Reprenons la formule propositionnelle de l'exemple 5.5 pour $n = 3$: la formule $\bigvee_{1 \le i \le 3} P_i \wedge Q_i$ est représentée dans la figure 3, où l'on a annoté chacune des sous-formules avec des noms de propositions en orange.

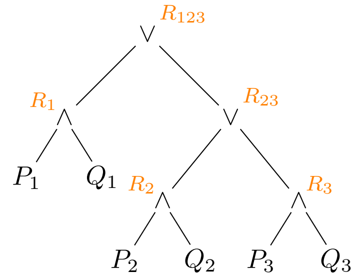


FIGURE 3. La formule propositionnelle $\bigvee_{1 \le i \le 3} P_i \wedge Q_i$ annotée.

Une formule propositionnelle en forme normale conjonctive équi-satisfiable (légèrement simplifiée par rapport à la transformation ci-dessus) est

$$R_{123} \wedge (\neg R_{123} \vee R_1 \vee R_{23}) \wedge (\neg R_{23} \vee R_2 \vee R_3) \wedge \bigwedge_{1 \le i \le 3} (\neg R_i \vee P_i) \wedge (\neg R_i \vee Q_i) ,$$

que l'on peut simplifier en

$$(R_1 \vee R_2 \vee R_3) \wedge \bigwedge_{1 \le i \le 3} (\neg R_i \vee P_i) \wedge (\neg R_i \vee Q_i) .$$

La forme clausale correspondante est

$$\{\{R_1, R_2, R_3\}, \{\neg R_1, P_1\}, \{\neg R_1, Q_1\}, \{\neg R_2, P_2\}, \{\neg R_2, Q_2\}, \{\neg R_3, P_3\}, \{\neg R_3, Q_3\}\} .$$

5.2.3. *Format DIMACS.* Les solveurs SAT emploient un format de fichier standard pour les formules propositionnelles en forme clausale, appelé « format DIMACS » du nom du *Center for Discrete Mathematics and Theoretical Computer Science* qui avait organisé les premières compétitions internationales. Voici par exemple comment représenter la forme clausale de l'exemple 5.8 en format DIMACS :

example-5-8.cnf
```
c exemple 5.8 en format DIMACS
c   table des propositions
c   R₁  P₁  Q₁  R₂  P₂  Q₂  R₃  P₃  Q₃
c   1   2   3   4   5   6   7   8   9
c
p cnf 9 7
 1 4 7 0
-1 2 0
-1 3 0
-4 5 0
-4 6 0
-7 8 0
-7 9 0
```

Le fichier commence par cinq lignes de commentaires, qui débutent par le caractère « c ». En format DIMACS, les propositions sont représentées par des entiers strictement positifs, et on a

ajouté en commentaire comment les noms des propositions de l'exemple 5.8 ont été numérotés pour faciliter la lecture de l'exemple (mais rien n'impose de le faire). La sixième ligne est le véritable début du fichier, qui commence par « `p cnf` » suivi de deux nombres :

— « 9 » correspond au nombre de propositions utilisées dans la forme clausale ;
— « 7 » correspond au nombre de clauses.

Les clauses occupent les lignes suivantes. Chaque clause se termine par le caractère « 0 », et consiste en une séquence de nombres entiers non nuls : un nombre positif n désigne la proposition numérotée par n, tandis qu'un nombre négatif $-n$ désigne la négation de cette proposition. Par exemple, à la huitième ligne, « `-1` » correspond à $\neg R_1$, tandis que « 2 » correspond à P_1.

5.2.4. *Implémentation de la forme clausale en Java.* Voyons comment implémenter la mise sous forme clausale équi-satisfiable en Java. Dans l'esprit du format DIMACS, nous travaillons maintenant avec une représentation des littéraux comme des entiers. Nous allons représenter une clause comme un objet de la classe `Collection<Integer>`. Voici une classe pour les clauses.

Clause
```
import java.util.*;
public class Clause extends HashSet<Integer> {
    public Clause () {
        super();
    }
    // ⋮
}
```

Une forme clausale peut être représentée comme un objet de `Collection<Clause>`. Voici une implémentation possible.

DIMACS
```
import java.util.*;
public class DIMACS {
    protected int nprops;                        // nombre de propositions
    private Map<String,Integer> propnames;  // nom de proposition ↦ entier
    private Collection<Clause> clauses;      // ensemble de clauses
    public DIMACS (int nprops, Map<String,Integer> propnames) {
        this.nprops = nprops;
        this.propnames = propnames;
        this.clauses = new HashSet<Clause>();
    }
    public boolean add(Clause c) {
        return clauses.add(c);
    }
    // ⋮
}
```

Pour construire un objet de la classe `DIMACS` depuis une formule propositionnelle, nous allons travailler sur une formule en forme normale négative. On suppose pour cela et que l'on dispose des deux méthodes suivantes.

Formule
```
protected abstract int setid(int n, Map<String,Integer> p);
protected int getid() { /* ... */ }
```

La première méthode attribue des entiers strictement positifs distincts à chaque nœud interne de la formule, ainsi qu'à chaque occurrence d'une même proposition ; de plus, la table p est mise à jour pour se souvenir de quel nom de proposition est associé à quel entier. La seconde retourne simplement l'entier associé au nœud courant.

L'étape suivante est de construire la formule propositionnelle $\psi = Q_\varphi \wedge \bigwedge_{\varphi' \text{ sous-formule de } \varphi}(Q_{\varphi'} \Rightarrow \psi_{\varphi'})$, ce qui est fait par la méthode `getDIMACS` ci-dessous.

Formule
```
protected abstract void addSousClauses(DIMACS clauses);
public DIMACS getDIMACS () {
    // passage sous forme normale négative
    Formule nnf = this.getNNF();
    // table qui associera un entier > 0 à chaque proposition
    Map<String,Integer> propnames = new HashMap<String,Integer>();
    // chaque nœud de `nnf' reçoit un entier non nul
    int nprops = nnf.setid(0, propnames);
    // ensemble initialement vide de clauses
    DIMACS dimacs = new DIMACS(nprops, propnames);
    // ajout de ⋀_{φ' sous-formule de φ}(Q_{φ'} ⇒ ψ_{φ'})
    nnf.addSousClauses(dimacs);
    // ajout de la clause Q_φ
    Clause c = new Clause();
    c.add(Integer.valueOf(nnf.getid()));
    dimacs.add(c);
    return dimacs;
}
```

L'idée ici est que `Formule.addSousClauses(clauses)` ajoute à `clauses` les clauses de la conjonction $\bigwedge_{\varphi' \text{ sous-formule de } \varphi}(Q_{\varphi'} \Rightarrow \psi_{\varphi'})$, et est implémentée différemment selon le type de la sous-formule φ'.

Formule.Et
```
protected void addSousClauses(DIMACS clauses) {
    // φ' = φ1 ∧ φ2
    phi1.addSousClauses(clauses);
    phi2.addSousClauses(clauses);
    // clause ¬Q_{φ'} ∨ Q_{φ1}
    Set<Integer> c1 = new Clause();
    c1.add(Integer.valueOf(-this.getid()));
    c1.add(Integer.valueOf(phi1.getid()));
    // clause ¬Q_{φ'} ∨ Q_{φ2}
    Clause c2 = new Clause();
    c2.add(Integer.valueOf(-this.getid()));
    c2.add(Integer.valueOf(phi2.getid()));
    // ajout des nouvelles clauses
    clauses.add(c1);
    clauses.add(c2);
}
```

Formule.Ou
```
protected void addSousClauses(DIMACS clauses) {
    // φ' = φ1 ∨ φ2
    phi1.addSousClauses(clauses);
    phi2.addSousClauses(clauses);
    // clause ¬Q_{φ'} ∨ Q_{φ1} ∨ Q_{φ2}
    Clause c = new Clause();
    c.add(Integer.valueOf(-this.getid()));
    c.add(Integer.valueOf(phi1.getid()));
    c.add(Integer.valueOf(phi2.getid()));
    // ajout de la nouvelle clause
```

```
    clauses.add(c);
}
```

Formule.Non

```
protected void addSousClauses(DIMACS clauses) {
    // φ' = ¬P
    // rien à faire
}
```

Formule.Proposition

```
protected void addSousClauses(DIMACS clauses) {
    // φ' = P
    // rien à faire
}
```

Enfin, voici une implémentation de `toString()` qui retourne une chaîne de caractères au format DIMACS.

Clause

```
public String toString() {
    String ret = "";
    for (Integer i : this)
        ret += i + " ";
    return ret + "0";
}
```

DIMACS

```
public String toString() {
    String out = "";
    // commentaires de début du fichier : table des numéros de propositions
    out += "c table des propositions\nc   ";
    for (Map.Entry<String,Integer> e : propnames.entrySet()) {
        String sid = e.getValue().toString();
        out += e.getKey();
        for (int i = 0; i < sid.length() - e.getKey().length(); i++)
            out += " ";
        out += " ";
    }
    out += "\nc   ";
    for (Map.Entry<String,Integer> e : propnames.entrySet()) {
        String sid = e.getValue().toString();
        out += sid;
        for (int i = 0; i < e.getKey().length() - sid.length(); i++)
            out += " ";
        out += " ";
    }
    // en-tête avec les nombres de propositions et de clauses
    out += "\np cnf "+ nprops +" "+ this.size() + "\n";
    // ajout des clauses au format DIMACS
    for (Clause c : clauses)
        ret += c + "\n";
    return out;
}
```

(DUPARC, 2015, sec. 2.10.1)

5.3. **Forme normale disjonctive.** Toujours en utilisant la distributivité, on obtient une mise sous *forme normale disjonctive* $\mathrm{dnf}(\varphi)$ pour toute φ en forme normale négative ; le cas le plus important de cette transformation est

$$\mathrm{dnf}(\varphi \wedge (\psi \vee \psi')) \stackrel{\text{def}}{=} \mathrm{dnf}((\psi \vee \psi') \wedge \varphi) \stackrel{\text{def}}{=} \mathrm{dnf}(\varphi \wedge \psi) \vee \mathrm{dnf}(\varphi \wedge \psi') .$$

Le résultat est une formule propositionnelle $\bigvee_{1\leq i\leq m}\bigwedge_{1\leq j\leq n_i}\ell_{i,j}$ où les $\ell_{i,j}$ sont des littéraux.

Étant donnée une telle formule, déterminer si elle est satisfiable peut être effectué en temps linéaire (en supposant un hachage parfait des noms de propositions) : une conjonction $\bigwedge_{1\leq j\leq n_i}\ell_{i,j}$ de littéraux est en effet satisfiable si et seulement si elle ne contient pas à la fois une proposition P et sa négation $\neg P$.

Comme la mise sous forme normale conjonctive, cette transformation peut avoir un coût exponentiel. Cependant, et contrairement à ce que nous venons de voir pour la forme normale conjonctive en section 5.2.2, on ne peut pas espérer avoir un algorithme en temps polynomial pour calculer une formule propositionnelle sous forme normale disjonctive équi-satisfiable avec une formule donnée en entrée (sous réserve que $\mathsf{P}\neq\mathsf{NP}$, une notion qui sera vue en cours de « calculabilité et complexité » en M1), puisque résoudre la satisfiabilité de cette forme normale disjonctive se fait ensuite en temps polynomial.

Une alternative pour calculer une forme normale disjonctive d'une formule φ est de calculer sa table de vérité et d'en déduire sa forme normale disjonctive complète *comme cela avait été fait dans la preuve du théorème 3.8 de complétude fonctionnelle ; voir l'exercice 3 du TD n°3. Cette forme complète est* canonique *: deux formules logiquement équivalentes ont la même forme normale disjonctive complète. La forme normale disjonctive complète est assez peu utile en pratique puisque le calcul de la table de vérité prend systématiquement un temps exponentiel en le nombre de propositions $|\mathrm{fp}(\varphi)|$.*

6. Modélisation

Résumé. De nombreux problèmes informatiques peuvent être exprimés comme la satisfiabilité d'une formule propositionnelle. Les *solveurs* SAT sont des logiciels dédiés à ce problème, qui prennent en entrée une formule propositionnelle sous forme normale conjonctive (écrite au format DIMACS), et cherchent à répondre si la formule est satisfiable ou non.

6.1. **Utilisation de solveurs** SAT. Les solveurs SAT sont des programmes qui déterminent si une formule propositionnelle φ donnée est satisfiable ou non, et si oui, fournissent une interprétation I qui satisfait la formule, c'est-à-dire telle que $I \models \varphi$. Les solveurs SAT prennent en entrée une forme clausale $\mathrm{Cl}(\varphi)$ au format DIMACS (c.f. section 5.2.3).

6.1.1. *Utilisation de MiniSAT.* MiniSAT (`http://minisat.se/`) est un solveur SAT facile à installer puisqu'il existe des paquets pour distributions GNU/Linux. Une invocation sur le fichier DIMACS de la section 5.2.3 est « `minisat exemple-5-8.cnf exemple-5-8.modele` ». Dans le cas où l'ensemble de clauses fourni en entrée est satisfiable, une interprétation partielle est retournée :

```
example-5-8.modele
SAT
-1 2 3 -4 5 6 7 8 9 0
```

L'interprétation est retournée sous la forme d'une clause DIMACS, où les propositions associées à 1 apparaissent comme des entiers positifs et celles associées à 0 comme des entiers négatifs. En l'occurrence, l'interprétation en termes de l'exemple 5.8 est

$$[0/R_1, 1/P_1, 1/Q_1, 0/R_2, 1/P_2, 1/Q_2, 1/R_3, 1/P_3, 1/Q_3] .$$

En terme de la formule originale de l'exemple 5.5, nous avons vu dans la preuve de la proposition 5.6 qu'il suffit d'ignorer les propositions fraîches ajoutées par la mise sous forme équisatisfiable ; l'interprétation partielle suivante est donc un modèle de $(P_1 \wedge Q_1) \vee (P_2 \wedge Q_2) \vee (P_3 \wedge Q_3)$:

$$[1/P_1, 1/Q_1, 1/P_2, 1/Q_2, 1/P_3, 1/Q_3] .$$

6.1.2. *Utilisation de Sat4j dans un programme Java.* Sat4j (`http://www.sat4j.org/`) est un solveur SAT plus récent écrit en Java, qui intègre les heuristiques utilisées par de nombreux solveurs comme MiniSAT et Glucose. Il peut être appelé directement depuis un programme Java qui construit les clauses successives comme des objets de type `int[]`. Voici comment appeler Sat4j depuis notre classe `DIMACS` :

```
DIMACS
import org.sat4j.minisat.*;
import org.sat4j.specs.*;
import org.sat4j.core.*;
// ⋮
    private int[] interpretation;
    public int[] modele() {
        return interpretation;
    }
    public boolean satisfiable() {
        ISolver solver = SolverFactory.newDefault();
        // initialisation du solver
        solver.newVar(nprops);
        solver.setExpectedNumberOfClauses(clauses.size());
```

```
        try {
            for (Clause c : clauses) // ajout des clauses
                solver.addClause(new VecInt(c.stream()
                    .mapToInt(i->i).toArray()));
            IProblem problem = solver;
            boolean ret = problem.isSatisfiable();
            if (ret)
                interpretation = problem.model();
            return ret;
        } catch (ContradictionException e) {
            return false;
        } catch (TimeoutException e) {
            System.err.println("timeout!");
            return false;
        }
    }
```

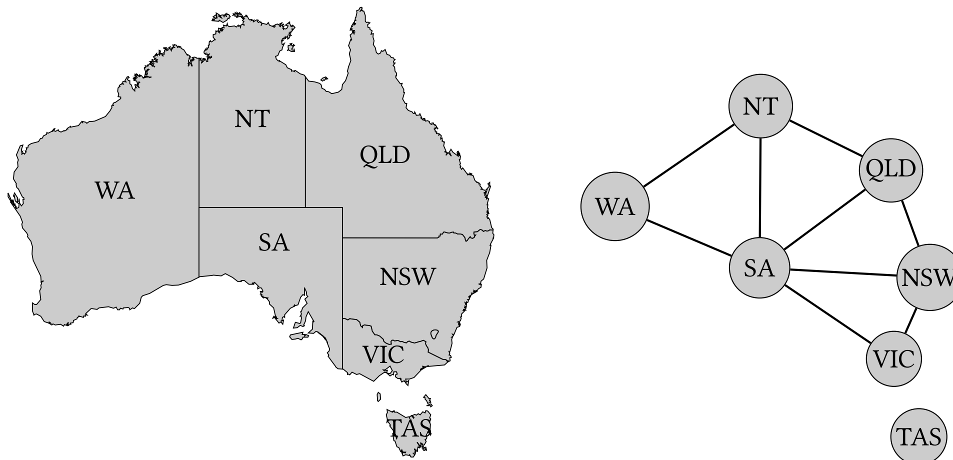


FIGURE 4. La carte des territoires de l'Australie[9] et le graphe associé.

6.2. **Exemple de modélisation : coloration de graphe.** Les solveurs SAT sont particulièrement utiles pour des problèmes combinatoires pour lesquels on ne connaît pas d'algorithme efficace dans le pire des cas – et on soupçonne même qu'il n'existe pas de tels algorithmes. Il s'avère en effet que les solveurs SAT permettent de résoudre ces problèmes dans les cas *qui apparaissent en pratique.*

Notre problème est le suivant : est-il possible de colorier la carte de l'Australie (voir la carte [9]) avec seulement trois couleurs, disons rouge, vert et bleu, de telle sorte que deux territoires adjacents aient des couleurs différentes ? Ce problème est en réalité un problème de coloriage d'un graphe non orienté comme illustré à droite de la figure 4 : peut-on associer une couleur à chaque sommet du graphe, de telle sorte que deux sommets adjacents n'aient pas la même couleur ? La réponse est « oui » et un tel coloriage est donné dans la figure 5.

Notre objectif est cependant de trouver automatiquement une telle solution pour n'importe quel graphe fini non orienté. Voyons comment procéder pour notre exemple.

9. Auteur de la carte à gauche de la figure 4 : Lokal_Profil, licence [CC BY-SA 3.0], via *Wikimedia Commons*. Les territoires sont : *Western Australia* (WA), *Northern Territory* (NT), *South Australia* (SA), *Queensland* (QLD), *New South Wales* (NSW), *Victoria* (VIC) et *Tasmania* (TAS) ; on a ignoré le petit territoire de la capitale.

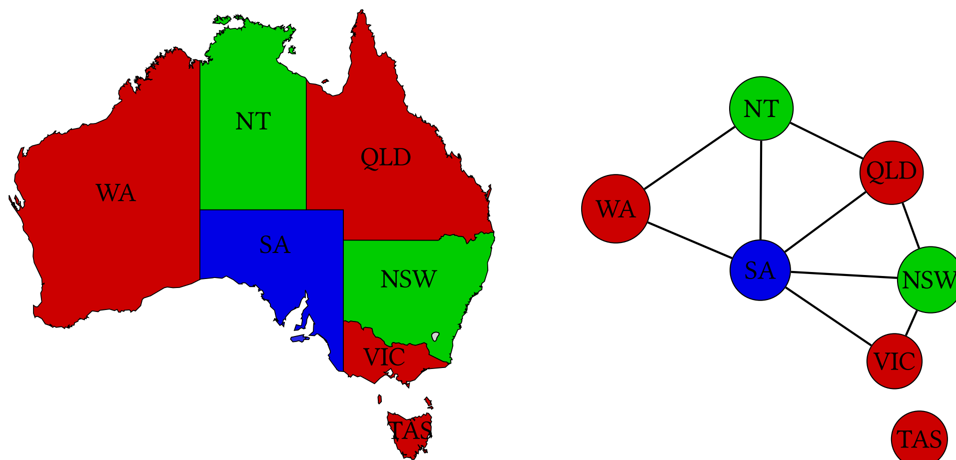


FIGURE 5. Un coloriage de la carte des territoires de l'Australie[9] et du graphe associé.

Choix des propositions. Nous travaillons avec des propositions $P_{v,c}$ où v est un sommet du graphe, c'est-à-dire un territoire dans $\{$WA, NT, SA, QLD, NSW, VIC, TAS$\}$ et c est une couleur dans $\{R, V, B\}$. Cela signifie qu'une interprétation I de ces propositions décrira une *relation* incluse dans $V \times C$: pour chaque paire (v, c), I va indiquer si le sommet v du graphe est colorié par la couleur c.

Voici le préambule d'un fichier au format DIMACS où on a indiqué en commentaire quel entier strictement positif est associé à chacune des paires $(v, c) \in V \times C$.

```
coloriage.cnf
c table des propositions
c  WA,R WA,V WA,B NT,R NT,V NT,V SA,R SA,V SA,B QLD,R QLD,V QLD,B
c  1    2    3    4    5    6    7    8    9    10    11    12
c  NSW,R NSW,V NSW,B VIC,R VIC,V VIC,B TAS,R TAS,V TAS,B
c  13    14    15    16    17    18    19    20    21
p cnf 21 55
```

Nous allons maintenant construire une formule propositionnelle qui sera satisfiable si et seulement s'il existe un coloriage du graphe. Toutes les interprétations I des propositions de la forme $P_{v,c}$ pour $v \in \{$WA, NT, SA, QLD, NSW, VIC, TAS$\}$ et $c \in \{R, V, B\}$ ne sont pas des coloriages. Plusieurs conditions doivent en effet être remplies.

Au moins une couleur par sommet. Nous ne devons pas laisser un territoire de l'Australie non colorié. Cela correspond à vérifier que pour chaque territoire $v \in \{$WA, NT, SA, QLD, NSW, VIC, TAS$\}$, au moins l'une des propositions parmi $P_{v,R}$, $P_{v,V}$ et $P_{v,B}$ est vraie dans I, soit la formule propositionnelle $P_{v,R} \vee P_{v,V} \vee P_{v,B}$. Voici les clauses correspondantes au format DIMACS.

```
coloriage.cnf
c  au moins une couleur par sommet
 1  2  3 0
 4  5  6 0
 7  8  9 0
10 11 12 0
13 14 15 0
16 17 18 0
19 20 21 0
```

Au plus une couleur par sommet. En effet, par exemple, le territoire de *New South Wales* ne peut pas être colorié à la fois en rouge et en vert. Cela correspond à vérifier que, pour chaque territoire $v \in \{\text{WA, NT, SA, QLD, NSW, VIC, TAS}\}$ et pour chaque paire de couleurs distinctes $c \neq c'$ issues de $\{R, V, B\}$, on n'ait pas à la fois $P_{v,c}$ et $P_{v,c'}$ vraies dans I. Cela correspond à la formule propositionnelle $\neg(P_{v,c} \wedge P_{v,c'})$, qui est équivalente à la clause $\neg P_{v,c} \vee \neg P_{v,c'}$. Voici les clauses correspondantes au format DIMACS pour WA, NT et SA.

coloriage.cnf
```
c au plus une couleur par sommet
 -1  -2 0
 -1  -3 0
 -2  -3 0
 -4  -5 0
 -4  -6 0
 -5  -6 0
 -7  -8 0
 -7  -9 0
 -8  -9 0
```

Avec ces deux types de clauses combinées, on garantit en fait qu'il existe une *fonction* entre les sommets du graphe et les couleurs.

Couleurs distinctes pour sommets adjacents. Enfin, nous devons vérifier que pour toute paire (u, v) de territoires adjacents, leurs couleurs associées sont distinctes, c'est-à-dire que I vérifie $\neg(P_{u,c} \wedge P_{v,c})$ pour tout arête (u, v) et toute couleur c ; cela s'écrit sous forme clausale comme $\neg P_{u,c} \vee \neg P_{v,c}$. Par exemple, pour les arêtes (WA, NT), (WA, SA) et (NT, SA), on aura au format DIMACS :

coloriage.cnf
```
c pas la même couleur sur deux sommets adjacents
c arête (WA,NT)
 -1  -4 0
 -2  -5 0
 -3  -6 0
c arête (WA,SA)
 -1  -7 0
 -2  -8 0
 -3  -9 0
c arête (NT,SA)
 -4  -7 0
 -5  -8 0
 -6  -9 0
```

En général, pour un graphe fini non orienté $G = (V, E)$ et un ensemble de couleurs C, le graphe est coloriable si et seulement si la formule propositionnelle suivante est satisfiable :

$$\Big(\bigwedge_{v \in V} \bigvee_{c \in C} P_{v,c} \Big) \wedge \Big(\bigwedge_{v \in V} \bigwedge_{c \neq c'} (\neg P_{v,c} \vee \neg P_{v,c'}) \Big) \wedge \Big(\bigwedge_{(u,v) \in E} \bigwedge_{c \in C} (\neg P_{u,c} \vee \neg P_{v,c}) \Big) . \qquad (4)$$

6.3. **Exemple de modélisation : dépendances logicielles.** Voici un problème concret en informatique : comment assurer que toutes les dépendances d'un logiciel soient correctement installées et configurées ? Cela touche les programmes installés sur un système, par exemple via un gestionnaire de paquets comme `apt` sur les systèmes Debian Linux et apparentés (comme Ubuntu), mais aussi les bibliothèques nécessaires pour compiler du code, par exemple via `autoconf` pour C/C++, `maven` pour Java, `opam` pour OCaml, `pip` pour Python, etc. Des problèmes surgissent assez vite avec tous ces systèmes du fait de *conflits* entre certaines versions de dépendances qui ne peuvent pas être installées simultanément ; on parle couramment de « *dependency hell* ».

(Mancinelli et al., 2006)

Comme nous allons le voir, on peut résoudre ces problèmes de dépendances à l'aide d'un modélisation en logique propositionnelle et d'un solveur SAT ; c'est d'ailleurs ce qui est fait au sein du logiciel Eclipse à l'aide de Sat4j pour la gestion des plugins.

Voici un scénario concret de problème de dépendances avec le système `apt`. Ce genre de problèmes peut cependant survenir avec la plupart des systèmes de gestion de dépendances. Si vous avez un système Linux basé sur Debian, vous pouvez même tester ce scénario en ajoutant les deux lignes suivantes à `/etc/apt/sources.list` avant de lancer `sudo apt-get update` :

/etc/apt/sources.list

```
deb [trusted=yes] https://www.irif.fr/~schmitz/teach/2020_lo5/debs ./
deb [trusted=yes] https://www.irif.fr/~schmitz/teach/2020_lo5/debs-old ./
```

Un paquet logiciel Debian contient des méta-informations ; voici par exemple les informations du paquet `foo` disponible aux adresses ci-dessus :

apt-cache show foo

```
Package: foo
Version: 1.0
Architecture: all
Maintainer: anon
Depends: bar (>= 2.1), baz | qux
Filename: ./foo-1.0.deb
Size: 704
MD5sum: b898166d798077e84317686d66d259e5
SHA1: e4c056543faf48b4ba97fd2b113fd05397ea8a7d
SHA256: 591489045bf2ce5bc7c5d09cb3e9dd6416939ee23a38f4cd3ecba80889d717f7
Description: dummy foo-1.0 package
Description-md5: c777289cc850cccf5b3b07c9c31902f2
```

Ce qui nous intéresse est le nom du paquet (`foo`), sa version (`1.0`), et ses dépendances (`bar (>= 2.1), baz | qux`) : le paquet `foo` dans sa version `1.0` dépend du paquet `bar` dans une version supérieure ou égale à `2.1`, et soit du paquet `baz` soit du paquet `qux`, sans contrainte de version.

À noter qu'un même paquet peut être disponible en plusieurs versions :

apt-cache show quxx

```
Package: quxx
Version: 1.4
Architecture: all
Maintainer: anon
Conflicts: quz
Filename: ./quxx-1.4.deb
Size: 664
MD5sum: ce56ed662469facfc2af728e75262849
SHA1: f1a91faf93a68b8eb626032607d148b494731124
SHA256: b7c4498b4b16c84e9315cf2ecea613ec11d943564ff2ef7d0c8b762fe10eced2
Description: dummy quxx-1.4 package
Description-md5: 0d88c6b67d64002702518fb93a4ebfee

Package: quxx
Version: 1.3
Architecture: all
Maintainer: anon
Filename: ./quxx-1.3.deb
Size: 680
MD5sum: 3553398d7d504fd1993c719436709f68
SHA1: c3bac75a683808f515656eaa3cf6460d71ab933b
```

```
SHA256: f58ac7d4ad72bf47b2ed4afaf97f9770f048076cf17ef53dae523cd8d66ac25a
Description: dummy quxx-1.3 package
Description-md5: 94ff5e3c00228e821c8163972d5fae10
```

Ici, quxx existe en version 1.3 et en version 1.4. Ces paquets n'ont pas de dépendances, mais quxx version 1.4 est en conflit avec quz : il ne peut pas être installé en même temps qu'aucune version de quz. Notons que les dépendances et conflits peuvent varier d'une version à l'autre d'un même paquet. La figure 6 résume les dépendances (en orange) et conflits (en rouge pointillé) entre les paquets de notre scénario.

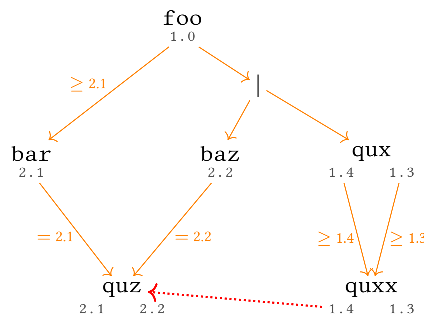


FIGURE 6. Les dépendances (en orange) et conflits (en rouge) entre paquets.

Si on essaie d'installer foo, on obtient un message d'erreur qui nous renseigne assez peu :

```
sudo apt-get install --dry-run foo
The following packages have unmet dependencies:
 foo : Depends: bar (>= 2.1) but it is not going to be installed
E: Unable to correct problems, you have held broken packages.
```

Si on analyse la situation à l'aide du graphe de la figure 6, on peut voir que, pour pouvoir installer foo, on doit installer bar, et que ce dernier exige l'installation de quz dans sa version 2.1. On doit aussi installer baz ou qux. Le paquet baz exige l'installation de quz dans sa version 2.2 : ceci est incompatible avec quz-2.1 qui est une version différente du même paquet, donc on ne peut pas installer bar et baz en même temps. Il reste donc qux, mais on voit qu'il dépend de quxx qui dans sa version 1.4 est en conflit avec quz, donc on ne peut pas installer bar et qux-1.4 en même temps. Ceci explique le message d'erreur d'apt. Il y aurait cependant une solution si on se permettait d'installer des paquets dans des versions qui ne sont pas les plus récentes : installer bar-2.1, quz-2.1, qux-1.3 et quxx-1.3.

Voyons maintenant comment modéliser notre scénario en logique propositionnelle. Formellement, on dispose d'un ensemble de paquets versionnés

$$\mathcal{D} \stackrel{\text{def}}{=} \{\texttt{foo-1.0}, \texttt{bar-2.1}, \texttt{baz-2.2}, \texttt{qux-1.3}, \texttt{qux-1.4}, \texttt{quxx-1.3}, \texttt{quxx-1.4}, \texttt{quz-2.1}, \texttt{quz-2.2}\}$$

et on cherche une *solution*, c'est-à-dire un sous-ensemble $\mathcal{E} \subseteq \mathcal{D}$ tel que $\mathcal{E}$ contienne foo-1.0 le paquet que nous voulons installer, et tel que les dépendances et conflits de la figure 6 soient respectés.

Choix des propositions. Nous allons utiliser des propositions Q_p pour chaque paquet versionné $p \in \mathcal{D}$. Une interprétation I de ces propositions définit alors le sous-ensemble $\{p \mid Q_p^I = 1\} \subseteq \mathcal{D}$. Les contraintes que nous allons écrire assureront ensuite que foo-1.0 appartienne au sous-ensemble ainsi défini et que les dépendances et conflits soient respectés.

Voici le préambule d'un fichier au format DIMACS où on a indiqué en commentaire quel entier strictement positif est associé à chaque paquet versionné $p \in \mathcal{D}$.

dependances.cnf
```
c Table des propositions : une proposition par version de paquet
c foo-1.0 bar-2.1 baz-2.2 qux-1.3 qux-1.4 quz-2.1 quz-2.2 quxx-1.3 quxx-1.4
c    1       2       3       4       5       6       7        8        9
c
p cnf 9 12
```

Nous allons maintenant écrire une formule propositionnelle qui sera satisfaite par une interprétation I si et seulement si I définit une solution.

Le paquet `foo-1.0` *appartient à la solution.* On garantit cela à l'aide de la formule $Q_{\texttt{foo-1.0}}$. En format DIMACS :

dependances.cnf
```
c on souhaite installer foo-1.0
1 0
```

Dépendances. Dans les cas les plus simples, pour chaque dépendance $p \rightarrow p'$ indiquée en orange dans la figure 6, on va ajouter une contrainte de la forme $Q_p \Rightarrow Q_{p'}$: si p est installé, alors on doit aussi installer p'. Cela donne par exemple la formule $Q_{\texttt{foo-1.0}} \Rightarrow Q_{\texttt{bar-2.1}}$.

De manière générale, une dépendance peut mener à une *disjonction* de paquets versionnés, et la contrainte logique correspondante est alors une implication d'une disjonction des propositions correspondantes ; par exemple $Q_{\texttt{foo-1.0}} \Rightarrow (Q_{\texttt{baz-2.2}} \vee Q_{\texttt{qux-1.3}} \vee Q_{\texttt{qux-1.4}})$.

Ces deux contraintes peuvent être mises sous forme normale conjonctive comme $\neg Q_{\texttt{foo-1.0}} \vee Q_{\texttt{bar-2.1}}$ et $\neg Q_{\texttt{foo-1.0}} \vee Q_{\texttt{baz-2.2}} \vee Q_{\texttt{qux-1.3}} \vee Q_{\texttt{qux-1.4}}$; voici leur traduction au format DIMACS :

dependances.cnf
```
c foo-1.0 dépend de : bar (>= 2.1), baz | qux
-1 2 0
-1 3 4 5 0
```

Les autres dépendances sont traitées de manière similaire.

Conflits déclarés. Il y a un conflit déclaré dans la figure 6 : quxx-1.4 est en conflit avec quz. On peut modéliser cela par la formule $Q_{\texttt{qux-1.4}} \Rightarrow \neg(Q_{\texttt{quz-2.1}} \vee Q_{\texttt{quz-2.2}})$: si on installe quxx-1.4, alors aucune version de quz ne peut être installée. Cette formule est logiquement équivalente à la formule $(\neg Q_{\texttt{qux-1.4}} \vee \neg Q_{\texttt{quz-2.1}}) \wedge (\neg Q_{\texttt{qux-1.4}} \vee \neg Q_{\texttt{quz-2.2}})$ sous forme normale conjonctive, et voici sa traduction au format DIMACS :

dependances.cnf
```
c quxx-1.4 est en conflit avec : quz
-9 -6 0
-9 -7 0
```

Conflits entre versions d'un même paquet. Enfin, il faut ajouter les conflits implicites : deux versions différentes d'un même paquet ne peuvent pas coexister. Par exemple, pour qux, cela revient à la formule propositionnelle $\neg(Q_{\texttt{qux-1.3}} \wedge Q_{\texttt{qux-1.4}})$, qui est logiquement équivalente à $\neg Q_{\texttt{qux-1.3}} \vee \neg Q_{\texttt{qux-1.4}}$ sous forme normale conjonctive ; voici sa traduction au format DIMACS :

dependances.cnf
```
c on ne peut pas avoir deux versions de qux
-4 -5 0
```

Des contraintes similaires doivent être écrites pour quxx et quz.

Si on appelle un solveur SAT comme MiniSAT sur le fichier DIMACS complet, celui-ci trouve un modèle

dependances.modele

```
SAT
1 2 -3 4 -5 6 -7 8 -9 0
```

Cette interprétation satisfait nos dépendances et conflits et correspond à la solution

$$\{\texttt{foo-1.0}, \texttt{bar-2.1}, \texttt{qux-1.3}, \texttt{quxx-1.3}, \texttt{quz-2.1}\} .$$

La commande `sudo apt-get install quxx=1.3 qux=1.3 quz=2.1 bar=2.1 foo` fonctionne maintenant et permet d'installer `foo`.

7. Satisfiabilité et recherche de modèle

Résumé. Une manière d'implémenter un solveur SAT est d'utiliser l'*algorithme DPLL* dû à Davis, Putnam, Logemann et Loveland. Cet algorithme travaille par *simplification* d'ensembles de clauses : la simplification d'un ensemble de clauses par un littéral ℓ élimine les clauses qui contiennent ℓ et retire $\overline{\ell}$ des clauses restantes. L'algorithme DPLL simplifie en priorité par les littéraux *unitaires* (ℓ est unitaire s'il existe une clause qui ne contient que ce littéral) et les littéraux *purs* (ℓ est pur si le littéral $\overline{\ell}$ n'apparaît nulle part dans l'ensemble de clauses).

Voir (Knuth, 2008, sec. 7.1.1), (Knuth, 2015, sec. 7.2.2.2) et (Kroening et Strichman, 2016, ch. 2) pour une présentation des aspects algorithmiques du problème de satisfiabilité, et (Perifel, 2014, thm. 3-V), (Arora et Barak, 2009, thm. 2.10), (Papadimitriou, 1993, thm. 8.2), (Carton, 2008, thm. 4.19) ou (Lassaigne et Rougemont, 2004, thm. 11.1) sur sa complexité et en particulier le théorème de Cook–Levin.

Le problème de *satisfiabilité* est le problème de décision suivant.

Problème (SAT).
entrée : une formule propositionnelle φ
question : φ est-elle satisfiable ?

Ce problème est résolu par les solveurs SAT ; de plus, si la réponse au problème est positive, ces solveurs fournissent une interprétation partielle I telle que $I \vDash \varphi$.

Dans de nombreuses utilisations de SAT, telles que les modélisations de la section 6, on suppose de plus que la formule φ est sous forme clausale, voire k-clausale pour un certain k fixé, auquel cas on parlera plutôt de « kSAT ». Comme vu dans le corollaire 5.7, SAT et 3SAT sont deux problèmes essentiellement équivalents.

7.1. **Recherche de modèle par énumération.** On peut résoudre automatiquement le problème SAT : puisque par la propriété 3.3, il suffit de trouver une interprétation partielle de domaine $\text{fp}(\varphi)$ qui satisfait φ, on peut simplement énumérer les $2^{|\text{fp}(\varphi)|}$ interprétations possibles. En termes de tables de vérité, cela revient à construire la table de φ et de tester si au moins une ligne met φ à 1.

On ne connaît pas d'algorithme pour SAT *qui travaille en temps polynomial dans le pire des cas – on soupçonne même qu'un tel algorithme en temps sous-exponentiel n'existe pas, ce qui est appelé l'«* exponential time hypothesis *».*

Exemple 7.1. Reprenons la forme clausale de l'exemple 3.12

$$F = \{\{P, Q, \neg R\}, \{Q, R\}, \{\neg P, \neg Q, R\}, \{\neg P, \neg R\}, \{P, \neg Q\}\} .$$

Sa table de vérité est donnée dans la table 6.

Table 6. La table de vérité de l'ensemble de clauses de l'exemple 3.12.

P	Q	R	$\{P, Q, \neg R\}$	$\{Q, R\}$	$\{\neg P, \neg Q, R\}$	$\{\neg P, \neg R\}$	$\{P, \neg Q\}$	F
1	1	1	1	1	1	0	1	0
1	1	0	1	1	0	1	1	0
1	0	1	1	1	1	0	1	0
1	0	0	1	0	1	1	1	0
0	1	1	1	1	1	1	0	0
0	1	0	1	1	1	1	0	0
0	0	1	0	1	1	1	1	0
0	0	0	1	0	1	1	1	0

On voit dans cette table que chaque ligne, c'est-à-dire chaque interprétation partielle de domaine $\{P, Q, R\}$, contient au moins une entrée 0 pour une des clauses de F. Par conséquent, la colonne pour F ne contient que des 0 : cette forme clausale est insatisfiable.

Plutôt que d'écrire explicitement la table de vérité, on peut aussi représenter les interprétations partielles de domaine $\text{fp}(\varphi)$ sous la forme d'un arbre : pour chaque proposition $P \in \text{fp}(\varphi)$, on

branche sur les deux choix $I \vDash \neg P$ et $I \vDash P$. L'arbre correspondant pour l'exemple 7.1 est donné dans la figure 7.

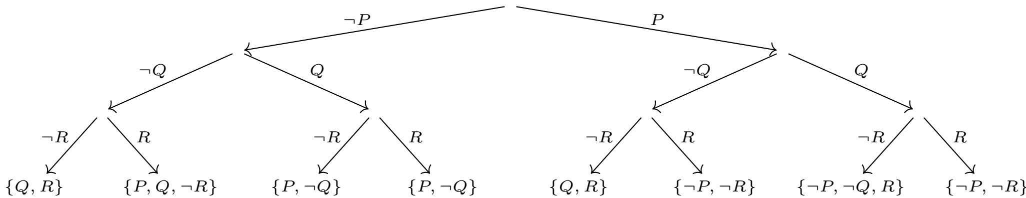


FIGURE 7. Un arbre de recherche de modèle pour l'exemple 7.1.

Chaque branche de l'arbre décrit une interprétation partielle sous la forme d'une liste de littéraux, et l'on a décoré chaque feuille de l'arbre par une clause non satisfaite par cette interprétation ; par exemple, la branche la plus à gauche étiquetée par $\neg P, \neg Q, \neg R$ correspond à l'interprétation partielle $[0/P, 0/Q, 0/R]$ (qui est la dernière ligne de la table 6), et cette interprétation ne satisfait pas la clause $\{Q, R\}$.

Cette représentation sous forme d'arbre de recherche a deux intérêts :
— d'une part, elle est moins longue à écrire à la main qu'une table de vérité,
— d'autre part, elle suggère un algorithme récursif qui explore l'arbre pour tester si une formule sous forme clausale est satisfiable ou non : chaque nœud interne de l'arbre correspond à un appel récursif d'une fonction pour tester la satisfiabilité, et aux feuilles on vérifie s'il existe au moins une clause satisfaite par l'interprétation partielle pour cette branche.

À noter que, dans le cas de formules propositionnelles sous forme clausale, il est très aisé de vérifier si une clause C est satisfaite ou non par une interprétation : $I \vDash C$ si et seulement s'il existe $\ell \in C$ tel que $I \vDash \ell$. Cela suggère le pseudo-code suivant pour évaluer la valeur de vérité d'une clause C sous une interprétation I :

Fonction `evalueclause`(C, I)
1 **pour tous les** $\ell \in C$ **faire**
2   **si** $I \vDash \ell$ **alors retourner** 1
3 **retourner** 0

Par suite, pour une formule propositionnelle sous forme clausale, c'est-à-dire un ensemble de clauses F, $I \vDash F$ si et seulement si $\forall C \in F, I \vDash C$:

Fonction `evalueclausal`(F, I)
1 **pour tous les** $C \in F$ **faire**
2   **si** not `evalueclause`(C, I) **alors retourner** 0
3 **retourner** 1

Le pseudo-code qui suit décrit l'algorithme récursif suggéré par les arbres de recherche, qui prend en entrée un ensemble de clauses F et une interprétation partielle I de domaine initialement vide.

Fonction `satisfiable`(F, I)
1 **si** $\mathrm{dom}(I) = \mathrm{fp}(F)$ **alors**
2 **retourner** `evalueclausal`(F, I)
3 **sinon**
4 **choisir** $P \in \mathrm{fp}(F) \setminus \mathrm{dom}(I)$
5 **retourner** `satisfiable`$(F, I[1/P])$ or `satisfiable`$(F, I[0/P])$

L'implémentation de la recherche de modèle par énumération en OCaml sera vue en TP du cours « programmation fonctionnelle ».

7.1.1. *Implémentation de la recherche par énumération en Java.*

Propositions et littéraux. Dans l'esprit du format DIMACS, chaque proposition P_i va être associée à un entier `i>0` ; le littéral P_i est alors représenté par l'entier `i` et le littéral $\neg P_i$ par l'entier `-i`.

Interprétations. Une interprétation est I maintenant représentée comme un tableau d'entiers ; si `nprops` est le nombre de propositions, alors `interpretation` sera un tableau de longueur `nprops`. L'index dans ce tableau d'un littéral ℓ représenté par un objet `l` de type `Integer` est fourni par la méthode suivante.

```
public static int index(Integer l) {
    int i = l.intValue();
    return (i > 0)? i-1: -i-1;
}
```

Pour un littéral ℓ représenté par un objet `l` de type `Integer`, $I \vDash \ell$ se vérifie en testant si `interpretation[index(l)] == l.intValue()`.

Clauses. Rappelons que nous représentons les clauses comme des objets de type `Clause`, lequel implémente `Collection<Integer>`. Une interprétation I est un modèle d'une clause C si et seulement si c'est un modèle d'au moins un littéral ℓ de C : $I \vDash C$ si et seulement si $\exists \ell \in C . I \vDash \ell$. Nous pouvons implémenter cela dans notre classe `Clause` par la méthode suivante.

```
Clause
public boolean evalue(int[] interpretation) {
    return this.stream().anyMatch(
        l -> interpretation[index(l)] == l.intValue());
}
```

Solveur SAT *naïf.* Voici enfin le code d'un solveur SAT naïf qui implémente la recherche de modèle par énumération. Ce solveur a accès aux trois champs suivants.

```
SolveurNaif
protected int nprops;
private Collection<Clause> clauses;
private int[] interpretation;
```

La méthode `satisfiable` construit successivement – via des appels récursifs – toutes les interprétations possibles des propositions, et retourne vrai dès qu'elle trouve un modèle.

```
SolveurNaif
private boolean satisfiable(int i) {
    // interprétation de toutes les propositions
    if (i == nprops)
        // est-ce que toutes les clauses sont satisfaites ?
        return clauses.stream().allMatch(c-> c.evalue(interpretation));
    else {
        // branchement :
        //  - tente de mettre la ième proposition à 1
        interpretation[i] = i+1;
        if (satisfiable(i+1))
```

```
            return true;
        // - restaure l'interprétation
        for (int j = i+1; j < nprops; j++)
            interpretation[j] = j+1;
        //  - tente de mettre la ième proposition à 0
        interpretation[i] = -i-1;
        return satisfiable(i+1);
    }
}
public boolean satisfiable() {
    return satisfiable(0);
}
```

Cette implémentation naïve d'un solveur SAT suffit pour des petits exemples, mais prend plus d'une seconde sur ma machine pour résoudre le problème de coloriage de la section 6.2.

7.2. **Recherche de modèle par simplification.** Un défaut de l'algorithme par énumération de la section précédente est qu'il attend d'avoir construit une interprétation partielle de domaine $\mathrm{fp}(\varphi)$ avant de tester si les clauses sont satisfaites. Pourtant, il est parfois possible de répondre plus tôt : par exemple, dans l'arbre de la figure 7, le nœud atteint en suivant le chemin $\neg P, Q$ correspond à une interprétation partielle $[0/P, 1/Q]$ qui ne satisfait pas la clause $\{P, \neg Q\}$, quel que soit le choix de l'interprétation de la proposition R.

Cependant, évaluer toutes les clauses C d'une formule à chaque nœud interne de l'arbre d'interprétation partielle I telle que $\mathrm{fp}(C) \subseteq \mathrm{dom}(I)$ serait coûteux. À la place, l'idée de la recherche de modèle par simplification est d'évaluer « partiellement » l'ensemble des clauses au fur et à mesure de l'exploration de l'arbre.

7.2.1. *Simplification de formes clausales.* La recherche de modèle par simplification simplifie un ensemble de clauses propositionnelles S jusqu'à obtenir une clause vide – auquel cas S n'était pas satisfiable – ou un ensemble vide de clauses – auquel cas S était satisfiable. Soit S un ensemble de clauses. On définit la *simplification* de S par un littéral ℓ comme l'ensemble de clauses

$$S[\top/\ell] \stackrel{\text{def}}{=} \{C \mid \big((C \cup \{\overline{\ell}\}) \in S \text{ ou } \overline{\ell} \notin C \in S\big) \text{ et } \ell \notin C\}$$

où l'on a éliminé les clauses de S contenant ℓ et simplifié les clauses de S de la forme $C \cup \{\overline{\ell}\}$ en leur enlevant $\overline{\ell}$. Par exemple,

$$\{\{P, Q\},\ \{\neg P, R\},\ \{P, \neg P\}\}[\top/P] = \{\{R\}\}\ ,$$
$$\{\{P, Q\},\ \{\neg P, R\},\ \{P, \neg P\}\}[\top/\neg P] = \{\{Q\}\}\ .$$

La simplification revient effectivement à substituer $\top$ à ℓ et $\bot$ à $\overline{\ell}$ dans toutes les clauses de S et à simplifier le résultat en utilisant les équivalences logiques $\varphi \vee \top \Leftrightarrow \top$, $\varphi \vee \bot \Leftrightarrow \varphi$ et $\varphi \wedge \top \Leftrightarrow \varphi$. La simplification d'un ensemble F de clauses par un littéral ℓ s'écrit en pseudo-code comme suit.

Fonction `simplifie`(F, ℓ)
1 $F' := \emptyset$
2 **pour tous les** $C \in F$ **faire**
3   **si** $\ell \notin C$ **alors** $F' := F' \cup \{C \setminus \{\overline{\ell}\}\}$
4 **retourner** F'

Pour une interprétation I, on note $I[1/\ell]$ pour l'interprétation qui associe 1 à P si $\ell = P$, 0 à P si $\ell = \neg P$, et Q^I à toute proposition $Q \notin \mathrm{fp}(\ell)$.

7.2.2. *Recherche par simplification.* Comme son nom l'indique, la *recherche par simplification* vise à trouver une interprétation qui satisfait toutes les clauses d'un ensemble F de clauses, en testant successivement si $F[\top/P]$ ou $F[\top/\neg P]$ est satisfiable ; voir la figure 8 (où les littéraux colorés en orange sont impactés par la prochaine simplification).

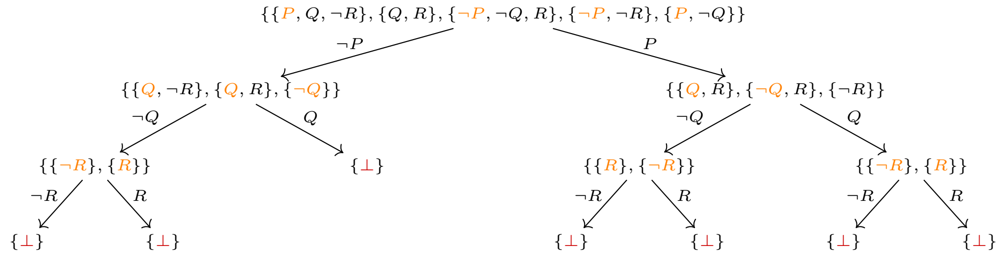


FIGURE 8. Un arbre de recherche par simplification pour l'exemple 7.1.

Cette recherche *réussit* s'il existe une branche qui termine sur l'ensemble vide $\emptyset$ de clauses, et *échoue* si toutes les branches aboutissent à un ensemble qui contient la clause vide (celle-ci est notée $\bot$). La recherche illustrée dans la figure 8 échoue : l'ensemble F de l'exemple 7.1 était bien insatisfiable.

Exemple 7.2. Voici un autre exemple de recherche par simplification. Soit l'ensemble de clauses $F \stackrel{\text{def}}{=} \{\{P, \neg R, \neg P\},\ \{Q, \neg R\},\ \{\neg Q\}\}$. Un exemple de recherche par simplification est donné dans la figure 9.

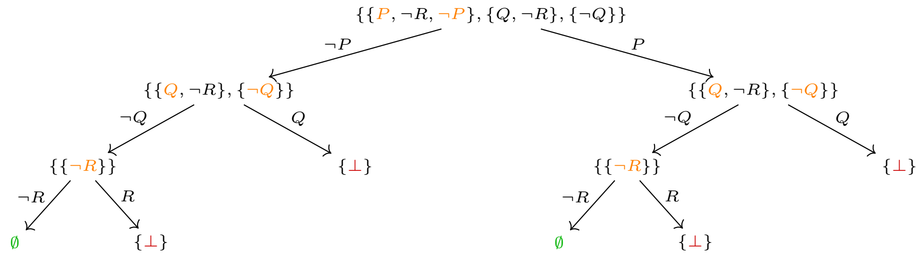


FIGURE 9. Un arbre de recherche par simplification pour l'exemple 7.2.

Cette recherche réussit, puisqu'il existe au moins une feuille étiquetée par l'ensemble vide de clauses, par exemple la feuille atteinte par le chemin $P, \neg Q, \neg R$; et en effet, toute interprétation qui étend $[1/P, 0/Q, 0/R]$ est un modèle de F.

Voici un pseudo-code pour un algorithme récursif de recherche par simplification.

Fonction `satisfiable`(F)
1 **si** $F = \emptyset$ **alors retourner** 1
2 **si** $\bot \in F$ **alors retourner** 0
3 **choisir** $P \in \mathrm{fp}(F)$
4 **retourner**
`satisfiable(simplifie(`F, P`))` or `satisfiable(simplifie(`$F, \neg P$`))`

Remarque 7.3. À noter dans ce pseudo-code que le choix de la proposition à utiliser pour simplifier à la ligne 3 n'a pas besoin d'être le même le long de toutes les branches. Par exemple, la figure 10

montre une recherche par simplification pour l'exemple 7.1 qui échoue plus rapidement que celle de la figure 8.

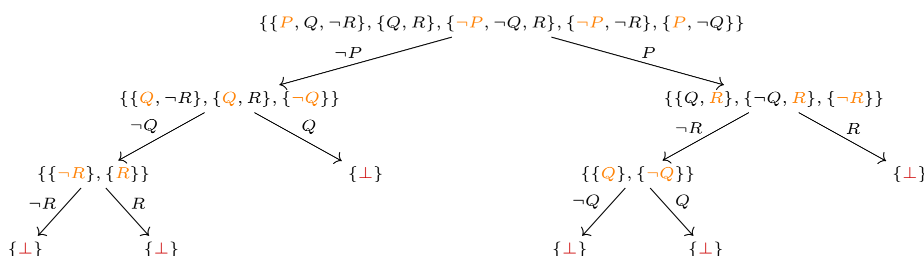


FIGURE 10. Un autre arbre de recherche par simplification pour l'exemple 7.1.

Exemple 7.4. Considérons la formule sous forme clausale

$$\{\{P,Q,R\},\{\neg P,Q,R\},\{R\},\{P,\neg Q,\neg R\},\{\neg P,\neg Q,\neg R\},\{\neg R\}\}\ .$$

Si on effectue les simplifications sur P puis Q puis R, l'arbre de recherche obtenu est celui de la figure 11, qui est aussi grand que celui d'une recherche par énumération exhaustive.

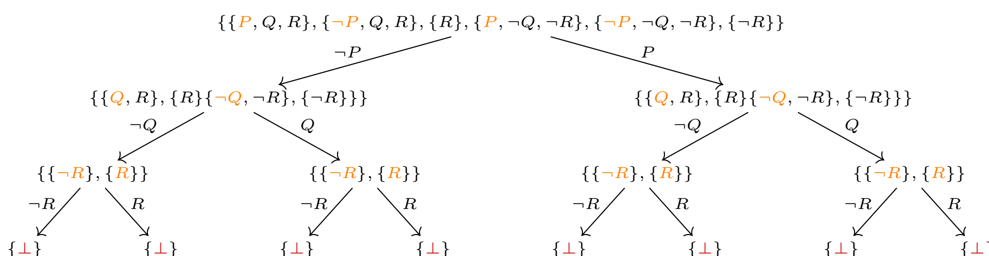


FIGURE 11. Un arbre de recherche par simplification pour l'exemple 7.4.

Il est pourtant possible de faire une recherche par simplification beaucoup plus efficace, en commençant par la proposition R, comme illustré dans la figure 12. En général, il peut être avantageux de simplifier en priorité sur des propositions qui apparaissent dans beaucoup de clauses.

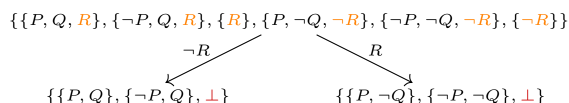


FIGURE 12. Un autre arbre de recherche par simplification pour l'exemple 7.4.

7.2.3. *Correction et complétude.* La recherche de modèle par simplification peut se comprendre par le biais de *règles de réécriture* qui agissent sur des ensembles de clauses. Les règles (split_P) et ($\text{split}_{\neg P}$) de la figure 13 cherchent à montrer qu'un ensemble fini F est satisfiable en le réduisant à l'ensemble vide de clauses. Les arbres des figures 8 à 12 recensent des réécritures possibles par ce système de règles.

Il est aisé de voir que F est satisfiable si et seulement si $F \to^*_{\text{spl}} \emptyset$ à l'aide des règles (split_P) et ($\text{split}_{\neg P}$) pour $P \in \text{fp}(F)$. En effet, notons $I[1/\ell]$ pour l'interprétation qui associe 1 à P si

$$F \to_{\text{spl}} F[\top/P] \qquad \text{où } P \in \text{fp}(F) \qquad (\text{split}_P)$$
$$F \to_{\text{spl}} F[\top/\neg P] \qquad \text{où } P \in \text{fp}(F) \qquad (\text{split}_{\neg P})$$

FIGURE 13. Règles de réécriture de la recherche par simplification.

$\ell = P$, 0 à P si $\ell = \neg P$, et Q^I à toute proposition $Q \notin \text{fp}(\ell)$. On a alors la conséquence suivante du lemme 4.7 de substitution propositionnelle

Propriété 7.5. *Pour toute ensemble S de clauses, toute interprétation I et tout littéral ℓ, $I \vDash S[\top/\ell]$ si et seulement si $I[1/\ell] \vDash S$.*

Une autre remarque importante est que la simplification par un littéral P ou $\neg P$ conduit à un ensemble de clauses où ni P ni $\neg P$ n'apparaît.

Propriété 7.6. *Soit S un ensemble de clauses et P une proposition. Alors $P \notin \text{fp}(S[\top/P])$ et $P \notin \text{fp}(S[\top/\neg P])$.*

Comme les règles de réécriture de la figure 13 n'appliquent la simplification qu'à une proposition $P \in \text{fp}(F)$, on a que $F \to_{\text{spl}} F'$ implique $\text{fp}(F) \supsetneq \text{fp}(F')$, donc on ne peut appliquer les règles qu'au plus $|\text{fp}(F)|$ fois à un ensemble F donné : on dit que ce système de règles de réécriture *termine*.

On en déduit le théorème suivant des propriétés 7.5 et 7.6.

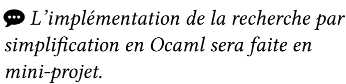
Théorème 7.7. *Soit F un ensemble fini de clauses. Alors les règles de la figure 13 sont correctes et complètes : F est satisfiable si et seulement si $F \to^*_{\text{spl}} \emptyset$.*

Démonstration. Pour la correction, c'est-à-dire pour montrer que $F \to^*_{\text{spl}} \emptyset$ implique F satisfiable, il suffit d'observer d'une part que l'ensemble vide de clauses $\emptyset$ est satisfiable (il est même valide), et d'autre part que si F' est satisfiable – disons par une interprétation I telle que $I \vDash F'$ – et $F \to_{\text{spl}} F'$ par une des règles de la figure 13, alors F est satisfiable :
- pour (split_P) : alors $F' = F[\top/P]$ pour la proposition P : on a $I[1/P] \vDash F$ par la propriété 7.5 ;
- pour $(\text{split}_{\neg P})$: alors $F' = F[\top/\neg P]$ pour la proposition P : on a $I[0/P] \vDash F$ par la propriété 7.5.

Une simple récurrence sur le nombre de réécritures dans $F \to^*_{\text{spl}} \emptyset$ démontre alors la correction.

Pour la complétude, c'est-à-dire pour montrer que F satisfiable implique $F \to^*_{\text{spl}} \emptyset$, supposons que $I \vDash F$. On ordonne $\text{fp}(F)$ de manière arbitraire comme $P_1 < \cdots < P_n$. Soit $F_0 \stackrel{\text{def}}{=} F$; on applique pour chaque $1 \leq i \leq n$ à la proposition P_i sur l'ensemble F_{i-1}
- soit (split_{P_i}) si $I \vDash P_i$ et alors $F_i \stackrel{\text{def}}{=} F_{i-1}[\top/P_i]$ et comme $I = I[1/P_i]$, $I \vDash F_i$ par la propriété 7.5 ;
- soit $(\text{split}_{\neg P_i})$ si $I \vDash \neg P_i$ et alors $F_i \stackrel{\text{def}}{=} F_{i-1}[\top/\neg P_i]$ et comme $I = I[1/\neg P_i]$, $I \vDash F_i$ par la propriété 7.5.

Comme $\text{fp}(F_i) = \{P_{i+1}, \ldots, P_n\}$ pour tout $0 \leq i \leq n$ par la propriété 7.6, F_n est un ensemble de clauses sans propositions. De plus, il ne peut pas contenir la clause vide puisque $I \vDash F_n$. Donc $F_n = \emptyset$. □


L'implémentation de la recherche par simplification en Ocaml sera faite en mini-projet.

7.2.4. *Implémentation de la recherche par simplification en Java.* Comme dans la section 7.1.1, nous utilisons des représentations proches de celles utilisées en pratique par les solveurs SAT, à savoir qu'un littéral est un entier non nul et une interprétation est un tableau d'entiers. Voici une implémentation possible de la simplification d'un ensemble de clauses, d'abord dans notre classe `Clause` et ensuite au niveau de notre solveur par simplification.

Clause
```
public boolean simplifie(int l) {
    if (this.contains(Integer.valueOf(l)))
        return true;
    this.remove(Integer.valueOf(-l));
    return false;
}
```

SolveurSplit
```
private SolveurSplit simplifie(int l) {
    interpretation[index(l)] = l;
    clauses.removeIf(c -> c.simplifie(l));
    return this;
}
```

Voici maintenant une implémentation de la recherche de modèle par simplification en Java.

SolveurSplit
```
private boolean satisfiable(int i) {
    // l'ensemble vide de clauses est satisfiable
    if (clauses.size() == 0) {
        // interprétation arbitraire des propositions restantes
        for (int j = i; j < nprops; j++)
            interpretation[j] = j+1;
        return true;
    }
    // un clause vide est insatisfiable
    if (clauses.stream().anyMatch(c -> c.size() == 0))
        return false;
    // branchement :
    SolveurSplit clone = this.clone();
    //  - tente de simplifier par le littéral P⊠
    if (this.simplifie(i+1).satisfiable(i+1))
        return true;
    // - restaure l'état du solveur
    this.interpretation = clone.interpretation;
    this.clauses = clone.clauses;
    //  - tente de simplifier par le littéral ¬P⊠
    return this.simplifie(-i-1).satisfiable(i+1);
}
public boolean satisfiable() {
    return satisfiable(0);
}
```

Bien qu'elle ne cherche pas à optimiser l'ordre des littéraux sur lesquels simplifier (voir la remarque 7.3 et l'exemple 7.4), cette implémentation résout les problèmes de la section 6 en moins d'une seconde sur ma machine. Elle n'est cependant pas capable de résoudre des problème de taille plus importante, de l'ordre de plusieurs dizaines de propositions et de clauses.

7.3. **Algorithme de Davis, Putnam, Logemann et Loveland.** Couramment appelé *DPLL* d'après ses inventeurs, cet algorithme sert d'inspiration aux solveurs SAT actuels. L'algorithme DPLL est un raffinement de la recherche de modèle par simplification de la section 7.2.

(Conchon et Simon, 2018, sec. 2.2.3), (Goubault-Larrecq et Mackie, 1997, sec. 2.4.3), (Harrisson, 2009, sec. 2.9)

L'idée de départ de l'algorithme DPLL provient de la remarque 7.3 : comment choisir les littéraux par lesquels simplifier l'ensemble de clauses ? Comme vu dans l'exemple 7.4, un mauvais choix va résulter en un arbre de recherche très grand, potentiellement aussi grand que l'arbre de recherche par énumération naïve de la section 7.1 ; à l'inverse, un bon choix peut mener beaucoup plus rapidement à la réponse. L'algorithme DPLL fournit des heuristiques pratiques pour choisir

ces littéraux. Voici deux telles heuristiques, qui identifient les littéraux *unitaires* et les littéraux *purs* de l'ensemble de clauses.

Littéraux unitaires. Dans un ensemble S de clauses, une clause C est *unitaire* si elle ne contient qu'un seul littéral, et on dit alors que ce littéral est *unitaire*. Formellement, ℓ est unitaire dans S si $S = S' \cup \{\{\ell\}\}$ pour un ensemble S'. Dans tout modèle I de S, c'est-à-dire si $I \vDash S$, un littéral unitaire ℓ sera nécessairement satisfait par $I : I \vDash \ell$. C'est donc un bon choix pour une simplification.

Par exemple, dans l'ensemble de clauses de l'exemple 7.4, les littéraux R et $\neg R$ sont unitaires.

Littéraux purs. On définit l'ensemble des littéraux *purs* d'un ensemble S de clauses comme l'ensemble des littéraux ℓ tels que $\overline{\ell}$ n'apparaît dans aucune clause de S :

$$\mathrm{Pur}(S) \stackrel{\text{def}}{=} \{\ell \mid \forall C \in S \,.\, \overline{\ell} \notin C\} \,.$$

Dans tout modèle I de S, c'est-à-dire si $I \vDash S$, si ℓ est un littéral pur de S, alors $I[1/\ell]$ est encore un modèle de S. C'est encore une fois un bon choix pour une simplification.

Exemple 7.8. Soit l'ensemble de clauses

$$F \stackrel{\text{def}}{=} \{\{P, \neg Q, R\}, \{P, \neg R\}, \{Q, R\}, \{P, \neg Q\}\} \,.$$

Alors $\mathrm{Pur}(F) = \{P\}$.

7.3.1. *Correction et complétude.* L'algorithme DPLL peut être vu comme une extension du système de règles de la figure 13, qui introduit deux nouvelles règles (unit) et (pure). Tout comme lors de la recherche de modèle par simplification, les règles de la figure 14 cherchent à montrer qu'un ensemble fini F de clauses est satisfiable en le réduisant à l'ensemble vide.

$$\begin{array}{rlll}
F \cup \{\{\ell\}\} \to_{\text{dpll}} & F[\top/\ell] & & \text{(unit)} \\
F \to_{\text{dpll}} & F[\top/\ell] & \text{où } \ell \in \mathrm{Pur}(F) & \text{(pure)} \\
F \to_{\text{dpll}} & F[\top/P] & \text{où } P \in \mathrm{fp}(F) & (\text{split}_P) \\
F \to_{\text{dpll}} & F[\top/\neg P] & \text{où } P \in \mathrm{fp}(F) & (\text{split}_{\neg P})
\end{array}$$

Figure 14. Règles de réécriture de DPLL.

On montre que ce système de règles reste correct ; la complétude découle directement de celle du système de recherche de modèle par simplification.

Théorème 7.9. *Soit F un ensemble fini de clauses. Alors les règles de la figure 14 sont correctes et complètes : F est satisfiable si et seulement si $F \to^*_{\text{dpll}} \emptyset$.*

Démonstration. Pour la correction, puisque la recherche de modèle par simplification était correcte (voir le théorème 7.7), il suffit de montrer que, si F' est satisfiable – disons par une interprétation I telle que $I \vDash F'$ – et $F \to_{\text{dpll}} F'$ par (unit) et (pure), alors F est satisfiable :

– pour (unit) : alors $F' = F''[\top/\ell]$ et $F = F'' \cup \{\{\ell\}\}$ pour un littéral unitaire ℓ de F'' : on a $I[1/\ell] \vDash F''$ par la propriété 7.5 et donc $I[1/\ell] \vDash F$;
– pour (pure) : alors $F' = F[\top/\ell]$ pour un littéral pur ℓ : on a $I[1/\ell] \vDash F$ par la propriété 7.5.

Pour la complétude, c'est-à-dire pour montrer que F satisfiable implique $F \to^*_{\text{dpll}} \emptyset$, cela découle du théorème 7.7. En effet, si F est satisfiable, alors $F \to^*_{\text{spl}} \emptyset$, c'est-à-dire $F \to^*_{\text{dpll}} \emptyset$ en n'utilisant que les règles (split_P) et ($\text{split}_{\neg P}$). □

7.3.2. *Algorithme DPLL.* L'intérêt des règles (unit) et (pure) est qu'elles sont *inversibles*, au sens suivant.

Proposition 7.10 (inversibilité)**.** *Soit F un ensemble de clauses. Si $F \xrightarrow{\text{(unit)}}_{\text{dpll}} F'$ ou $F \xrightarrow{\text{(pure)}}_{\text{dpll}} F'$ pour un ensemble de clauses F satisfiable, alors F' est aussi satisfiable.*

Démonstration. Supposons que I soit une interprétation telle que $I \vDash F$.

- Pour (unit) : alors $F = F \cup \{\{\ell\}\}$ et en particulier $I \vDash \ell$. Donc $I[1/\ell] = I \vDash F$ et par la propriété 7.5, $I \vDash F[\top/\ell] = F'$.
- Pour (pure) : alors $F' = F[\top/\ell]$ où ℓ est un littéral dans $\text{Pur}(F)$. Si $I \vDash \ell$, alors $I[1/\ell] = I \vDash F$ et par la propriété 7.5 $I \vDash F[\top/\ell] = F'$. Inversement, si $I \nvDash \ell$, alors comme $I \vDash F$, dans toutes les clauses $C \in F$, il existe un littéral $\ell_C \in C$ tel que $I \vDash \ell_C$ et forcément $\ell_C \neq \ell$ pour toutes les clauses $C \in F$. Mais alors $I[1/\ell] \vDash \ell_C$ pour toutes les clauses $C \in F$, et donc $I[1/\ell] \vDash F$. Par la propriété 7.5, $I \vDash F[\top/\ell] = F'$. □

L'inversibilité et la correction des règles (unit) et (pure) signifie que, si $F \xrightarrow{\text{(unit)}}_{\text{dpll}} F'$ ou $F \xrightarrow{\text{(pure)}}_{\text{dpll}} F'$, alors F est satisfiable si et seulement si F' l'est. Autrement dit, il n'est pas utile d'introduire des *points de choix* comme lors de l'utilisation des règles (split_P) et ($\text{split}_{\neg P}$), pour revenir en arrière s'il s'avérait que F' était insatisfiable. Les règles (unit) et (pure) peuvent donc être appliquées arbitrairement. La stratégie usuelle de l'algorithme DPLL est de les appliquer en priorité (et dans cet ordre) avant d'essayer (split_P) ou ($\text{split}_{\neg P}$), de manière à accélérer la recherche de preuve en éliminant des points de choix. Voici le pseudo-code de l'algorithme écrit en style récursif.

L'algorithme DPLL est implémenté sous forme itérative en utilisant du backtracking. *Les performances des solveurs* SAT *actuels tiennent à une gestion fine des retours aux points de choix, et lors de ces retours à l'ajout de nouvelles clauses inférées à partir de la branche d'échec, ceci afin de guider la recherche vers une branche de succès ; cette technique est appelée* « conflict-driven clause learning ». *Voir (Conchon et Simon, 2018, sec. 2.2) et (Zhang et al., 2001).*

Fonction `satisfiable`(F)
1. **si** $F = \emptyset$ **alors retourner** 1
2. **si** $\bot \in F$ **alors retourner** 0
3. **si** $\exists \ell \,.\, F = F \cup \{\{\ell\}\}$ **alors retourner** `satisfiable`(`simplifie`(F, ℓ))
4. **si** $\exists \ell \in \text{Pur}(F)$ **alors retourner** `satisfiable`(`simplifie`(F, ℓ))
5. **choisir** $P \in \text{fp}(F)$
6. **retourner** `satisfiable`(`simplifie`(F, P)) or `satisfiable`(`simplifie`$(F, \neg P)$)

Comme dans le cas de la recherche par simplification, les réécritures du système de la figure 14 sont de longueur au plus $|\text{fp}(F)|$.

Reprenons les différents exemples de cette section. Pour l'exemple de la figure 10, l'algorithme DPLL construit (entre autres) l'arbre de recherche de la figure 15. Pour l'exemple 7.2, l'arbre de recherche DPLL est celui de la figure 16. Pour l'exemple des figures 11 et 12, l'algorithme DPLL construit (entre autres) l'arbre de recherche de la figure 17. Pour l'exemple 7.8, l'algorithme DPLL construit l'arbre de recherche de la figure 18.

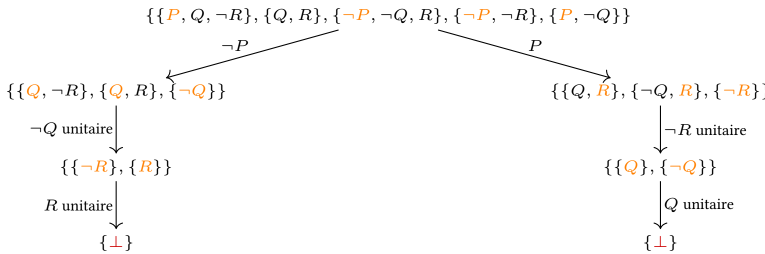


Figure 15. Un arbre de recherche DPLL pour l'exemple 7.1.

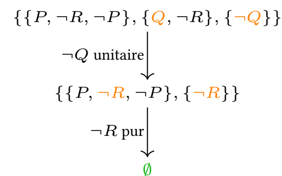


FIGURE 16. L'arbre de recherche DPLL pour l'exemple 7.2.

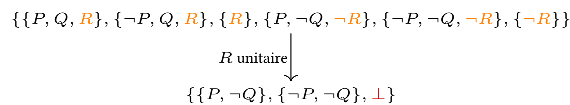


FIGURE 17. Un arbre de recherche DPLL pour l'exemple 7.4.

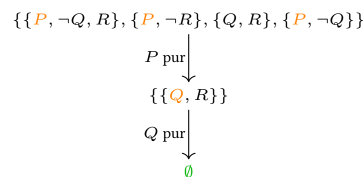


FIGURE 18. L'arbre de recherche DPLL pour l'exemple 7.8.

💬 *L'implémentation d'un algorithme DPLL récursif en Ocaml sera faite en mini-projet.*

7.3.3. *Implémentation d'un DPLL récursif en Java.* Voici enfin une implémentation assez naïve de l'algorithme DPLL en Java sous la forme d'une méthode récursive.

DPLLRec

```java
private DPLLRec simplifie(int l) {
    interpretation[index(l)] = l;
    clauses.removeIf(c -> c.simplifie(l));
    return this;
}
public boolean satisfiable() {
    // l'ensemble vide de clauses est satisfiable
    if (clauses.size() == 0)
        return true;
    // un clause vide est insatisfiable
    if (clauses.stream().anyMatch(c -> c.size() == 0))
        return false;
    // clause unitaire
    Optional<Clause> unitaire
        = clauses.stream().filter(c -> c.size() == 1).findAny();
    if (unitaire.isPresent()) {
        int l = unitaire.get().stream().findAny().get().intValue();
        return simplifie(l).satisfiable();
    }
    // littéral pur
    for (int i = 0; i < nprops; i++)
        if (interpretation[i] == 0) {
            final Integer l = Integer.valueOf(i+1);
            final Integer notl = Integer.valueOf(-i-1);
            if (clauses.stream().noneMatch(c ->
```

```
                    c.stream().anyMatch(j -> j.equals(notl))))
                return simplifie(l.intValue()).satisfiable();
            if (clauses.stream().noneMatch(c ->
                    c.stream().anyMatch(j -> j.equals(l))))
            return simplifie(notl.intValue()).satisfiable();
        }
    // branchement
    for (int i = 0; i <nprops; i++)
        if (interpretation[i] == 0) {
            DPLLRec clone = this.clone();
            //  - tente de simplifier par le littéral Pi
            if (this.simplifie(i+1).satisfiable())
                return true;
            // - restaure l'état du solveur
            this.interpretation = clone.interpretation;
            this.clauses = clone.clauses;
            //  - tente de simplifier par le littéral ¬Pi
            return this.simplifie(-i-1).satisfiable();
        }
    assert false;
    return false;
}
```

Comme précédemment, cette implémentation ne cherche pas à optimiser le choix des littéraux lors des branchements. Cette implémentation arrive à résoudre des problèmes de taille modeste, autour de quelques centaines de propositions et de clauses, qui sont hors de portée des implémentations à base de recherche par énumération ou par simplification.

8. Validité et recherche de preuve

Résumé. Le *calcul des séquents propositionnel* est un système de déduction qui manipule des séquents $\vdash \Gamma$, où Γ est un multi-ensemble fini de formules propositionnelles sous forme normale négative. Un séquent $\vdash \Gamma$ pour lequel il existe une dérivation dans le système de preuve est dit *prouvable*, ce qui est noté « $\vdash_{\mathbf{LK}_0} \Gamma$ ».

On peut implémenter la *recherche de preuve* dans le calcul des séquents propositionnel : cela tient à la propriété 8.7 de branches linéaires, et de plus on peut faire une implémentation sans retours sur erreurs grâce au lemme 8.8 d'inversibilité syntaxique.

Un séquent $\vdash \Gamma$ est *valide*, si pour toute interprétation I, il existe une formule propositionnelle $\vartheta \in \text{dom}(\Gamma)$ du séquent telle que $I \vDash \vartheta$. Par le théorème 8.10 de *correction*, si $\vdash \Gamma$ est prouvable, alors il est valide. Inversement, par le théorème 8.13 de *complétude*, si $\vdash \Gamma$ est valide, alors il est prouvable.

Le problème de *validité* est le problème de décision suivant.

Problème (VALIDITÉ).
instance : une formule propositionnelle φ
question : φ est-elle valide ?

Une façon de résoudre ce problème nous est offerte par la dualité entre satisfiabilité et validité (voir la propriété 3.11) : φ est valide si et seulement si $\neg\varphi$ n'est pas satisfiable. On peut donc résoudre VALIDITÉ à l'aide d'un solveur SAT.

Cependant, on souhaiterait aussi, dans le cas où φ est valide, avoir un *certificat* de validité, qui « explique » pourquoi la formule propositionnelle est valide ; on voudrait de plus pouvoir aisément vérifier, quand on dispose d'un tel certificat, que la formule propositionnelle était bien valide.

Dans le cas de SAT*, un certificat de satisfiabilité était simplement un modèle I de la formule, qui est de taille $|\text{fp}(\varphi)|$ et $I \vDash \varphi$ peut être vérifié en temps polynomial.*

Certificats d'insatisfiabilité. Or, dans les algorithmes que nous avons vus dans la section 7.3, dans le cas d'une formule propositionnelle insatisfiable, aucune information n'est retournée ; par exemple, MiniSat retourne seulement « UNSAT ». Retourner l'arbre de recherche entier du solveur SAT fournirait un certificat facile à vérifier, mais potentiellement beaucoup trop gros et de plus dépendant des heuristiques utilisées par chaque solveur.

Les solveurs SAT plus récents comme Glucose [10] peuvent fournir des certificats sous des formats standard. Le principe de base est le suivant : soit F un ensemble de clauses insatisfiable. Un certificat est alors un ensemble F' de clauses tel que

Voir par exemple Goldberg et Novikov (2003) et Wetzler, Heule et Hunt (2014) pour les certificats de solveurs SAT.

(1) toutes les clauses $C' \in F'$ sont des conséquences logiques de F, c'est-à-dire $F \cup \{\{\overline{\ell}\}\} \vDash \bot$ pour tout littéral $\ell \in C'$, et
(2) $F \cup F' \vDash \bot$.

L'espoir ici est que d'une part cet ensemble F' soit de petite taille et d'autre part que les tests d'insatisfiabilité $F \cup \{\{\overline{\ell}\}\} \vDash \bot$ et $F \cup F' \vDash \bot$ soient aisés, par exemple en n'utilisant que des règles unitaires (unit).

Systèmes de preuve. L'approche alternative que nous allons explorer dans cette section est plutôt de résoudre le problème de validité directement, à l'aide d'un *système de preuve*, c'est-à-dire de règles de déduction qui garantissent la validité. L'intérêt ici est que la preuve elle-même, c'est-à-dire l'arbre de dérivation dans le système de preuve, fait office de certificat. Un tel certificat est très facile à vérifier ; en revanche, il peut être de taille exponentielle en la taille de φ.

On soupçonne en fait que des certificats de taille polynomiale et vérifiables en temps polynomial n'existent pas pour VALIDITÉ*, car cela impliquerait* NP = coNP*, comme vous le verrez en cours de « calculabilité et complexité » en M1. Cette question de la « complexité des preuves » a été initiée par Cook et Reckhow.*

Voir (Duparc, 2015, sec. 3.5) ou (Goubault-Larrecq et Mackie, 1997, sec. 2.3.3) pour un calcul des séquents bilatère avec coupure ; d'autres familles de systèmes de preuve pour la logique classique propositionnelle sont les systèmes à la Hilbert *(ibid., sec. 2.3.1), la* déduction naturelle *(ibid., sec. 2.3.2), la* résolution propositionnelle *(ibid., sec. 2.4.2) et les* systèmes de tableaux *(ibid., sec. 2.4.1).*

8.1. Calcul des séquents propositionnel.

Le système de preuve que nous allons étudier dans ce cours est un système de calcul des séquents. Il y a quantité de variantes du calcul des séquents

10. `https://www.labri.fr/perso/lsimon/glucose/`

pour la logique classique propositionnelle. Le système que nous étudions ici est un calcul monolatère inversible sans coupure, qui a l'avantage de ne comporter que peu de règles et de se prêter particulièrement bien à la *recherche de preuve*.

Un *multi-ensemble* fini sur un ensemble S est formellement une fonction $m\colon S \to \mathbb{N}$ de domaine $\mathrm{dom}(m) \stackrel{\text{def}}{=} \{e \in S \mid m(e) > 0\}$ fini ; on peut aussi voir m comme une séquence finie de S^* modulo permutation.

Dans le calcul de la figure 19, un *séquent* est un multi-ensemble fini Γ de formules en forme normale négative, et est noté $\vdash \Gamma$. La virgule dénote l'union de multi-ensembles : par exemple, Γ, Δ, φ dénote le multi-ensemble avec $\Gamma(\varphi) + \Delta(\varphi) + 1$ occurrences de la formule propositionnelle φ, et $\Gamma(\psi) + \Delta(\psi)$ occurrences de $\psi \neq \varphi$. On note le séquent vide $\vdash \bot$, où $\bot(\varphi) \stackrel{\text{def}}{=} 0$ pour toute formule φ.

$$\dfrac{}{\vdash \Gamma, P, \neg P}\,\text{(ax)} \qquad \dfrac{\vdash \Gamma, \varphi \qquad \vdash \Gamma, \psi}{\vdash \Gamma, \varphi \wedge \psi}\,(\wedge) \qquad \dfrac{\vdash \Gamma, \varphi, \psi}{\vdash \Gamma, \varphi \vee \psi}\,(\vee)$$

FIGURE 19. Calcul des séquents propositionnel monolatère.

*Ce calcul ne contient pas ni la règle structurelle d'*échange*, qui est implicite parce que nous travaillons avec des multi-ensembles, ni la règle structurelle d'*affaiblissement*, qui est implicite dans la règle d'axiome* (ax)*, ni la règle structurelle de* contraction *ni la règle de* coupure*, mais toutes celles-ci sont admissibles.*

Une règle du calcul des séquents permet de déduire un séquent *conclusion* d'un nombre fini de séquents *prémisses*. Chaque règle comprend une *formule principale* dans sa conclusion, indiquée en orange dans les règles de la figure 19. Un séquent $\vdash \Gamma$ est *prouvable*, noté $\vdash_{\mathbf{LK}_0} \Gamma$, s'il en existe une dérivation dans le système de la figure 19.

Exemple 8.1. La loi de PEIRCE de l'exemple 5.2 est prouvable. La dérivation correspondante (avec la formule principale indiquée en orange à chaque étape) est :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg P, Q, P}\,\text{(ax)}}{\vdash \neg P \vee Q, P}\,(\vee) \qquad \dfrac{}{\vdash \neg P, P}\,\text{(ax)}}{\vdash (\neg P \vee Q) \wedge \neg P, P}\,(\wedge)}{\vdash ((\neg P \vee Q) \wedge \neg P) \vee P}\,(\vee)$$

Exemple 8.2. Considérons la formule propositionnelle $(P \wedge P) \Leftrightarrow P$. Sa forme normale négative est $(\neg P \vee \neg P \vee P) \wedge (\neg P \vee (P \wedge P))$. Cette formule est prouvable ; une dérivation est :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg P, \neg P, P}\,\text{(ax)}}{\vdash \neg P, \neg P \vee P}\,(\vee)}{\vdash \neg P \vee \neg P \vee P}\,(\vee) \qquad \dfrac{\dfrac{\dfrac{}{\vdash \neg P, P}\,\text{(ax)} \qquad \dfrac{}{\vdash \neg P, P}\,\text{(ax)}}{\vdash \neg P, P \wedge P}\,(\wedge)}{\vdash \neg P \vee (P \wedge P)}\,(\vee)}{\vdash (\neg P \vee \neg P \vee P) \wedge (\neg P \vee (P \wedge P))}\,(\wedge)$$

Exemple 8.3. Considérons la formule propositionnelle $(P \wedge (Q \vee R) \Rightarrow ((P \wedge Q) \vee (P \wedge R))$. Sa forme normale négative est $\neg P \vee (\neg Q \wedge \neg R) \vee (P \wedge Q) \vee (P \wedge R)$. Cette formule est prouvable ; une dérivation est :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg P, \neg Q \wedge \neg R, P, P \wedge R}\,\text{(ax)} \qquad \dfrac{\dfrac{}{\vdash \neg P, \neg Q \wedge \neg R, Q, P}\,\text{(ax)} \qquad \dfrac{\dfrac{}{\vdash \neg P, \neg Q, Q, R}\,\text{(ax)} \qquad \dfrac{}{\vdash \neg P, \neg R, Q, R}\,\text{(ax)}}{\vdash \neg P, \neg Q \wedge \neg R, Q, R}\,(\wedge)}{\vdash \neg P, \neg Q \wedge \neg R, Q, P \wedge R}\,(\wedge)}{\vdash \neg P, \neg Q \wedge \neg R, P \wedge Q, P \wedge R}\,(\wedge)}{\vdash \neg P, \neg Q \wedge \neg R, (P \wedge Q) \vee (P \wedge R)}\,(\vee)}{\vdash \neg P, (\neg Q \wedge \neg R) \vee (P \wedge Q) \vee (P \wedge R)}\,(\vee)}{\vdash \neg P \vee (\neg Q \wedge \neg R) \vee (P \wedge Q) \vee (P \wedge R)}\,(\vee)$$

Exemple 8.4. Considérons les trois formules propositionnelles suivantes :

$$\varphi_A \stackrel{\text{def}}{=} B \Rightarrow D\ , \qquad \varphi_B \stackrel{\text{def}}{=} \neg B \Rightarrow (\neg D \wedge \neg U)\ , \qquad \varphi_C \stackrel{\text{def}}{=} B \vee D \vee U\ .$$

On souhaite montrer que $B \wedge D$ est une conséquence logique de $\varphi_A \wedge \varphi_B \wedge \varphi_C$. Par le lemme 4.2 de déduction, $\varphi_A \wedge \varphi_B \wedge \varphi_C \vDash B \wedge D$ si et seulement si $(\varphi_A \wedge \varphi_B \wedge \varphi_C) \Rightarrow B \wedge D$ est valide, autrement dit si et seulement si sa forme normale négative $\overline{\varphi}_A \vee \overline{\varphi}_B \vee \overline{\varphi}_C \vee (B \wedge D)$ est valide, où

$$\overline{\varphi}_A = B \wedge \neg D\ , \qquad \overline{\varphi}_B = \neg B \wedge (D \vee U)\ , \qquad \overline{\varphi}_C = \neg B \wedge \neg D \wedge \neg U\ .$$

La dérivation ci-dessous montre que $\vdash_{\mathbf{LK}_0} \overline{\varphi}_A \vee \overline{\varphi}_B \vee \overline{\varphi}_C \vee (B \wedge D)$, ce qui implique par le théorème 8.10 de correction que la formule est bien valide et donc que $B \wedge D$ est bien une conséquence logique de $\varphi_A \wedge \varphi_B \wedge \varphi_C$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \overline{\varphi}_A, \neg B, \overline{\varphi}_C, B}\,(\text{ax}) \quad \dfrac{\dfrac{}{\vdash B, \neg B, \overline{\varphi}_C, D}\,(\text{ax}) \quad \dfrac{}{\vdash \neg D, \neg B, \overline{\varphi}_C, D}\,(\text{ax})}{\vdash \overline{\varphi}_A, \neg B, \overline{\varphi}_C, D}\,(\wedge)}{\vdash \overline{\varphi}_A, \neg B, \overline{\varphi}_C, B \wedge D}\,(\wedge) \quad \dfrac{\dfrac{\pi}{\vdash \overline{\varphi}_A, D, U, \overline{\varphi}_C, B \wedge D}}{\vdash \overline{\varphi}_A, D \vee U, \overline{\varphi}_C, B \wedge D}\,(\vee)}{\vdash \overline{\varphi}_A, \overline{\varphi}_B, \overline{\varphi}_C, B \wedge D}\,(\wedge)}{\vdash \overline{\varphi}_A, \overline{\varphi}_B, \overline{\varphi}_C \vee (B \wedge D)}\,(\vee)}{\vdash \overline{\varphi}_A, \overline{\varphi}_B \vee \overline{\varphi}_C \vee (B \wedge D)}\,(\vee)}{\vdash \overline{\varphi}_A \vee \overline{\varphi}_B \vee \overline{\varphi}_C \vee (B \wedge D)}\,(\vee)$$

où la dérivation π de $\vdash \overline{\varphi}_A, D, U, \overline{\varphi}_C, B \wedge D$ est

$$\dfrac{\dfrac{}{\vdash \overline{\varphi}_A, D, U, \neg D, B \wedge D}\,(\text{ax}) \quad \dfrac{\dfrac{}{\vdash \overline{\varphi}_A, D, U, \neg U, B \wedge D}\,(\text{ax}) \quad \dfrac{\dfrac{}{\vdash \overline{\varphi}_A, D, U, \neg B, B}\,(\text{ax}) \quad \dfrac{\pi'}{\vdash \overline{\varphi}_A, D, U, \neg B, D}}{\vdash \overline{\varphi}_A, D, U, \neg B, B \wedge D}\,(\wedge)}{\vdash \overline{\varphi}_A, D, U, \neg U \wedge \neg B, B \wedge D}\,(\wedge)}{\vdash \overline{\varphi}_A, D, U, \overline{\varphi}_C, B \wedge D}\,(\wedge)$$

et la dérivation π' de $\vdash \overline{\varphi}_A, D, U, \neg B, D$ est

$$\dfrac{\dfrac{}{\vdash B, D, U, \neg B, D}\,(\text{ax}) \quad \dfrac{}{\vdash \neg D, D, U, \neg B, D}\,(\text{ax})}{\vdash \overline{\varphi}_A, D, U, \neg B, D}\,(\wedge)$$

8.2. **Recherche de preuve.** La recherche de preuve en calcul des séquents propositionnel vise à répondre au problème de décision suivant.

Problème (PROUVABILITÉ).

instance : un séquent propositionnel $\vdash \Gamma$

question : $\vdash \Gamma$ est-il prouvable dans le calcul des séquents propositionnel ?

Comme pour le problème de satisfiabilité, on peut aussi retourner une preuve de $\vdash_{\mathbf{LK}_0} \varphi$ quand la réponse est positive. Comme nous le verrons dans la section 8.3, le calcul des séquents est correct et complet : en particulier, pour une formule propositionnelle φ en forme normale négative, $\vdash_{\mathbf{LK}_0} \varphi$ si et seulement si φ est valide. Résoudre PROUVABILITÉ revient donc à résoudre VALIDITÉ.

Recherche de preuve. La recherche de preuve tente de développer un arbre de dérivation avec $\vdash \Gamma$ pour conclusion, de la racine vers les feuilles. Une *branche* de recherche de preuve est une séquence potentiellement infinie de séquents $\vdash \Gamma = \vdash \Gamma_0, \vdash \Gamma_1, \ldots$ calculée par la recherche de preuve : pour tout $i > 0, \vdash \Gamma_i$ est une prémisse d'une règle dont $\vdash \Gamma_{i-1}$ est la conclusion. Une *branche d'échec* est une branche de recherche de preuve finie $\vdash \Gamma_0, \vdash \Gamma_1, \ldots, \vdash \Gamma_n$ où aucune règle ne s'applique à $\vdash \Gamma_n$, c'est-à-dire qu'il n'est le séquent conclusion d'aucune règle.

Points de choix. La recherche de preuve n'est pas déterministe : il peut y avoir plusieurs règles applicables pour un même séquent $\vdash \Gamma$. Par exemple, pour $\vdash \neg P \vee Q, R \vee P$, on peut commencer par sélectionner $\neg P \vee Q$ comme formule principale et appliquer $(\vee)$:

$$\dfrac{\vdash \neg P, Q, R \vee P}{\vdash \neg P \vee Q, R \vee P}\ (\vee)$$

Mais on aurait aussi pu sélectionner $R \vee P$ comme formule principale et appliquer $(\vee)$:

$$\dfrac{\vdash \neg P \vee Q, R, P}{\vdash \neg P \vee Q, R \vee P}\ (\vee)$$

En général – mais comme nous allons le voir, pas dans le calcul des séquents de la figure 19 – il est nécessaire pour dans un algorithme déterministe de recherche de preuve de mémoriser de tels *points de choix* pour pouvoir y revenir par *backtracking* en cas d'échec.

Exemple 8.5. La formule propositionnelle $\varphi_{\text{ex}} = (P \vee \neg Q) \wedge P$ de la figure 1 n'est pas prouvable : il n'y a aucun point de choix dans la recherche de preuve qui échoue ci-dessous :

$$\dfrac{\dfrac{\overset{\textbf{échec}}{\vdash P, \neg Q}}{\vdash P \vee \neg Q}\ (\vee) \qquad \overset{\textbf{échec}}{\vdash P}}{\vdash (P \vee \neg Q) \wedge P}\ (\wedge)$$

Les deux séquents $\vdash P, \neg Q$ et $\vdash P$ sont des ***séquents d'échec***, c'est-à-dire des séquents sur lequel aucune règle ne s'applique. Les deux branches de recherche de preuve sont des *branches d'échec*, car elles terminent sur des séquents d'échec.

Exemple 8.6. La formule propositionnelle $(P \vee \neg Q) \vee (\neg P \wedge R)$ n'est pas prouvable. En effet, une recherche de preuve pour cette formule commence nécessairement par appliquer $(\vee)$:

$$\dfrac{\vdash P \vee Q, \neg P \wedge \neg R}{\vdash (P \vee \neg Q) \vee (\neg P \wedge R)}\ (\vee)$$

Il y a là un point de choix, selon qu'on utilise $P \vee Q$ ou $\neg P \wedge R$ comme formule principale dans le séquent $\vdash P \vee Q, \neg P \wedge \neg R$. Si on fait le premier choix, la recherche de preuve échoue puisqu'une des branches (celle de droite) termine sur le séquent d'échec $\vdash P, \neg Q, R$ et est donc une branche d'échec :

$$\dfrac{\dfrac{\dfrac{}{\vdash P, \neg Q, \neg P}\ (\text{ax}) \qquad \overset{\textbf{échec}}{\vdash P, \neg Q, R}}{\vdash P, \neg Q, \neg P \wedge R}\ (\wedge)}{\vdash P \vee \neg Q, \neg P \wedge R}\ (\vee)$$

Si on fait l'autre choix, celui de considérer $\neg P \wedge R$ comme principale, la recherche de preuve échoue aussi :

$$\dfrac{\dfrac{\dfrac{}{\vdash P, \neg Q, \neg P}\ (\text{ax})}{\vdash P \vee \neg Q, \neg P}\ (\vee) \qquad \dfrac{\overset{\textbf{échec}}{\vdash P, \neg Q, R}}{\vdash P \vee \neg Q, R}\ (\vee)}{\vdash P \vee \neg Q, \neg P \wedge R}\ (\wedge)$$

Comme quel que soit le choix de formule principale, la recherche de preuve échoue, la formule n'est pas prouvable.

Propriété de branches finies. On peut montrer que les branches de recherche de preuve dans le calcul des séquents propositionnel de la figure 19 sont toutes finies, et même linéaires en la taille du séquent $\vdash \Gamma$ que l'on cherche à prouver.

On définit pour cela la *taille* d'un multi-ensemble Γ comme la somme des tailles de ses occurrences de formules $|\Gamma| \stackrel{\text{def}}{=} \sum_{\varphi \in \text{dom}(\Gamma)} |\varphi| \cdot \Gamma(\varphi)$, où la taille $|\varphi|$ d'une formule propositionnelle φ en forme normale négative est définie comme le nombre de nœuds étiquetés par $\wedge$ ou $\vee$ dans son arbre de syntaxe ; formellement :

$$|\ell| \stackrel{\text{def}}{=} 0\ , \qquad\qquad |\varphi \vee \psi| = |\varphi \wedge \psi| \stackrel{\text{def}}{=} 1 + |\varphi| + |\psi|\ .$$

Intuitivement, la taille d'une formule propositionnelle est le nombre de nœuds étiquetés par « $\vee$ » ou « $\wedge$ » de son arbre de syntaxe abstraite, et la taille d'un séquent est la somme des tailles de ses formules.

Propriété 8.7 (branches linéaires)**.** *Soit* $\vdash \Gamma$ *un séquent propositionnel. Alors toutes les branches d'une recherche de preuve pour* $\vdash \Gamma$ *sont de longueur au plus* $|\Gamma|$.

Démonstration. On peut vérifier que pour chacune des règles (ax), ($\vee$) et ($\wedge$), la taille de chaque séquent prémisse est strictement inférieure à celle du séquent conclusion. □

Voici comment la recherche de preuve pourrait s'écrire en pseudo-code, qui pour un séquent courant $\vdash \Gamma$ essaye tour à tour d'utiliser chaque formule $\varphi \in \mathrm{dom}(\Gamma)$ en guise de formule principale.

```
Fonction prouvable(⊢ Γ)
1  v := 0
2  pour tous les φ ∈ dom(Γ) faire
3      si φ = ℓ et ℓ̄ ∈ dom(Γ) alors
4          retourner 1
5      si φ = φ′ ∨ ψ alors
6          Δ := Γ \ φ
7          v := v or prouvable(⊢ Δ, φ′, ψ)
8      si φ = φ′ ∧ ψ alors
9          Δ := Γ \ φ
10         v := v or (prouvable(⊢ Δ, φ′) and prouvable(⊢ Δ, ψ))
11 retourner v
```

Par la propriété 8.7 de branches linéaires, cet algorithme termine bien : la profondeur des appels récursifs est bornée par $|\Gamma|$.

8.2.1. *Inversibilité et algorithme de recherche de preuve.* Nous allons maintenant voir que le choix d'une formule principale n'est pas important dans notre calcul des séquents, et que l'on peut donc considérablement simplifier le pseudo-code de `prouvable`. On peut commencer par observer que, dans nos deux exemples précédents, la recherche de preuve réussit pour nos deux choix de formule principale :

$$\dfrac{\dfrac{\dfrac{}{\vdash \neg P, Q, R, P}\,(\mathsf{ax})}{\vdash \neg P, Q, R \vee P}\,(\vee)}{\vdash \neg P \vee Q, R \vee P}\,(\vee) \qquad\qquad \dfrac{\dfrac{\dfrac{}{\vdash \neg P, Q, R, P}\,(\mathsf{ax})}{\vdash \neg P \vee Q, R, P}\,(\vee)}{\vdash \neg P \vee Q, R \vee P}\,(\vee)$$

Une règle de déduction est *syntaxiquement inversible* si, quand il existe une dérivation de son séquent conclusion, alors il existe une dérivation de chacun de ses séquents prémisses. La règle (ax) est bien sûr syntaxiquement inversible puisqu'elle n'a aucune prémisse. Mais ce qui est plus intéressant, c'est que toutes les autres règles de la figure 19 sont elles aussi syntaxiquement inversibles. Cela signifie que dans une recherche de preuve, on peut appliquer ces règles de manière gloutonne sans faire de *backtracking* – donc sans avoir à mémoriser de points de choix – et que si l'on arrive à un séquent pour lequel aucune règle ne s'applique, alors c'est qu'il n'y avait *pas* de preuve.

Lemme 8.8 (inversibilité syntaxique)**.** *Les règles du calcul des séquents propositionnel sont syntaxiquement inversibles.*

Démonstration. Comme déjà mentionné, la règle (ax) est syntaxiquement inversible par définition puisqu'elle n'a pas de prémisses.

Pour la règle $(\vee)$ **:** supposons qu'il existe une dérivation π du séquent $\vdash \Gamma, \varphi \vee \psi$. On montre par récurrence sur la profondeur de π que $\vdash_{\mathbf{LK_0}} \Gamma, \varphi, \psi$.

- Si π se termine par $(\vee)$ où $\varphi \vee \psi$ est principale, alors évidemment il existe une sous-dérivation de π pour sa prémisse $\vdash \Gamma, \varphi, \psi$.
- Sinon π se termine par une règle (R) où $\varphi \vee \psi$ n'est pas principale. Par inspection des règles, on est nécessairement dans une situation

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots \\ \vdash \Gamma_1, \varphi \vee \psi\end{array} \quad \cdots \quad \begin{array}{c}\pi_k\\ \vdots \\ \vdash \Gamma_k, \varphi \vee \psi\end{array}}{\vdash \Gamma, \varphi \vee \psi}\ \text{(R)}$$

où $0 \leq k \leq 2$ ($k = 0$ correspondant au cas de la règle (ax)). Pour tout $1 \leq i \leq k$, par hypothèse de récurrence sur π_i, il existe des dérivations π'_i de $\vdash \Gamma_i, \varphi, \psi$. On a donc la dérivation

$$\dfrac{\begin{array}{c}\pi'_1\\ \vdots \\ \vdash \Gamma_1, \varphi, \psi\end{array} \quad \cdots \quad \begin{array}{c}\pi'_k\\ \vdots \\ \vdash \Gamma_k, \varphi, \psi\end{array}}{\vdash \Gamma, \varphi, \psi}\ \text{(R)}$$

Pour la règle $(\wedge)$ **:** supposons qu'il existe une dérivation π du séquent $\vdash \Gamma, \varphi \wedge \psi$. On montre par récurrence sur la profondeur de π que $\vdash_{\mathbf{LK_0}} \Gamma, \varphi$ et $\vdash_{\mathbf{LK_0}} \Gamma, \psi$.

- Si π se termine par $(\wedge)$ où $\varphi \wedge \psi$ est principale, alors évidemment il existe des sous-dérivations de π pour chacune de ses prémisses $\vdash \Gamma, \varphi$ et $\vdash \Gamma, \psi$.
- Sinon π se termine par une règle (R) où $\varphi \wedge \psi$ n'est pas principale. Par inspection des règles, on est nécessairement dans une situation

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots \\ \vdash \Gamma_1, \varphi \wedge \psi\end{array} \quad \cdots \quad \begin{array}{c}\pi_k\\ \vdots \\ \vdash \Gamma_k, \varphi \wedge \psi\end{array}}{\vdash \Gamma, \varphi \wedge \psi}\ \text{(R)}$$

où $0 \leq k \leq 2$ ($k = 0$ correspondant au cas de la règle (ax)). Pour tout $1 \leq i \leq k$, par hypothèse de récurrence sur π_i, il existe des dérivations π'_i et π''_i de $\vdash \Gamma_i, \varphi$ et de $\vdash \Gamma_i, \psi$. On a donc les dérivations

$$\dfrac{\begin{array}{c}\pi'_1\\ \vdots \\ \vdash \Gamma_1, \varphi\end{array} \quad \cdots \quad \begin{array}{c}\pi'_k\\ \vdots \\ \vdash \Gamma_k, \varphi\end{array}}{\vdash \Gamma, \varphi}\ \text{(R)} \qquad \text{et} \qquad \dfrac{\begin{array}{c}\pi''_1\\ \vdots \\ \vdash \Gamma_1, \psi\end{array} \quad \cdots \quad \begin{array}{c}\pi''_k\\ \vdots \\ \vdash \Gamma_k, \psi\end{array}}{\vdash \Gamma, \psi}\ \text{(R)}$$

□

Exemple 8.9. Reprenons la recherche de preuve de l'exemple 8.6. Considérons l'application de la règle $(\vee)$ sur la formule principale $P \vee \neg Q$ dans

$$\dfrac{\dfrac{\dfrac{}{\vdash P, \neg Q, \neg P}\ \text{(ax)} \qquad \dfrac{\text{échec}}{\vdash P, \neg Q, R}}{\vdash P, \neg Q, \neg P \wedge R}\ (\wedge)}{\vdash P \vee \neg Q, \neg P \wedge R}\ (\vee)$$

Au moins une prémisse de cette règle n'est pas prouvable, à savoir $\vdash P, \neg Q, \neg P \wedge R$. Par le lemme 8.8 d'inversibilité syntaxique, la conclusion $\vdash P \vee \neg Q, \neg P \wedge R$ n'est pas prouvable. Il est donc inutile d'essayer de choisir la formule $\neg P \wedge R$ comme principale, puisque la recherche de preuve échouera aussi dans ce cas (comme on a pu le vérifier dans l'exemple 8.6).
De plus, quand on a comme ici le choix entre formule $\vee$ et une formule $\wedge$ comme principale, il vaut mieux choisir la formule $\vee$ qui mène à un arbre de recherche de preuve plus petit (comme on a aussi pu le voir dans l'exemple 8.6).

Grâce au lemme 8.8 d'inversibilité syntaxique, la recherche de preuve peut s'implémenter très aisément par un programme récursif dont voici le pseudo-code.

Fonction prouvable($\vdash \Gamma$)
1 **si** $\vdash \Gamma = \vdash \Delta, P, \neg P$ **alors**
2 **retourner** 1
3 **si** $\vdash \Gamma = \vdash \Delta, \varphi \vee \psi$ **alors**
4 **retourner** prouvable($\vdash \Delta, \varphi, \psi$)
5 **si** $\vdash \Gamma = \vdash \Delta, \varphi \wedge \psi$ **alors**
6 **retourner** prouvable($\vdash \Delta, \varphi$) and prouvable($\vdash \Delta, \psi$)
7 **retourner** 0

L'implémentation de la recherche de preuve en OCaml sera faite en mini-projet.

8.2.2. *Implémentation de la recherche de preuve en Java.* Pour notre implémentation en Java du calcul des séquents, nous allons représenter un séquent par une structure de données plus riche qu'une simple `Collection<Formule>`, en distinguant les littéraux, les formules $\wedge$ et les formules $\vee$. Notre classe `Sequent` est définie comme suit.

Sequent
```
import java.util.*;
public class Sequent {
    protected HashMap<String,Boolean> litteraux;
    protected LinkedList<Formule.Et> ets;
    protected LinkedList<Formule.Ou> ous;
    // ⋮
```

L'idée est qu'un littéral positif P du séquent soit associé par `litteraux` à `Boolean.TRUE`, tandis qu'un littéral négatif $\neg P$ le soit à `Boolean.FALSE`.

Cette structure permet de détecter immédiatement au moment où on ajoute un littéral au séquent s'il devient une instance de la règle d'axiome (ax) : cela se produit si on ajoute P et que $\neg P$ était déjà présent, ou si on ajoute $\neg P$ et que P était déjà présent. Voici le code Java correspondant.

Sequent
```
// retourne faux si le séquent devient une instance de la règle d'axiome
public boolean add(Formule phi) {
    return phi.addToSequent(this);
}
```
Formule
```
protected abstract boolean addToSequent(Sequent seq);
```
Formule.Et
```
protected boolean addToSequent(Sequent seq) {
    seq.ets.add(this);
    return true;
}
```
Formule.Ou
```
protected boolean addToSequent(Sequent seq) {
    seq.ous.add(this);
    return true;
}
```
Formule.Non
```
protected boolean addToSequent(Sequent seq) {
    // vérifie que nous sommes bien en forme normale négative
```

```
    assert (phi1.getClass() == Formule.Proposition.class);
    Formule.Proposition p = (Formule.Proposition) phi1;
    Boolean existe = seq.litteraux.put(p.nom,Boolean.FALSE);
    return existe == null || !existe.booleanValue();
}
```

Formule.Proposition

```
protected boolean addToSequent(Sequent seq) {
    Boolean existe = seq.litteraux.put(nom,Boolean.TRUE);
    return existe == null || existe.booleanValue();
}
```

Plutôt que l'algorithme récursif de la section 8.2.1, nous implémentons un algorithme itératif avec une file d'attente.

Sequent

```
public boolean prouvable() {
    Queue<Sequent> file = new LinkedList<Sequent>(); // file d'attente
    file.add(this);
    // recherche de preuve
    while (!file.isEmpty()) {
        // le séquent à prouver
        Sequent seq = file.poll();
        // cas d'une branche d'échec
        if (seq.ous.isEmpty() && seq.ets.isEmpty())
            return false;
        // choix d'une formule principale
        Formule.Ou phi = seq.ous.poll();
        if (phi != null) {
            // si la règle d'axiome ne s'applique pas, ajout à la file
            if (seq.add(phi.phi1) && seq.add(phi.phi2))
                file.add(seq);
        }
        else {
            Formule.Et psi = seq.ets.poll();
            Sequent clone = (Sequent) seq.clone();
            // si la règle d'axiome ne s'applique pas, ajout à la file
            if (seq.add(psi.phi1))
                file.add(seq);
            // si la règle d'axiome ne s'applique pas, ajout à la file
            if (clone.add(psi.phi2))
                file.add(clone);
        }
    }
    return true;
}
```

Par le théorème 8.10 de correction et le théorème 8.13 de complétude qui seront démontrés ci-après, pour vérifier qu'une formule propositionnelle φ est valide, il suffit de vérifier que le séquent $\vdash \varphi$ est prouvable.

Formule

```
public boolean valide() {
    Sequent seq = new Sequent();
    seq.add(this.getNNF());
    return seq.prouvable();
}
```

8.3. **Correction et complétude.** Un séquent $\vdash \Gamma$ est *satisfait* par une interprétation I, ce qu'on écrit $I \models \Gamma$, s'il existe une formule témoin $\vartheta \in \mathrm{dom}(\Gamma)$ telle que $I \models \vartheta$. On dit qu'un séquent $\vdash \Gamma$ est *valide* et on écrit $\models \Gamma$ si pour toute interprétation I, $I \models \Gamma$.

Notons que ces définitions généralisent bien les notions de satisfiabilité et de validité des formules propositionnelles. En effet, si $\Gamma = \varphi$ une formule propositionnelle en forme normale négative, par la définition ci-dessus, $\vdash \varphi$ est satisfait par une interprétation I si et seulement s'il existe une formule témoin $\vartheta \in \mathrm{dom}(\Gamma)$ telle que $I \models \vartheta$; comme $\mathrm{dom}(\Gamma) = \{\varphi\}$, le seul choix possible pour ϑ est $\vartheta = \varphi$, et donc I satisfait $\vdash \varphi$ si et seulement si $I \models \varphi$.

La correction et la complétude du calcul des séquents montrent qu'un séquent est prouvable si et seulement s'il est valide. La correction se montre par une simple induction sur les dérivations en calcul des séquents.

Théorème 8.10 (correction). *Si* $\vdash_{\mathbf{LK}_0} \Gamma$*, alors* $\models \Gamma$.

Démonstration. On procède par induction structurelle sur une dérivation de $\vdash \Gamma$, en montrant pour chaque règle du calcul des séquents que si les prémisses sont valides, alors la conclusion l'est aussi.

Pour (ax) : alors $\Gamma = \Gamma', P, \neg P$ pour une formule φ. Pour toute interprétation I, soit $I \models P$, soit $I \models \neg P$; dans tous les cas $I \models \Gamma$.

Pour $(\vee)$ **:** alors $\Gamma = \Gamma', \varphi \vee \psi$ où $\vdash_{\mathbf{LK}_0} \Gamma', \varphi, \psi$. Soit I une interprétation quelconque ; alors par hypothèse d'induction $I \models \Gamma', \varphi, \psi$. Il existe donc une formule témoin $\vartheta \in \mathrm{dom}(\Gamma') \cup \{\varphi, \psi\}$ telle que $I \models \vartheta$. Si $\vartheta \in \{\varphi, \psi\}$, alors $I \models \varphi \vee \psi$, sinon $\vartheta \in \mathrm{dom}(\Gamma')$ et alors $I \models \Gamma'$; dans tous les cas $I \models \Gamma$.

Pour $(\wedge)$ **:** alors $\Gamma = \Gamma', \varphi \wedge \psi$ où $\vdash_{\mathbf{LK}_0} \Gamma', \varphi$ et $\vdash_{\mathbf{LK}_0} \Gamma', \psi$. Soit I une interprétation quelconque ; alors par hypothèse d'induction $I \models \Gamma', \varphi$ et $I \models \Gamma', \psi$. Il existe donc une formule témoin $\vartheta \in \mathrm{dom}(\Gamma') \cup \{\varphi\}$ telle que $I \models \vartheta$ et une formule témoin $\vartheta' \in \mathrm{dom}(\Gamma') \cup \{\psi\}$ telle que $I \models \vartheta'$. Si $\vartheta \in \mathrm{dom}(\Gamma')$ ou $\vartheta' \in \mathrm{dom}(\Gamma')$, alors $I \models \Gamma'$. Sinon, $\vartheta = \varphi$ et $\vartheta' = \psi$ et donc $I \models \varphi \wedge \psi$. Dans tous les cas $I \models \Gamma$. □

Pour la complétude du calcul des séquents, c'est-à-dire montrer que si un séquent est valide, alors il est prouvable, rappelons qu'une *branche d'échec* est une branche de recherche de preuve finie $\vdash \Gamma_0, \vdash \Gamma_1, \ldots, \vdash \Gamma_n$ où aucune règle ne s'applique au séquent $\vdash \Gamma_n$; on appelle un tel séquent un *séquent d'échec*.

Propriété 8.11 (contre-modèle des séquents d'échec). *Soit* $\vdash \Gamma$ *un séquent d'échec. Alors il en existe un* contre-modèle*, c'est-à-dire une interprétation* I *telle que* $I \not\models \Gamma$.

Démonstration. Par définition d'un séquent d'échec, aucune règle ne s'applique à $\vdash \Gamma$. Par suite,
- comme ni $(\vee)$ ni $(\wedge)$ ne s'applique, $\mathrm{dom}(\Gamma)$ ne contient que des littéraux : $\mathrm{dom}(\Gamma) = \{\ell_1, \ldots, \ell_k\}$;
- comme (ax) ne s'applique pas non plus, $\mathrm{dom}(\Gamma)$ ne contient pas à la fois un littéral et sa négation.

Il existe donc une interprétation I telle que $I \not\models \ell_j$ pour tout $1 \leq j \leq k$, c'est-à-dire telle que $I \not\models \ell_1, \ldots, \ell_k$ et donc $I \not\models \Gamma$. □

Par exemple, les branches d'échec de l'exemple 8.6 se terminent sur le séquent d'échec $\vdash P, \neg Q, R$, et toute interprétation qui étend $[0/P, 1/Q, 0/R]$ en est un contre-modèle.

L'idée de la preuve de complétude qui va suivre est que l'existence d'un contre-modèle pour le séquent d'échec $\vdash \Gamma_n$ d'une branche d'échec $\vdash \Gamma_0, \vdash \Gamma_1, \ldots, \vdash \Gamma_n$ implique l'existence d'un contre-modèle pour le séquent $\vdash \Gamma_0$. Nous nous appuyons pour cela sur une version « sémantique » de l'inversibilité de notre calcul des séquents. On dit pour cela qu'une règle est *sémantiquement inversible* si, quand sa conclusion est valide, alors chacune de ses prémisses est aussi valide.

Lemme 8.12 (inversibilité sémantique)**.** *Les règles du calcul des séquents sont sémantiquement inversibles. Plus précisément, si une interprétation I est un contre-modèle d'une prémisse, alors c'est aussi un contre-modèle de la conclusion.*

Démonstration.

Pour la règle (ax) : comme elle n'a pas de prémisse, elle est bien sémantiquement inversible.

Pour la règle $(\vee)$ **:** supposons $I \not\models \Gamma, \varphi, \psi$, donc $I \not\models \Gamma$, $I \not\models \varphi$ et $I \not\models \psi$. On en déduit que $I \not\models \varphi \vee \psi$, et comme $I \not\models \Gamma$, que $I \not\models \Gamma, \varphi \vee \psi$.

Pour la règle $(\wedge)$ **:** supposons que $I \not\models \Gamma, \varphi$ (le cas où $I \not\models \Gamma, \psi$ est similaire). Donc $I \not\models \Gamma$ et $I \not\models \varphi$. On en déduit que $I \not\models \varphi \wedge \psi$ et donc, puisque $I \not\models \Gamma$, que $I \not\models \Gamma, \varphi \wedge \psi$. □

Dans l'exemple 8.6, toute interprétation I qui étend $[0/P, 1/Q, 0/R]$ est un contre-modèle du séquent $\vdash P, \neg Q, R$, et par applications successives du lemme 8.12 d'inversibilité sémantique, on en déduit que I est un contre-modèle du séquent initial $\vdash (P \vee \neg Q) \vee (\neg P \wedge R)$, qui n'est donc pas valide. C'est ce raisonnement que nous généralisons pour démontrer le théorème de complétude.

Théorème 8.13 (complétude)**.** *Si* $\models \Gamma$, *alors* $\vdash_{\mathbf{LK}_0} \Gamma$.

Démonstration. On procède par contraposition : on suppose $\vdash \Gamma$ non prouvable, et on montre que $\vdash \Gamma$ n'est pas valide, en exhibant un contre-modèle du séquent $\vdash \Gamma$, c'est-à-dire une interprétation I telle que $I \not\models \Gamma$.

On commence par observer que si $\vdash \Gamma$ n'est pas prouvable, alors il existe une branche d'échec. En effet, par la propriété 8.7, les branches de recherche de preuve sont finies, donc elles terminent toutes soit par un séquent d'échec, soit par une instance de la règle d'axiome. Mais si $\vdash \Gamma$ avait une recherche de preuve où toutes les branches se terminaient par une instance de la règle d'axiome, alors cette recherche aurait construit une dérivation et donc $\vdash \Gamma$ serait prouvable.

Soit donc $\vdash \Gamma_0, \ldots, \vdash \Gamma_n$ une branche d'échec où $\vdash \Gamma_0 = \vdash \Gamma$ et où $\vdash \Gamma_n$ est un séquent d'échec. Par la propriété 8.11 de contre-modèle des séquents d'échec, il existe un contre-modèle I de $\vdash \Gamma_n$, tel que $I \not\models \Gamma_n$.

On montre par récurrence décroissante sur $n \geq i \geq 0$ que $I \not\models \Gamma_i$. Pour le cas de base au rang $i = n$, on avait bien choisi I tel que $I \not\models \Gamma_n$. Pour l'étape de récurrence au rang $i - 1$, on sait par hypothèse de récurrence que $I \not\models \Gamma_i$, et par le lemme 8.12 d'inversibilité sémantique, $I \not\models \Gamma_{i-1}$. Pour conclure, on a montré qu'il existait un contre-modèle I de $\vdash \Gamma_0$, c'est-à-dire de $\vdash \Gamma$, qui n'est donc pas valide. □

On peut remarquer que, puisque le calcul des séquents est correct et complet, l'inversibilité syntaxique et l'inversibilité sémantique sont deux propriétés équivalentes : la première parle de prouvabilité, *tandis que la seconde parle de* validité.

9. * Clauses de Horn

Résumé. Une *clause de Horn* est une clause où au plus un littéral est positif. On représente habituellement une telle clause sous la forme d'une implication $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$, ou sous la forme $\bot \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ s'il n'y a aucun littéral positif (une telle clause est alors appelée *négative*).

Pour tout ensemble S de clauses de Horn, il existe une interprétation I_S telle que S soit satisfiable si et seulement si $I_S \vDash S$, et qui est *minimale* au sens où elle met le moins de propositions possible à 1 (voir le théorème 9.9). En fait, quand S est satisfiable, cette interprétation met une proposition à 1 si et seulement si la proposition est une conséquence logique de S (voir la proposition 9.10).

L'ensemble des propositions vraies dans I_S peut-être calculé en *temps linéaire* en la taille de l'ensemble de clauses S (voir la section 9.2.3). Cela permet de résoudre HornSAT – la variante de SAT où les clauses fournies en entrée sont toutes des clauses de Horn – en temps linéaire dans le pire des cas.

Une *clause de Horn* est une clause où au plus un littéral est positif.

Exemple 9.1. Considérons la formule propositionnelle

$$Q \wedge ((Q \wedge (P \vee R)) \Rightarrow S) \wedge \neg(P' \wedge Q) \wedge S \wedge ((P \Rightarrow \neg R) \vee P) \wedge ((Q \wedge S) \Rightarrow R)\,.$$

Cette formule est logiquement équivalente à la formule sous forme normale conjonctive

$$Q \wedge (\neg Q \vee \neg P \vee S) \wedge (\neg Q \vee \neg R \vee S) \wedge (\neg P' \vee \neg Q) \wedge S \wedge (\neg P \vee \neg R \vee P) \wedge (\neg Q \vee \neg S \vee R)\,,$$

et sous forme clausale comme

$$\{\{Q\},\ \{\neg Q, \neg P, S\},\ \{\neg Q, \neg R, S\},\ \{\neg P', \neg Q\},\ \{S\}, \{\neg P, \neg R, P\},\ \{\neg Q, \neg S, R\}\}\,.$$

Chaque clause dans cette forme clausale contient au plus un littéral positif : c'est donc un ensemble de clauses de Horn.

On appelle une clause de Horn sans littéral positif $\neg Q_1 \vee \cdots \vee \neg Q_k$ une clause *négative* et on l'écrit plutôt sous la forme $\bot \Leftarrow Q_1 \wedge \cdots \wedge Q_k$; une telle clause non négative peut aussi être vue comme une règle de déduction $\frac{Q_1 \cdots Q_k}{\bot}$. On appelle une clause de Horn avec un littéral positif $P \vee \neg Q_1 \vee \cdots \vee \neg Q_k$ une clause *non négative* et on l'écrit plutôt sous la forme $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$; une telle clause non négative peut aussi être vue comme une règle de déduction $\frac{Q_1 \cdots Q_k}{P}$.

En intelligence artificielle, on parle souvent de base de connaissance *(ou KB de l'anglais «* knowledge base *») pour un ensemble de clauses de Horn.*

Exemple 9.2. Les clauses de l'exemple 9.1 peuvent être écrites de manière équivalente comme l'ensemble de clauses de Horn

$$\{Q,\ S \Leftarrow Q \wedge P,\ S \Leftarrow Q \wedge P',\ \bot \Leftarrow P' \wedge Q,\ S,\ P \Leftarrow P \wedge R,\ R \Leftarrow Q \wedge S\}\,.$$

On peut aussi voir cet ensemble comme le système de déduction fini ci-dessous.

$$\frac{}{Q} \qquad \frac{Q \quad P}{S} \qquad \frac{Q \quad P'}{S} \qquad \frac{P' \quad Q}{\bot} \qquad \frac{}{S} \qquad \frac{P \quad R}{P} \qquad \frac{Q \quad S}{R}$$

Les clauses de Horn ont plusieurs intérêts :
- leur problème de satisfiabilité peut être résolu de manière très efficace, à savoir en temps linéaire,
- un système de déduction sur un ensemble fini n'est jamais qu'un ensemble de clauses de Horn,
- c'est le format employé en programmation logique, par exemple par le langage Prolog, et en bases de données par le langage Datalog.

9.1. **Modèle minimal.** Dans cette section, on va utiliser un ordre partiel sur les interprétations (similaire à celui utilisé sur les fonctions booléennes) : $I \leq I'$ si et seulement si, pour toute proposition $P \in \mathcal{P}_0$, $I \vDash P$ implique $I' \vDash P$. On note I_0 pour l'interprétation constante égale à 0, c'est-à-dire telle que $I_0 \nvDash P$ pour toute proposition $P \in \mathcal{P}_0$; alors $I_0 \leq I$ pour toute interprétation $I \in \mathbb{B}^{\mathcal{P}_0}$.

La propriété centrale des clauses de HORN est qu'elles ont un *modèle minimal* pour cet ordre $\leq$ sur les interprétations.

L'ensemble $\mathbb{B}^{\mathcal{P}_0}$ des interprétations ordonnées par $\leq$ forme un treillis complet *où, pour toute proposition $P \in \mathcal{P}_0$, $(I \vee I') \vDash P$ si $I \vDash P$ ou $I' \vDash P$, et $(I \wedge I') \vDash P$ si $I \vDash P$ et $I' \vDash P$.*

Propriété 9.3 (modèle minimal)**.** *Soit S un ensemble de clauses de HORN. Si S est satisfiable, alors il a un modèle minimal.*

9.1.1. *Les fonctions f_S.* Une des manières de démontrer la propriété 9.3 est de construire explicitement ce modèle minimal. Soit S un ensemble de clauses de HORN. Cet ensemble détermine une fonction $f_S : \mathbb{B}^{\mathcal{P}_0} \to \mathbb{B}^{\mathcal{P}_0}$ définie pour toute interprétation $I \in \mathbb{B}^{\mathcal{P}_0}$ et toute proposition $P \in \mathcal{P}_0$ par

$$f_S(I) \vDash P \quad \text{si} \quad \exists (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in S \,.\, \forall 1 \leq i \leq k \,.\, I \vDash Q_i \,. \tag{5}$$

À noter que si $S' \subseteq S$ est le sous-ensemble des clauses non-négatives de S, alors $f_{S'} = f_S$: les clauses négatives n'interviennent pas dans l'équation (5).

On peut aussi démontrer la propriété 9.3 en montrant que l'ensemble des modèles Sat(S) *d'un ensemble S de clauses de HORN est fermé par $\wedge$ (KNUTH, 2008, sec. 7.1.1, thm. H).*

Exemple 9.4. Reprenons les clauses de l'exemple 9.2 et considérons l'interprétation I telle que $I \vDash P$ et $I \vDash R$ et ce sont les seules propositions satisfaites par I ; autrement dit, $I \stackrel{\text{def}}{=} I_0[1/P, 1/R]$. Alors $f_S(I) \vDash Q$, $f_S(I) \vDash S$ et $f_S(I) \vDash P$ en utilisant les clauses Q, S et $P \Leftarrow P \wedge R$ de l'ensemble de clauses, et ce sont les seules propositions satisfaites par $f_S(I)$; autrement dit $f_S(I) = I_0[1/Q, 1/S, 1/P]$. Intuitivement, cela correspond aux trois propositions que l'on peut déduire en *une étape* de déduction depuis les propositions satisfaites par I.

Propriété 9.5 (monotonie de f_S)**.** *La fonction f_S est* monotone *: $I \leq I'$ implique $f_S(I) \leq f_S(I')$.*

Démonstration. Supposons $I \leq I'$ et soit P une proposition telle que $f_S(I) \vDash P$: on cherche à montrer que $f_S(I') \vDash P$ – ce qui montrera $f_S(I) \leq f_S(I')$. Par définition de f_S dans l'équation (5), puisque $f_S(I) \vDash P$, il existe une clause $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ dans S telle que $I \vDash Q_i$ pour tout $1 \leq i \leq k$. Comme $I \leq I'$, on a aussi $I' \vDash Q_i$ pour tout $1 \leq i \leq k$. Par définition de f_S, on a bien $f_S(I') \vDash P$. □

Exemple 9.6. Pour compléter l'exemple 9.4, considérons I' telle que $I' \vDash Q$, $I' \vDash P$, $I' \vDash R$ et $I' \vDash S$ et ce sont les seules propositions satisfaites par I' ; autrement dit, $I' \stackrel{\text{def}}{=} I[1/Q, 1/S]$. Alors $f_S(I') \vDash Q$, $f_S(I') \vDash S$, $f_S(I') \vDash P$, et $f_S(I') \vDash R$ et ce sont les seules propositions satisfaites par $f_S(I')$. On a $f_S(I') = f_S(I)[1/R]$ qui vérifie bien $f_S(I) \leq f_S(I')$.

Lemme 9.7. *Soit S' un ensemble de clauses non négatives de HORN. Alors $I \vDash S'$ si et seulement si $f_{S'}(I) \leq I$.*

Démonstration. Si $I \vDash S'$, alors pour toute clause $C = (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in S'$, $I \vDash C$. Dès lors, si $f_S(I) \vDash P$, c'est-à-dire si $I \vDash Q_i$ pour tout $1 \leq i \leq k$, alors $I \vDash P$. Cela montre $f_{S'}(I) \leq I$.

Inversement, supposons que $f_{S'}(I) \leq I$, et considérons une clause non négative $C = (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in S'$: on veut montrer que $I \vDash C$.

- S'il existe $1 \leq i \leq k$ tel que $I \nvDash Q_i$, alors $I \vDash C$;
- sinon, nécessairement $I \vDash Q_i$ pour tout $1 \leq i \leq k$, et alors $f_S(I) \vDash P$ et comme $f_S(I) \leq I$ on en déduit $I \vDash P$ et donc encore une fois $I \vDash C$. □

À titre d'exemple, l'interprétation $I \stackrel{\text{def}}{=} I_0[1/P, 1/R]$ de l'exemple 9.4 est telle que $f_S(I) \not\leq I$ et effectivement $I \nvDash Q$. L'interprétation $I' \stackrel{\text{def}}{=} I_0[1/P, 1/Q, 1/R, 1/S]$ de l'exemple 9.6 est telle que $f_S(I') \leq I'$ et effectivement I' satisfait toutes les clauses de l'exemple 9.2.

9.1.2. *Itérations de la fonction f_S.* En itérant la fonction f_S à partir de l'interprétation I_0, par monotonie de f_S on définit une chaîne d'interprétations $I_0 \leq f_S(I_0) \leq f_S^2(I_0) \leq \cdots$ dont la limite est l'interprétation $I_S \stackrel{\text{def}}{=} \bigvee_{n\in\mathbb{N}} f_S^n(I_0)$, qui est telle que, pour toute proposition $P \in \mathcal{P}_0$,

$$I_S \vDash P \text{ si et seulement si } \exists n \in \mathbb{N} \,.\, f_S^n(I_0) \vDash P \,. \tag{6}$$

Dans le cas de l'ensemble de clauses de l'exemple 9.2, on obtient la chaîne $I_0 \leq I_0[1/Q, 1/S] \leq I_0[1/Q, 1/R, 1/S] \leq I_0[1/Q, 1/R, 1/S] \leq \cdots$ qui se stabilise sur $I_S = I_0[1/Q, 1/R, 1/S]$.

Le lemme 9.8 montre que l'interprétation I_S est le plus petit point fixe *de la fonction f_S dans le treillis complet $(\mathbb{B}^{\mathcal{P}_0}, \leq)$.*

Lemme 9.8. *Soit S un ensemble de clauses de* Horn*. Alors I_S est la plus petite interprétation telle que $f_S(I_S) \leq I_S$.*

Démonstration. Montrons tout d'abord que $f_S(I_S) \leq I_S$. Considérons pour cela une proposition P telle que $f_S(I_S) \vDash P$ et montrons que $I_S \vDash P$. Par définition de f_S dans l'équation (5), puisque $f_S(I_S) \vDash P$, il existe une clause non négative $C = (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in S$ où $I_S \vDash Q_i$ pour tout $1 \leq i \leq k$. Par l'équation (6), pour chaque $1 \leq i \leq k$ il existe donc un indice n_i tel que $f_S^{n_i}(I_0) \vDash Q_i$. Dès lors, si l'on définit $n \stackrel{\text{def}}{=} \max_{1\leq i\leq k} n_i$, $f_S^n(I_0) \vDash Q_i$ pour tout $1 \leq i \leq k$ par monotonie de f_S, donc $f_S^{n+1}(I_0) \vDash P$, ce qui implique $I_S \vDash P$ par l'équation (6).

Soit maintenant I une interprétation telle que $f_S(I) \leq I$. On montre par récurrence sur n que $f_S^n(I_0) \leq I$ pour tout n, ce qui impliquera $I_S \leq I$ d'après l'équation (6). Pour le cas de base au rang $n = 0$, $I_0 \leq I$. Pour l'étape de récurrence $n + 1$, $f_S^n(I_0) \leq I$ par hypothèse de récurrence au rang $n + 1$, donc $f_S^{n+1}(I_0) = f_S(f_S^n(I_0)) \leq f_S(I)$ par monotonie de f_S, et donc finalement $f_S^{n+1}(I_0) \leq I$ puisque $f_S(I) \leq I$. □

On a même $f_S(I') = I'$, autrement dit I' est un point fixe *de la fonction f_S, mais ce n'est pas le plus petit.*

À noter que l'interprétation $I' \stackrel{\text{def}}{=} I_0[1/P, 1/Q, 1/R, 1/S]$ de l'exemple 9.6 était bien telle que $f_S(I') \leq I'$, mais ce n'est pas la plus petite interprétation avec cette propriété : $I_S \stackrel{\text{def}}{=} I_0[1/Q, 1/R, 1/S]$ est plus petite.

Nous déduisons maintenant une preuve de la propriété 9.3 : le modèle minimal de S est l'interprétation I_S que nous avons construite.

Théorème 9.9. *Soit S un ensemble de clauses de* Horn*. Alors S est satisfiable si et seulement si $I_S \vDash S$, et dans ce cas I_S est le modèle minimal de S.*

Démonstration. Soit $S' \subseteq S$ le sous-ensemble des clauses non négatives de S. Cet ensemble est toujours satisfiable, par exemple par l'interprétation I_1 telle que $I \vDash P$ pour toute proposition $P \in \mathcal{P}_0$. Par les lemmes 9.7 et 9.8 et en notant que $f_S = f_{S'}$, I_S est donc la plus petite interprétation telle que $I_S \vDash S'$.

Supposons $I_S \vDash S$. Alors S est satisfiable, et il reste à montrer que I_S est le modèle minimal de S. En effet, si $I \leq I_S$ est une interprétation telle que $I \vDash S$, alors $I \vDash S'$ et par les lemmes 9.7 et 9.8, $I_S \leq I$.

Supposons maintenant S satisfiable et montrons que $I_S \vDash S$. Tout modèle I de S est en particulier un modèle de S', donc $I_S \leq I$ puisque I_S est minimal. De plus, pour toute clause négative $C = (\bot \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in S$, comme $I \vDash C$, il existe $1 \leq i \leq k$ tel que $I \nvDash Q_k$. Donc $I_S \nvDash Q_k$ et $I_S \vDash C$. □

9.2. **HornSAT.** Le problème de satisfiabilité associé aux clauses de Horn est la restriction suivante de SAT :

Problème (HornSAT)**.**

instance : un ensemble fini F de clauses de Horn

question : F est-il satisfiable ?

Une manière naturelle de résoudre HornSAT est de calculer l'interprétation I_F définie par l'équation (6) : alors par le théorème 9.9, F est satisfiable si et seulement si $I_F \vDash F$, et cette dernière étape se vérifie en temps linéaire une fois qu'on a calculé I_F.

9.2.1. *Conséquences logiques d'un ensemble de clauses de Horn.* Un autre problème souvent considéré sur des ensembles S de clauses de Horn est de déterminer quelles en sont les *conséquences logiques* au sens de la section 4.1.

On a vu dans le théorème 9.9 que tout ensemble satisfiable S de clauses de Horn a un modèle minimal, qui peut être calculé d'après l'équation (6) en itérant la fonction f_S de l'équation (5) sur l'interprétation I_0. Un autre point de vue sur ce modèle minimal est qu'il met à vrai une proposition P si et seulement si cette proposition est une conséquence logique de l'ensemble S.

Proposition 9.10. *Soit S un ensemble de clauses de Horn. Si S est satisfiable, alors pour toute proposition $P \in \mathcal{P}_0$, $I_S \vDash P$ si et seulement si $S \vDash P$.*

⚠ *Si S est insatisfiable, alors $S \vDash P$ pour tout $P \in \mathcal{P}_0$.*

Démonstration. Supposons S satisfiable.

Montrons tout d'abord que $S \vDash P$ implique $I_S \vDash P$ pour toute proposition P. D'après le théorème 9.9, puisque S est satisfiable, $I_S \vDash S$. Par définition d'une conséquence logique, $I_S \vDash P$.

Montrons maintenant que $I_S \vDash P$ implique $S \vDash P$ pour toute proposition P. Si on note I l'interprétation telle que $I \vDash P$ si et seulement si $S \vDash P$, cela revient à montrer que $I_S \leq I$. Par le lemme 9.8, il suffit donc de montrer que $f_S(I) \leq I$.

Soit P une proposition telle que $f_S(I) \vDash P$: on veut montrer que $S \vDash P$ (ce qui montrera bien $I \vDash P$). Pour montrer $S \vDash P$, par définition d'une conséquence logique, on suppose que $I' \vDash S$ pour une interprétation I' et on montre que $I' \vDash P$.

Puisque $f_S(I) \vDash P$, par définition de f_S dans l'équation (5), il existe une clause non-négative $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ dans S telle que $I \vDash Q_i$ pour tout $1 \leq i \leq k$, c'est-à-dire par définition de I telle que $S \vDash Q_i$ pour tout $1 \leq i \leq k$. Chaque Q_i est donc une conséquence logique de S ; puisque $I' \vDash S$ on a donc $I' \vDash Q_i$ pour tout $1 \leq i \leq k$. De plus, $I' \vDash S$ donc $I' \vDash (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k)$: cela montre bien $I' \vDash P$. □

Voici un problème de décision typiquement considéré en intelligence artificielle.

Problème (Conséquence de Horn).
instance : un ensemble fini F de clauses de Horn non-négatives et une proposition P
question : est-ce que P est une conséquence logique de F ?

Comme F ne contient que des clauses non-négatives, il est satisfiable. Le proposition 9.10 s'applique donc et montre qu'il suffit de calculer I_F le modèle minimal de F et de vérifier si $I_F \vDash P$. Cela peut aussi être résolu à l'aide d'un solveur pour HornSAT : en effet, par le lemme 4.2 de déduction, l'ensemble de clauses $F \cup \{\bot \Leftarrow P\}$ est insatisfiable si et seulement si $F \vDash P$.

9.2.2. *Algorithme naïf.* Pour les deux problèmes et , on a vu que si on est capable de calculer le modèle minimal I_F d'un ensemble F de clauses de Horn non-négatives, alors il est ensuite aisé de répondre.

Fixons un tel ensemble F de clauses non-négatives. Les interprétations $I \in \mathbb{B}^{\mathrm{fp}(F)}$ vont être représentées comme des ensembles E dans $2^{\mathrm{fp}(F)}$: on aura alors $P \in E$ si et seulement $I \vDash P$. Le calcul de I_F par itération de la fonction f_F comme défini dans les équations (5) et (6) va être remplacé par un calcul équivalent sur des ensembles. Définissons

$$E_{\leq n} \stackrel{\text{def}}{=} \{P \in \mathrm{fp}(F) \mid \exists j \leq n \,.\, f_F^j(I_0) \vDash P\} \tag{7}$$

ainsi que la fonction $g_F : 2^{\mathrm{fp}(F)} \to 2^{\mathrm{fp}(F)}$ par

$$g_F(E) \stackrel{\text{def}}{=} \{P \in \mathrm{fp}(F) \mid \exists P \Leftarrow Q_1 \wedge \cdots \wedge Q_k \,.\, \forall 1 \leq i \leq k \,.\, Q_i \in E\}\ . \tag{8}$$

Propriété 9.11. $E_{\leq 0} = \emptyset$ *et pour tout* $n \in \mathbb{N}$, $E_{\leq n+1} = E_{\leq n} \cup g_F(E_{\leq n})$.

Démonstration. Par récurrence sur n. Pour le cas de base $n = 0$,

$$\begin{aligned} E_{\leq 0} &= \{P \in \text{fp}(F) \mid \exists j \leq 0 \,.\, f_F^j(I_0) \vDash P\} && \text{par (7)} \\ &= \{P \in \text{fp}(F) \mid I_0 \vDash P\} \\ &= \emptyset \,. \end{aligned}$$

Pour l'étape de récurrence $n + 1$,

$$\begin{aligned} E_{\leq n+1} &= \{P \in \text{fp}(F) \mid \exists j \leq n+1 \,.\, f_F^j(I_0) \vDash P\} && \text{par (7)} \\ &= \{P \in \text{fp}(F) \mid \exists j \leq n \,.\, f_F^j(I_0) \vDash P\} \cup \{P \in \text{fp}(F) \mid f_F^{n+1}(I_0) \vDash P\} \\ &= E_{\leq n} \cup \{P \in \text{fp}(F) \mid f_F^{n+1}(I_0) \vDash P\} && \text{par h.r.} \\ &= E_{\leq n} \cup \{P \in \text{fp}(F) \mid f_F(f_F^n(I_0)) \vDash P\} \\ &= E_{\leq n} \cup g_F(\{P \in \text{fp}(F) \mid f_F^n(I_0) \vDash P\}) && \text{par (5) et (8)} \\ &= E_{\leq n} \cup g_F(\{P \in \text{fp}(F) \mid \exists j \leq n \,.\, f_F^j(I_0) \vDash P\}) && \text{car } \forall j \leq n \,.\, f_F^j(I_0) \leq f_F^n(I_0) \\ &= E_{\leq n} \cup g_F(E_{\leq n}) \,. && \text{par h.r.} \end{aligned}$$

□

Exemple 9.12. Reprenons l'exemple 9.2 sans sa clause négative : $F \stackrel{\text{def}}{=} \{Q,\ S \Leftarrow Q \wedge P,\ S \Leftarrow Q \wedge P',\ S,\ P \Leftarrow P \wedge R,\ R \Leftarrow Q \wedge S\}$. On a les ensembles suivants : $E_{\leq 0} = \emptyset$, $E_{\leq 1} = \{Q, S\}$, $E_{\leq 2} = \{Q, R, S\} = E_{\leq n}$ pour tout $n \geq 2$.

La propriété 9.11 fournit un premier algorithme naïf pour calculer le modèle minimal d'un ensemble F de clauses non-négatives.

Fonction horn(F)
1 $E := \emptyset$; $E' := \emptyset$
2 **répéter**
3     $E := E \cup E'$
4     $E' := \emptyset$
5   **pour chaque** $C = (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in F$ **faire**
6   **si** $(P \notin E)$ and $(\forall 1 \leq i \leq k \,.\, Q_i \in E)$ **alors** $E' := E' \cup \{P\}$
7 **jusqu'à** $E' = \emptyset$
8 **retourner** E

Au $(n+1)^{\text{e}}$ passage dans le corps de la boucle principale des lignes 3 à 6, la variable E contient $E_{\leq n}$, tandis que la variable E' contient à la fin de la boucle

$$E'_{n+1} \stackrel{\text{def}}{=} g_F(E_{\leq n}) \setminus E_{\leq n} \,. \tag{9}$$

Dans l'exemple 9.12, on a ainsi $E'_1 = \{Q, S\}$, $E'_2 = \{R\}$ et $E'_3 = \emptyset$.

On a donc $E_{\leq n} = \bigcup_{0<j\leq n} E'_j$: les ensembles E'_j forment une partition de $E_{\leq n}$, et l'algorithme s'arrête dès que $E'_j = \emptyset$. Autrement dit, on ajoute à chaque nouveau passage dans la boucle principale au moins une proposition parmi fp(F) à l'ensemble E, donc on peut faire au plus $|\text{fp}(F)| + 1$ passages dans la boucle principale.

Chacun de ces passages passe par la boucle des lignes 5 et 6, qui, du fait des tests $Q_i \in E$ pour tout $1 \leq i \leq k$ et pour toute clause $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ de F, a un coût en $O(\|F\| \cdot e(|\text{fp}(F)|)$ si

- on définit la *taille* de F comme la somme $\|F\| \stackrel{\text{def}}{=} \sum_{C \in S} |C|$ des tailles de ses clauses (c'est-à-dire la somme des nombres de littéraux présents dans chaque clause) et
- $e(|\text{fp}(F)|)$ borne le coût d'accès dans un sous-ensemble E de fp(F). Ce coût peut varier : par exemple, avec un arbre binaire de recherche, il est en $O(\log |\text{fp}(F)|)$, tandis qu'avec une table de hachage il est en $O(1)$ si on suppose un hachage parfait mais peut atteindre

$O(|\text{fp}(F)|)$ dans le pire des cas sinon ; enfin si $|\text{fp}(F)|$ n'est pas trop grand on peut aussi représenter E comme une tableau de booléens (*bit vector*) indexé par $\text{fp}(F)$ avec des accès en $O(1)$.

La ligne 3 a quant à elle un coût en $|\text{fp}(F)| \cdot e(|\text{fp}(F)|)$ si on ajoute un à un les éléments de E' dans E. La complexité en temps de cet algorithme naïf est donc en $O(|\text{fp}(F)| \cdot \|F\| \cdot e(|\text{fp}(F)|)$, qui est au moins quadratique.

9.2.3. *Algorithme en temps linéaire.* Le principal problème dans l'algorithme précédent est qu'il faut passer en revue toutes les clauses $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ de F à la ligne 5 puis pour chaque Q_i de tester si $Q_i \in E$ à la ligne 6.

Cet algorithme est dû à Dowling *et* Gallier *(1984) dans sa présentation pour* HornSAT ; *c.f. (*Knuth*, 2008, sec. 7.1.1, algo. C). Le même algorithme était cependant déjà connu pour un problème équivalent sur les grammaires algébriques, c.f. (*Harrison*, 1978, p. 97), (*Sippu *et* Soisalon-Soininen*, 1988, thm. 4.14).*

Une idée pourrait être d'utiliser la simplification de clauses comme vue dans l'algorithme DPLL : à la ligne 3, à chaque fois qu'on ajoute une proposition Q à l'ensemble E, on pourrait *simplifier* l'ensemble de clauses F par le littéral P. Pour une clause non-négative de Horn $P \Leftarrow Q_1 \wedge \cdots \wedge Q_{i-1} \wedge Q_i \wedge Q_{i+1} \wedge \cdots \wedge Q_k$, sa simplification par Q_i est la clause $P \Leftarrow Q_1 \wedge \cdots \wedge Q_{i-1} \wedge Q_{i+1} \wedge \cdots \wedge Q_k$. Cela permet d'éviter de tester si $Q_i \in E$ lors des tests de la ligne 6. Pour l'exemple 9.12, les ensembles de clauses sur lesquels on travaille deviennent alors $F_0 = F = \{Q,\ S \Leftarrow Q \wedge P,\ S \Leftarrow Q \wedge P',\ S,\ P \Leftarrow P \wedge R, R \Leftarrow Q \wedge S\}$, puis $F_1 = \{P \Leftarrow P \wedge R,\ R\}$ après simplification par Q et S, et enfin $F_2 = \{P \Leftarrow P\}$ après simplification par R. Cette étape de simplification nécessite cependant toujours une passe sur toutes les clauses restantes à chaque passage par la boucle principale.

Le principe de l'algorithme en temps linéaire est de construire un tableau *contient* qui donne pour chaque proposition $Q \in \text{fp}(F)$ la liste des clauses $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ de F telles que $Q = Q_i$ pour un certain i. De plus, au lieu de simplifier les clauses, on va utiliser un *compteur* qui dit pour chaque clause $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ combien de propositions parmi $Q_1, \ldots, Q_k$ n'ont pas encore été ajoutées à l'ensemble E. La figure 20 illustre ces structures pour l'exemple 9.12.

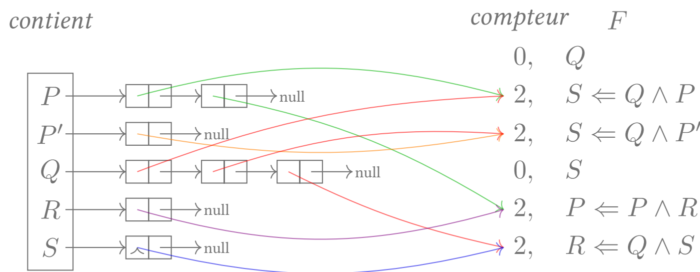


Figure 20. Les structures de données initiales de l'algorithme en temps linéaire.

Ces structures initiales sont construites aux lignes 2 à 9 de l'algorithme suivant, dans lesquelles on initialise aussi E et E' pour contenir toutes les propositions dont le compte est initialement zéro (Q et S dans l'exemple de la figure 20).

Fonction horn(F)

```
1  E := ∅; E′ := ∅
2  pour chaque Q ∈ fp(F) faire
3      contient[Q] := null      (liste des clauses contenant ¬Q, initialement vide)
4  pour chaque C = (P ⇐ Q_1 ∧ ··· ∧ Q_k) ∈ F faire
5      compteur[C] := k      (nombre de littéraux à traiter de la clause C)
6      pour chaque 1 ≤ i ≤ k faire
7          contient[Q_i].add(C)      (ajout à la liste)
8      si k = 0 alors
9          E := E ∪ {P}; E′ := E′ ∪ {P}
```

À l'issue de cette initialisation, on a les invariants suivants pour $n = 1$, qui vont rester vrais à chaque n^{e} entrée ligne 12 dans la boucle principale donnée ci-dessous :

(1) $E = \bigcup_{j \le n} E'_j$,

(2) $E' = E'_n$

(3) $\forall C = (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in F$, *compteur*$[C] = \left|\{1 \le i \le k \mid Q_i \notin \bigcup_{0<j<n} E'_j\}\right|$.

```
12 tant que E′ ≠ ∅ faire
13     E″ := ∅
14     pour chaque Q ∈ E′ faire
15         pour chaque C = (P ⇐ Q_1 ∧ ··· ∧ Q_k) ∈ contient[Q] faire
16             si P ∉ E alors
17                 compteur[C] := compteur[C] − 1
18                 si compteur[C] = 0 alors      (c'est-à-dire si Q_i ∈ E pour tout 1 ≤ i ≤ k)
19                     E := E ∪ {P}; E″ := E″ ∪ {P}
20     E′ := E″
21 retourner E
```

La mise à jour de E'' à la ligne 19 se fait en effet quand *compteur*$[C] = 0$, c'est-à-dire quand toutes les propositions Q_i de la clause C sont dans E. L'algorithme termine quand $E' = \emptyset$.

En ce qui concerne la complexité de l'algorithme, la phase d'initialisation est en temps $O(\|F\| \cdot e(|\text{fp}(F)|))$. Pour la boucle principale, il faut remarquer que, comme les ensembles E' successifs, on ne passe par la ligne 14 qu'au plus une fois par proposition $Q \in \text{fp}(F)$ pendant l'exécution. Cela signifie que pour chaque clause $C = (P \Leftarrow Q_1 \wedge \cdots \wedge Q_k) \in F$, on ne passe par la ligne 15 qu'au plus k fois sur la totalité de l'exécution de l'algorithme ; la boucle principale s'exécute donc elle aussi en $O(\|F\| \cdot e(|\text{fp}(F)|))$.

Si on utilise un tableau de booléens pour représenter l'ensemble E, alors $e(|\text{fp}(F)|)$ est en temps constant $O(1)$ et cet algorithme travaille en temps linéaire dans le pire des cas.

Proposition 9.13. *Soit F un ensemble fini de clauses de Horn. Alors on peut calculer I_F en temps déterministe $O(\|F\| \cdot e(|\text{fp}(F)|))$ où $e(|\text{fp}(F)|)$ borne le coût des opérations d'accès dans des sous-ensembles de* fp(F).

9.2.4. *Implémentation de l'algorithme en temps linéaire en Java.* Voici comment on pourrait implémenter l'algorithme de la section 9.2.3 en Java. Pour commencer, on peut étendre la classe `Clause` de la section 5.2.4 pour traiter les clauses de Horn. On code ici encore les clauses en « style DIMACS », c'est-à-dire avec des entiers strictement positifs pour coder les propositions.

Une clause de la forme $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ ou de la forme $\bot \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ est représentée par un objet de la classe `HornClause` suivante.

HornClause.java
```
public class HornClause {
    // La proposition P (ou 0 pour ⊥ dans une clause négative).
    private int head;
    // L'ensemble de propositions {Q1,...,Qk}.
    private Set<Integer> tail;

    public int head() {
        return head;
    }
    public Set<Integer> tail() {
        return tail;
    }
```

Afin d'implémenter le *compteur* associé à chaque clause, on ajoute aussi un attribut `count` ainsi qu'une méthode pour le décrémenter à la ligne 17 de l'algorithme.

HornClause.java
```
// Le nombre de propositions Q1,...,Qk qui restent à traiter ; initialement k.
private int count;
// Décrémente le compte associé à la clause et retourne true si ce
// compte tombe à 0.
protected boolean decrement() {
    count--;
    return count == 0;
}
```

Voici enfin une implémentation possible de l'algorithme de la section 9.2.3. Une différence avec cet algorithme est qu'on ne suppose pas que l'ensemble de clauses F (représenté par `clauses`) ne contienne uniquement des clauses non-négatives. La variable E, qui représente le modèle minimal si F est satisfiable, est vue comme un `int[]`, tandis que les variables E' et E'' sont des `LinkedList<Integer>`.

Horn.java
```
private Collection<HornClause> clauses; // l'ensemble F
private int[] interpretation;           // l'ensemble E
public boolean satisfiable() {
    @SuppressWarnings("unchecked")
    final Collection<HornClause>[] contient =
        (Collection<HornClause>[]) Array.newInstance((Class<LinkedList<HornClause>>)
        new LinkedList<HornClause>().getClass(), nprops);
    Collection<Integer> nouveaux = new LinkedList<Integer>(); // E' := ∅

    // initialisation
    for (int i = 0; i < nprops; i++) {
        // pour tout Q ∈ fp(F)
        interpretation[i] = -i-1; // Q ∉ E
        contient[i] = new LinkedList<HornClause>(); // contient[Q] := null
    }
    for (HornClause c : clauses) {
        // pour tout C ∈ F
        if (c.tail().size() == 0) {
            if (c.head() == 0)
```

```
                return false;
            interpretation[c.head()-1] = c.head();    //  E := E ∪ {P}
            nouveaux.add(Integer.valueOf(c.head())); // E' := E' ∪ {P}
        }
        else
            for (Integer q : c.tail())
                // pour tout 1 ≤ i ≤ k
                contient[q.intValue()-1].add(c); // contient[Q_i].add(C)
    }

    // boucle principale
    while (!nouveaux.isEmpty()) {
        // tant que E' ≠ ∅
        Collection<Integer> tmp = new LinkedList<Integer>(); // E'' := ∅
        for (Integer q : nouveaux)
            // pour tout Q ∈ E'
            for (HornClause c : contient[q.intValue()-1]) {
                // pour tout C ∈ contient[Q]
                int p = c.head();
                if ((p == 0 || interpretation[p-1] < 0) &&
                    c.decrement()) {
                    if (p == 0)
                        return false;
                    interpretation[p-1] = p;      //  E := E ∪ {P}
                    tmp.add(Integer.valueOf(p)); // E'' := E'' ∪ {P}
                }
            }
        nouveaux = tmp; // E' := E''
    }
    return true;
}
```

9.2.5. *Implémentation de l'algorithme en temps linéaire en OCaml.* Voici pour finir comment on pourrait implémenter l'algorithme de la section 9.2.3 en OCaml. Une clause de Horn de la forme $P \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ ou $\bot \Leftarrow Q_1 \wedge \cdots \wedge Q_k$ va être représentée par une paire `(p,[q₁;…;qₖ])` de type `int * int list`.

On définit aussi un type

```
horn.ml
type hc = { p : int; mutable compteur : int }
```

pour représenter la proposition P (possiblement $\bot$, qui est codé par 0) et la valeur du *compteur* de la clause. Le tableau *contient* est alors un `hc list array` qui fournit pour chaque proposition Q une liste de structures de type `hc`. L'ensemble E est représenté par un `bool array`, tandis que les ensembles E' et E'' le sont par des `int list`.

```
horn.ml
let hornsat: int list list -> bool array = function clauses ->
  let horns     = List.map horn_of_clause clauses in
  let nprops    = 1 + (List.fold_left max 0
                         (List.map (fun i -> if (i<0) then -i else i)
                            (List.flatten clauses))) in
  let e         = Array.make nprops false (* E := ∅ *) in
  let cur       = ref [] (* E' := ∅ *) in
  let contient = Array.make nprops [] (* contient[Q] := null pour tout Q ∈ fp(F) *)in
  (* initialisation pour chaque clause (P, [Q₁;…;Qₖ]) *)
```

```
let initialise (p, qs) =
  let hc = { p = p; compteur = List.length qs } in
  (* la même paire { p; compteur } va être partagée *)
  List.iter (fun q -> contient.(q) <- (ref hc)::contient.(q)) qs;
  (* si le compteur est initialement nul, ajout de P à E et à E′ *)
  if (hc.compteur = 0) then (e.(p) <- true; cur := p::(!cur))
in
List.iter initialise horns;
(* boucle principale *)
while ((List.length !cur) > 0) do (* tant que E′ ≠ ∅ *)
  let tmp = ref [] (* E″ := ∅ *) in
  List.iter (fun q -> (* pour tout Q ∈ E′ *)
      List.iter (fun hc -> (* pour tout C ∈ contient[Q] *)
          if (not e.(!hc.p)) then (* si P ∉ E *)
            (!hc.compteur <- !hc.compteur - 1; (* compteur[Q] := compteur[Q] − 1 *)
             if (!hc.compteur = 0) then
               (* si le compteur tombe à zéro, ajout de P à E et à E″ *)
               (e.(!hc.p) <- true; tmp := (!hc.p)::(!tmp))))
        contient.(q)) !cur;
  cur := !tmp
done;
e (* retourner E *)
```

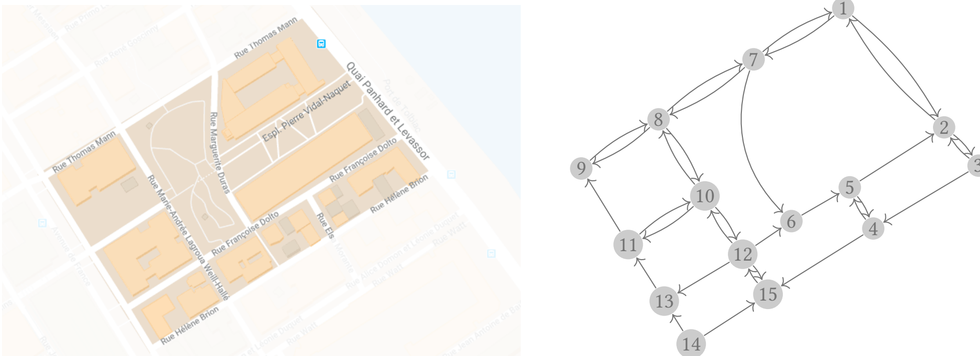


FIGURE 21. Plan du campus[11] et graphe de circulation automobile.

9.3. **Exemple de modélisation : accessibilité dans un graphe orienté.** Commençons par un exemple très simple de modélisation d'un problème sous la forme d'une question de satisfiabilité d'un ensemble de clauses de HORN propositionnelles : celui de l'accessibilité dans un graphe fini orienté. De manière plus concrète, considérons le graphe de la figure 21, qui représente le graphe de circulation automobile autour du campus des Grands Moulins [11]. Le problème informatique que l'on souhaite résoudre est l'existence d'un chemin d'un sommet du graphe à un autre. Par exemple, on peut aller du sommet 14 au sommet 1, mais l'inverse n'est pas possible.

L'idée générale de la modélisation à l'aide de formules propositionnelles est d'utiliser une proposition par sommet du graphe et d'encoder la relation d'accessibilité à l'aide de clauses de HORN. Dans le cas de la figure 21 avec le sommet 1 en guise de source, cela signifie construire l'ensemble de propositions $\{P_1, P_2, \dots, P_{15}\}$ et l'ensemble F suivant de clauses de HORN, qui contient une implication $P_v \Leftarrow P_u$ si et seulement s'il existe un arc (u, v) dans le graphe.

Comme vu dans le cours d'« éléments d'algorithmique », le problème d'accessibilité peut être résolu efficacement par des algorithmes de parcours de graphe.

La complexité du problème d'accessibilité dans un graphe orienté est un problème classique en complexité algorithmique, que vous verrez dans le cours « calculabilité et complexité »; c.f. (PERIFEL, 2014, prop. 4AL) et (ARORA et BARAK, 2009, thm. 4.18).

Cet ensemble F pourrait aussi s'écrire comme un ensemble de clauses utilisent au plus deux littéraux : chaque formule propositionnelle $P_v \Leftarrow P_u$ est équivalente à $\neg P_u \lor P_v$. On parle alors du problème « 2SAT ». Les liens entre accessibilité et 2SAT sont bien connus, voir (CARTON, 2008, thm. 4.47), (PAPADIMITRIOU, 1993, pp. 184–187) ou (KNUTH, 2008, sec. 7.1.1, thm. K).

11. Données géographiques © 2019 Google.

$$F \stackrel{\text{def}}{=} \{ \quad P_2 \Leftarrow P_1, \quad P_7 \Leftarrow P_1, \quad P_1 \Leftarrow P_2, \quad P_3 \Leftarrow P_2, \quad P_2 \Leftarrow P_3, \quad P_4 \Leftarrow P_3,$$
$$P_5 \Leftarrow P_4, \quad P_{15} \Leftarrow P_4, \quad P_2 \Leftarrow P_5, \quad P_4 \Leftarrow P_5, \quad P_5 \Leftarrow P_6, \quad P_1 \Leftarrow P_7,$$
$$P_6 \Leftarrow P_7, \quad P_8 \Leftarrow P_7, \quad P_7 \Leftarrow P_8, \quad P_9 \Leftarrow P_8, \quad P_{10} \Leftarrow P_8, \quad P_8 \Leftarrow P_9,$$
$$P_8 \Leftarrow P_{10}, \quad P_{11} \Leftarrow P_{10}, \quad P_{12} \Leftarrow P_{10}, \quad P_9 \Leftarrow P_{11}, \quad P_{10} \Leftarrow P_{11}, \quad P_6 \Leftarrow P_{12},$$
$$P_{10} \Leftarrow P_{12}, \quad P_{13} \Leftarrow P_{12}, \quad P_{15} \Leftarrow P_{12}, \quad P_{11} \Leftarrow P_{13}, \quad P_{13} \Leftarrow P_{14}, \quad P_{15} \Leftarrow P_{14},$$
$$P_{12} \Leftarrow P_{15}\}$$

Cet ensemble de clauses va nous permettre de répondre au problème d'accessibilité grâce à la propriété suivante.

Propriété 9.14. *Soit $G = (V, E)$ un graphe fini dirigé et $s \in V$ un sommet source. Soit $\{P_v \mid v \in V\}$ l'ensemble des propositions associé à son ensemble de sommets et $F \stackrel{\text{def}}{=} \{P_v \Leftarrow P_u \mid (u, v) \in E\}$ l'ensemble de clauses de Horn associé à son ensemble d'arcs. Alors, pour tout sommet $v \in V$, $F \cup \{P_s\} \models P_v$ si et seulement s'il existe un chemin dirigé de s à v dans G.*

Dès lors, il existe un chemin dirigé d'un sommet source $s \in V$ à un sommet cible $t \in V$ si et seulement si $F \cup \{P_s\} \models P_t$.

Voici par exemple un fichier au format DIMACS qui vérifie que le sommet 14 n'est pas accessible depuis le sommet 1 dans le graphe de la figure 21. En effet, un chemin de s à t existe si et seulement si $F \cup \{P_s\} \models P_t$ d'après la propriété 9.14, si et seulement si $F \cup \{P_s\} \cup \{\neg P_t\}$ est insatisfiable par le lemme 4.2 de déduction.

accessibilite.cnf

```
p cnf 15 33
c  arcs du graphe
 -1   2 0
 -1   7 0
 -2   1 0
 -2   3 0
 -3   2 0
 -3   4 0
 -4   5 0
 -4  15 0
 -5   2 0
 -5   4 0
 -6   5 0
 -7   1 0
 -7   8 0
 -7   6 0
 -8   7 0
 -8   9 0
 -8  10 0
 -9   8 0
-10   8 0
-10  11 0
-10  12 0
-11   9 0
-11  10 0
-12   6 0
-12  10 0
-12  13 0
-12  15 0
-13  11 0
-14  13 0
```

```
-14 15 0
-15 12 0
c  sommet source
1 0
c  sommet destination
-14 0
```

En général, pour un graphe fini dirigé $G = (V, E)$, un sommet source $s \in V$ et un sommet cible $t \in V$, il existe un chemin dirigé de s à t si et seulement si la formule propositionnelle suivante n'est pas satisfiable :

$$P_s \land \neg P_t \land \bigwedge_{(u,v)\in E} (\neg P_u \lor P_v) \ . \tag{10}$$

Partie 3. Logique classique du premier ordre

La logique propositionnelle de la partie 2 est limitée à l'expression de propriétés de propositions booléennes (interprétées comme vraies ou fausses). Nous allons l'étendre et étudier la logique du premier ordre (aussi appelée « calcul des prédicats »). Outre les connecteurs booléens déjà rencontrés ($\land$, $\lor$, $\neg$, $\Rightarrow$, …), la logique du premier ordre permet de quantifier ($\forall$, $\exists$) sur des éléments et dispose d'un vocabulaire enrichi par des symboles auxiliaires de fonction et de relation. Par exemple, les formules (1), (2) et (3) de l'introduction étaient des formules de la logique du premier ordre, où les symboles H, M, V, A et R étaient des symboles de relation.

10. Structures

Résumé. Une *signature* du premier ordre $L = (\mathcal{F}, \mathcal{P})$ est constituée de symboles de fonction $f \in \mathcal{F}$ et de symboles de relation $R \in \mathcal{P}$. On associe une *arité* dans $\mathbb{N}$ à chaque symbole et on note « $\mathcal{F}_n$ » l'ensemble des symboles de fonction d'arité n et « $\mathcal{P}_m$ » l'ensemble des symboles de relation d'arité m ; une fonction d'arité zéro est une *constante* et une relation d'arité zéro est une *proposition*.

Une *interprétation* I d'une signature $(\mathcal{F}, \mathcal{P})$ est définie par un domaine D_I non vide, d'une fonction $f^I : D_I^n \to D_I$ pour tout $f \in \mathcal{F}_n$ et tout n et d'une relation $R^I : D_I^m \to \mathbb{B}$ pour tout $R \in \mathcal{P}_m$ et tout m.

Les formules du premier ordre sont interprétées sur des structures très générales, utilisables pour modéliser des structures mathématiques (ordres, groupes, corps, etc.) ou des structures relationnelles comme utilisées en bases de données.

10.1. **Signatures.** On définit tout d'abord une *signature* du premier ordre (aussi appelée un « langage du premier ordre ») comme une paire $L = (\mathcal{F}, \mathcal{P})$ formée de deux ensembles dénombrables $\mathcal{F}$ de symboles de fonction et $\mathcal{P} \neq \emptyset$ de symboles de relation (aussi appelés « symboles de prédicat »), avec $\mathcal{F} \cap \mathcal{P} = \emptyset$, et tous deux munis d'une fonction d'*arité* $r : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$. On note $\mathcal{F}_n \stackrel{\text{def}}{=} \{f \in \mathcal{F} \mid r(f) = n\}$ et $\mathcal{P}_n \stackrel{\text{def}}{=} \{R \in \mathcal{P} \mid r(R) = n\}$ leurs restrictions aux symboles d'arité n. Une fonction d'arité zéro est appelée une *constante* ; une relation d'arité zéro est appelée une *proposition*.

On peut aussi travailler avec des signatures non dénombrables, mais les preuves nécessitent alors d'utiliser des notions de théorie des ensembles comme le lemme de Zorn.

10.2. **Interprétations.** Une *interprétation* I d'une signature $L = (\mathcal{F}, \mathcal{P})$ (aussi appelée une « L-structure ») est constituée d'un ensemble $D_I \neq \emptyset$, appelé son domaine ou son support, d'une fonction $f^I : D_I^{r(f)} \to D_I$ pour chaque $f \in \mathcal{F}$, et d'une relation $R^I : D_I^{r(R)} \to \mathbb{B}$ pour chaque $R \in \mathcal{P}$. Une telle interprétation est notée $I = (D_I, (f^I)_{f \in \mathcal{F}}, (R^I)_{R \in \mathcal{P}})$; dans le cas où $=^I$ est l'égalité sur D_I, on l'omet dans cette notation.

(Duparc, 2015, sec. 11.1)

L'ensemble des formules valides change si l'on permet un domaine vide.

En théorie des modèles et en théorie des modèles finis (Chang et Keisler, 1990 ; Ebbinghaus, Flum et Thomas, 1994 ; Libkin, 2004), le symbole d'égalité est toujours présent et interprété comme l'égalité sur D_I ; voir la section 15.1.3.

Exemple 10.1 (interprétations propositionnelles)**.** Posons $\mathcal{F} \stackrel{\text{def}}{=} \emptyset$ et $\mathcal{P} \stackrel{\text{def}}{=} \{P^{(0)}, Q^{(0)}\}$. Dans cette signature, il n'y a pas de symbole de fonction, et il y a deux symboles de relation P et Q, tous deux d'arité 0. Une interprétation possible pour cette signature est I de domaine $D_I \stackrel{\text{def}}{=} \{\bullet\}$ avec $P^I \stackrel{\text{def}}{=} 1$ et $Q^I \stackrel{\text{def}}{=} 0$.

Exemple 10.2 (ordres)**.** Posons $\mathcal{F} \stackrel{\text{def}}{=} \emptyset$ et $\mathcal{P} \stackrel{\text{def}}{=} \{<^{(2)}, =^{(2)}\}$ où l'exposant « $^{(2)}$ » indique un symbole d'arité 2. Dans cette signature, il n'y a pas de symboles de fonction, et il y a deux symboles de relation d'arité 2. Une interprétation $I = (\mathbb{Q}, <)$ pour cette signature est définie par $D_I \stackrel{\text{def}}{=} \mathbb{Q}$, $<^I \stackrel{\text{def}}{=} <$ l'ordre habituel sur les rationnels et $=^I$ l'égalité sur les rationnels.

Exemple 10.3 (arithmétique)**.** Posons $\mathcal{F} \stackrel{\text{def}}{=} \{+^{(2)}, \times^{(2)}\}$ et $\mathcal{P} \stackrel{\text{def}}{=} \{=^{(2)}\}$. Cette signature comprend deux fonctions binaires $+$ et $\times$, et une relation binaire $=$. Une interprétation $I = (\mathbb{N}, +, \times)$ pour cette signature est définie par $D_I \stackrel{\text{def}}{=} \mathbb{N}$, $+^I\colon (n,m) \mapsto n+m$, $\times^I\colon (n,m) \mapsto nm$ et $=^I$ l'égalité sur $\mathbb{N}$.

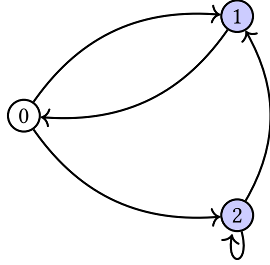


Figure 22. Un graphe fini orienté.

Exemple 10.4 (graphe fini orienté)**.** Considérons le graphe fini orienté avec des sommets coloriés de la figure 22 et voyons comment le décrire comme une structure sur une signature appropriée. On pose pour cela $\mathcal{F} \stackrel{\text{def}}{=} \emptyset$ et $\mathcal{P} \stackrel{\text{def}}{=} \{E^{(2)}, =^{(2)}, B^{(1)}\}$ où E va représenter la relation d'adjacence du graphe et B indiquer pour chaque sommet s'il est bleu ou non. Le graphe de la figure 22 correspond alors à l'interprétation I de domaine $D_I \stackrel{\text{def}}{=} \{0,1,2\}$ avec $E^I \stackrel{\text{def}}{=} \{(0,1),(0,2),(1,0),(2,1),(2,2)\}$, $B^I \stackrel{\text{def}}{=} \{1,2\}$, et $=^I$ l'égalité entre sommets.

Exemple 10.5 (base de donnée relationnelle)**.** Posons $L \stackrel{\text{def}}{=} \{\mathcal{F}, \mathcal{P})$ où

$$\mathcal{F} \stackrel{\text{def}}{=} \{\textit{shining}^{(0)}, \textit{player}^{(0)}, \textit{easyrider}^{(0)}, \textit{apocalypsenow}^{(0)}, \textit{kubrick}^{(0)}, \textit{altman}^{(0)}, \textit{hopper}^{(0)}, \textit{nicholson}^{(0)}, \textit{robbins}^{(0)}, \textit{coppola}^{(0)}, \textit{champo}^{(0)}, \textit{odeon}^{(0)}\}$$

et

$$\mathcal{P} \stackrel{\text{def}}{=} \{\textit{Film}^{(3)}, \textit{Seance}^{(2)}, =^{(2)}\}\ .$$

Une interprétation I possible pour cette signature a pour domaine

$$D_I \stackrel{\text{def}}{=} \{\textit{Shining}, \textit{The Player}, \textit{Easy Rider}, \textit{Apocalypse Now}, \textsc{Kubrick}, \textsc{Altman}, \textsc{Hopper}, \textsc{Nicholson}, \textsc{Robbins}, \textsc{Hopper}, \text{Le Champo}, \text{Odéon}\}$$

où les constantes sont interprétées de manière évidente, et

$$\textit{Film}^I \stackrel{\text{def}}{=} \{(\textit{Shining}, \textsc{Kubrick}, \textsc{Nicholson}),\ (\textit{The Player}, \textsc{Altman}, \textsc{Robbins}), (\textit{Easy Rider}, \textsc{Hopper}, \textsc{Nicholson}),\ (\textit{Easy Rider}, \textsc{Hopper}, \textsc{Hopper}), (\textit{Apocalypse Now}, \textsc{Coppola}, \textsc{Hopper})\}\ ,$$

$$\textit{Seance}^I \stackrel{\text{def}}{=} \{(\text{Le Champo}, \textit{Shining}),\ (\text{Le Champo}, \textit{Easy Rider}),\ (\text{Le Champo}, \textit{The Player}), (\text{Odéon}, \textit{Easy Rider})\}$$

et $=^I$ est l'égalité sur D_I. Cette interprétation correspond aux tables « Films » et « Séances » de la base de donnée ci-après.

💬 *En général, une base de donnée relationnelle peut-être vue comme une interprétation de domaine fini sur une signature dotée des symboles de constantes adéquats, d'un symbole de relation par table, et du symbole d'égalité ; on appelle cela une* signature relationnelle.

	Films	
titre	réalisation	interprète
Shining	Kubrick	Nicholson
The Player	Altman	Robbins
Easy Rider	Hopper	Nicholson
Easy Rider	Hopper	Hopper
Apocalypse Now	Coppola	Hopper

Séances	
cinéma	titre
Le Champo	*Shining*
Le Champo	*Easy Rider*
Le Champo	*The Player*
Odéon	*Easy Rider*

10.3. **Représentation des interprétations en OCaml.** Voyons comment on pourrait représenter les interprétations en OCaml. Une manière peu élégante mais qui a le mérite d'être simple est de se dire qu'une interprétation I prend un symbole de relation R ou un symbole de fonction f vu comme une chaîne de caractères, ainsi que n éléments du domaine D_I sous la forme d'une liste, et retourne soit une valeur booléenne (dans le cas de R) soit un élément du domaine (dans le cas de f). Si le type générique `'e` désigne le type des éléments du domaine D_I, une interprétation contient donc une fonction de type `string -> 'e list -> bool` qui interprète les symboles de relation et une fonction de type `string -> 'e list -> 'e` qui interprète les symboles de fonction.

Pour pouvoir implémenter la sémantique plus tard dans la section 12.3, nous allons aussi avoir besoin d'une manière d'énumérer les éléments du domaine D_I. Quand ce domaine est fini, cela pourrait prendre la forme d'une liste de type `'e list`, mais quand D_I est infini cela ne convient guère. Il y a en fait un type de donnée pour des séquences (potentiellement) infinies dans la bibliothèque standard d'OCaml : le module Seq. Une séquence d'éléments de type `'e` est alors de type `'e t`.

Nos interprétations seront donc des triplets comprenant la séquence des éléments du domaine (de type `'e t`), l'interprétation des symboles de relation (de type `string -> 'e list -> bool`), et l'interprétation des symboles de fonction (de type `string -> 'e list -> 'e`).

```
firstorder.ml
type 'e interpretation =
  'e t * (string -> 'e list -> bool) * (string -> 'e list -> 'e)
```

Prenons l'exemple 10.1. Pour représenter $D_I = \{\bullet\}$, on peut utiliser `string` en guise de type `'e` d'éléments et construire la séquence qui ne contient que la chaîne de caractères "•" à l'aide de la fonction `List.to_seq`. L'interprétation des symboles de relation ne doit s'inquiéter que de P et Q, tandis qu'il n'y a pas de symbole de fonction.

```
firstorder.ml
let props : string interpretation =
  (to_seq ["•"],
   (fun s l -> match (s,l) with
              | ("P", []) -> true
              | ("Q", []) -> false
              | _         -> failwith "erreur dans l'interprétation"),
   (fun _                 -> failwith "erreur dans l'interprétation"))
```

L'exemple 10.4 est un autre exemple d'interprétation I de domaine fini, qui est l'ensemble des sommets d'un graphe dirigé. On peut représenter les éléments de D_I comme des entiers, et voici la représentation de l'interprétation en OCaml.

```
firstorder.ml
let graph_10_4 : int interpretation =
  (to_seq [0;1;2],
```

```
  (fun s l -> match (s,l) with
             | ("=", [u;v]) -> u = v
             | ("E", [u;v]) -> List.mem (u,v) [(0,1);(0,2);(1,0);(2,1);(2,2)]
             | ("B", [u])   -> List.mem u     [1;2]
             | _            -> failwith "erreur dans l'interprétation"),
  (fun _                    -> failwith "erreur dans l'interprétation"))
```

Un dernier exemple d'interprétation avec un domaine fini est celui de la base de données relationnelle de l'exemple 10.5. Dans ce cas, on va représenter les éléments du domaine comme des `string`.

firstorder.ml

```
let database : string interpretation =
  let domain_list = ["Shining";"The Player";"Easy Rider";"Apocalypse Now";
                     "Kubrick";"Altman";"Hopper";"Nicholson";"Robbins";
                     "Coppola";"Le Champo";"Odéon"] in
  (to_seq domain_list,
   (fun s l -> match (s,l) with
             | ("=", [s;t])      -> s = t
             | ("Film", [t;r;i]) -> List.mem (t,r,i)
                                    [("Shining","Kubrick","Nicholson");
                                     ("The Player","Altman","Robbins");
                                     ("Easy Rider","Hopper","Nicholson");
                                     ("Easy Rider","Hopper","Hopper");
                                     ("Apocalypse Now","Coppola","Hopper")]
             | ("Seance", [c;t]) -> List.mem (c,t)
                                    [("Le Champo","Shining");
                                     ("Le Champo","Easy Rider");
                                     ("Le Champo","The Player");
                                     ("Odéon","Easy Rider")]
             | _                 -> failwith "erreur dans l'interprétation"),
   (fun s l -> match (s, l) with
             | (s, []) when (List.mem s domain_list) -> s
             | _                 -> failwith "erreur dans l'interprétation"))
```

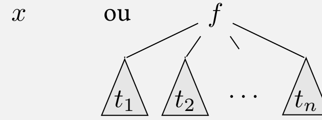
Pour un exemple d'interprétation sur un domaine infini, voyons le cas de l'arithmétique sur les entiers naturels décrit dans l'exemple 10.3. On a $D_I = \mathbb{N}$ et on peut représenter ces entiers naturels comme des `int`. Une énumération des entiers naturels peut être implémentée à l'aide de la fonction `Seq.unfold`.

firstorder.ml

```
let naturals = unfold (fun i -> if (i < 0) then None else Some(i, i+1)) 0
let natarith : int interpretation =
  (naturals,
   (fun s l -> match (s,l) with
             | ("=", [x;y]) -> x = y
             | _            -> failwith "erreur dans l'interprétation"),
   (fun s l -> match (s,l) with
             | ("+", [i;j]) -> (i + j)
             | ("·", [i;j]) -> (i * j)
             | _            -> failwith "erreur dans l'interprétation"))
```

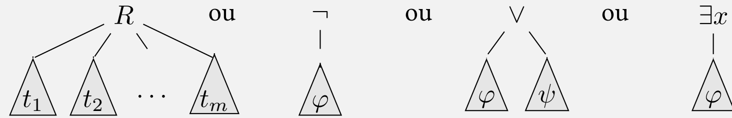
11. Syntaxe

Résumé. Fixons une signature $L = (\mathcal{F}, \mathcal{P})$ et un ensemble infini dénombrable de *variables* X. L'ensemble $T(\mathcal{F}, X)$ des *termes* sur $\mathcal{F}$ et X est l'ensemble des arbres de la forme



où $x \in X$, $n \in \mathbb{N}$, $f \in \mathcal{F}_n$ et $t_1, t_2, \ldots, t_n$ sont des termes.

Une *formule* est un arbre de la forme



où $m \in \mathbb{N}$, $R \in \mathcal{P}_m$, $t_1, t_2, \ldots, t_m \in T(\mathcal{F}, X)$, $x \in X$ et φ et ψ sont des formules. Une formule de la forme « $R(t_1, t_2, \ldots, t_m)$ » est dite *atomique*.

Un terme sans variable est un *terme clos* et on note « $T(\mathcal{F})$ » pour l'ensemble des termes clos sur $\mathcal{F}$. Une variable x qui apparaît dans une formule mais pas sous un quantificateur $\exists x$ est dite *libre* ; sinon elle est *liée*. On note « $\text{fv}(\varphi)$ » pour l'ensemble des variables libres de φ et « $\text{bv}(\varphi)$ » pour son ensemble de variables liées. Une formule sans variable libre est dite *close*.

(Duparc, 2015, sec. 10.3), (David, Nour et Raffalli, 2003, sec. 1.2), (Goubault-Larrecq et Mackie, 1997, sec. 6.1), (Harrisson, 2009, sec. 3.1).

La logique propositionnelle *(aussi appelée « calcul des propositions ») de la partie 2 est obtenue en posant $\mathcal{F} = \emptyset$ et $\mathcal{P} = \mathcal{P}_0$, et en interdisant la quantification dans la syntaxe des formules.*

11.1. **Formules.** Soit X un ensemble infini dénombrable de symboles de variables. Les formules de la logique du premier ordre sur une signature $L = (\mathcal{F}, \mathcal{P})$ (aussi appelées « L-formules ») sont définies par la syntaxe abstraite

$$
\begin{aligned}
t &::= x \mid f(t_1, \ldots, t_m) && \text{(termes)}\\
\alpha &::= R(t_1, \ldots, t_m) && \text{(formules atomiques)}\\
\varphi &::= \alpha \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi && \text{(formules)}
\end{aligned}
$$

où $x \in X$, $m \in \mathbb{N}$, $f \in \mathcal{F}_m$ et $R \in \mathcal{P}_m$. L'ensemble des termes sur X et $\mathcal{F}$ est aussi dénoté $T(\mathcal{F}, X)$. Un *littéral* est une formule atomique α ou sa négation $\neg\alpha$.

Comme dans le cas de la logique propositionnelle, on peut ajouter d'autres opérateurs booléens à cette syntaxe minimale. Alternativement, ces symboles peuvent être définis dans la logique :

$$
\varphi \wedge \psi \stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi) \qquad \forall x.\varphi \stackrel{\text{def}}{=} \neg\exists x.\neg\varphi \qquad \varphi \Rightarrow \psi \stackrel{\text{def}}{=} \neg\varphi \vee \psi
$$

$$
\top \stackrel{\text{def}}{=} \forall x.R(x, \ldots, x) \vee \neg R(x, \ldots, x) \qquad \bot \stackrel{\text{def}}{=} \neg\top
$$

où R est un symbole arbitraire de $\mathcal{P}$ (qui est supposé non vide).

Comme pour les formules propositionnelles, les formules de la logique du premier ordre sont donc des arbres de syntaxe abstraite. Par exemple, la figure 23 décrit la formule du buveur $\exists x.(B(x) \Rightarrow \forall y.B(y))$. On a matérialisé dans cette figure les liens (en pointillés rouges) entre quantification et occurrences de chaque variable.

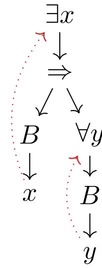


FIGURE 23. L'arbre de syntaxe abstraite de la formule du buveur $\exists x.(B(x) \Rightarrow \forall y.B(y))$.

11.2. **Variables libres et variables liées.** L'ensemble $\mathrm{fv}(t)$ des *variables libres* (« *free variables* » en anglais) d'un terme t est défini inductivement par

(DUPARC, 2015, sec. 10.6), (GOUBAULT-LARRECQ et MACKIE, 1997, def. 6.3)

$$\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\} , \qquad \mathrm{fv}(f(t_1, \ldots, t_m)) \stackrel{\text{def}}{=} \bigcup_{1 \le i \le m} \mathrm{fv}(t_i) .$$

Un terme t sans variable libre (*i.e.* $\mathrm{fv}(t) = \emptyset$) est dit *clos* ; l'ensemble des termes clos est noté $T(\mathcal{F})$.

Les ensembles $\mathrm{fv}(\varphi)$ des *variables libres* et $\mathrm{bv}(\varphi)$ des *variables liées* (« *bound variables* » en anglais) d'une formule φ sont définis inductivement par

$$\begin{aligned}
\mathrm{fv}(R(t_1, \ldots, t_m)) &\stackrel{\text{def}}{=} \bigcup_{1 \le i \le m} \mathrm{fv}(t_i) , & \mathrm{bv}(R(t_1, \ldots, t_m)) &\stackrel{\text{def}}{=} \emptyset , \\
\mathrm{fv}(\neg\varphi) &\stackrel{\text{def}}{=} \mathrm{fv}(\varphi) , & \mathrm{bv}(\neg\varphi) &\stackrel{\text{def}}{=} \mathrm{bv}(\varphi) \\
\mathrm{fv}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \mathrm{fv}(\varphi) \cup \mathrm{fv}(\psi) , & \mathrm{bv}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \mathrm{bv}(\varphi) \cup \mathrm{bv}(\psi) , \\
\mathrm{fv}(\exists x.\varphi) &\stackrel{\text{def}}{=} \mathrm{fv}(\varphi) \setminus \{x\} & \mathrm{bv}(\exists x.\varphi) &\stackrel{\text{def}}{=} \{x\} \cup \mathrm{bv}(\varphi) .
\end{aligned}$$

Une formule φ sans variable libre (c'est-à-dire telle que $\mathrm{fv}(\varphi) = \emptyset$) est dite *close*. Par exemple, la formule $\exists x.(B(x) \Rightarrow \forall y.B(y))$ représentée dans la figure 23 est close. Si φ est le nom d'une formule qui n'est pas close, on note couramment « $\varphi(x_1, \ldots, x_n)$ » pour expliciter que $x_1, \ldots, x_n$ est une énumération des variables de $\mathrm{fv}(\varphi) = \{x_1, \ldots, x_n\}$, voir exemples 12.5 et 13.6 pour des exemples d'utilisation de cette notation.

11.3. **Syntaxe en OCaml.** Une représentation possible des termes et des formules en OCaml est d'employer des chaînes de caractères pour les symboles de fonction et les symboles de relation.

11.3.1. *Syntaxe abstraite en OCaml.* Voici les types OCaml pour les termes et les formules. Les symboles de fonction $f \in \mathcal{F}$, de relation $R \in \mathcal{P}$ et de variables $x \in X$ sont tous représentés à l'aide de chaînes de caractères.

firstorder.ml
```
type term = Var of string | Fun of string * term list
type atom = string * term list
type fo = Atom of atom | Non of fo | Ou of fo * fo | Ex of string * fo
```

Avec ces déclarations de type, la formule du buveur $\exists x.(B(x) \Rightarrow \forall y.B(y))$ représentée dans la figure 23 s'écrit en syntaxe restreinte comme $\exists x.(\neg B(x) \vee \neg(\exists y.\neg B(y)))$ et se définit en OCaml comme suit :

firstorder.ml
```
let buveur = Ex ("x",
                 Ou (Non (Atom ("B", [Var "x"])),
                     Non (Ex ("y", Non (Atom ("B", [Var "y"]))))))
```

11.3.2. *Syntaxe concrète en OCaml.* Le passage d'une formule, définie par son arbre de syntaxe abstraite, à sa syntaxe concrète sous forme d'une chaîne de caractères se fait par récursion sur les termes et formules.

```
firstorder.ml
let rec string_of_term = function
    Var x -> x
  | Fun (f, tl)   -> f^(string_of_terml tl)
and string_of_terml = function
    []   -> ""
  | [t]  -> "("^(string_of_term t)^")"
  | t::l -> (List.fold_left
                      (fun s t' -> s^", "^(string_of_term t'))
                      ("("^(string_of_term t)) l)^")"
and string_of_fo = function
    Atom (r, tl) -> r^(string_of_terml tl)
  | Non (phi)    -> "¬"^(string_of_fo phi)
  | Ou (phi,psi) -> "("^(string_of_fo phi)^" ∨ "^(string_of_fo psi)^")"
  | Ex (x, phi)  -> "(∃"^x^"."^(string_of_fo phi)^")"
```

11.3.3. *Variables libres et variables liées en OCaml.* Le calcul des ensembles de variables libres et de variables liées se fait par récurrence sur les termes et formules.

```
firstorder.ml
let rec aux_tv t vars = match t with
  | Var x -> x::vars
  | Fun (_,tl) -> List.fold_left (fun vs t' -> aux_tv t' vs) vars tl
and aux_fv phi vars = match phi with
  | Atom(_, tl) -> List.fold_left (fun vs t' -> aux_tv t' vs) vars tl
  | Non(phi')    -> aux_fv phi' vars
  | Ou(phi',psi)-> aux_fv phi' (aux_fv psi vars)
  | Ex(x, phi') -> List.filter (fun y -> (x <> y)) (aux_fv phi' vars)
and aux_bv phi vars = match phi with
  | Atom(_, _) -> vars
  | Non(phi')    -> aux_bv phi' vars
  | Ou(phi',psi)-> aux_bv phi' (aux_bv psi vars)
  | Ex(x, phi') -> x::(aux_bv phi' vars)

let fv phi = List.sort_uniq compare (aux_fv phi [])
and bv phi = List.sort_uniq compare (aux_bv phi [])
```

12. Sémantique

Résumé. Soit $L = (\mathcal{F}, \mathcal{P})$ une signature du premier ordre et $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$ l'ensemble des *valeurs de vérité*, où 0 désigne « faux » et 1 désigne « vrai ». Étant donnée une *interprétation* I de L et une *valuation* $\rho\colon X \to D_I$ des variables,

- la *sémantique* d'un terme t est un élément $[\![t]\!]^I_\rho \in D_I$ et
- la *sémantique* d'une formule φ est une valeur de vérité $[\![\varphi]\!]^I_\rho \in \mathbb{B}$.

Dans les deux cas, ces sémantiques ne dépendent que de la valuation des variables libres de t et φ (propriété 12.1). On note « $I, \rho \vDash \varphi$ » si $[\![\varphi]\!]^I_\rho = 1$.

Une formule est *satisfiable* s'il existe une interprétation I et une valuation ρ telles que $I, \rho \vDash \varphi$. Une interprétation I est un *modèle* d'une formule φ (noté « $I \vDash \varphi$ ») si pour toute valuation ρ, on a $I, \rho \vDash \varphi$. Une formule φ est une *conséquence logique* d'un ensemble de formules S (noté « $S \vDash \varphi$ ») si, pour toute interprétation I et pour toute valuation ρ telles que $I, \rho \vDash \psi$ pour toute formule $\psi \in S$, on a $I, \rho \vDash \varphi$. Enfin, une formule φ est *valide* (noté « $\vDash \varphi$ ») si pour toute interprétation I et toute valuation ρ, on a $I, \rho \vDash \varphi$.

Fixons $L = (\mathcal{F}, \mathcal{P})$ une signature du premier ordre. On note $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$ pour l'ensemble des *valeurs de vérité* (aussi appelé « algèbre de Boole »), muni des opérations $\text{not} : \mathbb{B} \to \mathbb{B}$ et $\text{or}\colon \mathbb{B}^2 \to \mathbb{B}$, définies par $\text{not}\, 1 = 0 \text{ or } 0 = 0$ et $\text{not}\, 0 = 1 \text{ or } 1 = 1 \text{ or } 0 = 0 \text{ or } 1 = 1$. Pour une famille $(v_j)_{j \in J}$ de valeurs de vérité $v_j \in \mathbb{B}$ indexée par un ensemble J, on définit aussi la disjonction distribuée par $\text{Or}_{j \in J}\, v_j = 1$ si et seulement s'il existe $j \in J$ tel que $v_j = 1$.

(David, Nour et Raffalli, 2003, sec. 2.2), (Goubault-Larrecq et Mackie, 1997, sec. 6.2), (Harrisson, 2009, sec. 3.3).

12.1. **Sémantique.** Une *valuation* dans I est une fonction $\rho\colon X \to D_I$. On notera $\rho[e/x]$ pour la valuation qui associe e à x et $\rho(y)$ à $y \neq x$. Pour un terme t, sa sémantique $[\![t]\!]^I_\rho$ dans une interprétation I pour une valuation ρ est un élément de D_I défini inductivement par

voir aussi (Duparc, 2015, sec. 11.3) pour une définition de la sémantique en termes de jeux

$$[\![x]\!]^I_\rho \stackrel{\text{def}}{=} \rho(x)\ , \qquad [\![f(t_1, \dots, t_m)]\!] \stackrel{\text{def}}{=} f^I([\![t_1]\!]^I_\rho, \dots, [\![t_m]\!]^I_\rho)$$

pour tout m et tout $f \in \mathcal{F}_m$.

La sémantique $[\![\varphi]\!]^I_\rho$ d'une formule du premier ordre φ dans une interprétation I pour une valuation ρ est une valeur de vérité dans $\mathbb{B}$ définie inductivement par

$$[\![R(t_1, \dots, t_m)]\!]^I_\rho \stackrel{\text{def}}{=} R^I([\![t_1]\!]^I_\rho, \dots, [\![t_m]\!]^I_\rho)\ , \qquad [\![\neg\varphi]\!]^I_\rho \stackrel{\text{def}}{=} \text{not}\, [\![\varphi]\!]^I_\rho\ ,$$
$$[\![\varphi \vee \psi]\!]^I_\rho \stackrel{\text{def}}{=} [\![\varphi]\!]^I_\rho \text{ or } [\![\psi]\!]^I_\rho\ , \qquad [\![\exists x.\varphi]\!]_\rho \stackrel{\text{def}}{=} \underset{e \in D_I}{\text{Or}}\, [\![\varphi]\!]^I_{\rho[e/x]}\ .$$

On dit que (I, ρ) *satisfait* φ, noté $I, \rho \vDash \varphi$, si $[\![\varphi]\!]^I_\rho = 1$; cette écriture peut être définie de manière équivalente par

$$\begin{aligned}
&I, \rho \vDash R(t_1, \dots, t_m) && \text{si } ([\![t_1]\!]^I_\rho, \dots, [\![t_m]\!]^I_\rho) \in R^I\ ,\\
&I, \rho \vDash \neg\varphi && \text{si } I, \rho \nvDash \varphi\ ,\\
&I, \rho \vDash \varphi \vee \psi && \text{si } I, \rho \vDash \varphi \text{ ou } I, \rho \vDash \psi\ ,\\
&I, \rho \vDash \exists x.\varphi && \text{si } \exists e \in D_I . I, \rho[e/x] \vDash \varphi\ .
\end{aligned}$$

12.2. **Modèles, satisfiabilité et validité.** Une formule φ est *satisfiable* s'il existe une interprétation I et une valuation ρ telle que $I, \rho \vDash \varphi$. On dit que I est un *modèle* de φ, noté $I \vDash \varphi$, si pour toute valuation ρ, $I, \rho \vDash \varphi$. Une formule φ est *valide*, ce qui est noté $\vDash \varphi$, si pour toute interprétation I et toute valuation ρ, $I, \rho \vDash \varphi$; autrement dit, φ est valide si pour toute interprétation I, I est un modèle de φ.

Pour un ensemble de formules S, une interprétation I et une valuation ρ, on écrit $I, \rho \vDash S$ si $I, \rho \vDash \psi$ pour toutes les formules $\psi \in S$. Si de plus φ est une formule, on écrit $S \vDash \varphi$ si pour

toute paire (I, ρ) telle que $I, \rho \models S$ on a $I, \rho \models \varphi$. Un ensemble S de formules est *insatisfiable* s'il n'existe pas I et ρ telles que $I, \rho \models S$, ou de manière équivalente si $S \models \bot$.

Notons que la satisfaction d'une formule ne dépend que de la valuation de ses variables libres.

Propriété 12.1. *Pour toute formule φ (resp. terme t), toute interprétation I, et toutes valuations ρ et ρ' telles que pour tout $x \in \text{fv}(\varphi)$ (resp. $x \in \text{fv}(t)$) $\rho(x) = \rho'(x)$, $[\![\varphi]\!]^I_\rho = [\![\varphi]\!]^I_{\rho'}$ (resp. $[\![t]\!]^I_\rho = [\![t]\!]^I_{\rho'}$).*

Démonstration. Par induction structurelle sur φ (resp. t). □

En particulier, par la propriété 12.1, φ avec $\text{fv}(\varphi) = \{x_1, \dots, x_n\}$ est satisfiable si et seulement la formule close $\exists x_1 \dots \exists x_n.\varphi$ est satisfiable, et φ est valide si et seulement si la formule close $\forall x_1 \dots \forall x_n.\varphi$ est valide.

12.3. **Implémentation de la sémantique en OCaml.** On a déjà vu comment définir le type `'e interpretation` dans la section 10.3 pour une interprétation I dans laquelle les éléments du domaine D_I ont le type générique `'e`.

Une valuation est une fonction $\rho\colon X \to D_I$. Comme nous avons représenté les variables $x \in X$ comme des chaînes de caractères, une valuation va donc être une fonction de type `string -> 'e`.

firstorder.ml
```
type 'e valuation = string -> 'e
```

La mise à jour $\rho[e/x]$ d'une valuation ρ va servir pour la sémantique des formules de la forme $\exists x.\varphi$.

firstorder.ml
```
let upd: 'e valuation -> 'e -> string -> 'e valuation =
  fun rho e x -> (fun y -> if (x = y) then e else rho y)
```

La sémantique $[\![t]\!]^I_\rho$ d'un terme t dans une interprétation I et pour une valuation ρ des variables peut maintenant s'implémenter par

firstorder.ml
```
let rec tevalue: 'e interpretation -> 'e valuation -> term -> 'e =
  fun ((_,_,funs) as i) rho t ->
  match t with
  | Var x      -> rho x
  | Fun(f,tl) -> funs f (List.map (fun t -> tevalue i rho t) tl)
```

Pour la sémantique des formules du premier ordre, il va nous falloir une implémentation de l'opérateur $\text{Or}_{e \in D_I}$. C'est cette implémentation qui a motivé l'ajout d'une séquence de type `'e t` (qui est un type défini dans le module Seq) à nos interprétations pour énumérer les éléments du domaine D_I. Voici son code OCaml, qui parcourt les éléments x de la séquence et retourne `true` dès que `predicate x` est vrai, et retourne `false` si la séquence termine avant cela. À noter que cette fonction ne termine pas si la séquence est infinie et le prédicat est faux sur tous ses éléments.

firstorder.ml
```
let rec exists: ('e -> bool) -> 'e t -> bool = fun predicate domain ->
  match domain () with
  | Nil -> false
  | Cons (x, next) -> if (predicate x)
                      then true
                      else exists predicate next
```

La sémantique $[\![\varphi]\!]^I_\rho$ d'une formule du premier ordre φ dans une interprétation I et pour une valuation ρ des variables peut ensuite s'implémenter par

firstorder.ml
```
let rec evalue:'e interpretation -> 'e valuation -> fo -> bool =
  fun ((dom,rels,funs) as i) rho phi ->
  match phi with
  | Atom(r, tl) -> rels r (List.map (fun t -> tevalue i rho t) tl)
  | Non(phi')   -> not(evalue i rho phi')
  | Ou(phi',psi)-> ((evalue i rho phi') || (evalue i rho psi))
  | Ex(x, phi') -> exists (fun e -> evalue i (upd rho e x) phi') dom
```

12.4. **Exemples de formules.** Voici quelques exemples de formules du premier ordre.

Exemple 12.2 (loi de Peirce)**.** Prenons la signature $L = (\emptyset, \{P^{(0)}, Q^{(0)}\}$ de l'exemple 10.1. La *formule de Peirce* $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ déjà présentée dans l'exemple 3.10 est une formule du premier ordre sur cette signature. Elle est encore valide : pour toute interprétation I et pour toute valuation ρ, soit $P^I = 1$ et alors $I, \rho \vDash P$ et donc $I, \rho \vDash ((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$, soit $P^I = 0$ et alors $I, \rho \nvDash P$ donc $I, \rho \vDash P \Rightarrow Q$, donc $I, \rho \nvDash (P \Rightarrow Q) \Rightarrow P$, et enfin $I, \rho \nvDash ((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$.

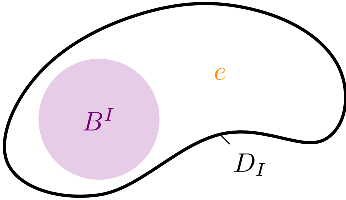
Exemple 12.3 (formule du buveur)**.** Considérons la signature $L = (\emptyset, \{B^{(1)}\})$ avec une relation unaire B. La formule $\exists x.(B(x) \Rightarrow \forall y.B(y))$, qui se lit habituellement « il existe un individu tel que s'il boit alors tout le monde boit », est valide.

On notera que cette formule n'est pas valide si l'on permet un domaine d'interprétation vide. Elle n'est pas non plus valide en logique intuitionniste du premier ordre.

Nous allons montrer que dans toute interprétation I et pour toute valuation ρ, $I, \rho \vDash \exists x.(B(x) \Rightarrow \forall y.B(y))$. Cela revient à montrer que pour toute interprétation I, il existe un élément $e \in D_I$ tel que $I, \rho[e/x] \vDash B(x) \Rightarrow \forall y.B(y)$, c'est-à-dire tel que

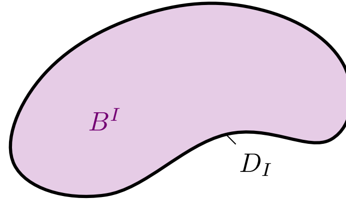
$$I, \rho[e/x] \nvDash B(x) \qquad \text{ou} \qquad I, \rho[e/x] \vDash \forall y.B(y) \; .$$

Il y a alors deux cas selon l'interprétation B^I du symbole de relation B :



soit $B^I \subsetneq D_I$: il existe un élément « sobre » $e \in D_I$ tel que $e \notin B^I$, et on a bien

$$I, \rho[e/x] \nvDash B(x)$$



soit $B^I = D_I$: comme $D_I \neq \emptyset$, on peut choisir n'importe quel $e \in D_I$ et on a bien

$$I, \rho[e/x] \vDash \forall y.B(y)$$

Exemple 12.4 (requêtes sur un graphe)**.** Considérons à nouveau la signature et l'interprétation I de l'exemple 10.4 qui représente le graphe de la figure 22. On peut par exemple déterminer s'il existe un sommet de degré sortant un par la formule close

$$\exists x.(\exists y.E(x,y) \land \forall z.y = z \lor \neg E(x,z)) \; . \tag{11}$$

Cette formule va s'évaluer à 1 dans l'interprétation I et pour n'importe quelle valuation ρ (puisque la formule est close) : I est un donc modèle de la formule. En effet,

$$\begin{aligned}
I,\rho &\models \exists x.(\exists y.E(x,y) \land \forall z.y = z \lor \neg E(x,z))\\
\text{car } I,\rho[1/x] &\models \exists y.E(x,y) \land \forall z.y = z \lor \neg E(x,z)\\
\text{car } I,\rho[1/x,0/y] &\models E(x,y) \land \forall z.y = z \lor \neg E(x,z)\\
\text{car d'une part } I,\rho[1/x,0/y] &\models E(x,y) \text{ puisque } (1,0) \in E^I,\\
\text{et d'autre part } I,\rho[1/x,0/y] &\models \forall z.y = z \lor \neg E(x,z) :\\
\text{en effet } I,\rho[1/x,0/y,0/z] &\models y = z \lor \neg E(x,z) \text{ puisque } y \text{ et } z \text{ sont valués à } 0\\
\text{puis } I,\rho[1/x,0/y,1/z] &\models y = z \lor \neg E(x,z) \text{ puisque } (1,1) \notin E^I\\
\text{et enfin } I,\rho[1/x,0/y,2/z] &\models y = z \lor \neg E(x,z) \text{ puisque } (1,2) \notin E^I.
\end{aligned}$$

Comme vous l'apprendrez en cours de bases de données, le calcul relationel *– qui est le nom que l'on donne à la logique du premier ordre sur une signature relationnelle – est équivalent à l'*algèbre relationnelle *– qui sont les opérations implémentées au sein des systèmes de gestion de base de données – ce résultat est connu comme le théorème de Codd et est ce qui permet de répondre aux requêtes SQL.*

Exemple 12.5 (requêtes sur une base de données)**.** Considérons la signature et l'interprétation I de l'exemple 10.5. Une formule du premier ordre permet d'exprimer des *requêtes* sur la base de données correspondante : si $x_1,\dots,x_n$ sont les variables libres de $\varphi(x_1,\dots,x_n)$, l'ensemble de n-uplets

$$\varphi(I) \stackrel{\text{def}}{=} \{(e_1,\dots,e_n) \mid I,[e_1/x_1,\dots,e_n/x_n] \models \varphi\}$$

est le résultat de la requête. Le langage SQL de requêtes sur une base de données permet d'exprimer des requêtes de la logique du premier ordre, et nous allons donner des exemples de requêtes en SQL pour chaque formule.

— Les titres des films présents dans la base de données :

$$\exists r \exists i.\mathit{Film}(x,r,i) \tag{12}$$

avec pour résultat $\{$*Shining*, *The Player*, *Easy Rider*, *Apocalypse Now*$\}$. Cette requête peut être exprimée en SQL sur la base de données de l'exemple 10.5 par

```
SELECT DISTINCT titre
  FROM Films
```

— Tous les cinémas qui diffusent un film avec Hopper :

$$\exists t \exists r.\mathit{Film}(t,r,\mathit{hopper}) \land \mathit{Seance}(x,t) \tag{13}$$

avec pour résultat $\{$Le Champo, Odéon$\}$. La requête SQL correspondante est

```
SELECT DISTINCT cinema
  FROM Films NATURAL JOIN Seances
 WHERE Films.interprete = 'Hopper'
```

— Tous les cinémas qui diffusent un film de Kubrick avec Hopper :

$$\exists t.\mathit{Film}(t,\mathit{kubrick},\mathit{hopper}) \land \mathit{Seance}(x,t) \tag{14}$$

avec pour résultat $\emptyset$. La requête SQL correspondante est

```
SELECT DISTINCT cinema
  FROM Films NATURAL JOIN Seances
 WHERE Films.interprete = 'Hopper'
   AND Films.realisation = 'Kubrick'
```

— Les interprètes qui ont joué dans un film dirigé par Kubrick ou par Coppola :

$$\exists t.\mathit{Film}(t,\mathit{kubrick},x) \lor \mathit{Film}(t,\mathit{coppola},x) \tag{15}$$

avec pour résultat $\{$Nicholson, Hopper$\}$. La requête SQL correspondante est

```sql
SELECT DISTINCT interprete
  FROM Films
 WHERE Films.realisation = 'Kubrick'
    OR Films.realisation = 'Coppola'
```

— Les réalisateurs qui ont joué dans un film qu'ils ont dirigé :

$$\exists t.\mathit{Film}(t, x, x) \tag{16}$$

avec pour résultat $\{$Hopper$\}$. Une requête SQL correspondante est

```sql
SELECT DISTINCT F1.realisation
  FROM Films F1 JOIN Films F2 ON F1.realisation = F2.interprete
                             AND F1.titre = F2.titre
```

— Les réalisateurs et les cinémas qui diffusent leurs films :

$$\exists t \exists i.\mathit{Film}(t, x, i) \wedge \mathit{Seance}(y, t) \tag{17}$$

avec pour résultat $\{$(Kubrick, Le Champo), (Altman, Le Champo), (Hopper, Le Champo), (Hopper, Odéon)$\}$. Une requête SQL correspondante est

```sql
SELECT DISTINCT realisation, cinema
  FROM Films NATURAL JOIN Seances
```

— Les réalisateurs dont les films passent dans tous les cinémas :

$$\forall c.(\exists t.\mathit{Seance}(c, t)) \Rightarrow (\exists t \exists i.\, \mathit{Film}(t, x, i) \wedge \mathit{Seance}(c, t)) \tag{18}$$

avec pour résultat $\{$Hopper$\}$. Une requête SQL possible pour cela est

```sql
SELECT DISTINCT F1.realisation
  FROM Films F1
 WHERE NOT EXISTS
       (SELECT S1.cinema
          FROM Seances S1
        EXCEPT
        SELECT S2.cinema
          FROM Seances S2 NATURAL JOIN Films F2
         WHERE F1.realisation = F2.realisation)
```

À noter ici que par souci de lisibilité, on aurait pu définir une formule $\mathit{cinema}(c) \stackrel{\text{def}}{=} \exists t.\mathit{Seance}(c, t)$ et écrire notre requête (18) comme

$$\forall c.\mathit{cinema}(c) \Rightarrow \exists t \exists i.\mathit{Film}(t, x, i) \wedge \mathit{Seance}(c, t)\ . \tag{19}$$

Comme la langue française est ambiguë, on aurait aussi pu comprendre cette requête comme

$$\exists t \exists i.\mathit{Film}(t, x, i) \wedge \forall c.\mathit{cinema}(c) \Rightarrow \mathit{Seance}(c, t) \tag{20}$$

qui retourne aussi $\{$Hopper$\}$.

— Les interprètes qui ont joué dans un film de Hopper mais pas dans un film de Kubrick :

$$(\exists t.\mathit{Film}(t, \mathit{hopper}, x)) \wedge \neg(\exists t.\mathit{Film}(t, \mathit{kubrick}, x)) \tag{21}$$

avec pour résultat $\{$Hopper$\}$. Une requête SQL possible pour cela est

```sql
SELECT interprete
  FROM Films
 WHERE realisation = 'Hopper'
EXCEPT
SELECT interprete
  FROM Films
 WHERE realisation = 'Kubrick'
```

— Les interprètes qui n'ont joué que dans un seul film :

$$\exists t \exists r. \mathit{Film}(t, r, x) \land \forall t' \forall r'. \mathit{Film}(t', r', x) \Rightarrow t = t' \tag{22}$$

avec pour résultat $\{$Robbins$\}$. Une requête SQL possible pour cela est

```
SELECT interprete
  FROM Films
EXCEPT
SELECT F1.interprete
  FROM Films F1, Films F2
 WHERE F1.interprete = F2.interprete
   AND F1.titre <> F2.titre
```

Exemple 12.6 (propriétés des ordres)**.** Dans la signature de l'exemple 10.2, l'interprétation I de domaine $\mathbb{Q}$ est un modèle de la formule

$$\forall x \exists y. y < x \tag{23}$$

qui exprime le fait que l'ordre n'est pas borné à gauche ; la formule est donc satisfiable. Il en serait de même si on avait pris $D_I \stackrel{\text{def}}{=} \mathbb{Z}$ et $<^I$ l'ordre sur les entiers relatifs. En revanche, si on avait pris $D_I \stackrel{\text{def}}{=} \mathbb{N}$, la formule (23) serait devenue fausse : cette formule n'est donc pas valide. La formule close

$$\forall x \forall y. x < y \Rightarrow \exists z. x < z \land z < y \tag{24}$$

exprime que l'ordre est dense. Elle est vraie sur $\mathbb{Q}$ mais pas sur $\mathbb{Z}$ et est donc satisfiable mais pas valide. La formule

$$\forall x \exists y \forall z. x < y \land \neg(x < z \land z < y) \tag{25}$$

est au contraire vraie sur $\mathbb{Z}$ mais fausse sur $\mathbb{Q}$.

13. Substitutions

Résumé. Une *substitution* est une fonction σ de domaine fini qui associe à toute variable $x \in X$ un terme $\sigma(x) \in T(\mathcal{F}, X)$; par extension, $t\sigma$ est le terme t dans lequel toutes les occurrences de chaque variable x ont été remplacées par $\sigma(x)$.

Une substitution σ est *applicable* à une formule φ si elle n'interagit pas avec les variables liées de φ ; on note alors « $\varphi\sigma$ » pour la formule φ dans laquelle toutes les occurrences de chaque variable x ont été remplacées par $\sigma(x)$. Modulo *α-renommage*, qui consiste à renommer les variables liées de φ, on peut toujours appliquer une substitution (c.f. propriété 13.3).

Soit I une interprétation et ρ une valuation. On dénote par « $\rho\sigma$ » la valuation qui associe pour toute variable $x \in X$ l'élément $(\rho\sigma)(x) \stackrel{\text{def}}{=} [\![\sigma(x)]\!]^I_\rho$; alors le lemme 13.5 de substitution dit que $[\![\varphi\sigma]\!]^I_\rho = [\![\varphi]\!]^I_{\rho\sigma}$.

Une *substitution* est une fonction $\sigma\colon X \to T(\mathcal{F}, X)$ de *domaine* $\operatorname{dom}(\sigma) \stackrel{\text{def}}{=} \{x \in X \mid \sigma(x) \neq x\}$ fini. On note aussi σ comme $[\sigma(x_1)/x_1, \ldots, \sigma(x_n)/x_n]$ où $x_1, \ldots, x_n$ sont les variables distinctes de $\operatorname{dom}(\sigma)$.

(Duparc, 2015, sec. 10.7), (Goubault-Larrecq et Mackie, 1997, def. 6.4), (Harrisson, 2009, sec. 3.4).

Une substitution définit une fonction $t \mapsto t\sigma$ de $T(\mathcal{F}, X)$ dans $T(\mathcal{F}, X)$ par induction sur le terme t

$$x\sigma \stackrel{\text{def}}{=} \sigma(x) \;, \qquad f(t_1, \ldots, t_m)\sigma \stackrel{\text{def}}{=} f(t_1\sigma, \ldots, t_m\sigma)$$

pour tout $m \in \mathbb{N}$ et tout $f \in \mathcal{F}_m$. Une substitution à image dans $T(\mathcal{F})$ est dite *close*.

Ces définitions s'étendent aussi aux formules. Cependant, du fait de la présence de variables liées dans les formules, une substitution n'est pas applicable à n'importe quelle formule. Pour une substitution σ et une formule φ, on note $\operatorname{rv}(\sigma) \stackrel{\text{def}}{=} \bigcup_{x \in \operatorname{dom}(\sigma)} \operatorname{fv}(\sigma(x))$ son ensemble de variables images (« *range variables* » en anglais). On dit que σ est *applicable* à φ si

$$(\operatorname{dom}(\sigma) \cup \operatorname{rv}(\sigma)) \cap \operatorname{bv}(\varphi) = \emptyset \;,$$

et dans ce cas on définit $\varphi\sigma$ comme l'application de σ à φ par induction sur la formule φ

$$R(t_1, \ldots, t_m)\sigma \stackrel{\text{def}}{=} R(t_1\sigma, \ldots, t_m\sigma) \;, \qquad (\neg\varphi)\sigma \stackrel{\text{def}}{=} \neg(\varphi\sigma) \;,$$
$$(\varphi \vee \psi)\sigma \stackrel{\text{def}}{=} (\varphi\sigma) \vee (\psi\sigma) \;, \qquad (\exists x.\varphi)\sigma \stackrel{\text{def}}{=} \exists x.(\varphi\sigma) \;,$$

où $m \in \mathbb{N}$ et $R \in \mathcal{P}_m$.

La *composition* $\sigma\sigma'$ de deux substitutions σ et σ' est définie par $\varphi(\sigma\sigma') \stackrel{\text{def}}{=} (\varphi\sigma)\sigma'$. Par exemple $(B(x) \vee B(y))[y/x, z/y] = B(y) \vee B(z)$, $(B(x) \vee B(y))[y/x][z/y] = B(z) \vee B(z)$, et $(B(x) \vee B(y))[z/x] = B(z) \vee B(y)$.

13.1. **Lemme de substitution.** Les valuations fournissent le pendant côté modèles des substitutions côté formules. Pour une substitution σ et une valuation ρ, notons $\sigma\rho$ la valuation $x \mapsto [\![\sigma(x)]\!]^I_\rho$ pour tout $x \in X$. Nous préparons ici le terrain pour le lemme 13.5 de substitution, qui sera énoncé sous une forme plus générale un peu plus loin.

⚠ *Une substitution propositionnelle τ reliait deux interprétations I et $I\tau$; par contraste, une substitution σ relie deux valuations ρ et $\rho\sigma$.*

Lemme 13.1. *Pour tout terme t, toute substitution σ, toute interprétation I, et toute valuation ρ, $[\![t\sigma]\!]^I_\rho = [\![t]\!]^I_{\sigma\rho}$.*

Démonstration. Par induction structurelle sur t.

Pour le cas de base d'une variable x, $[\![x\sigma]\!]^I_\rho = [\![\sigma(x)]\!]^I_\rho = (\sigma\rho)(x) = [\![x]\!]^I_{\sigma\rho}$.

Pour un terme $f(t_1, \ldots, t_m)$, $[\![f(t_1, \ldots, t_m)\sigma]\!]^I_\rho = [\![f(t_1\sigma, \ldots, t_m\sigma)]\!]^I_\rho = f^I([\![t_1\sigma]\!]^I_\rho, \ldots, [\![t_m\sigma]\!]^I_\rho) \stackrel{\text{h.i.}}{=} f^I([\![t_1]\!]^I_{\sigma\rho}, \ldots, [\![t_m]\!]^I_{\sigma\rho}) = [\![f(t_1, \ldots, t_m)]\!]^I_{\sigma\rho}$. □

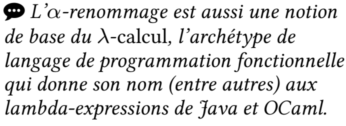
Lemme 13.2. *Pour toute formule φ, toute substitution σ applicable à φ, toute interprétation I, et toute valuation ρ, $[\![\varphi\sigma]\!]^I_\rho = [\![\varphi]\!]^I_{\sigma\rho}$.*

Démonstration. Par induction structurelle sur φ.

Pour une formule atomique $R(t_1,\dots,t_m)$, une négation $\neg\varphi$, ou une disjonction $\varphi \vee \psi$, cela découle du lemme 13.1 et de l'hypothèse d'induction. Pour une quantification $\exists x.\varphi$,

$$[\![(\exists x.\varphi)\sigma]\!]^I_\rho = [\![\exists x.(\varphi\sigma)]\!]^I_\rho = \operatorname*{Or}_{e\in D_I} [\![\varphi\sigma]\!]^I_{\rho[e/x]} \overset{\text{h.i.}}{=} \operatorname*{Or}_{e\in D_I} [\![\varphi]\!]^I_{\sigma(\rho[e/x])} = [\![\exists x.\varphi]\!]^I_{\sigma\rho} ,$$

où il faut montrer pour justifier cette dernière étape que, pour toute variable libre $z \in \mathrm{fv}(\varphi)$, $(\sigma(\rho[e/x]))(z) = ((\sigma\rho)[e/x])(z)$, ce qui permettra de conclure par la propriété 12.1. Si $z = x$, alors $(\sigma(\rho[e/x]))(x) = [\![\sigma(x)]\!]^I_{\rho[e/x]} = [\![x]\!]^I_{\rho[e/x]} = e = [\![x]\!]^I_{(\sigma\rho)[e/x]} = ((\sigma\rho)[e/x])(x)$ où $\sigma(x) = x$ puisque, comme σ est applicable à $\exists x.\varphi$ et $x \in \mathrm{bv}(\exists x.\varphi)$, $x \notin \mathrm{dom}(\sigma)$. Si $z \neq x$, alors $(\sigma(\rho[e/x]))(z) = [\![\sigma(z)]\!]^I_{\rho[e/x]} = [\![\sigma(z)]\!]^I_\rho = ((\sigma\rho)[e/x])(z)$, où l'égalité centrale repose sur le fait que $x \notin \mathrm{fv}(\sigma(z))$, qui à son tour découle de $\mathrm{bv}(\exists x.\varphi) \cap \mathrm{rv}(\sigma) = \emptyset$ puisque σ est applicable à $\exists x.\varphi$. □


L'α-renommage est aussi une notion de base du λ-calcul, l'archétype de langage de programmation fonctionnelle qui donne son nom (entre autres) aux lambda-expressions de Java et OCaml.

13.2. **α-renommages.** Quand une substitution σ n'est pas applicable à une formule φ, il est cependant possible d'effectuer un α-renommage à φ pour obtenir une formule φ' sur laquelle appliquer σ. On raisonnera donc implicitement non sur des formules mais sur des classes de formules équivalentes à α-renommage près. Il est important dans ce cas que cette opération définisse une congruence (l'α-congruence) sur les formules pour pouvoir continuer à raisonner inductivement, et que l'opération préserve la sémantique des formules.

On définit l'opération d'α-*renommage* comme une règle de réécriture sur les formules

$$\exists x.\varphi \to_\alpha \exists y.\varphi[y/x] \qquad (\alpha\text{-renommage})$$

où $y \notin \mathrm{fv}(\exists x.\varphi)$ et $[y/x]$ est applicable à φ. À noter que l'α-renommage n'impacte que les variables liées de φ, ce qui explique pourquoi il en préserve la sémantique (voir le lemme 13.4 ci-dessous).

On définit l'α-*congruence* $=_\alpha$ comme la plus petite congruence entre formules engendrée par $\to_\alpha$, c'est-à-dire la plus petite relation réflexive, symétrique et transitive telle que $\varphi \to_\alpha \psi$ implique $\varphi =_\alpha \psi$ et telle que si $\varphi =_\alpha \psi$ et $\varphi' =_\alpha \psi'$, alors $\neg\varphi =_\alpha \neg\psi$, $\varphi \vee \varphi' =_\alpha \psi \vee \psi'$ et $\exists x.\varphi =_\alpha \exists x.\psi$.

En appliquant des α-renommages inductivement sur les sous-formules croissantes de φ, on peut au besoin renommer toutes les variables liées de φ, et on déduit :

Propriété 13.3 (applicabilité). *Pour toute substitution σ et toute formule φ, il existe une formule φ' telle que $\varphi =_\alpha \varphi'$ et que σ soit applicable à φ'. De plus, si $\varphi =_\alpha \varphi''$ et σ est aussi applicable à φ'', alors $\varphi' =_\alpha \varphi''$ et $\varphi'\sigma =_\alpha \varphi''\sigma$.*

Il reste à vérifier que les α-renommages n'impactent pas la sémantique des formules.

Lemme 13.4 (α-congruence). *Soient φ et ψ deux formules telles que $\varphi =_\alpha \psi$. Alors pour toute interprétation I et toute valuation ρ, $[\![\varphi]\!]^I_\rho = [\![\psi]\!]^I_\rho$.*

Démonstration. Par induction sur la congruence $=_\alpha$.

Le cas de base est celui d'un α-renommage $\exists x.\varphi \to_\alpha \exists y.\varphi[y/x]$ avec $y \notin \mathrm{fv}(\exists x.\varphi)$ et $[y/x]$ applicable. Puisque $[y/x]$ est applicable, par le lemme 13.2,

$$[\![\exists y.\varphi[y/x]]\!]^I_\rho = \operatorname*{Or}_{e\in D_I} [\![\varphi[y/x]]\!]^I_{\rho[e/y]} = \operatorname*{Or}_{e\in D_i} [\![\varphi]\!]^I_{[y/x](\rho[e/y])} = \operatorname*{Or}_{e\in D_i} [\![\varphi]\!]^I_{\rho[e/x]} = [\![\exists x.\varphi]\!]^I_\rho$$

où nous devons justifier pour l'avant-dernière étape que, pour toute variable libre $z \in \mathrm{fv}(\varphi)$, $([y/x](\rho[e/y]))(z) = (\rho[e/x])(z)$, ce qui permettra de conclure ce cas de base par la propriété 12.1. Si $x = z$, alors $([y/x](\rho[e/y]))(x) = [\![y]\!]^I_{\rho[e/y]} = e = [\![x]\!]^I_{\rho[e/x]} = (\rho[e/x])(x)$. Si $x \neq z$, alors

d'une part $([y/x](\rho[e/y]))(z) = [\![z]\!]^I_{\rho[e/y]} = [\![z]\!]^I_\rho$ puisque $y \notin \text{fv}(\exists x.\varphi)$ et donc $y \neq z$, et d'autre part $(\rho[e/x])(z) = [\![z]\!]^I_{\rho[e/x]} = [\![z]\!]^I_\rho$.

Le cas de base de la réflexivité ainsi que les étapes d'induction pour la symétrie, la transitivité, et la congruence pour $\neg$, $\vee$ et $\exists$ sont évidents puisque $[\![\varphi]\!]^I_\rho = [\![\psi]\!]^I_\rho$ vue comme une relation entre formules est aussi une congruence. □

La propriété 13.3 et le lemme 13.4 justifient un abus de notation : nous écrirons désormais « $\varphi\sigma$ » pour une formule $\varphi'\sigma$ où $\varphi =_\alpha \varphi'$ et σ est applicable à φ'. Le lemme 13.1 ainsi que le lemme 13.4 d'α-congruence combiné au lemme 13.2 nous permettent alors d'énoncer le lemme de substitution.

Lemme 13.5 (substitution)**.** *Pour tout terme t, toute formule φ, toute substitution σ, toute interprétation I, et toute valuation ρ, $[\![t\sigma]\!]^I_\rho = [\![t]\!]^I_{\sigma\rho}$ et $[\![\varphi\sigma]\!]^I_\rho = [\![\varphi]\!]^I_{\sigma\rho}$.*

(Goubault-Larrecq et Mackie, 1997, thm. 6.8).

On utilisera par la suite deux identités aisément démontrables par induction sur φ : en appliquant au besoin une α-congruence à φ pour rendre les substitutions applicables, si $x \notin \text{fv}(\varphi)$, alors

$$\varphi[t/x] =_\alpha \varphi \ , \tag{26}$$

et si $x \notin \text{fv}(\varphi) \setminus \{y\}$, alors

$$\varphi[x/y][t/x] =_\alpha \varphi[t/y] \ . \tag{27}$$

Exemple 13.6 (propriétés arithmétiques)**.** Dans la signature arithmétique avec $\mathcal{F} \stackrel{\text{def}}{=} \{+^{(2)}, \times^{(2)}\}$ et $\mathcal{P} \stackrel{\text{def}}{=} \{=^{(2)}\}$ de l'exemple 10.3 et l'interprétation $I = (\mathbb{N}, +, \times)$, on peut définir dans la logique :

— le nombre 0 par une formule avec une variable libre

$$\mathit{zero}(x) \stackrel{\text{def}}{=} x + x = x \ . \tag{28}$$

Pour rappel, si « φ » est le nom d'une formule qui n'est pas close, on peut expliciter que $\text{fv}(\varphi) = \{x_1, \ldots, x_n\}$ en écrivant « $\varphi(x_1, \ldots, x_n)$ ». C'est ce que l'on a fait ici avec la formule « *zero* ». De manière très naturelle, on écrit alors « $\mathit{zero}(y)$ » pour le résultat $\mathit{zero}[y/x]$ du renommage de x en y dans la formule *zero*.

Pour revenir à la sémantique de notre formule (28), on a $I \models \exists x.\mathit{zero}(x) \wedge \forall y.(\mathit{zero}(y) \Rightarrow x = y)$, c'est-à-dire qu'il y a exactement un élément e du domaine $D_I \stackrel{\text{def}}{=} \mathbb{N}$ tel que $I, \rho[e/x] \models \mathit{zero}(x)$: c'est $e = 0$.

— de même, le nombre 1 par une formule avec une variable libre

$$\mathit{un}(x) \stackrel{\text{def}}{=} \neg\mathit{zero}(x) \wedge x \times x = x \tag{29}$$

et alors $I \models \exists x.\mathit{un}(x) \wedge \forall y.(\mathit{un}(y) \Rightarrow x = y)$;

— l'ordre strict par une formule avec deux variables libres

$$x < y \stackrel{\text{def}}{=} \exists z.\neg\mathit{zero}(z) \wedge y = x + z \ ; \tag{30}$$

— le prédicat

$$\mathit{pair}(x) \stackrel{\text{def}}{=} \exists y \exists z.x = y \times (z + z) \wedge \mathit{un}(z) \ ; \tag{31}$$

— le prédicat

$$\mathit{premier}(x) \stackrel{\text{def}}{=} \neg\mathit{zero}(x) \wedge \neg\exists y \exists z.x = y \times z \wedge \neg\mathit{un}(y) \wedge \neg\mathit{un}(z) \ ; \tag{32}$$

— une formule close qui dit que tout nombre premier supérieur à 2 est impair :

$$\forall x.(\mathit{premier}(x) \wedge \exists y.x > y + y \wedge \mathit{un}(y)) \Rightarrow \neg\mathit{pair}(x) \ . \tag{33}$$

13.3. Implémentation des substitutions en OCaml. Nous allons représenter une substitution $\sigma = [t_1/x_1, \ldots, t_n/x_n]$ comme une liste d'associations `sa=[(x₁,t₁);…;(xₙ,tₙ)]` :

firstorder.ml
```
type sassoc = (string * term) list
```

La fonction de type $X \to T(\mathcal{F}, X)$ associée est alors la suivante :

firstorder.ml
```
type substitution = string -> term
let subst_of_sassoc: sassoc -> substitution =
  fun sa -> fun x -> match List.assoc_opt x sa with
   | None   -> Var x
   | Some t -> t
```

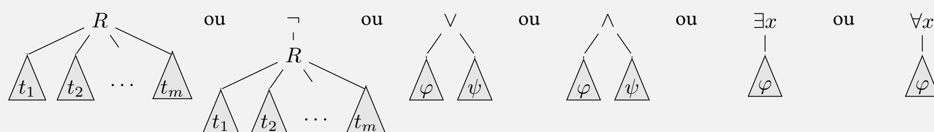
Nous allons implémenter simultanément les substitutions et les α-renommages. Pour cela, on dispose d'une séquence `variables` de type `string t` de noms de variables de X, ainsi que d'une fonction `find : ('a -> bool) -> 'a t -> 'a` qui trouve un élément satisfaisant un certain prédicat à l'intérieur d'une séquence. On a également implémenté une fonction `rv : sassoc -> string list` qui calcule l'ensemble $\text{dom}(\sigma) \cup \bigcup_{x\in\text{dom}(\sigma)} \text{fv}(\sigma(x))$ des noms de variables « réservées » qui ne doivent pas être liées dans une formule φ pour la substitution soit applicable à φ.

La fonction `alpha : string -> fo -> string list -> string * fo` prend en argument le nom x de la variable et la formule φ d'une formule $\exists x.\varphi$, ainsi qu'une liste de noms de variables à éviter, et retourne une paire $(y, \varphi[y/x])$ telle que $\exists x.\varphi \to_\alpha \exists y.\varphi[y/x]$.

firstorder.ml
```
(* α-renommage d'une formule ∃x.φ *)
let rec alpha x phi context =
  (* liste des noms de variables que l'on ne peut pas utiliser *)
  let forbidden =
    List.rev_append (fv (Ex(x,phi))) (List.rev_append context (bv phi)) in
  (* on trouve un nom de variable y utilisable *)
  let y =
    find (fun v -> not(List.mem v forbidden)) (append (to_seq[x]) variables) in
  (* on renvoie la paire (y,φ[y/x]) *)
  (y, fsubst [(x,Var y)] phi)
  and
(* application d'une substitution à un terme t *)
    tsubst sa t = match t with
    | Var x -> subst_of_sassoc sa x
    | Fun (f,tl) -> Fun (f, List.map (tsubst sa) tl)
  and
(* application d'une substitution à une formule φ *)
    fsubst sa phi = match phi with
    | Atom(r, tl) -> Atom (r, List.map (tsubst sa) tl)
    | Non(phi)    -> Non(fsubst sa phi)
    | Ou(phi,psi) -> Ou(fsubst sa phi, fsubst sa psi)
    | Ex(x, phi)  -> (* α-renommage à la volée *)
       let (y, phi') = alpha x phi (rv sa) in Ex(y, fsubst sa phi')
```

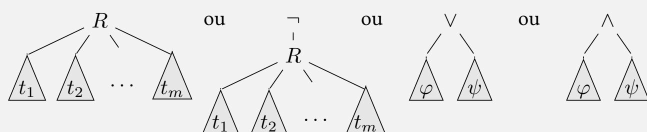
14. Formes normales

Résumé. On peut mettre n'importe quelle formule sous *forme normale négative* en « poussant » les négations vers les feuilles grâce aux dualités de de Morgan ; la formule obtenue est alors de la forme

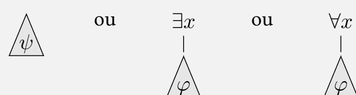


où $m \in \mathbb{N}$, $R \in \mathcal{P}_m$, $t_1, t_2, \ldots, t_m \in T(\mathcal{F}, X)$ et φ et ψ sont des formules sous forme normale négative. Les formules de la forme « $R(t_1, t_2, \ldots, t_m)$ » ou « $\neg R(t_1, t_2, \ldots, t_m)$ » sont appelées des *littéraux*.

Une formule est *sans quantificateur* si elle est de la forme



où $m \in \mathbb{N}$, $R \in \mathcal{P}_m$, $t_1, t_2, \ldots, t_m \in T(\mathcal{F}, X)$ et φ et ψ sont des formules sans quantificateurs. Une formule est sous *forme prénexe* si elle est de la forme



où φ est sous forme prénexe et ψ est sans quantificateurs. On peut mettre une formule sous forme prénexe à partir d'une formule sous forme normale négative en « tirant » les quantificateurs vers la racine.

Une formule sous forme prénexe est *universelle* si elle est de la forme « $\forall x_1 \forall x_2 \ldots \forall x_n.\psi$ » où ψ est sans quantificateurs. La *skolémisation* d'une formule produit une formule équi-satisfiable sous forme universelle.

Comme en logique propositionnelle, les formules du premier ordre peuvent être mises sous différentes formes normales préservant la sémantique (formules *équivalentes*) ou préservant la satisfiabilité (formules *équi-satisfiables*).

(David, Nour et Raffalli, 2003, sec. 2.6), (Goubault-Larrecq et Mackie, 1997, sec. 6.5.1)

14.1. **Forme normale négative.** Une formule du premier ordre est en *forme normale négative* si elle respecte la syntaxe abstraite

$$\ell ::= \alpha \mid \neg\alpha \qquad \text{(littéraux)}$$
$$\varphi ::= \ell \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \mid \forall x.\varphi \qquad \text{(formules en forme normale négative)}$$

où α est une formule atomique et $x \in X$. En d'autres termes, les négations ne peuvent apparaître que devant des formules atomiques. La mise sous forme normale négative procède en « poussant » les négations vers les feuilles ; pour une formule φ, on notera $\mathrm{nnf}(\varphi)$ sa forme normale

négative obtenue inductivement par

$$\begin{aligned}
\mathrm{nnf}(\alpha) &\stackrel{\text{def}}{=} \alpha\;, & \mathrm{nnf}(\neg\alpha) &\stackrel{\text{def}}{=} \neg\alpha\;,\\
\mathrm{nnf}(\varphi\vee\psi) &\stackrel{\text{def}}{=} \mathrm{nnf}(\varphi)\vee\mathrm{nnf}(\psi)\;, & \mathrm{nnf}(\varphi\wedge\psi) &\stackrel{\text{def}}{=} \mathrm{nnf}(\varphi)\wedge\mathrm{nnf}(\psi)\;,\\
\mathrm{nnf}(\neg(\varphi\vee\psi)) &\stackrel{\text{def}}{=} \mathrm{nnf}(\neg\varphi)\wedge\mathrm{nnf}(\neg\psi)\;, & \mathrm{nnf}(\neg(\varphi\wedge\psi)) &\stackrel{\text{def}}{=} \mathrm{nnf}(\neg\varphi)\vee\mathrm{nnf}(\neg\psi)\;,\\
\mathrm{nnf}(\exists x.\varphi) &\stackrel{\text{def}}{=} \exists x.\mathrm{nnf}(\varphi)\;, & \mathrm{nnf}(\forall x.\varphi) &\stackrel{\text{def}}{=} \forall x.\mathrm{nnf}(\varphi)\;,\\
\mathrm{nnf}(\neg\exists x.\varphi) &\stackrel{\text{def}}{=} \forall x.\mathrm{nnf}(\neg\varphi)\;, & \mathrm{nnf}(\neg\forall x.\varphi) &\stackrel{\text{def}}{=} \exists x.\mathrm{nnf}(\neg\varphi)\;,\\
\mathrm{nnf}(\neg\neg\varphi) &\stackrel{\text{def}}{=} \mathrm{nnf}(\varphi)\;.
\end{aligned}$$

En termes algorithmiques, cette mise sous forme normale négative se fait en temps linéaire. Elle préserve visiblement la sémantique des formules : φ et $\mathrm{nnf}(\varphi)$ sont équivalentes. On notera en général

$$\overline{\varphi} \stackrel{\text{def}}{=} \mathrm{nnf}(\neg\varphi)$$

pour la forme normale négative de la négation de φ.

Exemple 14.1. La formule du buveur de l'exemple 12.3 s'écrit sous forme normale négative comme $\exists x.(\neg B(x) \vee \forall y.B(y))$.

Exemple 14.2. La formule (24) de l'exemple 12.6 s'écrit sous forme normale négative comme

$$\forall x\forall y.\neg(x < y) \vee \exists z.x < z \wedge z < y\;. \tag{34}$$

Exemple 14.3. La requête (18) de l'exemple 12.5 s'écrit sous forme normale négative comme

$$\forall c.(\forall t.\neg \mathit{Seance}(c,t)) \vee (\exists t\exists i.\mathit{Film}(t,x,i) \wedge \mathit{Seance}(c,t))\;. \tag{35}$$

La requête (21) s'écrit sous forme normale négative comme

$$(\exists t.\mathit{Film}(t,\mathit{hopper},x)) \wedge (\forall t.\neg \mathit{Film}(t,\mathit{kubrick},x))\;. \tag{36}$$

La requête (22) s'écrit sous forme normale négative comme

$$\exists t\exists r.\mathit{Film}(t,r,x) \wedge \forall t'\forall r'.\neg \mathit{Film}(t',r',x) \vee t = t'\;. \tag{37}$$

On peut observer par induction structurelle sur φ que

Propriété 14.4. *Pour toute formule φ en forme normale négative et toute substitution σ, $(\overline{\varphi})\sigma = \overline{(\varphi\sigma)}$.*

La *profondeur* $p(\varphi)$ d'une formule φ en forme normale négative est définie inductivement par $p(\ell) \stackrel{\text{def}}{=} 0$ pour un littéral, $p(\varphi\vee\psi) = p(\varphi\wedge\psi) \stackrel{\text{def}}{=} 1+\max\{p(\varphi), p(\psi)\}$, et $p(\exists x.\varphi) = p(\forall x.\varphi) \stackrel{\text{def}}{=} 1 + p(\varphi)$.

14.1.1. *Implémentation de la forme normale négative en OCaml.* Afin d'implémenter la mise sous forme normale négative en OCaml, nous commençons par étendre la syntaxe des formules pour permettre explicitement les formules $\varphi \wedge \psi$ et les formules $\forall x.\varphi$.

firstorder.ml

```
type fo =
  Atom of atom
| Non of fo
| Ou of fo * fo
| Et of fo * fo     (* formules φ∧ψ *)
| Ex of string * fo
| Pt of string * fo (* formules ∀x.φ *)
```

Toutes les fonctions vu dans les sections précédentes sont ensuite modifiées pour traiter cette syntaxe étendue des formules.

Le calcul de nnf(φ) pour une formule φ est ensuite une implémentation directe de la définition.

firstorder.ml
```
let rec nnf_of_fo: fo -> fo = function
    Atom(r, tl)        -> Atom(r, tl)
  | Ou(phi, psi)       -> Ou(nnf_of_fo phi, nnf_of_fo psi)
  | Et(phi, psi)       -> Et(nnf_of_fo phi, nnf_of_fo psi)
  | Ex(x, phi)         -> Ex(x, nnf_of_fo phi)
  | Pt(x, phi)         -> Pt(x, nnf_of_fo phi)
  | Non(Atom(r, tl))   -> Non(Atom(r, tl))
  | Non(Non(phi))      -> nnf_of_fo phi
  | Non(Ou(phi, psi))  -> Et(nnf_of_fo (Non(phi)), nnf_of_fo (Non(psi)))
  | Non(Et(phi, psi))  -> Ou(nnf_of_fo (Non(phi)), nnf_of_fo (Non(psi)))
  | Non(Ex(x, phi))    -> Pt(x, nnf_of_fo (Non(phi)))
  | Non(Pt(x, phi))    -> Ex(x, nnf_of_fo (Non(phi)))
```

14.2. **Forme prénexe.** Une formule est sous *forme prénexe* si elle est de la forme $Q_1x_1 \dots Q_nx_n.\psi$ où $Q_i \in \{\forall, \exists\}$ pour tout $1 \le i \le n$ et ψ est sans quantificateur, c'est-à-dire ψ respecte la syntaxe abstraite

(Duparc, 2015, sec. 11.6), (David, Nour et Raffalli, 2003, sec. 2.6.2), (Goubault-Larrecq et Mackie, 1997, thm. 6.10), (Harrisson, 2009, sec. 3.5)

$$\psi ::= \ell \;\mid\; \psi \vee \psi \;\mid\; \psi \wedge \psi \,. \qquad \text{(formules sans quantificateur)}$$

Propriété 14.5 (forme prénexe)**.** *Pour toute formule φ, on peut construire une formule prénexe équivalente.*

Démonstration. Sans perte de généralité, on peut supposer que φ est déjà sous forme normale négative. On procède alors par induction sur φ :

Cas de ℓ : Alors ℓ est déjà sous forme prénexe.

Cas de $\varphi \vee \varphi'$: Par hypothèse d'induction, φ et φ' sont équivalentes à $Q_1x_1 \dots Q_nx_n.\psi$ et $Q'_1y_1 \dots Q'_my_m.\psi'$ où ψ et ψ' sont sans quantificateur. Par le lemme 13.4 d'α-congruence, on peut supposer que ces formules utilisent des ensembles disjoints de variables : $\{x_1, \dots, x_n\} \cap \text{fv}(\psi') = \emptyset$ et $\text{fv}(\psi) \cap \{y_1, \dots, y_m\} = \emptyset$.

Montrons que la formule $(Q_1x_1 \dots Q_nx_n.\psi) \vee (Q'_1y_1 \dots Q'_my_m.\psi')$ est équivalente à $Q_1x_1 \dots Q_nx_n.Q'_1y_1 \dots Q'_my_m.(\psi \vee \psi')$, qui sera donc équivalente à $\varphi \vee \varphi'$. On procède pour cela par récurrence sur $n+m$. Le cas de base pour $n+m=0$ est trivial. Pour l'étape de récurrence, il suffit de montrer que pour toutes formules θ et θ' et $x \notin \text{fv}(\theta')$,

— $(\exists x.\theta) \vee \theta'$ avec $x \notin \text{fv}(\theta')$ est équivalente à $\exists x.\theta \vee \theta'$: en effet, pour toute interprétation I et toute valuation ρ,

$$\begin{aligned}
[\![(\exists x.\theta) \vee \theta']\!]^I_\rho &= (\mathop{\text{Or}}_{e \in D_I} [\![\theta]\!]^I_{\rho[e/x]}) \text{ or } [\![\theta']\!]^I_\rho \\
&= (\mathop{\text{Or}}_{e \in D_I} [\![\theta]\!]^I_{\rho[e/x]}) \text{ or } (\mathop{\text{Or}}_{e \in D_I} [\![\theta']\!]^I_{\rho[e/x]}) \qquad \text{(car } D_I \neq \emptyset\text{, et par la propriété 12.1)} \\
&= \mathop{\text{Or}}_{e \in D_I} [\![\theta \vee \theta']\!]^I_{\rho[e/x]} \\
&= [\![\exists x.(\theta \vee \theta')]\!]^I_\rho \,.
\end{aligned}$$

– $(\forall x.\theta) \vee \theta'$ avec $x \notin \mathrm{fv}(\theta')$ est équivalente à $\forall x.\theta \vee \theta'$: en effet, pour toute interprétation I et toute valuation ρ,

$$\begin{aligned}
[\![(\forall x.\theta) \vee \theta']\!]^I_\rho &= (\mathop{\mathrm{And}}_{e \in D_I} [\![\theta]\!]^I_{\rho[e/x]}) \text{ or } [\![\theta']\!]^I_\rho \\
&= \mathop{\mathrm{And}}_{e \in D_I} ([\![\theta]\!]^I_{\rho[e/x]} \text{ or } [\![\theta']\!]^I_\rho) && \text{(par distributivité)} \\
&= \mathop{\mathrm{And}}_{e \in D_I} ([\![\theta]\!]^I_{\rho[e/x]} \text{ or } [\![\theta']\!]^I_{\rho[e/x]}) && \text{(par la propriété 12.1 puisque } x \notin \mathrm{fv}(\theta')\text{)} \\
&= [\![\forall x.(\theta \vee \theta')]\!]^I_\rho \;.
\end{aligned}$$

Cas de $\varphi \wedge \varphi'$: Similaire au cas précédent.

Cas de $\exists x.\varphi$: Par hypothèse d'induction, φ est équivalente à $Q_1 x_1 \ldots Q_n x_n.\psi$ où ψ est sans quantificateur. Alors $\exists x.\varphi$ est équivalente à $\exists x.Q_1 x_1 \ldots Q_n x_n.\psi$.

Cas de $\forall x.\varphi$: Similaire au cas précédent. □

Exemple 14.6. La négation de la formule du buveur de l'exemple 12.3 s'écrit sous forme normale négative comme $\forall x.(B(x) \wedge \exists y.\neg B(y))$. Une forme prénexe de cette négation est alors $\forall x.\exists y.(B(x) \wedge \neg B(y))$.

Exemple 14.7. La formule (34) de l'exemple 14.2 s'écrit sous forme prénexe comme

$$\forall x \forall y \exists z.\neg(x < y) \vee (x < z \wedge z < y) \;. \tag{38}$$

Exemple 14.8. Mettons les formules de l'exemple 14.3 sous forme prénexe. Pour la requête (35), il faut procéder à un α-renommage sur une des sous-formules qui quantifient sur t ; par exemple :

$$\forall c.(\forall t.\neg \mathit{Seance}(c,t)) \vee (\exists t' \exists i.\mathit{Film}(t',x,i) \wedge \mathit{Seance}(c,t') \;. \tag{39}$$

On peut maintenant appliquer la propriété 14.5 à (39) :

$$\forall c \forall t \exists t' \exists i.\neg \mathit{Seance}(c,t) \vee (\mathit{Film}(t',x,i) \wedge \mathit{Seance}(c,t')) \;. \tag{40}$$

On procède de même pour (36) :

$$\exists t \forall t'.\mathit{Film}(t,\mathit{hopper},x) \wedge \neg \mathit{Film}(t',\mathit{kubrick},x) \;. \tag{41}$$

Et aussi pour (37) :

$$\exists t \exists r \forall t' \forall r'.\mathit{Film}(t,r,x) \wedge (\neg \mathit{Film}(t',r',x) \vee t = t') \;. \tag{42}$$

14.2.1. *Implémentation de la forme prénexe en OCaml.* L'implémentation le calcul de la forme prénexe d'une formule φ en forme normale négative va suivre les arguments de la preuve de la propriété 14.5 : une formule $(\exists x.\psi) \vee \varphi$ où $x \notin \mathrm{fv}(\varphi)$ est logiquement équivalente à $\exists x.\psi \vee \varphi$, et il en est de même modulo commutativité du $\vee$, ainsi qu'en remplaçant $\vee$ par $\wedge$ ou $\exists$ par $\forall$. Afin de garantir $x \notin \mathrm{fv}(\varphi)$, on procède à un α-renommage à la volée de la formule $\exists x.\psi$.

firstorder.ml
```
let rec pnf_of_nnf: fo -> fo = function
  | Atom(_,_) as phi -> phi
  | Non(_)    as phi -> phi
  | Ex(x,phi) -> Ex(x, pnf_of_nnf phi)
  | Pt(x,phi) -> Pt(x, pnf_of_nnf phi)
  | Ou(phi,psi) -> (match (pnf_of_nnf phi, pnf_of_nnf psi) with
                     | (phi, Ex(x,psi)) -> (* α-renommage à la volée *)
                       let (y,psi') = alpha x psi (fv phi) in
                       Ex(y, pnf_of_nnf (Ou(phi, psi')))
                     | (Ex(x,psi),phi) -> (* α-renommage à la volée *)
                       let (y,psi') = alpha x psi (fv phi) in
```

```
                              Ex(y, pnf_of_nnf (Ou(psi', phi)))
                            | (phi, Pt(x,psi)) -> (* α-renommage à la volée *)
                              let (y,psi') = alpha x psi (fv phi) in
                              Pt(y, pnf_of_nnf (Ou(phi, psi')))
                            | (Pt(x,psi),phi) -> (* α-renommage à la volée *)
                              let (y,psi') = alpha x psi (fv phi) in
                              Pt(y, pnf_of_nnf (Ou(psi', phi)))
                            | (phi,psi) -> Ou(phi,psi))
  | Et(phi,psi) -> (match (pnf_of_nnf phi, pnf_of_nnf psi) with
                            | (phi, Ex(x,psi)) -> (* α-renommage à la volée *)
                              let (y,psi') = alpha x psi (fv phi) in
                              Ex(y, pnf_of_nnf (Et(phi, psi')))
                            | (Ex(x,psi),phi) -> (* α-renommage à la volée *)
                              let (y,psi') = alpha x psi (fv phi) in
                              Ex(y, pnf_of_nnf (Et(psi', phi)))
                            | (phi, Pt(x,psi)) -> (* α-renommage à la volée *)
                              let (y,psi') = alpha x psi (fv phi) in
                              Pt(y, pnf_of_nnf (Et(phi, psi')))
                            | (Pt(x,psi),phi) -> (* α-renommage à la volée *)
                              let (y,psi') = alpha x psi (fv phi) in
                              Pt(y, pnf_of_nnf (Et(psi', phi)))
                            | (phi,psi) -> Et(phi,psi))
let pnf_of_fo: fo -> fo = function phi -> pnf_of_nnf(nnf_of_fo phi)
```

14.3. **Skolémisation.** Une formule $\varphi = Q_1 x_1 \ldots Q_n x_n.\psi$ sous forme prénexe est *existentielle* si $Q_i = \exists$ pour tout $1 \leq i \leq n$, et *universelle* si $Q_i = \forall$ pour tout $1 \leq i \leq n$. La skolémisation d'une formule φ est une formule universelle équi-satisfiable.

(David, Nour et Raffalli, 2003, sec. 2.7), (Goubault-Larrecq et Mackie, 1997, thm. 6.12), Harrisson, 2009, [sec. 3.6

On étend pour cela l'ensemble $\mathcal{F}$ des symboles de fonction. Pour toute formule φ et toute variable x, on fixe une énumération $\vec{x}_\varphi$ de fv$(\exists x.\varphi)$ et on ajoute un nouveau symbole de fonction $f_{\varphi,x} \notin \mathcal{F}$ d'arité $|\text{fv}(\exists x.\varphi)|$. Soit $\mathcal{F}' \stackrel{\text{def}}{=} \mathcal{F} \uplus \{f_{\varphi,x} \mid f \text{ formule sur } (\mathcal{F},\mathcal{P}) \text{ et } x \in X\}$.

La *skolémisation* $s(\varphi)$ d'une formule φ en forme normale négative est alors définie par

$$\begin{aligned}
s(\ell) &\stackrel{\text{def}}{=} \ell\ ,\\
s(\varphi \vee \psi) &\stackrel{\text{def}}{=} s(\varphi) \vee s(\psi)\ ,\\
s(\varphi \wedge \psi) &\stackrel{\text{def}}{=} s(\varphi) \wedge s(\psi)\ ,\\
s(\exists x.\varphi) &\stackrel{\text{def}}{=} (s(\varphi))[f_{\varphi,x}(\vec{x}_\varphi)/x]\\
s(\forall x.\varphi) &\stackrel{\text{def}}{=} \forall x.s(\varphi)\ .
\end{aligned}$$

On peut noter que, si φ était sous forme prénexe, alors $s(\varphi)$ est universelle.

Théorème 14.9 (Skolem)**.** *Soit φ une formule du premier ordre. Alors on peut construire une formule universelle équi-satisfiable.*

Démonstration. On peut supposer φ sous forme prénexe. Il suffit alors de montrer que φ et $s(\varphi)$ sont équi-satisfiables.

Commençons par montrer que, si φ est satisfiable, alors $s(\varphi)$ l'est aussi. Pour toute interprétation I sur la signature $(\mathcal{F},\mathcal{P})$, on construit une interprétation I' sur la signature $(\mathcal{F}',\mathcal{P})$. L'interprétation I' a le même domaine $D_{I'} \stackrel{\text{def}}{=} D_I$ que I et interprète les symboles de $\mathcal{F}$ et $\mathcal{P}$ de la même manière : $f^{I'} \stackrel{\text{def}}{=} f^I$ et $R^{I'} \stackrel{\text{def}}{=} R^I$ pour tout $f \in \mathcal{F}$ et $R \in \mathcal{P}$. Il faut maintenant fournir une interprétation des symboles $f_{\varphi,x}$. Considérons pour cela une formule φ et une variable x. Soit $x_1,\ldots,x_n$ l'énumeration de fv$(\exists x.\varphi)$. Pour tout tuple $(e_1,\ldots,e_n) \in D_I^n$, on définit

$$D_{I,\varphi,x}(e_1,\ldots,e_n) \stackrel{\text{def}}{=} \{e \in D_I \mid I,\rho[e_1/x_1,\ldots,e_n/x_n,e/x] \vDash \varphi\}\ . \tag{43}$$

Comme $D_I \neq \emptyset$, on peut choisir un élément $e_\emptyset \in D_I$. On étend alors l'interprétation I : si $D_{I,\varphi,x}(e_1,\ldots,e_n)$ est non vide, on choisit un représentant $f^{I'}_{\varphi,x}(e_1,\ldots,e_n) \in D_{I,\varphi,x}(e_1,\ldots,e_n)$, et sinon on pose $f^{I'}_{\varphi,x}(e_1,\ldots,e_n) = e_\emptyset$.

Montrons maintenant par induction sur φ que, pour toute interprétation I et toute valuation ρ, $I,\rho \vDash \varphi$ implique $I',\rho \vDash s(\varphi)$. On ne traite ici que le cas d'une formule $\exists x.\varphi$. Soit $x_1,\ldots,x_n$ l'énumération de fv$(\exists x.\varphi)$. On a les implications

$$\begin{array}{lll} & I,\rho \vDash \exists x.\varphi & \\ \Rightarrow & \exists e \in D_I,\ I,\rho[e/x] \vDash \varphi & \\ \Rightarrow & I,\rho[f^{I'}_{\varphi,x}(\rho(x_1),\ldots,\rho(x_n))/x] \vDash \varphi & \\ \Rightarrow & I',\rho[f^{I'}_{\varphi,x}(\rho(x_1),\ldots,\rho(x_n))/x] \vDash s(\varphi) & \text{par hyp. ind.} \\ \Rightarrow & I',\rho \vDash (s(\varphi))[f_{\varphi,x}(x_1,\ldots,x_n)/x] & \text{par le lemme 13.5} \\ \Rightarrow & I',\rho \vDash s(\exists x.\varphi)\ . & \end{array}$$

Inversement, montrons que si $s(\varphi)$ est satisfiable, alors φ l'est aussi. Pour une interprétation I sur la signature $(\mathcal{F}',\mathcal{P})$, on définit l'interprétation $I_{|\mathcal{F}}$ comme sa restriction à la signature $(\mathcal{F},\mathcal{P})$; autrement dit, on « oublie » dans $I_{|\mathcal{F}}$ les interprétations des symboles de fonction $f_{\varphi,x}$. Montrons par induction sur φ que $I,\rho \vDash s(\varphi)$ implique $I_{|\mathcal{F}},\rho \vDash \varphi$. Encore une fois, nous ne traitons que le cas d'une formule $\exists x.\varphi$ où $x_1,\ldots,x_n$ est l'énumération de fv$(\exists x.\varphi)$. On a les implications

$$\begin{array}{lll} & I,\rho \vDash s(\exists x.\varphi) & \\ \Rightarrow & I,\rho \vDash (s(\varphi))[f_{\varphi,x}(x_1,\ldots,x_n)/x] & \\ \Rightarrow & I,\rho[f^{I}_{\varphi,x}(\rho(x_1),\ldots,\rho(x_n))/x] \vDash s(\varphi) & \text{par le lemme 13.5} \\ \Rightarrow & I_{|\mathcal{F}},\rho[f^{I}_{\varphi,x}(\rho(x_1),\ldots,\rho(x_n))/x] \vDash \varphi & \text{par hyp. ind.} \\ \Rightarrow & I_{|\mathcal{F}},\rho \vDash \exists x.\varphi\ . & \end{array}$$

□

⚠ La skolémisation préserve la satisfiabilité, mais pas la validité. Ainsi, la formule du buveur est valide, mais la formule (44) ne l'est pas : l'interprétation I avec $D_I = \{e_1,e_2\}$, $a^I = e_1$ et $B^I = \{e_1\}$ en est un contre-modèle.

Exemple 14.10. La formule du buveur $\exists x.(B(x) \Rightarrow \forall y.B(y))$ se skolémise en introduisant un symbole de constante a pour $f_{(B(x)\Rightarrow\forall y.B(y)),x}$ où fv$(\exists x.(B(x) \Rightarrow \forall y.B(y))) = \emptyset$:

$$B(a) \Rightarrow \forall y.B(y)\ . \tag{44}$$

Exemple 14.11. La formule (38) de l'exemple 14.7 se skolémise en introduisant une fonction f d'arité deux pour $f_{\varphi,z}$ où $\varphi = \neg(x < y) \vee (x < z \wedge z < y)$ et fv$(\exists z.\varphi) = \{x,y\}$:

$$\forall x\forall y.\neg(x < y) \vee (x < f(x,y) \wedge f(x,y) < y)\ . \tag{45}$$

⚠ Les requêtes que nous obtenons ici ont encore x en variable libre. Pour obtenir une formule universelle équi-satisfiable close*, il faudrait dans chaque cas introduire une constante a et remplacer x par a.*

Exemple 14.12. La requête (40) de l'exemple 14.8 se skolémise en introduisant une fonction f d'arité 3 pour $f_{\varphi,t'}$ où $\varphi = \exists i.\neg\mathit{Seance}(c,t) \vee (\mathit{Film}(t',x,i) \wedge \mathit{Seance}(c,t')$ et fv$(\exists t.\varphi) = \{x,c,t\}$, ainsi qu'une fonction g d'arité 4 pour $f_{\psi,i}$ où $\psi = \neg\mathit{Seance}(c,t) \vee (\mathit{Film}(t',x,i) \wedge \mathit{Seance}(c,t')$ et fv$(\exists i.\psi) = \{x,c,t,t'\}$:

$$\forall c\forall t.\neg\mathit{Seance}(c,t) \vee (\mathit{Film}(f(x,c,t),x,g(x,c,t,f(x,c,t))) \wedge \mathit{Seance}(c,f(x,c,t)))\ . \tag{46}$$

La requête (41) se skolémise en introduisant une fonction h d'arité 1 :

$$\forall t'.\mathit{Film}(h(x),\mathit{hopper},x) \wedge \neg\mathit{Film}(t',\mathit{kubrick},x)\ . \tag{47}$$

La requête (42) se skolémise en introduisant une fonction f' d'arité 1 et une fonction g' d'arité 2 :

$$\forall t'\forall r'.\mathit{Film}(f'(x),g'(x,f'(x))) \wedge (\neg\mathit{Film}(t',r',x) \vee f'(x) = t')\ . \tag{48}$$

14.3.1. *Implémentation de la skolémisation en OCaml.* Pour implémenter la skolémisation, c'est-à-dire la fonction $s(\varphi)$ définie plus haut, nous allons avoir besoin de générer des symboles de fonction $f_{\varphi,x}$. On implémente pour cela une séquence `functions` de type `string t` qui liste des symboles de fonctions. Il faut de plus éviter qu'un nom de fonction $f_{\varphi,x}$ soit en conflit avec un symbole de fonction utilisé dans φ ; la fonction `ff : fo -> string list` retourne pour cela l'ensemble des symboles de fonction utilisés dans la formule passée en argument.

firstorder.ml
```
let skolem_of_nnf : fo -> fo =
  let rec aux = fun forbidden -> function
      Atom(_,_) as phi -> phi
    | Non(_)    as phi -> phi
    | Ou(phi, psi)     -> Ou(aux forbidden phi, aux forbidden psi)
    | Et(phi, psi)     -> Et(aux forbidden phi, aux forbidden psi)
    | Pt(x, phi)       -> Pt(x, aux forbidden phi)
    | Ex(x, phi)       ->
       (* appel récursif pour construire s(φ) *)
       let sphi = aux forbidden phi in
       (* la séquence de variables x⃗_φ sous forme de liste *)
       let args = List.rev_map (fun x -> Var x) (fv (Ex(x,phi))) in
       (* le symbole de fonction f_φ,x *)
       let fphix =
         find (fun f -> not(List.mem f (List.rev_append forbidden (ff sphi))))
           functions in
       (* application de la substitution [f_φ,x(x⃗_φ)/x] *)
       fsubst [(x, Fun(fphix, args))] sphi
  in fun phi -> aux (ff phi) phi
let skolem phi = skolem_of_nnf(nnf_of_fo phi)
```

14.4. * **Forme clausale.** Une *clause* est une formule close universelle $\forall x_1 \dots \forall x_n.\psi$ où ψ est une disjonction de littéraux. Une formule est sous *forme clausale* si c'est une conjonction de clauses, que l'on présente aussi comme un ensemble de clauses.

(DAVID, NOUR et RAFFALLI, 2003, sec. 7.4.2)

La mise sous forme clausale en logique du premier ordre sert en particulier de préliminaire à l'utilisation des preuves par résolution ; voir (DAVID, NOUR et RAFFALLI, 2003, sec 7.4), (GOUBAULT-LARRECQ et MACKIE, 1997, sec. 7.3)

Théorème 14.13 (forme clausale). *Soit φ une formule. Alors on peut construire un ensemble équi-satisfiable $Cl(\varphi)$ de clauses.*

Démonstration. Soit φ une formule. En quantifiant existentiellement ses variables libres $\text{fv}(\varphi) = \{x_1, \dots, x_n\}$, on obtient une formule close équi-satisfiable $\varphi' \stackrel{\text{def}}{=} \exists x_1 \dots \exists x_n.\varphi$. Par le théorème 14.9 de SKOLEM, on construit une formule universelle close équi-satisfiable $s(\varphi')$. Il reste simplement à mettre $s(\varphi')$ sous forme normale conjonctive, en utilisant le fait que $\forall x.\varphi \wedge \psi$ est équivalente à $(\forall x.\varphi) \wedge (\forall x.\psi)$. On obtient ainsi une écriture de $s(\varphi)$ comme $\bigwedge_i \forall \vec{x}. \bigvee_{j_i} \ell_{j_i}$, et on définit alors $\text{Cl}(\varphi) \stackrel{\text{def}}{=} \{\forall \vec{x}. \bigvee_{j_i} \ell_{j_i}\}_i$. □

Voir la section 5.2.1 pour la mise sous forme normale conjonctive.

Exemple 14.14. La forme clausale de la *négation* $\neg(\exists x.B(x) \Rightarrow \forall y.B(y))$ de la formule du buveur est $\neg B(a) \wedge \forall x.B(x)$.

14.4.1. *Implémentation de la forme clausale en OCaml.* Voyons comment implémenter la construction donnée dans la démonstration du théorème 14.13. La première étape est de construire $\varphi' \stackrel{\text{def}}{=} \exists x_1 \dots \exists x_n.\varphi$ où $\text{fv}(\varphi) = \{x_1, \dots, x_n\}$; on implémente cela dans la fonction `close` ci-dessous.

firstorder.ml
```
(* quantification existentielle des variables libres *)
let close: fo -> fo = fun phi ->
  let rec quantify vars phi = match vars with
```

```
    | []    -> phi
    | x::xs -> quantify xs (Ex(x,phi))
  in quantify (fv phi) phi
```

La deuxième étape est de skolémiser φ' en une formule universelle close équi-satisfiable.

```
firstorder.ml
(* formule universelle close équi-satisfiable *)
let universal: fo -> fo = fun phi -> skolem(close(phi))
```

Pour la mise sous forme clausale, on va définir une clause comme une liste de littéraux :

```
firstorder.ml
type litteral = Pos of string * term list | Neg of string * term list
type clause   = litteral list
```

On définit des fonctions auxiliaires pour manipuler des formules sous forme clausale.

```
firstorder.ml
(* calcul de la forme normale conjonctive de (⋀⋁ℓ_{i,j}) ∧ (⋀⋁ℓ'_{k,l}) *)
let cnf_and_cnf: clause list -> clause list -> clause list =
  List.rev_append

(* calcul de la forme normale conjonctive de (⋀⋁ℓ_{i,j}) ∨ (⋀⋁ℓ'_{k,l}) *)
let cnf_or_cnf: clause list -> clause list -> clause list =
  (* calcul de la forme normale conjonctive de
     (⋁ℓ_j) ∨ (⋀⋁ℓ'_{k,l}) ⇔ ⋀(⋁ℓ_j ∨ ⋁ℓ'_{k,l}) *)
  let cnf_or_cl: clause list -> clause -> clause list =
    fun clauses cl -> List.rev_map (List.rev_append cl) clauses in
  fun clauses1 clauses2 -> concat_map (cnf_or_cl clauses1) clauses2

(* calcul de la forme normale conjonctive de ¬(⋀⋁ℓ_{i,j}) *)
let not_cnf: clause list -> clause list = fun clauses ->
  (* si ⊥ apparaît dans la conjonction, alors on retourne ∅ *)
  if List.mem [] clauses then [] else
    (* ¬(⋀⋁ℓ_{i,j}) ⇔ ⋁⋀ℓ̄_{i,j} *)
    let negclauses = List.rev_map
                       (fun cl -> List.rev_map (function
                                    Pos(r,tl) -> [Neg(r,tl)]
                                  | Neg(r,tl) -> [Pos(r,tl)]) cl)
                       clauses in
    let rec distribute = function
        []         -> [[]]
      | cnf::cnfs -> cnf_or_cnf cnf (distribute cnfs) in
    distribute negclauses
```

La mise sous forme clausale proprement dite est en fait une mise sous forme conjonctive de la sous-formule sans quantificateur ψ de $s(\varphi') = \forall y_1 \cdots \forall y_m.\psi$.

```
firstorder.ml
let clauses_of_fo: fo -> clause list = fun phi ->
(* descendre dans une formule universelle close
   jusqu'à la formule ψ sans quantificateur *)
let rec qf_of_prenex: fo -> fo = function
    Ex(_, psi) -> qf_of_prenex psi
  | Pt(_, psi) -> qf_of_prenex psi
  | psi        -> psi
in
```

```
let psi = qf_of_prenex (universal phi) in
(* mise sous forme clausale de ψ (qui est en nnf et sans quantificateur) *)
let rec cnf_of_qf: fo -> clause list = function
    Atom(r,tl)       -> [[Pos(r,tl)]]
  | Non(Atom(r,tl)) -> [[Neg(r,tl)]]
  | Et(psi1,psi2)   -> cnf_and_cnf (cnf_of_qf psi1) (cnf_of_qf psi2)
  | Ou(psi1,psi2)   -> cnf_or_cnf (cnf_of_qf psi1) (cnf_of_qf psi2)
  | _               -> failwith "incorrect argument of cnf_of_qf"
in cnf_of_qf psi
```

14.5. * **Modèles de Herbrand.** Nous avons vu dans la définition de la sémantique des formules du premier ordre que les interprétations pouvaient être extrêmement variées : il suffit d'un domaine non vide et d'interprétations des symboles de fonction et de relation. Si on cherche à construire un modèle d'une formule, cela laisse un choix considérable. Nous allons voir dans cette section qu'en fait, on peut se restreindre à des interprétations très particulières, appelées *interprétations de Herbrand*, dans lesquelles le domaine est tout simplement l'ensemble des termes clos.

(Goubault-Larrecq et Mackie, 1997, sec. 5.2), (Harrisson, 2009, sec. 3.7)

Supposons pour cela $\mathcal{F}_0 \neq \emptyset$, de telle sorte que l'ensemble $T(\mathcal{F})$ des termes clos soit non vide. L'interprétation d'un symbole de fonction f d'arité n dans une interprétation de Herbrand va simplement être la fonction qui associe le terme $f(t_1, \ldots, t_n)$ aux termes $t_1, \ldots, t_n$. Afin d'interpréter les symboles de relation, on va définir la *base de Herbrand*

$$\mathcal{B} \stackrel{\text{def}}{=} \{R(t_1, \ldots, t_m) \mid m \in \mathbb{N}, R \in \mathcal{P}_m, t_1, \ldots, t_m \in T(\mathcal{F})\}$$

comme l'ensemble des instanciations des symboles de relation de $\mathcal{P}$.

Une *interprétation de Herbrand* est une fonction $H\colon \mathcal{B} \to \mathbb{B}$ qui associe une valeur de vérité à chaque élément de la base de Herbrand. On identifie H avec l'interprétation de domaine $D_H \stackrel{\text{def}}{=} T(\mathcal{F})$ qui associe

$$f^H(t_1, \ldots, t_n) \stackrel{\text{def}}{=} f(t_1, \ldots, t_n) , \qquad R^H(t_1, \ldots, t_m) \stackrel{\text{def}}{=} H(R(t_1, \ldots, t_m))$$

à tout $f \in \mathcal{F}_n$ et $R \in \mathcal{P}_m$ pour tout n, m.

Théorème 14.15 (Herbrand)**.** *Soit S un ensemble de formules universelles closes. Alors S est satisfiable si et seulement si S est satisfiable dans une interprétation de Herbrand.*

Démonstration. Seul le sens direct nécessite une preuve. Soit I une interprétation telle que $I \models S$. On définit l'interprétation de Herbrand H par $H(R(t_1, \ldots, t_m)) \stackrel{\text{def}}{=} [\![R(t_1, \ldots, t_m)]\!]^I$ pour tous $m \in \mathbb{N}$, $R \in \mathcal{P}_m$, et $t_1, \ldots, t_m \in T(\mathcal{F})$; comme les formules et termes en question sont clos, il n'est pas utile de préciser sous quelle valuation.

Les formules de S doivent être closes : $S = \{P(x), \neg P(a)\}$ avec $\mathcal{F} = \{a\}$ est satisfiable mais ne l'est pas dans une interprétation de Herbrand.

Montrons que $H \models S$. Pour toute formule close $\forall x_1 \ldots \forall x_n . \varphi \in S$ où φ est sans quantificateur et $\text{fv}(\varphi) = \{x_1, \ldots, x_n\}$, et pour tous $t_1, \ldots, t_n \in D_H = T(\mathcal{F})$, on définit $\sigma \stackrel{\text{def}}{=} [t_1/x_1, \ldots, t_n/x_n]$ – qui peut être vue à la fois comme une substitution close et comme une valuation dans D_H – et $\rho \stackrel{\text{def}}{=} [[\![t_1]\!]^I/x_1, \ldots, [\![t_n]\!]^I/x_n]$ une valuation dans I. On montre $[\![\varphi]\!]^H_\sigma = [\![\varphi]\!]^I_\rho$ par induction sur φ ; par suite, comme $I \models \forall x_1 \ldots \forall x_n . \varphi$, on en déduit $H \models \forall x_1 \ldots \forall x_n . \varphi$ comme voulu.

- Pour une formule atomique α, par définition de H, $[\![\alpha]\!]^H_\sigma = [\![\alpha\sigma]\!]^H$ par le lemme 13.5, et cette sémantique est égale à $[\![\alpha\sigma]\!]^I$ par définition de R^H, qui n'est autre que $[\![\alpha]\!]^I_\rho$ par le lemme 13.5.
- Pour une négation $\neg\varphi$, $[\![\neg\varphi]\!]^H_\sigma = \text{not}\,[\![\varphi]\!]^H_\sigma \stackrel{\text{h.i.}}{=} \text{not}\,[\![\varphi]\!]^I_\rho = [\![\neg\varphi]\!]^I_\rho$.
- Pour une disjonction $\varphi \vee \psi$, $[\![\varphi \vee \psi]\!]^H_\sigma = [\![\varphi]\!]^H_\sigma \text{ or } [\![\psi]\!]^H_\sigma \stackrel{\text{h.i.}}{=} [\![\varphi]\!]^I_\rho \text{ or } [\![\psi]\!]^I_\rho = [\![\varphi \vee \psi]\!]^I_\rho$. □

Le théorème de Herbrand permet de « réduire » la logique du premier ordre à la logique propositionnelle. Pour une formule universelle close $\varphi = \forall\vec{x}.\varphi'$ où φ' est sans quantificateur, on note $\varphi\Sigma \stackrel{\text{def}}{=} \{\varphi'\sigma \mid \sigma \text{ substitution close}\}$ l'ensemble de ses *instances de Herbrand*. Pour

un ensemble de formules universelles closes S, on écrit $S\Sigma \stackrel{\text{def}}{=} \bigcup_{\varphi \in S} \varphi\Sigma$. Une formule de $S\Sigma$ peut ainsi être vue comme une formule propositionnelle avec la base de Herbrand $\mathcal{B}$ comme ensemble de propositions.

Propriété 14.16. *Soit S un ensemble de formules universelles closes et H une interprétation de Herbrand. Alors $H \models S$ si et seulement si $H \models S\Sigma$.*

Démonstration. Pour toute formule close $\varphi = \forall x_1 \ldots \forall x_n.\varphi' \in S$ où φ' est sans quantificateur et $\text{fv}(\varphi') = \{x_1, \ldots, x_n\}$, $H \models \varphi$ si et seulement si pour tous termes clos $t_1, \ldots, t_n \in D_H$, $H, [t_1/x_1, \ldots, t_n/x_n] \models \varphi'$, si et seulement si pour tous termes clos $t_1, \ldots, t_n \in T(\mathcal{F})$, $H \models \varphi'[t_1/x_1, \ldots, t_n/x_n]$, si et seulement si pour toute formule $\psi \in \varphi\Sigma$, $H \models \psi$. □

Corollaire 14.17. *Soit S un ensemble de formules universelles closes. Alors S est satisfiable si et seulement si $S\Sigma$ est propositionnellement satisfiable sur l'ensemble de propositions $\mathcal{B}$.*

Démonstration. Par le théorème 14.15, S est satisfiable si et seulement s'il existe une interprétation de Herbrand H telle que $H \models S$. Il suffit alors d'appliquer la propriété 14.16 à S et H. □

15. Théories et modèles

Résumé. Une *théorie* est un ensemble T de formules closes (c'est-à-dire de formules sans variables libres) tel que, si φ est close et $T \vDash \varphi$, alors $\varphi \in T$. On définit habituellement des théories de deux manières :

(1) comme l'ensemble des formules closes $\mathrm{Th}(I) \stackrel{\text{def}}{=} \{\varphi \text{ close} \mid I \vDash \varphi\}$ dont une interprétation I est un modèle, ou

(2) comme l'ensemble des formules closes $\mathrm{Th}(A) \stackrel{\text{def}}{=} \{\varphi \text{ close} \mid A \vDash \varphi\}$ des conséquences logiques d'une *axiomatisation* A (c'est-à-dire que A est un ensemble de formules closes).

Dans une signature L qui contient le symbole d'égalité $=$, une interprétation est *normale* si l'interprétation $=^I$ du symbole est l'égalité sur le domaine. Si une interprétation I satisfait les *axiomes de congruence*, alors $=^I$ est une congruence et on peut construire l'interprétation *quotient* $I/{=^I}$, qui est une interprétation normale qui satisfait les mêmes formules que I (c.f. lemme 15.5).

Une théorie T est *cohérente* si elle ne contient pas à la fois une formule φ et sa négation $\neg\varphi$. Elle est *complète* si, pour toute formule close φ, $\varphi \in T$ ou $\neg\varphi \in T$. Elle est *décidable* s'il existe un algorithme (qui dépend de T) qui prend en entrée une formule close φ et répond si $\varphi \in T$ ou non.

Toutes les théories ne sont pas décidables. Une manière de montrer qu'une théorie T est décidable est de montrer qu'elle permet l'*élimination des quantificateurs*, c'est-à-dire de montrer que pour toute formule φ (pas nécessairement close), il existe une formule ψ sans quantificateurs telle que $T \vDash \varphi \Leftrightarrow \psi$. C'est le cas par exemple de la théorie $\mathrm{Th}(A_{\mathrm{oldns}})$ des ordres linéaires denses non bornés stricts (c.f. lemme 15.12) et de la théorie $\mathrm{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$ de l'arithmétique linéaire sur les rationnels (c.f. lemme 15.15).

(Duparc, 2015, sec. 11.7), (David, Nour et Raffalli, 2003, chap. 3), (Goubault-Larrecq et Mackie, 1997, sec. 6.4)

Prise en isolation, une formule du premier ordre utilise des symboles *non interprétés*, qui peuvent être interprétés de manière arbitraire et potentiellement contre-intuitive. Ainsi, quand nous avons défini les signatures des exemples 10.2, 10.3 et 10.5, nous avions une intuition de quelles interprétations seraient « raisonnables », mais rien n'oblige les interprétations à respecter cette intuition. Dans la signature $\mathcal{F} \stackrel{\text{def}}{=} \emptyset$ et $\mathcal{P} \stackrel{\text{def}}{=} \{<^{(2)}, =^{(2)}\}$ de l'exemple 10.2, les formules closes

$$\exists x.\neg(x = x) \qquad\qquad \exists x.x < x$$

sont satisfiables, par exemple par l'interprétation I de domaine $D_I = \{e\}$ telle que $<^I = \{(e, e)\}$ et $=^I = \emptyset$. De manière similaire, dans la signature de l'exemple 10.5, la formule close

$$\mathit{hopper} = \mathit{easyrider}$$

est satisfiable, par exemple par l'interprétation I de domaine $D_I = \{e\}$ telle que tous les symboles de constantes soient interprétées comme e ($\mathit{shining}^I = \mathit{player}^I = \cdots = \mathit{odeon}^I = e$), et $\mathit{Film}^I = \mathit{Seance}^I = \emptyset$ et $=^I = \{(e, e)\}$.

Le problème que nous rencontrons avec ces exemples est que nous voulons restreindre nos interprétations à avoir une forme particulière, pour interdire les interprétations « contre-intuitives ».

15.1. **Théories logiques.** Rappelons que pour un ensemble S de formules et une formule φ, φ est une conséquence logique de S, noté $S \vDash \varphi$, si pour toute interprétation I, si I est un modèle de toutes les formules de S alors I est un modèle de φ. En symboles, $S \vDash \varphi$ si $I \vDash S$ (c'est-à-dire si pour toute valuation ρ et toute formule $\psi \in S$, $I, \rho \vDash \psi$) implique $I \vDash \varphi$. Rappelons aussi qu'une formule *close* est une formule sans variable libre.

(Harrisson, 2009, def. 5.4).

À noter que certains ouvrages définissent une théorie *comme un ensemble de formules closes, sans demander qu'il soit fermé par conséquence logique ; c'est le cas par exemple de (Duparc, 2015, sec. 11.7), (David, Nour et Raffalli, 2003, def. 2.5.1) et (Chang et Keisler, 1990, sec. 1.4). Dans ces notes nous parlerons plutôt d'*axiomatisations *pour de tels ensembles de formules ; voir la section 15.1.2 ci-dessous.*

Une *théorie* est un ensemble T de formules closes sur une même signature $(\mathcal{F}, \mathcal{P})$ du premier ordre, tel que, si φ est une formule close et $T \vDash \varphi$, alors $\varphi \in T$. Par exemple, l'ensemble de toutes les formules closes est une théorie – d'un intérêt toutefois assez limité.

15.1.1. *Théories de structures.* Pour une interprétation I, on définit la *théorie de* I comme l'ensemble

$$\mathrm{Th}(I) \stackrel{\text{def}}{=} \{\varphi \text{ close} \mid I \vDash \varphi\} \tag{49}$$

des formules closes dont I est un modèle. Plus généralement, pour une classe $\mathcal{K}$ d'interprétations sur la même signature, la *théorie de* $\mathcal{K}$ est l'ensemble

$$\mathrm{Th}(\mathcal{K}) \stackrel{\text{def}}{=} \{\varphi \text{ close} \mid \forall I \in \mathcal{K} \,.\, I \vDash \varphi\} \tag{50}$$

des formules closes qui sont vraies dans toutes les interprétations dans $\mathcal{K}$; avec ces notations, $\mathrm{Th}(I) = \mathrm{Th}(\mathcal{K})$ pour la classe $\mathcal{K} \stackrel{\text{def}}{=} \{I\}$ qui ne contient que I. On vérifie aisément que $\mathrm{Th}(\mathcal{K})$ est bien une théorie : si $\mathrm{Th}(\mathcal{K}) \vDash \varphi$, c'est-à-dire si pour toute interprétation I, $I \vDash \mathrm{Th}(\mathcal{K})$ implique $I \vDash \varphi$, alors en particulier pour toute interprétation $I \in \mathcal{K}$, $I \vDash \mathrm{Th}(\mathcal{K})$ par définition de $\mathrm{Th}(\mathcal{K})$, donc $I \vDash \varphi$ et donc $\varphi \in \mathrm{Th}(\mathcal{K})$.

Exemple 15.1. Prenons comme dans l'exemple 10.1 une signature $\mathcal{F} = \emptyset$ et $\mathcal{P} = \{P^{(0)}, Q^{(0)}\}$ avec deux propositions P et Q, et soit l'interprétation de domaine $D_I = \{\bullet\}$ où $P^I = 1$ et $Q^I = 0$. Alors $\mathrm{Th}(I)$ contient en particulier les formules P, $\neg Q$, $P \vee Q \vee Q$, $\neg\neg P$, $Q \vee \exists x.P$, etc.

15.1.2. *Théories axiomatiques.* L'approche précédente, qui définit des théories à partir d'une classe d'interprétations $\mathcal{K}$, se heurte rapidement au problème suivant : comment définir la classe $\mathcal{K}$ elle-même ? Ainsi, pour la signature de l'exemple 10.2, on pourrait souhaiter ne considérer que des *ordres*. Pour cela, il est pratique de spécifier directement dans la logique quelles sont les propriétés attendues de nos interprétations. Par exemple, on peut demander pour la signature de l'exemple 10.2 que $\forall x.x = x$.

En général, une *axiomatisation* A est un ensemble de formules closes sur une signature $(\mathcal{F}, \mathcal{P})$ du premier ordre. La *théorie de* A est alors définie comme l'ensemble

$$\mathrm{Th}(A) \stackrel{\text{def}}{=} \{\varphi \text{ close} \mid A \vDash \varphi\} \tag{51}$$

des conséquences logiques closes de A ; on parle parfois de « théorie axiomatique » pour une telle théorie. Lorsque A est un ensemble fini, on dit que $\mathrm{Th}(A)$ est *définissable.* Lorsque A est *récursif* – c'est-à-dire lorsqu'il existe un algorithme qui prend en entrée une formule ψ et répond si $\psi \in A$ ou non –, on dit que $\mathrm{Th}(A)$ est *récursivement axiomatisable.* Bien sûr, si $\mathrm{Th}(A)$ est définissable, alors elle est récursivement axiomatisable.

Exemple 15.2 (théorie de l'égalité pure)**.** Considérons la signature $\mathcal{P} = \{=^{(2)}\}$. Nous définissons une axiomatisation A_{eq} telle que $I \vDash A_{\text{eq}}$ si et seulement si $=^I$ est une relation d'*équivalence* sur D_I :

$$\forall x. \quad x = x \qquad \text{(réflexivité de =)}$$

$$\forall x \forall y. \quad x = y \Rightarrow y = x \qquad \text{(symétrie de =)}$$

$$\forall x \forall y \forall z. \quad (x = y \wedge y = z) \Rightarrow x = z \qquad \text{(transitivité de =)}$$

Par exemple, l'interprétation I dépeinte dans la figure 24 et définie par $D_I \stackrel{\text{def}}{=} \{a, b, c\}$ et $=^I \stackrel{\text{def}}{=} \{(a,a), (a,b), (b,a), (b,b), (c,c)\}$ est un modèle de A_{eq}.

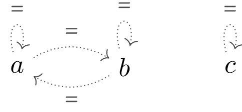


FIGURE 24. Une interprétation I telle que $I \vDash A_{\text{eq}}$.

Exemple 15.3 (théorie des congruences)**.** Considérons une signature $L = (\mathcal{F}, \mathcal{P})$ telle que $=^{(2)} \in \mathcal{P}$. Nous définissons une axiomatisation $A_{\text{cgr}}(L)$ telle que $I \vDash A_{\text{cgr}}(L)$ si et seulement si $=^I$ est une *congruence* sur D_I : on définit pour cela $A_{\text{cgr}}(L)$ comme A_{eq} à laquelle on ajoute, pour tout symbole de fonction f d'arité n dans $\mathcal{F}$

$$\forall x_1 \dots \forall x_n . \forall y_1 \dots \forall y_n . \quad x_1 = y_1 \wedge \dots \wedge x_n = y_n \Rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \tag{f-congruence}$$

et pour tout symbole de relation $R \neq =$ d'arité m dans $\mathcal{P}$

$$\forall x_1 \dots \forall x_m . \forall y_1 \dots \forall y_m . \quad x_1 = y_1 \wedge \dots \wedge x_m = y_m \Rightarrow (R(x_1, \dots, x_m) \Rightarrow R(y_1, \dots, y_m)) \tag{R-congruence}$$

Par exemple, pour $L_{\text{os}} \stackrel{\text{def}}{=} (\emptyset, \{<^{(2)}, =^{(2)}\})$, l'interprétation I de domaine $D_I \stackrel{\text{def}}{=} \{a, b, c, d\}$ où

$$<^I \stackrel{\text{def}}{=} \{(a,c), (b,c), (a,d), (b,d), (c,d)\} \qquad =^I \stackrel{\text{def}}{=} \{(a,a), (a,b), (b,a), (b,b), (c,c), (d,d)\}$$

est un modèle de $A_{\text{cgr}}(L_{\text{os}})$ (voir la figure 25).

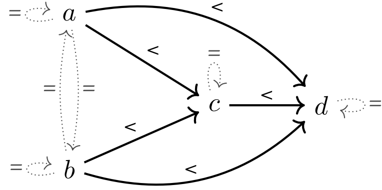


FIGURE 25. Une interprétation I telle que $I \vDash A_{\text{cgr}}(L_{\text{os}})$.

15.1.3. *Interprétations normales.* Soit $L = (\mathcal{F}, \mathcal{P})$ une signature du premier ordre qui contient le symbole d'égalité, c'est-à-dire tel que $=^{(2)} \in \mathcal{P}$. Une interprétation I de L est dite *normale* si $=^I$ est la relation d'égalité sur D_I. Quand on utilise la logique du premier ordre avec le symbole $=^{(2)}$, on souhaite généralement que celui-ci soit interprété comme l'égalité, donc ne travailler qu'avec des interprétations normales.

(Harrisson, 2009, sec. 4.1)

Comme vu dans les exemples 15.2 et 15.3, même en présence des axiomes de l'égalité, les interprétations ne sont pas forcément normales. Une manière de construire une interprétation normale à partir d'un modèle de $A_{\text{cgr}}(L)$ est de faire son quotient, c'est-à-dire d'identifier les éléments de D_I qui sont équivalents par la relation $=^I$.

Plus précisément, si une interprétation I est un modèle de $A_{\text{cgr}}(L)$, c'est-à-dire si $=^I$ est une congruence, alors on peut définir son *quotient* $I/{=^I}$. Le domaine $D_{I/=^I} \stackrel{\text{def}}{=} D_I/{=^I}$ de ce quotient est l'ensemble des classes d'équivalences de D_I ; on écrit $[e]_{=^I}$ pour la classe d'équivalence qui contient l'élément $e \in D_I$. Pour chaque symbole de fonction $f \in \mathcal{F}$ d'arité n, son interprétation dans $I/{=^I}$ est telle que $f^{I/=^I}([e_1]_{=^I}, \dots, [e_n]_{=^I}) \stackrel{\text{def}}{=} [e]_{=^I}$ si $f^I(e_1, \dots, e_n) = e$. Pour chaque symbole de relation $R \in \mathcal{P} \setminus \{=\}$ d'arité m, son interprétation dans $I/{=^I}$ est telle que $R^{I/=^I}([e_1]_{=^I}, \dots, [e_m]_{=^I}) = 1$ si $R^I(e_1, \dots, e_m) = 1$. Enfin, l'interprétation du symbole

d'égalité dans $I/{=^I}$ est l'égalité entre classes d'équivalences : l'interprétation $I/{=^I}$ est bien une interprétation normale.

Exemple 15.4 (quotients)**.** Les quotients des interprétations des figures 24 et 25 sont représentées dans les figures 26 et 27 respectivement. Dans ces figures, $[a]_{=^I} = [b]_{=^I} = \{a, b\}$ et $[c]_{=^I} = \{c\}$.

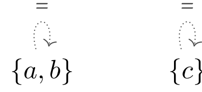


FIGURE 26. Le quotient $I/{=^I}$ de l'interprétation de la figure 24.

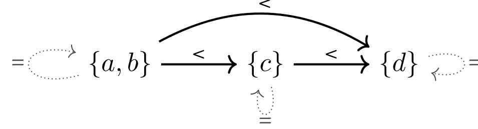


FIGURE 27. Le quotient $I/{=^I}$ de l'interprétation de la figure 25.

L'intérêt de cette construction du quotient $I/{=^I}$ d'une interprétation I est que l'on ne peut pas « distinguer » le quotient de l'interprétation I à l'aide d'une formule du premier ordre ; on dit dans ce cas que I et $I/{=^I}$ sont *élémentairement équivalents*.

Lemme 15.5 (équivalence élémentaire d'un quotient)**.** *Soit I une L-interprétation telle que $I \vDash A_{\text{cgr}}(L)$ et $I/{=^I}$ son quotient. Pour toute valuation $\rho\colon X \to D_I$, on définit $[\rho]_{=^I} : X \to D_I/{=^I}$ par $[\rho]_{=^I}(x) \stackrel{\text{def}}{=} [\rho(x)]_{=^I}$ pour tout $x \in X$. Alors, pour tout terme t et pour toute formule du premier ordre φ, $[[\![t]\!]^I_\rho]_{=^I} = [\![t]\!]^{I/=^I}_{[\rho]_{=^I}}$ et $[\![\varphi]\!]^I_\rho = [\![\varphi]\!]^{I/=^I}_{[\rho]_{=^I}}$.*

Démonstration. Le lemme se démontre par induction sur les termes t et les formules φ. Pour le cas de base des termes,

$$[\![x]\!]^{I/=^I}_{[\rho]_{=^I}} = [\rho]_{=^I}(x) = [\rho(x)]_{=^I} = [[\![x]\!]^I_\rho]_{=^I} \ .$$

Pour l'étape d'induction des termes, si $f \in \mathcal{F}_n$,

$$\begin{aligned}
[\![f(t_1, \dots, t_n)]\!]^{I/=^I}_{[\rho]_{=^I}} &= f^{I/=^I}([\![t_1]\!]^{I/=^I}_{[\rho]_{=^I}}, \dots, [\![t_n]\!]^{I/=^I}_{[\rho]_{=^I}}) \\
&= f^{I/=^I}([[\![t_1]\!]^I_\rho]_{=^I}, \dots, [[\![t_n]\!]^I_\rho]_{=^I}) && \text{par hyp. ind.} \\
&= [f^I([\![t_1]\!]^I_\rho, \dots, [\![t_n]\!]^I_\rho)]_{=^I} && \text{par def. de } f^{I/=^I} \\
&= [[\![f(t_1, \dots, t_n)]\!]^I_\rho]_{=^I} \ .
\end{aligned}$$

Pour le cas de base des formules, si $R \in \mathcal{P} \setminus \{=\}$ est d'arité m,

$$\begin{aligned}
[\![R(t_1, \dots, t_m)]\!]^{I/=^I}_{[\rho]_{=^I}} &= R^{I/=^I}([\![t_1]\!]^{I/=^I}_{[\rho]_{=^I}}, \dots, [\![t_n]\!]^{I/=^I}_{[\rho]_{=^I}}) \\
&= R^{I/=^I}([[\![t_1]\!]^I_\rho]_{=^I}, \dots, [[\![t_n]\!]^I_\rho]_{=^I}) && \text{par hyp. ind.} \\
&= R^I([\![t_1]\!]^I_\rho, \dots, [\![t_n]\!]^I_\rho) && \text{par def. de } R^{I/=^I} \\
&= [\![R(t_1, \dots, t_m)]\!]^I_\rho \ .
\end{aligned}$$

Dans le cas d'une formule atomique qui utilise le symbole d'égalité,

$$\begin{aligned}
[\![t = t']\!]^{I/=^I}_{[\rho]_{=^I}} &= ([\![t]\!]^{I/=^I}_{[\rho]_{=^I}} = [\![t']\!]^{I/=^I}_{[\rho]_{=^I}}) && \text{car } I/=^I \text{ est normale}\\
&= ([[\![t]\!]^I_\rho]_{=^I} = [[\![t']\!]^I_\rho]_{=^I}) && \text{par hyp. ind.}\\
&= ([\![t]\!]^I_\rho =^I [\![t']\!]^I_\rho) && \text{par déf. de } [.]_{=^I}\\
&= [\![t = t']\!]^I_\rho\,.
\end{aligned}$$

Les cas de la négation et de la disjonction sont faciles, et il reste l'étape d'induction pour

$$\begin{aligned}
[\![\exists x.\varphi]\!]^{I/=^I}_{[\rho]_{=^I}} &= \operatorname*{Or}_{e' \in D_I/=^I} [\![\varphi]\!]^{I/=^I}_{[\rho]_{=^I}[e'/x]}\\
&= \operatorname*{Or}_{e \in D_I} [\![\varphi]\!]^{I/=^I}_{[\rho]_{=^I}[[e]_{=^I}/x]} && \text{par déf. de } D_I/=^I \text{ et de Or}\\
&= \operatorname*{Or}_{e \in D_I} [\![\varphi]\!]^{I/=^I}_{[\rho[e/x]]_{=^I}} && \text{par déf. de } [\rho]_{=^I}\\
&= \operatorname*{Or}_{e \in D_I} [\![\varphi]\!]^{I}_{\rho[e/x]} && \text{par hyp. ind.}\\
&= [\![\exists x.\varphi]\!]^I_\rho\,.
\end{aligned}$$

□

Le lemme 15.5 permet de raisonner sur les interprétations normales à la place des interprétations : pour toute formule φ sur une signature L avec le symbole d'égalité,

- il existe une interprétation normale I et une valuation ρ telles que $I, \rho \vDash \varphi$ si et seulement s'il existe une interprétation I' et une valuation ρ' telles que $I', \rho' \vDash \varphi \wedge \bigwedge_{\psi \in A_{\text{cgr}}(L)} \psi$;
- il existe une interprétation normale I telle que, pour toute valuation ρ, $I, \rho \vDash \varphi$ si et seulement s'il existe une interprétation I' telle que, pour toute valuation ρ', $I', \rho' \vDash \varphi \wedge \bigwedge_{\psi \in A_{\text{cgr}}(L)} \psi$;
- pour toute interprétation normale I et pour toute valuation ρ, $I, \rho \vDash \varphi$ si et seulement si, pour toute interprétation I' et pour toute valuation ρ', $I', \rho' \vDash \left(\bigwedge_{\psi \in A_{\text{cgr}}(L)} \psi\right) \Rightarrow \varphi$.

Autrement dit, les notions de satisfiabilité, d'existence de modèle et de validité restreintes aux interprétations normales se réduisent aux notions correspondantes pour des interprétations arbitraires.

Les ouvrages de théorie des modèles comme (Chang et Keisler, 1990; Ebbinghaus, Flum et Thomas, 1994) ou de théorie des modèles finis comme (Libkin, 2004) travaillent uniquement avec des interprétations normales.

Exemple 15.6 (théorie des ordres stricts)**.** Reprenons la signature $L_{\text{os}} = (\mathcal{F}, \mathcal{P})$ où $\mathcal{F} = \emptyset$ et $\mathcal{P} = \{<^{(2)}, =^{(2)}\}$ de l'exemple 10.2. Nous allons définir une axiomatisation A_{os} telle que, pour toute interprétation normale I, $I \vDash A_{\text{os}}$ si et seulement si D_I est strictement ordonné par $<^I$. Nous ajoutons pour cela à $A_{\text{cgr}}(L_{\text{os}})$ de l'exemple 15.3 les deux formules suivantes :

$$\forall x. \qquad \neg(x < x) \qquad \text{(irréflexivité de <)}$$

$$\forall x \forall y \forall z. \qquad (x < y \wedge y < z) \Rightarrow x < z \qquad \text{(transitivité de <)}$$

Par exemple, les interprétations des figures 25, 27 et 28 sont des modèles de A_{os}. Si $I \vDash A_{\text{os}}$, on appelle $<^I$ un *pré-ordre strict* ; si I est de plus une interprétation normale, alors $<^I$ est un *ordre (partiel) strict.*

Exemple 15.7 (théorie des ordres linéaires stricts)**.** Continuons l'exemple 15.6. Nous définissions une axiomatisation A_{ols} qui contient les formules de A_{os} plus la formule suivante :

$$\forall x \forall y. x < y \vee x = y \vee y < x \qquad \text{(totalité de <)}$$

L'interprétation I de la figure 28 n'est pas un modèle de A_{ols} puisque $a \not<^I b$, $a \neq^I b$ et $b \not<^I a$. Les interprétations des figures 25 et 27 sont en revanche des modèles de A_{ols}. Si $I \vDash A_{\text{ols}}$, on appelle $<^I$ un *pré-ordre total strict* ; si de plus I est normale, alors on appelle $<^I$ un *ordre linéaire strict.*

FIGURE 28. Une interprétation I telle que $<^I$ soit un pré-ordre strict (à gauche) et son quotient $I/=^I$ (à droite).

Exemple 15.8 (axiomatiser une interprétation)**.** Reprenons maintenant la signature L et l'interprétation I de l'exemple 10.5. On souhaite définir une axiomatisation A qui contient $A_{\text{cgr}}(L)$ et telle que $\text{Th}(I) = \text{Th}(A)$. Pour cela, on va définir A de telle sorte que, pour toute interprétation I', $I' \vDash A$ si et seulement si $I'/=^{I'}$ est isomorphe à I ; dès lors, $A \vDash \varphi$

— ssi pour toute interprétation I', si $I' \vDash A$ alors $I' \vDash \varphi$ (par déf. de $A \vDash \varphi$),
— ssi pour toute interprétation I', si $I'/=^{I'}$ et I sont isomorphes, alors $I' \vDash \varphi$ (par hyp. sur $I' \vDash A$),
— ssi pour toute interprétation I', si $I'/=^{I'}$ et I sont isomorphes, alors $I'/=^{I'} \vDash \varphi$ (par le lemme 15.5),
— ssi $I \vDash \varphi$.

Nous allons commencer par imposer que $D_{I'}/=^{I'}$ soit inclus dans D_I par la formule $\forall x. \bigvee_{c \in \mathcal{F}_0} x = c$, c'est-à-dire

$$\forall x. x = \textit{shining} \lor x = \textit{player} \lor x = \textit{easyrider} \lor x = \textit{apocalypsenow} \lor x = \textit{kubrick} \lor x = \textit{altman} \\ \lor x = \textit{hopper} \lor x = \textit{nicholson} \lor x = \textit{robbins} \lor x = \textit{coppola} \lor x = \textit{champo} \lor x = \textit{odeon}$$

À noter que cette formule garantit en particulier que pour toute constante $c \in \mathcal{F}_0$, il existe $d \in \mathcal{F}_0$ telle que $(c^{I'}, d^{I'}) \in \{=^{I'}\}$.

Nous forçons aussi que pour toutes constantes $c \neq d$ de $\mathcal{F}_0$, leurs interprétations $c^{I'}$ et $d^{I'}$ soient distinctes, par la formule $\bigwedge_{c \neq d \in \mathcal{F}_0} \neg(c = d)$, c'est-à-dire

$$\neg(\textit{shining} = \textit{player}) \land \neg(\textit{shining} = \textit{easyrider}) \land \neg(\textit{shining} = \textit{apocalypsenow}) \land \neg(\textit{shining} = \textit{kubrick}) \\
\land \neg(\textit{shining} = \textit{altman}) \land \neg(\textit{shining} = \textit{hopper}) \land \neg(\textit{shining} = \textit{nicholson}) \land \neg(\textit{shining} = \textit{robbins}) \\
\land \neg(\textit{shining} = \textit{coppola}) \land \neg(\textit{shining} = \textit{champo}) \land \neg(\textit{shining} = \textit{odeon}) \land \neg(\textit{player} = \textit{easyrider}) \\
\land \neg(\textit{player} = \textit{apocalypsenow}) \land \neg(\textit{player} = \textit{kubrick}) \land \neg(\textit{player} = \textit{altman}) \land \neg(\textit{player} = \textit{hopper}) \\
\land \neg(\textit{player} = \textit{nicholson}) \land \neg(\textit{player} = \textit{robbins}) \land \neg(\textit{player} = \textit{coppola}) \land \neg(\textit{player} = \textit{champo}) \\
\land \neg(\textit{player} = \textit{odeon}) \land \neg(\textit{easyrider} = \textit{apocalypsenow}) \land \neg(\textit{easyrider} = \textit{kubrick}) \land \neg(\textit{easyrider} = \textit{altman}) \\
\land \neg(\textit{easyrider} = \textit{hopper}) \land \neg(\textit{easyrider} = \textit{nicholson}) \land \neg(\textit{easyrider} = \textit{robbins}) \land \neg(\textit{easyrider} = \textit{coppola}) \\
\land \neg(\textit{easyrider} = \textit{champo}) \land \neg(\textit{easyrider} = \textit{odeon}) \land \neg(\textit{apocalypsenow} = \textit{kubrick}) \land \neg(\textit{apocalypsenow} = \textit{altman}) \\
\land \neg(\textit{apocalypsenow} = \textit{hopper}) \land \neg(\textit{apocalypsenow} = \textit{nicholson}) \land \neg(\textit{apocalypsenow} = \textit{robbins}) \land \neg(\textit{apocalypsenow} = \textit{coppola}) \\
\land \neg(\textit{apocalypsenow} = \textit{champo}) \land \neg(\textit{apocalypsenow} = \textit{odeon}) \land \neg(\textit{kubrick} = \textit{altman}) \land \neg(\textit{kubrick} = \textit{hopper}) \\
\land \neg(\textit{kubrick} = \textit{nicholson}) \land \neg(\textit{kubrick} = \textit{robbins}) \land \neg(\textit{kubrick} = \textit{coppola}) \land \neg(\textit{kubrick} = \textit{champo}) \\
\land \neg(\textit{kubrick} = \textit{odeon}) \land \neg(\textit{altman} = \textit{hopper}) \land \neg(\textit{altman} = \textit{nicholson}) \land \neg(\textit{altman} = \textit{robbins}) \\
\land \neg(\textit{altman} = \textit{coppola}) \land \neg(\textit{altman} = \textit{champo}) \land \neg(\textit{altman} = \textit{odeon}) \land \neg(\textit{hopper} = \textit{nicholson}) \\
\land \neg(\textit{hopper} = \textit{robbins}) \land \neg(\textit{hopper} = \textit{coppola}) \land \neg(\textit{hopper} = \textit{champo}) \land \neg(\textit{hopper} = \textit{odeon}) \\
\land \neg(\textit{nicholson} = \textit{robbins}) \land \neg(\textit{nicholson} = \textit{coppola}) \land \neg(\textit{nicholson} = \textit{champo}) \land \neg(\textit{nicholson} = \textit{odeon}) \\
\land \neg(\textit{robbins} = \textit{coppola}) \land \neg(\textit{robbins} = \textit{champo}) \land \neg(\textit{robbins} = \textit{odeon}) \land \neg(\textit{coppola} = \textit{champo}) \\
\land \neg(\textit{coppola} = \textit{odeon}) \land \neg(\textit{champo} = \textit{odeon})$$

Cela garantit que $D_{I'}$ contienne D_I.

Il reste à imposer que les relations $\textit{Film}^{I'}$ et $\textit{Seance}^{I'}$ soient les bonnes. Pour la relation *Film*, on écrit pour cela

$$\begin{aligned}\forall x\forall y\forall z.\textit{Film}(x,y,z) \Leftrightarrow \big(\ &(x=\textit{shining}\wedge y=\textit{kubrick}\wedge z=\textit{nicholson})\\ &\vee(x=\textit{player}\wedge y=\textit{altman}\wedge z=\textit{robbins})\\ &\vee(x=\textit{easyrider}\wedge y=\textit{hopper}\wedge z=\textit{nicholson})\\ &\vee(x=\textit{easyrider}\wedge y=\textit{hopper}\wedge z=\textit{hopper})\\ &\vee(x=\textit{apocalypsenow}\wedge y=\textit{coppola}\wedge z=\textit{hopper})\big)\end{aligned}$$

Pour la relation *Seance*, on écrit de manière similaire

$$\begin{aligned}\forall x\forall y.\textit{Seance}(x,y,z) \Leftrightarrow \big(\ &(x=\textit{champo}\wedge y=\textit{shining})\\ &\vee(x=\textit{champo}\wedge y=\textit{easyrider})\\ &\vee(x=\textit{champo}\wedge y=\textit{player})\\ &\vee(x=\textit{odeon}\wedge y=\textit{easyrider})\big)\end{aligned}$$

15.1.4. *Cohérence, complétude et décidabilité.* On dit qu'une théorie T est *cohérente* si elle ne contient pas à la fois une formule φ et sa négation $\neg\varphi$. Une théorie incohérente ne peut pas avoir de modèle : pour toute interprétation I, $I \not\models T$ (sans quoi on aurait $I \models \varphi$ et $I \models \neg\varphi$, ce qui est absurde). Mais une théorie incohérente T contient alors toutes les formules closes puisqu'elle est fermée par conséquence logique : pour toute formule close ψ, on aura en effet $T \models \psi$ puisque pour toute interprétation I, $I \models T$ (qui est faux) impliquera bien $I \models \psi$.

La définition qui nous intéresse ici le plus est celle de décidabilité, *qui est un problème algorithmique.*
Mais ces trois définitions sont les clefs pour comprendre le « programme de Hilbert *» visant à assurer les fondements des mathématiques elles-mêmes, les réponses négatives apportées à ce programme par les deux théorèmes d'incomplétude de* Gödel, *et enfin la naissance de la notion de* calculabilité *avec la réponse négative à l'*Entscheidungsproblem *apportée par* Church *via le λ-calcul et par* Turing *via la notion de machine de* Turing. *Voir la section 15.3 pour plus de détails.*

Propriété 15.9. *Une théorie est cohérente si et seulement si elle a au moins un modèle, si et seulement si elle ne contient pas toutes les formules closes.*

Par suite, si $\mathcal{K} \neq \emptyset$ est une classe d'interprétations, alors $\mathrm{Th}(\mathcal{K})$ est forcément cohérente ; en revanche, si A est une axiomatisation, alors $\mathrm{Th}(A)$ est cohérente si et seulement si A a un modèle.

On dit qu'une théorie T est *complète* si, pour toute formule close φ, on a $\varphi \in T$ ou $\neg\varphi \in T$. Observons que si $\mathcal{K}$ est une classe d'interprétations, alors $\mathrm{Th}(\mathcal{K})$ est forcément complète : pour toute formule close φ, soit $\mathcal{K} \models \varphi$, soit $\mathcal{K} \not\models \varphi$. En revanche, pour une axiomatisation A donnée, $\mathrm{Th}(A)$ n'est pas forcément complète. Par exemple, considérons l'axiomatisation A_{eq} de l'exemple 15.2 et la formule $\varphi = \forall x\forall y.x = y$ qui exprime le fait que D_I contient exactement une classe d'équivalence pour $=^I$: $|D_I/{=^I}| = 1$. Alors $A_{\mathrm{eq}} \not\models \varphi$ puisqu'il existe des modèles avec strictement plus d'une classe d'équivalence, et $A_{\mathrm{eq}} \not\models \neg\varphi$ puisqu'il en existe avec exactement une classe d'équivalence.

On dit qu'une théorie T est *décidable* s'il existe un algorithme qui prend en entrée une formule φ et répond si $\varphi \in T$, c'est-à-dire un algorithme qui résout le problème de décision suivant, aussi connu comme le « *Entscheidungsproblem* ».

En théorie des modèles, on dit qu'une théorie T décide *une formule φ si $\varphi \in T$ ou $\neg\varphi \in T$, ce qui n'a pas le même sens ! Une théorie complète décide donc toutes les formules closes.*

Problème (DÉCISION DE T).
instance : une formule close φ
question : est-ce que $\varphi \in T$?

Si $T = \mathrm{Th}(A)$ est une théorie axiomatique définissable, c'est-à-dire si A est fini, alors $\varphi \in T$ si et seulement si $A \models \varphi$, ce qui d'après le lemme 4.2 de déduction est le cas si et seulement si $\big(\bigwedge_{\psi\in A}\psi\big) \Rightarrow \varphi$ est valide. La DÉCISION DE T peut ainsi se « réduire » au problème suivant dans le cas d'une théorie définissable :

Problème (VALIDITÉ).
instance : une formule φ
question : φ est-elle valide ?

En particulier, si $T = \mathrm{Th}(\emptyset)$ est la théorie de l'ensemble vide d'axiomes, la DÉCISION DE T est *exactement* le problème de validité d'une formule du premier ordre, puisqu'à partir d'une

formule φ quelconque il suffit de la clore par des quantifications universelles pour obtenir une formule équi-valide.

Dans le cas de la théorie $\mathrm{Th}(I)$ d'une interprétation I fixée, la décidabilité de $\mathrm{Th}(I)$ revient à l'existence d'un algorithme qui prend une formule close φ en entrée et répond si $I \vDash \varphi$, c'est-à-dire qu'une telle théorie est décidable si et seulement s'il existe un algorithme pour le problème suivant.

Problème (ÉVALUATION DANS I).
instance : une formule close φ
question : est-ce que $I \vDash \varphi$?

Si le problème d'ÉVALUATION DANS I était aisé à résoudre dans le cas de la logique propositionnelle (c.f. section 3), ce n'est malheureusement plus le cas dans le cas de la logique du premier ordre, et il existe des théories indécidables. Nous allons commencer par voir dans la section 15.2 une technique algorithmique qui permet de décider certaines théories. Mais voici une dernière remarque, qui fait appel à des notions que nous n'avons pas couvertes dans ces notes, et ne sera pas démontrée.

(David, Nour et Raffalli, 2003, thm. 3.6.5)

Propriété 15.10. *Si une théorie T est récursivement axiomatisable et complète, alors elle est décidable.*

(Harrisson, 2009, sec. 5.6), (David, Nour et Raffalli, 2003, sec. 3.7), (Chang et Lee, 1973, sec. 1.5)

15.2. **Élimination des quantificateurs et décidabilité.** Rappelons qu'une formule ψ est *sans quantificateurs* si elle respecte la syntaxe abstraite

$$\psi ::= \ell \mid \psi \vee \psi \mid \psi \wedge \psi \mid \bot \mid \top \qquad \text{(formules sans quantificateur)}$$

où ℓ est un littéral.

Une théorie T permet l'*élimination des quantificateurs* si pour toute formule φ, il existe une formule ψ sans quantificateur telle que φ et ψ sont *équivalentes modulo* T, c'est-à-dire telle que $T \vDash \varphi \Leftrightarrow \psi$. Cette élimination est *effective* s'il existe un algorithme d'élimination des quantificateurs pour T, c'est-à-dire un algorithme qui prend une formule φ en entrée et retourne une telle formule ψ. Si la formule φ est close, alors ψ ne contiendra aucune variable, et il est alors (généralement) assez aisé de déterminer si $T \vDash \psi$.

Appelons une formule *primitive existentielle* si elle est de la forme $\varepsilon = \exists x.\ell_1 \wedge \cdots \wedge \ell_n$ où $\ell_1, \ldots, \ell_n$ sont des littéraux tels que $x \in \mathrm{fv}(\ell_i)$ pour tout $1 \le i \le n$. Supposons que pour toute formule primitive existentielle ε, on dispose d'un algorithme qui retourne une formule sans quantificateur $\mathrm{qe}_T(\varepsilon)$ équivalente modulo T. Alors on a un algorithme général d'élimination des quantificateurs pour la théorie T :

Lemme 15.11. *Si une formule sans quantificateur $\mathrm{qe}_T(\varepsilon)$ équivalente modulo T à ε peut être calculée pour toute formule primitive existentielle ε, alors une formule sans quantificateur $\mathrm{qe}_T(\varphi)$ équivalente modulo T à φ peut être calculée pour toute formule φ.*

Démonstration. On définit $\mathrm{qe}_T(\varphi)$ par induction structurelle sur φ.

— Pour une formule atomique α, elle est déjà sans quantificateurs et il n'y a rien à faire : on pose $\mathrm{qe}_T(\alpha) \stackrel{\text{def}}{=} \alpha$.
— Pour le cas d'une formule $\neg\varphi$, on pose $\mathrm{qe}_T(\neg\varphi) \stackrel{\text{def}}{=} \neg\mathrm{qe}_T(\varphi)$ qui est bien équivalente modulo T à $\neg\varphi$.
— Pour le cas d'une formule $\varphi \vee \psi$, on pose $\mathrm{qe}_T(\varphi \vee \psi) \stackrel{\text{def}}{=} \mathrm{qe}_T(\varphi) \vee \mathrm{qe}_T(\psi)$ qui est bien équivalente modulo T à $\varphi \vee \psi$.
— Pour le cas d'une formule $\exists x.\varphi$, on construit une formule sans quantificateur $\mathrm{qe}_T(\varphi)$ équivalente modulo T par hypothèse d'induction. On a alors

$$T \vDash (\exists x.\varphi) \Leftrightarrow (\exists x.\mathrm{qe}_T(\varphi))$$

et on procède comme suit.

(1) On met $\mathrm{qe}_T(\varphi)$ sous forme normale disjonctive : $\mathrm{qe}_T(\varphi)$ est équivalente (en particulier modulo T) à $\bigvee_{1\leq j\leq m}\bigwedge_{1\leq i\leq n}\ell_{i,j}$ pour des littéraux $\ell_{i,j}$, et donc

$$T \vDash (\exists x.\varphi) \Leftrightarrow (\exists x. \bigvee_{1\leq j\leq m}\bigwedge_{1\leq i\leq n}\ell_{i,j})\;.$$

(2) Comme on a l'équivalence logique $(\exists x.\psi\vee\psi') \Leftrightarrow (\exists x.\psi)\vee(\exists x.\psi')$ pour toutes formules ψ, ψ', on en déduit que

$$T \vDash (\exists x.\varphi) \Leftrightarrow \bigvee_{1\leq j\leq m}\left(\exists x. \bigwedge_{1\leq i\leq n}\ell_{i,j}\right)\;,$$

et il suffit donc de construire une formule sans quantificateurs pour chacune des formules $\exists x.\bigwedge_{1\leq i\leq n}\ell_{i,j}$.

(3) Enfin, si ψ, ψ' sont des formules telles que $x \notin \mathrm{fv}(\psi)$, on a l'équivalence logique $(\exists x.\psi\wedge\psi') \Leftrightarrow (\psi\wedge\exists x.\psi')$. Dès lors, pour chaque formule $\exists x.\bigwedge_{1\leq i\leq n}\ell_{i,j}$, soient $I_j \stackrel{\text{def}}{=} \{1\leq i\leq n \mid x\in\mathrm{fv}(\ell_{i,j})\}$ et $\bar{I}_j \stackrel{\text{def}}{=} \{1\leq i\leq n \mid x\notin\mathrm{fv}(\ell_{i,j})\}$ les indices des littéraux où x est une variable libre ou non. On a donc

$$T \vDash (\exists x.\varphi) \Leftrightarrow \bigvee_{1\leq j\leq m}\left((\bigwedge_{i\in\bar{I}_j}\ell_{i,j})\wedge(\exists x.\bigwedge_{i\in I_j}\ell_{i,j})\right)\;.$$

(4) Chacune des formules $\exists x.\bigwedge_{i\in I_j}\ell_{i,j}$ est primitive existentielle, et d'après l'énoncé du lemme on peut calculer une formule sans quantificateur $\mathrm{qe}_T(\exists x.\bigwedge_{i\in I_j}\ell_{i,j})$ équivalente modulo T. On pose donc

$$\mathrm{qe}_T(\varphi) \stackrel{\text{def}}{=} \bigvee_{1\leq j\leq m}\left((\bigwedge_{i\in\bar{I}_j}\ell_{i,j})\wedge\mathrm{qe}_T(\exists x.\bigwedge_{i\in I_j}\ell_{i,j})\right)$$

qui est bien équivalente modulo T à $\exists x.\varphi$. □

15.2.1. *Implémentation en OCaml de l'élimination des quantificateurs.* La démonstration ci-dessus du lemme 15.11 explique comment construire une formule sans quantificateurs $\mathrm{qe}_T(\varphi)$ logiquement équivalente à φ modulo une théorie T, pour peu que l'on sache construire une formule sans quantificateurs $\mathrm{qe}_T(\varepsilon)$ qui soit logiquement équivalente à une formule primitive existentielle $\varepsilon = \exists x.\ell_1\wedge\cdots\wedge\ell_n$ où $x\in\mathrm{fv}(\ell_i)$ pour tout $1\leq i\leq n$, aussi modulo la théorie T. L'emploi d'une forme normale disjonctive à l'étape (1) est crucial dans cette démonstration, mais travailler avec des formes normales disjonctives n'est pas très adapté pour les applications que nous verrons dans la section 16.1 sur les solveurs SMT.

L'implémentation que nous allons voir travaille *de manière duale* : on suppose que l'on sait construire une formule sans quantificateurs $\mathrm{qe}_T(\upsilon)$ qui soit logiquement équivalente à une formule *primitive universelle* $\upsilon = \forall x.\ell_1\vee\cdots\vee\ell_n$ où $x\in\mathrm{fv}(\ell_i)$ pour tout $1\leq i\leq n$. Une telle formule est équivalente à la négation de la formule primitive existentielle $\exists x.\overline{\ell}_1\wedge\cdots\wedge\overline{\ell}_n$. Toute la démonstration du lemme 15.11 peut être refaite dans ce cadre, par dualité. Cela va permettre d'utiliser des formules sous forme normale conjonctive, comme déjà fait dans la section 14.4.1.

Pour rappel, on a défini des types `litteral` et `clause` comme suit

```
firstorder.ml
type litteral = Pos of string * term list | Neg of string * term list
type clause   = litteral list
```

ainsi que des fonctions auxiliaires `cnf_and_cnf`, `cnf_or_cnf` et `not_cnf` pour calculer la conjonction, la disjonction et la négation d'une formule sous forme normale conjonctive représentée comme un `clause list`.

La fonction qe ci-dessous va prendre en argument une fonction `qeprun` de type `string -> clause -> clause list` ainsi qu'une formule φ. La fonction `qeprun` implémente le calcul de $\mathrm{qe}_T(\upsilon)$ où $\upsilon = \forall x.\ell_1 \vee \cdots \vee \ell_n$ est une formule primitive universelle, représentée par son nom de variable x et sa liste de littéraux.

firstorder.ml
```
let rec qe: (string -> clause -> clause list) -> fo -> clause list =
  fun qeprun phi -> match phi with
  | Atom(r,tl)      -> [[Pos(r,tl)]]
  | Non(phi)        -> not_cnf(qe qeprun phi)
  | Et(phi,psi)     -> cnf_and_cnf (qe qeprun phi) (qe qeprun psi)
  | Ou(phi,psi)     -> cnf_or_cnf  (qe qeprun phi) (qe qeprun psi)
  | Ex(x,phi)       -> not_cnf(qe qeprun (Pt(x,Non(phi))))
  | Pt(x,phi)       ->
     (* cas d'une formule ∀x.φ *)
     begin
       (* étape (1) : mise sous forme normale conjonctive de qe_T(φ)
          c'est le cas par construction : qe_T(φ) = ∀x.⋀_{1≤i≤n} ⋁_{1≤j≤m} ℓ_{i,j}  *)
       let qephi: clause list = qe qeprun phi in
       (* étape (2) : on a (∀x.⋀_{1≤i≤n} ⋁_{1≤j≤m} ℓ_{i,j}) ⇔ (⋀_{1≤i≤n} ∀x.⋁_{1≤j≤m} ℓ_{i,j})
          et on va traiter chaque formule ∀x.⋁_{1≤j≤m} ℓ_{i,j} séparément *)
       concat_map (fun cl ->
           (* étape (3) : on distingue les littéraux ℓ_{i,j} pour 1 ≤ j ≤ m
              tels que x ∈ fv(ℓ_{i,j}), qu'on met dans `xfv' ;
              les autres vont dans `xnfv' *)
           let (xfv,xnfv) =
             List.partition (function
                   Pos(r, tl) ->
                    List.mem x
                      (List.fold_left (fun vs t' -> aux_tv t' vs) [] tl)
                 | Neg(r, tl) ->
                    List.mem x
                      (List.fold_left (fun vs t' -> aux_tv t' vs) [] tl))
               cl in
           (* étape (4) : on appelle qeprun sur chaque disjonction
              des littéraux qui ont x comme variable libre *)
           let xfv': clause list = qeprun x xfv in
           (* finalement, on distribue les littéraux dans lesquels
              x n'apparaît pas *)
           List.rev_map (fun cl -> List.rev_append cl xnfv) xfv') qephi
     end
```

(Harrisson, 2009, sec. 5.6), (David, Nour et Raffalli, 2003, sec. 3.7.2), (Chang et Lee, 1973, sec. 1.5)

15.2.2. *Théorie des ordres linéaires denses non bornés.* Reprenons la signature avec $\mathcal{F} = \emptyset$ et $\mathcal{P} = \{<^{(2)}, =^{(2)}\}$ et l'axiomatisation A_{ols} des ordres linéaires stricts des exemples 15.2, 15.3, 15.6 et 15.7. Nous définissons l'axiomatisation A_{oldns} des ordres linéaires denses non bornés en ajoutant à A_{ols} les deux formules suivantes :

$$\forall x \forall y \exists z.(x < y) \Rightarrow (x < z \wedge z < y) \qquad \text{(densité)}$$

$$\forall x \exists y \exists z. z < x \wedge x < y \qquad \text{(non borné)}$$

Observons que l'interprétation I de domaine $\mathbb{Q}$ où $<^I$ est l'ordre strict sur les rationnels et $=^I$ l'égalité sur $\mathbb{Q}$ est un modèle de A_{ols}, qui est donc une théorie cohérente.

Lemme 15.12. *La théorie* $\mathrm{Th}(A_{\text{oldns}})$ *permet effectivement l'élimination des quantificateurs.*

Démonstration. Avant de commencer la démonstration, on peut remarquer que

(0) un littéral négatif sur $(\mathcal{F}, \mathcal{P})$ est

– soit de la forme $\neg(x = y)$ et alors il est équivalent à $x < y \vee y < x$ modulo A_{oldns} par totalité de $<$,
– soit de la forme $\neg(x < y)$ et alors il est équivalent à $x = y \vee y < x$ modulo A_{oldns} par totalité de $<$.

Par le lemme 15.11, il suffit de pouvoir calculer une formule $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\varepsilon)$ pour une formule $\varepsilon = \exists x.\psi$ primitive existentielle. Par la remarque (0) précédente, tous les littéraux dans ψ peuvent être supposés positifs.

(A) S'il existe un littéral $x < x$ dans ψ, on a alors ε équivalente à $\bot$ modulo $\text{Th}(A_{\text{oldns}})$ par irréflexivité de $<$ et on pose $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\varepsilon) \stackrel{\text{def}}{=} \bot$.

(B) S'il existe un littéral $x = y$ dans ψ avec x un nom de variable différent de y, on a alors ε équivalente à $\psi[y/x]$ modulo $\text{Th}(A_{\text{oldns}})$ par transitivité de $=$ et par R-congruence et on pose $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\varepsilon) \stackrel{\text{def}}{=} \psi[y/x]$.

(C) Sinon, un littéral de la forme $x = x$ dans ψ peut être simplement supprimé par réflexivité de $=$,

(D) et il ne reste plus que des littéraux de la forme $y < x$ ou de la forme $x < z$: on a une formule équivalente modulo $\text{Th}(A_{\text{oldns}})$ à

$$\exists x.(\bigwedge_{1\leq i\leq m} y_i < x) \wedge (\bigwedge_{1\leq j\leq n} x < z_j)$$

où $x \neq y_i$ et $x \neq z_j$ pour tous $1 \leq i \leq m$ et $1 \leq j \leq n$.

(a) Si $m = 0$ ou $n = 0$, puisque $<$ est non borné on peut poser $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\varepsilon) \stackrel{\text{def}}{=} \top$.

(b) Sinon, par transitivité de $<$ et densité on définit notre formule comme $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\varepsilon) \stackrel{\text{def}}{=} \bigwedge_{1\leq i\leq m} \bigwedge_{1\leq j\leq n} y_i < z_j$. □

Exemple 15.13. Considérons la formule $\exists x\exists y\exists z.x < y \wedge y < z \wedge \neg(x < z)$. Cette formule contredit la transitivité de $<$, et nous allons montrer que $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\varphi) = \bot$.
Par les arguments du lemme 15.11 (étapes 2 et 3), cette formule est équivalente à

$$\exists x\exists y.x < y \wedge \exists z.y < z \wedge \neg(x < z)$$

où la formule $\exists z.y < z \wedge \neg(x < z)$ est primitive existentielle avec z qui apparaît dans chacun des deux littéraux.
L'étape 0 du lemme 15.12 nous fait par traduire $\neg(x < z)$ en $z < x \vee x = z$ modulo $\text{Th}(A_{\text{oldns}})$: on obtient par distributivité deux nouvelles formules primitives existentielles $\exists z.y < z \wedge z < x$ et $\exists z.y < z \wedge x = z$. Ces formules correspondent respectivement aux cas (D.b) et (B) et on a $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\exists z.y < z \wedge z < x) = y < x$ et $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\exists z.y < z \wedge x = z) = y < x$. Notre formule équivalente modulo $\text{Th}(A_{\text{oldns}})$ est maintenant

$$\exists x\exists y.x < y \wedge (y < x \vee y < x)$$

(que l'on peut évidemment simplifier) et où la formule $\exists y.x < y \wedge y < x$ est primitive existentielle avec y qui apparaît dans chacun de ses deux littéraux.
Cette formule correspond au cas (D.b) du lemme 15.12 et on obtient $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\exists y.x < y \wedge y < x) = x < x$. Notre formule est maintenant

$$\exists x.x < x$$

qui est primitive existentielle avec x qui apparaît dans son unique littéral.
En appliquant le cas (A) du lemme 15.12, on obtient $\text{qe}_{\text{Th}(A_{\text{oldns}})}(\exists x.x < x) = \bot$, qui est bien la formule sans quantificateur qu'on espérait trouver.

Théorème 15.14. *La théorie* $\text{Th}(A_{oldns})$ *est décidable.*

Cette preuve montre que, pour toute formule close φ, soit $\mathrm{qe}_{\mathrm{Th}(A_{oldns})}(\varphi) = \top$ soit $\mathrm{qe}_{\mathrm{Th}(A_{oldns})}(\varphi) = \bot$, et donc soit $\mathrm{Th}(A_{oldns}) \vDash \varphi \Leftrightarrow \top$ soit $\mathrm{Th}(A_{oldns}) \vDash \varphi \Leftrightarrow \bot$, donc soit $\mathrm{Th}(A_{oldns}) \vDash \varphi$ soit $\mathrm{Th}(A_{oldns}) \vDash \neg\varphi$: la théorie $\mathrm{Th}(A_{oldns})$ est donc aussi complète.

Démonstration. Par le lemme 15.12, pour toute formule close φ, on peut calculer une formule sans variables $\mathrm{qe}_{\mathrm{Th}(A_{\mathrm{oldns}})}(\varphi)$ équivalente modulo $\mathrm{Th}(A_{\mathrm{oldns}})$. Il suffit donc d'être capable de déterminer si $\mathrm{Th}(A_{\mathrm{oldns}}) \vDash \psi$ pour ψ sans variables. Comme $\mathcal{F}$ est vide, $T(\mathcal{F})$ l'ensemble des termes clos est lui aussi vide : ψ est soit $\top$ soit $\bot$. □

La démonstration du lemme 15.12 peut être traduite en une implémentation en OCaml de l'élimination des quantificateurs pour la théorie $\mathrm{Th}(A_{oldns})$. En suivant la démarche de la section 15.2.1, il nous faut implémenter une fonction `qeprun_oldns` de type `string -> clause -> clause list` qui prend en entrée une formule primitive universelle $\forall x. \bigvee_{1\leq j\leq m} \ell_j$ où $x \in \mathrm{fv}(\ell_j)$ pour tout $1 \leq j \leq m$ (représentée par le nom de variable x et la liste `[`ℓ_1 `;…;` ℓ_m`]`), et retourne une formule sans quantificateur sous forme d'une liste de clauses.

L'étape (0) du lemme 15.12, puisque nous travaillons de manière duale, est de n'avoir plus que des littéraux *négatifs*.

firstorder.ml
```
let negate_oldns: clause -> clause list = fun cl ->
  let (posl,negl) = List.partition
                      (function Pos(_,_) -> true | _ -> false) cl in
  let rec negative: clause -> clause list -> clause list = fun cl acc ->
    match cl with
    | [] -> acc
    (* (y < z) ⇔ ((y ≠ z) ∧ (z ≮ y))  *)
    | Pos("<",[y;z])::cl ->
       cnf_or_cnf
         [[Neg("=",[y;z])];[Neg("<",[z;y])]]
         (negative cl acc)
    (* (y = z) ⇔ ((y ≮ z) ∧ (z ≮ y))  *)
    | Pos("=",[y;z])::cl ->
       cnf_or_cnf
         [[Neg("<",[y;z])];[Neg("<",[z;y])]]
         (negative cl acc)
    | l::_ -> failwith ("invalid litteral"^(string_of_litteral l))
  in negative posl [negl]
```

L'analyse des cas (A), (B), (C) et (D) de la démonstration du lemme 15.12 s'applique ensuite à cette forme duale. On utilise dans le cas (B) de cette implémentation une fonction `csubst` qui applique une substitution à une clause.

firstorder.ml
```
let qe_oldns: string -> clause -> clause list = fun x cl ->
  let rec aux: clause -> clause -> clause -> clause list =
    fun less more cl -> match cl with
    (* A : (∀x.x ≮ x ∨ ψ) est toujours vraie : on retourne ∅ *)
    | Neg("<",[Var y;Var z])::_ when (x=y && x=z) -> []
    (* B : (∀x.(x ≠ y) ∨ ψ) ⇔ (∀x.(x = y) ⇒ ψ) : on retourne {ψ[y/x]} *)
    | Neg("=",[Var y;Var z])::cl' when (x=y) ->
       [csubst [(x,Var z)] (List.rev_append (List.rev_append less more) cl')]
    | Neg("=",[Var y;Var z])::cl' when (x=z) ->
       [csubst [(x,Var y)] (List.rev_append (List.rev_append less more) cl')]
    (* C : (∀x.x ≠ x ∨ ψ) ⇔ (∀x.ψ) *)
    | Neg("=",[Var y;Var z])::cl' when (x=y && x=z) -> aux less more cl'
    (* D : tri des littéraux restants : less = ⋁_i y_i ≮ x ; more = ⋁_j x ≮ z_j *)
    | Neg("<",[Var y;Var z])::cl' when (x=y) ->
       aux less (Neg("<",[Var y;Var z])::more) cl'
    | Neg("<",[Var y;Var z])::cl' when (x=z) ->
```

```
      aux (Neg("<",[Var y;Var z])::less) more cl'
    | l::_ -> failwith  ("invalid litteral"^(string_of_litteral l))
    (* on retourne {⋁_i ⋁_j y_i ≮ z_j} *)
    | [] -> [concat_map (function Neg("<",[Var y;Var x]) ->
                          List.rev_map (function Neg("<",[Var x;Var z]) ->
                                            Neg("<",[Var y;Var z])
                                                  | _ -> failwith "invalid litteral")
                            more
                              | _ -> failwith "invalid litteral")
               less]
  in concat_map (aux [] []) (negate_oldns cl)
```

15.2.3. *Théorie de l'arithmétique linéaire rationnelle.* La section 15.2.2 était l'occasion de montrer la décidabilité d'une théorie axiomatique. Ici, nous allons montrer la décidabilité de la théorie d'une structure dénotée par $(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$. Nous considérons pour cela la signature infinie $\mathcal{F} = \{1^{(0)}, (q\cdot^{(1)})_{q\in\mathbb{Q}}, +^{(2)}\}$ et $\mathcal{P} = \{<^{(2)}, =^{(2)}\}$. Notre interprétation a pour domaine $\mathbb{Q}$ l'ensemble des nombres rationnels. L'interprétation de la constante 1 est le nombre rationnel $1 \in \mathbb{Q}$, chacune des fonctions unaires $q\cdot$ pour $q \in \mathbb{Q}$ est interprétée comme la fonction $x \mapsto q \cdot x$, la fonction binaire $+$ est interprétée comme l'addition entre rationnels, et les deux relations $<$ et $=$ sont interprétées respectivement comme l'ordre strict et l'égalité sur $\mathbb{Q}$. À noter que la structure $(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$ est en particulier un modèle de A_{odlns}.

Lemme 15.15 (Fourier-Motzkin)**.** *La théorie* $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$ *permet effectivement l'élimination des quantificateurs.*

Démonstration. La démonstration procède essentiellement comme celle du lemme 15.12. Par le lemme 15.11, il suffit de pouvoir calculer une formule $\text{qe}_{\text{Th}(\mathbb{Q},1,(q\cdot)_{q\in\mathbb{Q}},+,<)}(\varepsilon)$ pour une formule primitive existentielle $\varepsilon = \exists x.\psi$. Comme dans le lemme 15.12 (étape 0), on peut se ramener au cas où ψ ne comprend que des littéraux positifs, qui sont donc de la forme $t < t'$ ou $t = t'$ avec $x \in \text{fv}(t)$ ou $x \in \text{fv}(t')$. Ces inéquations et équations sur $\mathbb{Q}$ peuvent de plus être réécrites de manière équivalente comme des littéraux de la forme $x < t$, $t < x$ ou $x = t$; par exemple, $5x + y < 2x - y + z$ peut se réécrire comme $x < -\frac{2}{3}y + \frac{1}{3}z$.

On applique les mêmes étapes de raisonnement que dans la preuve du lemme 15.12. (A) Si ψ contient un littéral $x < x$, alors $\text{qe}_{\text{Th}(\mathbb{Q},1,(q\cdot)_{q\in\mathbb{Q}},+,<)}(\varepsilon) = \bot$ convient. (B) Si ψ contient un littéral $x = t$ pour t différent de x, alors $\text{qe}_{\text{Th}(\mathbb{Q},1,(q\cdot)_{q\in\mathbb{Q}},+,<)}(\varepsilon) = \psi[t/x]$ convient. (C) Un littéral $x = x$ peut simplement être supprimé, et il nous reste donc (D) le cas où ε est de la forme

$$\exists x.(\bigwedge_{1\leq i\leq m} t_i < x) \wedge (\bigwedge_{1\leq j\leq n} x < t'_j)$$

où $x \notin \text{fv}(t_i)$ et $x \notin \text{fv}(t'_j)$ pour tout i, j. Comme dans le lemme 15.12, si $m = 0$ ou $n = 0$ (D.a), alors comme $\mathbb{Q}$ est non borné, $\text{qe}_{\text{Th}(\mathbb{Q},1,(q\cdot)_{q\in\mathbb{Q}},+,<)}(\varepsilon) = \top$ convient. Et comme $\mathbb{Q}$ est dense et $<$ est transitif, dans le cas contraire (D.b), $\text{qe}_{\text{Th}(\mathbb{Q},1,(q\cdot)_{q\in\mathbb{Q}},+,<)}(\varepsilon) = \bigwedge_{1\leq i\leq m}\bigwedge_{1\leq j\leq n} t_i < t'_j$ convient. □

Théorème 15.16. *La théorie* $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$ *est décidable.*

Démonstration. Par les lemmes 15.11 et 15.15, savoir si $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <) \vDash \varphi$ pour φ close se ramène savoir si $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <) \vDash \text{qe}_{\text{Th}(\mathbb{Q},1,(q\cdot)_{q\in\mathbb{Q}},+,<)}(\varphi)$, donc pour une formule sans variable. Cette formule est une combinaison booléenne de littéraux de la forme $t < t'$ ou $t = t'$ où t et t' n'utilisent que la constante 1, l'addition $+$ et les multiplications par des constantes de $\mathbb{Q}$, et il suffit donc de calculer les valeurs de t et t' dans $\mathbb{Q}$. □

La démonstration du lemme 15.15 peut elle aussi être traduite en une implémentation en OCaml de l'élimination des quantificateurs pour la théorie $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$. Toujours en

suivant la démarche de la section 15.2.1, il nous faut implémenter une fonction `qeprun_lqa` de type `string -> clause -> clause list` qui prend en entrée une formule primitive universelle $\forall x. \bigvee_{1\leq j\leq m} \ell_j$ où $x \in \mathrm{fv}(\ell_j)$ pour tout $1 \leq j \leq m$ (représentée par le nom de variable `x` et la liste `[`ℓ_1 `;…;` ℓ_m`]`), et retourne une formule sans quantificateur sous forme d'une liste de clauses.

(Goubault-Larrecq et Mackie, 1997, thm. 6.20), (Börger, Grädel et Gurevich, 1997, sec. 2.1)

15.3. * **Indécidabilité.** Il existe d'autres théories décidables, dont notablement
- l'*arithmétique de Presburger* $\mathrm{Th}(\mathbb{Z}, +, <)$, c.f. (Harrisson, 2009, sec. 5.7 ; David, Nour et Raffalli, 2003, sec. 3.7.5 ; Carton, 2008, thm. 3.63) et
- l'*arithmétique des réels* $\mathrm{Th}(\mathbb{R}, +, \times)$, c.f. (Harrisson, 2009, sec. 5.9 ; David, Nour et Raffalli, 2003, sec. 3.7.4).

Cependant, les théories, et en particulier les théories arithmétiques, deviennent indécidables dès qu'elles sont assez « riches ». En particulier, toute théorie qui étend l'*arithmétique élémentaire* de la section 15.3.1 ci-dessous est indécidable. Comme le concept d'indécidabilité sera plutôt exploré en M1 dans le cours « calculabilité et complexité », dans cette section nous allons seulement donner des exemples sans faire de preuves.

(Cori et Lascar, 2003, not. 1.5), (Duparc, 2015, ex. 488), (David, Nour et Raffalli, 2003, déf. 3.4.1)

15.3.1. *Indécidabilité de la théorie de l'arithmétique élémentaire.* Posons $\mathcal{F} = \{0^{(0)}, s^{(1)}, +^{(2)}, \times^{(2)}\}$ et $\mathcal{P} = \{=^{(2)}\}$. L'*arithmétique élémentaire* Q est l'axiomatisation finie suivante :

$$\begin{aligned}
\forall x.&\quad \neg(s(x) = 0)\\
\forall x.&\quad \neg(x = 0) \Rightarrow \exists y.x = s(y)\\
\forall x\forall y.&\quad s(x) = s(y) \Rightarrow x = y\\
\forall x.&\quad x + 0 = 0\\
\forall x\forall y.&\quad x + s(y) = s(x + y)\\
\forall x.&\quad x \times 0 = 0\\
\forall x\forall y.&\quad x \times s(y) = (x \times y) + x
\end{aligned}$$

à laquelle on ajoute les axiomes de l'égalité A_{eq} de l'exemple 15.2 ainsi que les axiomes de congruence suivants :

$$\begin{aligned}
\forall x\forall x'\forall y.&\quad (x = x' \wedge s(x) = y) \Rightarrow s(x') = y\\
\forall x\forall x'\forall y\forall z.&\quad (x = x' \wedge x + y = z) \Rightarrow x' + y = z\\
\forall x\forall x'\forall y\forall z.&\quad (x = x' \wedge y + x = z) \Rightarrow y + x' = z\\
\forall x\forall x'\forall y\forall z.&\quad (x = x' \wedge x \times y = z) \Rightarrow x' \times y = z\\
\forall x\forall x'\forall y\forall z.&\quad (x = x' \wedge y \times x = z) \Rightarrow y \times x' = z
\end{aligned}$$

Le résultat suivant énonce que cette « petite » axiomatisation définit une théorie $\mathrm{Th}(Q)$ qui est indécidable. Les premières démonstrations de résultats de ce type sont célèbres et ont été obtenues par Church en 1936 et par Turing en 1937, pour des théories arithmétiques récursivement axiomatisables plus riches que $\mathrm{Th}(Q)$.

(Cori et Lascar, 2003, thm. 4.1), (David, Nour et Raffalli, 2003, thm. 3.6.6)

Théorème 15.17 (Mostowski, Robinson et Tarski)**.** *Si T est une théorie cohérente qui contient Q, alors T est indécidable.*

Une conséquence de ce théorème est qu'il n'existe pas d'algorithme qui résout le problème de VALIDITÉ. En effet, si un tel algorithme existait, alors pour n'importe quelle formule close φ' pour laquelle on souhaiterait savoir si $\varphi' \in \mathrm{Th}(Q)$, on pourrait donner la formule $\varphi \stackrel{\text{def}}{=} (\bigwedge_{\psi\in Q} \psi) \Rightarrow \varphi'$ en entrée à l'algorithme pour la validité. Comme $\vDash \varphi$ si et seulement si $Q \vDash \varphi'$, l'existence d'un tel algorithme contredirait le théorème de Mostowski, Robinson et Tarski.

(Duparc, 2015, thm. 489), (Goubault-Larrecq et Mackie, 1997, thm. 6.20), (Cori et Lascar, 2003, cor. 4.2), (David, Nour et Raffalli, 2003, cor. 3.6.7)

Théorème 15.18 (indécidabilité de la VALIDITÉ)**.** *Il n'existe pas d'algorithme qui prend en entrée une formule du premier ordre et retourne si oui ou non elle est valide.*

15.3.2. *Indécidabilité des équations diophantiennes.* Une *équation diophantienne* est une équation $p(x_1, \ldots, x_n) = 0$ où p est un polynôme à coefficients dans $\mathbb{Z}$ pour laquelle on cherche une solution $(x_1, \ldots, x_n)$ dans $\mathbb{Z}^n$. Par exemple, $3x_1^2 - 2x_1x_2 - x_2^2x_3 - 7 = 0$ est une équation diophantienne qui a une solution $(1, 2, -2)$, tandis que $x_1^2 + x_2^2 + 1 = 0$ n'a pas de solution. Le problème de décision correspondant est connu comme « dixième problème de Hilbert ».

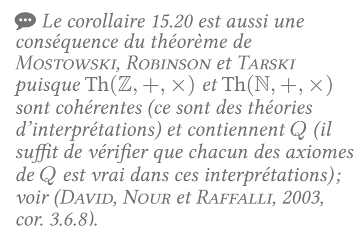
Problème (ÉQUATION DIOPHANTIENNE)**.**
instance : une équation diophantienne $p(x_1, \ldots, x_n) = 0$
question : existe-t'il une solution dans $\mathbb{Z}^n$?

Une conséquence d'un résultat de Matiassevitch en 1970 est qu'il n'existe pas d'algorithme capable de dire s'il existe au moins une solution à une équation diophantienne.

Théorème 15.19 (Matiassevitch)**.** *Il n'existe pas d'algorithme qui prend en entrée une équation diophantienne et retourne si oui ou non il en existe une solution.*

Une conséquence du théorème de Matiassevitch est que les théories arithmétiques sur les entiers ou sur les naturels sont indécidables. Ici, on prend $\mathcal{F} = \{+^{(2)}, \times^{(2)}\}$ et $\mathcal{P} = \{=^{(2)}\}$ et on dénote par $(\mathbb{Z}, +, \times)$ l'interprétation sur les entiers de cette signature et par $(\mathbb{N}, +, \times)$ celle sur les naturels.

Corollaire 15.20. *Les théories* $\mathrm{Th}(\mathbb{Z}, +, \times)$ *et* $\mathrm{Th}(\mathbb{N}, +, \times)$ *sont indécidables.*



Le corollaire 15.20 est aussi une conséquence du théorème de Mostowski, Robinson et Tarski puisque $\mathrm{Th}(\mathbb{Z}, +, \times)$ *et* $\mathrm{Th}(\mathbb{N}, +, \times)$ *sont cohérentes (ce sont des théories d'interprétations) et contiennent Q (il suffit de vérifier que chacun des axiomes de Q est vrai dans ces interprétations); voir (David, Nour et Raffalli, 2003, cor. 3.6.8).*
L'intérêt de passer par le théorème de Matiassevitch est que les formules φ pour lesquelles on ne sait pas répondre si $\varphi \in \mathrm{Th}(\mathbb{Z}, +, \times)$ ni si $\varphi \in \mathrm{Th}(\mathbb{N}, +, \times)$ sont particulièrement simples.

Démonstration. Comme vu dans l'exemple 13.6, on peut exprimer par les formules $\mathit{zero}(z) \stackrel{\text{def}}{=} z + z = z$ et $\mathit{un}(u) \stackrel{\text{def}}{=} \neg\mathit{zero}(u) \land u \times u = u$ que les variables z et u soient valuées à 0 et 1 respectivement.

Étant donnée une équation diophantienne $p(x_1, \ldots, x_n) = 0$, on peut l'écrire de manière équivalente comme $p_1(x_1, \ldots, x_n) = p_2(x_1, \ldots, x_n)$ où tous les coefficients de p_1 et p_2 sont dans $\mathbb{N}$. Par exemple, $3x_1^2 - 2x_1x_2 - x_2^2x_3 - 7 = 0$ s'écrit de manière équivalente comme $3x_1^2 = 2x_1x_2 + x_2^2x_3 + 7$.

On peut alors construire une formule universelle close

$$\varphi \stackrel{\text{def}}{=} \forall z \forall u.(\mathit{zero}(z) \land \mathit{un}(u)) \Rightarrow \forall x_1 \ldots \forall x_n.\neg(t_1(x_1, \ldots, x_n, z, u) = t_2(x_1, \ldots, x_n, z, u))$$

où t_1 et t_2 sont des termes qui représentent les polynômes p_1 et p_2. Par exemple, $3x_1^2$ est représenté par $x_1 \times x_1 + x_1 \times x_1 + x_1 \times x_1$, et $2x_1x_2 + x_2^2x_3 + 7$ est représenté par $x_1 \times x_2 + x_1 \times x_2 + x_2 \times x_2 \times x_3 + u + u + u + u + u + u + u$. La formule φ est telle que $\varphi \in \mathrm{Th}(\mathbb{Z}, +, \times)$ si et seulement si l'équation diophantienne $p(x_1, \ldots, x_n) = 0$ n'a pas de solution. Dès lors, si $\mathrm{Th}(\mathbb{Z}, +, \times)$ était décidable, cela contredirait le théorème de Matiassevitch.

Pour la théorie $\mathrm{Th}(\mathbb{N}, +, \times)$, on fait le même raisonnement en observant que l'équation diophantienne $p(x_1, \ldots, x_n) = 0$ a une solution dans $\mathbb{Z}^n$ si et seulement si $p(y_1 - z_1, \ldots, y_n - z_n) = 0$ a une solution dans $\mathbb{N}^{2n}$. □

Une conséquence du corollaire 15.20 est que le problème d'ÉVALUATION DANS I ne peut pas être résolu algorithmiquement quand $I = (\mathbb{Z}, +, \times)$ ou $I = (\mathbb{N}, +, \times)$.

Théorème 15.21 (indécidabilité de l'ÉVALUATION)**.** *Pour $I = (\mathbb{Z}, +, \times)$ ou $I = (\mathbb{N}, +, \times)$, il n'existe pas d'algorithme qui prend en entrée une formule φ et retourne si $I \vDash \varphi$ ou non, et ce même si φ est universelle.*

Corollaire 15.22. *Les théories* $\mathrm{Th}(\mathbb{Z}, +, \times)$ *et* $\mathrm{Th}(\mathbb{N}, +, \times)$ *ne sont pas récursivement axiomatisables.*

Démonstration. Si $\mathrm{Th}(\mathbb{Z}, +, \times)$ (ou $\mathrm{Th}(\mathbb{N}, +, \times)$) était récursivement axiomatisable, alors comme elle est complète – car c'est la théorie d'une interprétation – elle serait décidable par la propriété 15.10, contredisant le corollaire 15.20. □

Voir la page Wikipédia pour mieux comprendre les tenants et aboutissants de ces théorèmes.

15.3.3. *Théorèmes d'incomplétude.* Difficile de ne pas mentionner pour finir un résultat fameux de Gödel : ses deux théorèmes d'incomplétude. Nous sommes armés pour démontrer le premier de ces deux théorèmes, dans une version améliorée par Rosser.

(Cori et Lascar, 2003, thm. 4.3), (Goubault-Larrecq et Mackie, 1997, thm. 6.36), (Duparc, 2015, thm. 487), (David, Nour et Raffalli, 2003, thm. 3.6.9.1)

Théorème 15.23 (d'incomplétude de Gödel–Rosser)**.** *Soit T une théorie récursivement axiomatisable et cohérente contenant Q. Alors T n'est pas complète.*

Démonstration. Soit T une théorie récursivement axiomatisable et cohérente contenant Q. Par le théorème 15.17 de Mostowski, Robinson et Tarski, T est une théorie indécidable. Or, par la propriété 15.10, si T était complète, elle serait décidable : T est donc incomplète. □

Le second de ces deux théorèmes nécessiterait une démonstration un peu technique, où l'on « arithmétise » l'existence d'une preuve de cohérence de la théorie T comme une formule close $\mathrm{Coh}(T)$ (dans la signature de l'arithmétique). Accessoirement, la démonstration utilise une théorie axiomatique plus riche que $\mathrm{Th}(Q)$ appelée l'*arithmétique de Peano*, et que l'on notera ici PA.

(Cori et Lascar, 2003, thm. 4.5), (Goubault-Larrecq et Mackie, 1997, thm. 6.36), (Duparc, 2015, thm. 491), (David, Nour et Raffalli, 2003, thm. 3.6.9.2)

Théorème 15.24 (d'incomplétude de Gödel)**.** *Soit T une théorie récursivement axiomatisable et cohérente contenant* PA. *Alors T ne contient pas* $\mathrm{Coh}(T)$.

Autrement dit, bien qu'il soit possible d'*exprimer* par une formule logique $\mathrm{Coh}(T)$ le fait que la théorie T soit cohérente, on ne pourra pas le déduire comme une conséquence « interne » de la théorie T.

16. Satisfiabilité modulo théorie

Résumé. Un *solveur modulo théorie* pour une théorie T est un programme qui prend en entrée une formule φ et répond s'il existe une interprétation I et une valuation ρ telles que $I \vDash \psi$ pour toute formule $\psi \in T$ et $I, \rho \vDash \varphi$ (et retourne I et ρ le cas échéant).

En pratique, les solveurs modulo théorie savent traiter des formules φ qui sont des conjonctions de littéraux, font appel à un solveur SAT pour pouvoir traiter des formules sans quantificateurs, et peuvent utiliser l'élimination des quantificateurs pour traiter des formules générales.

Les solveurs modulo théorie utilisent un langage standard appelé *SMT-LIB* pour décrire la signature $L = (\mathcal{F}, \mathcal{P})$, la théorie T, et la formule φ.

Rappelons qu'une formule φ est satisfiable s'il existe une interprétation I et une valuation ρ telles que $I, \rho \vDash \varphi$. Comme vu dans la section 15, savoir si une formule prise en isolation est satisfiable n'est souvent pas suffisant, car on souhaite restreindre les interprétations possibles des symboles de fonction et de relation utilisées par la formule. On fixe alors une théorie T pour expliciter les propriétés de nos interprétations ; le problème considéré en pratique est donc le suivant.

Problème (SATISFIABILITÉ MODULO T).

instance : une formule φ

question : existe-t'il une interprétation I et une valuation ρ telles que $I \vDash T$ et $I, \rho \vDash \varphi$?

Un *témoin* de SATISFIABILITÉ MODULO T est alors une interprétation I et une valuation ρ telles que $I \vDash T$ et $I, \rho \vDash \varphi$.

Dans beaucoup de cas d'intérêt pratique, la théorie T elle-même est la théorie $\mathrm{Th}(I)$ d'une interprétation I fixée, comme $(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$ ou $(\mathbb{R}, +, \times)$. Dans ce cas le problème de SATISFIABILITÉ MODULO T revient à trouver s'il existe une valuation ρ telle que $I, \rho \vDash \varphi$.

16.1. Utilisation de solveurs SMT.

Un *solveur modulo théorie* (SMT) est un programme qui cherche à résoudre la problème de SATISFIABILITÉ MODULO T et retourne un témoin le cas échéant. Il existe de nombreux solveurs SMT. Dans ces notes nous utiliserons le solveur Z3 [12]. La lecture du tutoriel `https://jfmc.github.io/z3-play/` est chaudement recommandée.

16.1.1. *Principes de base des solveurs SMT.*

Dans leur forme la plus basique, les solveurs SMT ont été développés pour résoudre le problème de SATISFIABILITÉ MODULO T pour des formules φ sans quantificateur sur une théorie T fixée. Ainsi, on pourrait chercher à déterminer si la formule

$$x + y \geq 0 \wedge (x \neq z \Rightarrow y + z = -1) \wedge z > 3t \tag{52}$$

est satisfiable dans la théorie de l'arithmétique linéaire rationnelle $\mathrm{Th}(\mathbb{Q}, 1, (q\cdot)_{q\in\mathbb{Q}}, +, <)$.

Les solveurs SMT implémentent en réalité des solveurs pour des formules qui sont des conjonctions de littéraux. Dans le cas de la formule (52), on pourrait alors convertir la formule sous forme normale disjonctive

$$(x + y \geq 0 \wedge \neg(x \neq z) \wedge z > 3t) \vee (x + y \geq 0 \wedge y + z = -1 \wedge z > 3t)$$

puis essayer de résoudre chacune des deux conjonctions de littéraux indépendamment. Cela revient à résoudre les deux systèmes d'équation suivants dans $\mathbb{Q}$, ce qui se fait aisément :

$$\begin{cases} x + y \geq 0 \\ x - z = 0 \\ z - 3t > 0 \end{cases} \qquad \begin{cases} x + y \geq 0 \\ y + z - 1 = 0 \\ z - 3t > 0 \end{cases}$$

12. `https://github.com/Z3Prover/z3`

Solveurs SMT hors ligne. Un défaut d'une telle approche est que la mise sous forme normale disjonctive peut avoir un coût exponentiel. Les solveurs SMT s'appuient à la place sur des solveurs SAT pour traiter la « partie booléenne » de la formule. On procède pour cela comme suit.

(1) Chaque littéral de la formule d'origine est associé à une proposition. Par exemple, pour la formule (52), on obtient ainsi une formule $P_1 \wedge (P_2 \Rightarrow P_3) \wedge P_4$ où P_1 représente le littéral $x + y \geq 0$, P_2 représente $x \neq z$, P_3 représente $y + z = -1$ et P_4 représente $z > 3t$.

(2) La formule propositionnelle est convertie en forme normale conjonctive – ce qui donne la formule $P_1 \wedge (\neg P_2 \vee P_3) \wedge P_4$ pour notre exemple – et est donnée en entrée à un solveur SAT. Si cette formule propositionnelle est insatisfiable, alors la formule de départ l'était aussi.

(3) Sinon, le solveur SAT fournit un modèle de la formule propositionnelle, par exemple l'interprétation partielle $[1/P_1, 0/P_2, 1/P_3, 1/P_4]$. Ce modèle correspond à une conjonction de littéraux $x + y \geq 0 \wedge \neg(x \neq z) \wedge y + z = -1 \wedge z > 3t$ que le solveur SMT tente de satisfaire dans la théorie $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q \in \mathbb{Q}}, +, <)$; en l'occurrence, il s'agit simplement de résoudre un système d'inéquations linéaires sur les rationnels :

$$\begin{cases} x + y \geq 0 \\ x - z = 0 \\ y + z - 1 = 0 \\ z - 3t > 0 \end{cases}$$

Si cette conjonction de littéraux est satisfiable, la formule de départ l'était aussi.

(4) Sinon, et c'est le cas de notre exemple, on fait la conjonction entre la formule d'origine et la négation de notre conjonction de littéraux de l'étape (3) ; pour notre exemple, cela donne la formule $(x + y \geq 0 \wedge (x \neq z \Rightarrow y + z = -1) \wedge z > 3t) \wedge \neg(x + y \geq 0 \wedge \neg(x \neq z) \wedge y + z = -1 \wedge z > 3t)$. Le résultat est une nouvelle formule qui est satisfiable si et seulement si la formule d'origine l'était, et on recommence à l'étape (1) ci-dessus.

En pratique, plutôt que de faire la conjonction avec $\neg(x + y \geq 0 \wedge \neg(x \neq z) \wedge y + z = -1 \wedge z > 3t)$ en entier, le solveur SMT va trouver un sous-ensemble des littéraux qui ne peut pas être satisfait et ajouter la négation de ce sous-ensemble ; par exemple $\neg(x+y \geq 0 \wedge \neg(x \neq z) \wedge y+z = -1)$.

Les solveurs SAT *actuels basés sur des techniques de «* conflict-driven clause learning *» sont capables d'ajouter des nouvelles clauses en cours d'exécution, ce qui rend l'étape (4) efficace. Les solveurs SMT actuels sont plutôt* en ligne *pour faire dialoguer efficacement le solveur* SAT *et la recherche de modèle dans la théorie.*

Exemple 16.1. Si on poursuit sur notre exemple, nous avons maintenant (2) une formule propositionnelle en forme normale conjonctive $P_1 \wedge (\neg P_2 \vee P_3) \wedge P_4 \wedge (\neg P_1 \vee P_2 \vee \neg P_3 \vee \neg P_4)$ pour laquelle un solveur SAT retourne par exemple $[1/P_1, 0/P_2, 0/P_3, 1/P_4]$, et (3) la conjonction de littéraux $x + y \geq 0 \wedge \neg(x \neq z) \wedge \neg(y + z = -1) \wedge z > 3t$ est maintenant satisfiable dans $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q \in \mathbb{Q}}, +, <)$, par exemple par la valuation partielle $[0/x, 0/y, 0/z, -\frac{1}{6}/t]$, qui satisfait donc aussi la formule d'origine (52).

16.1.2. *Élimination des quantificateurs.* Dans le cas où la théorie T permet effectivement l'élimination des quantificateurs, comme les théories $\text{Th}(A_{\text{oldns}})$ des ordres linéaires denses non bornés stricts et $\text{Th}(\mathbb{Q}, 1, (q\cdot)_{q \in \mathbb{Q}}, +, <)$ de l'arithmétique linéaire rationnelle de la section 15.2, mais aussi les théories $\text{Th}(\mathbb{Z}, +, <)$ de l'arithmétique de Presburger et $\text{Th}(\mathbb{R}, +, \times)$ de l'arithmétique des réels, on dispose d'un algorithme qui, pour une formule φ donnée, retourne une formule sans quantificateurs $\text{qe}_T(\varphi)$ équivalente modulo T, c'est-à-dire telle que $T \vDash \varphi \Leftrightarrow \text{qe}_T(\varphi)$.

Cela permet d'appliquer notre solveur SMT à la formule $\text{qe}_T(\varphi)$. En effet, en général si $T \vDash \varphi \Leftrightarrow \psi$ alors la formule φ est satisfiable modulo T si et seulement si la formule ψ l'est, et de plus les mêmes témoins I, ρ peuvent être utilisés pour les deux formules. Cela découle des définitions : si $T \vDash \varphi \Leftrightarrow \psi$, c'est-à-dire si pour toute interprétation I telle que $I \vDash T$ et pour toute valuation ρ on a $I, \rho \vDash \varphi \Leftrightarrow \psi$, alors pour toute interprétation I telle que $I \vDash T$ et pour toute valuation ρ, $I, \rho \vDash \varphi$ si et seulement si $I, \rho \vDash \psi$.

16.1.3. *SMT-LIB.* Un solveur SMT comme Z3 peut être utilisé au travers de son API; il en existe pour plusieurs langages de programmation [13]. Si de telles APIs rendent les solveurs SMT directement accessibles depuis un langage de programmation, elles ont le défaut de dépendre à la fois du solveur et du langage.

Une approche générique est d'avoir un format commun pour tous les solveurs SMT, comme le format DIMACS l'était pour les solveurs SAT. Dans le cas des solveurs SMT, ce format commun s'appelle *SMT-LIB* [14] et est en fait un langage pour définir des théories logiques et des formules, et pour écrire des scripts très simples.

(Barrett, Fontaine et Tinelli, 2017)

Écrire des formules en SMT-LIB. Commençons par un exemple très simple : la formule (52). Voici le code correspondant en SMT-LIB.

eq52.smt2
```
(declare-const x Real)
(declare-const y Real)
(declare-const z Real)
(declare-const t Real)
(assert (and (>= (+ x y) 0) (=> (distinct x z) (= (+ y z) -1)) (> z (* 3 t))))
(check-sat)
(get-value (x y z t))
(exit)
```

Dans ce code, on commence par déclarer les quatre variables libres x, y, z et t. Le langage SMT-LIB force à donner des *types* à tous ses objets; ici nous utilisons le type `Real` pour toutes nos variables (il n'y a pas de type pour les rationnels, mais cela revient au même). Les formules dont on souhaite vérifier la satisfiabilité sont écrites en « notation préfixe » et ajoutées par la commande `(assert` φ`)` ; à noter que les opérateurs comme `and` peuvent prendre plus de deux arguments. On vérifie la satisfiabilité par `(check-sat)`, puis `(get-value ...)` permet d'afficher la valuation. Si on appelle Z3 sur ce fichier, il répond correctement que la formule est satisfiable et fournit une valuation :

```
sat
((x 0.0)
 (y 0.0)
 (z 0.0)
 (t (- (/ 1.0 6.0))))
```

Formules quantifiées en SMT-LIB. Voyons comment écrire les formules de l'exemple 12.6 en SMT-LIB. La formule (23) $\forall x \exists y . y < x$ s'écrit comme suit.

order.smt2
```
;; équation (23)
(assert (forall ((x Real)) (exists ((y Real)) (< y x))))
(check-sat-using (then qe smt))
```

Il y a deux différences par rapport au code précédent. D'une part, comme la formule est close, il n'est pas nécessaire de déclarer de variables libres comme des constantes. D'autre part, comme la formule utilise des quantificateurs, il faut indiquer à Z3 qu'il doit utiliser l'élimination des quantificateurs avant de chercher un modèle : c'est ce que fait la commande `(check-sat-using (then qe smt))`.

La formule (24) $\forall x \forall y . x < y \Rightarrow \exists z . x < z \wedge z < y$ s'écrit comme suit.

13. `https://github.com/Z3Prover/z3#z3-bindings`
14. `http://smtlib.cs.uiowa.edu/`

order.smt2
```
;; équation (24) sur les réels
(assert (forall ((x Real) (y Real)) (=> (< x y)
  (exists ((z Real)) (and (< x z) (< z y))))))
(check-sat-using (then qe smt))
```

Cette formule est satisfiable dans un ordre dense comme $(\mathbb{Q},<)$ ou $(\mathbb{R},<)$, mais ne l'est pas dans un ordre discret comme $(\mathbb{Z},<)$; Z3 répond `unsat` pour le code SMT-LIB suivant :

order.smt2
```
;; équation (24) sur les entiers
(assert (forall ((x Int) (y Int)) (=> (< x y)
  (exists ((z Int)) (and (< x z) (< z y))))))
(check-sat-using (then qe smt))
```

Inversement, la formule (25) $\forall x \exists y \forall z. x < y \wedge \neg(x < z \wedge z < y)$ n'est pas satisfiable dans $(\mathbb{R},<)$; Z3 répond `unsat` pour le code SMT-LIB suivant :

order.smt2
```
;; équation (25)
(assert (forall ((x Real)) (exists ((y Real)) (and (< x y)
  (not (exists ((z Real)) (and (< x z) (< z y))))))))
(check-sat-using (then qe smt))
```

Déclarer des symboles. Reprenons maintenant la formule du buveur $\exists x.B(x) \Rightarrow \forall y.B(y)$ de l'exemple 12.3. Celle-ci est valide si et seulement si sa négation est insatisfiable, ce que nous allons vérifier avec Z3. La formule utilise un symbole B de relation unaire *non interprété*, qu'il va falloir déclarer : on déclare pour cela un nouveau type D par la commande `(declare-sort D)`.

buveur.smt2
```
; exemple 12.3
; on appelle D notre domaine
(declare-sort D)
; le symbole de relation `B' prend un élément du domaine et retourne
; une valeur de vérité
(declare-fun B (D) Bool)
; la formule du buveur est valide si et seulement si sa négation
; est insatisfiable
(assert (not (exists ((x D)) (=> (B x) (forall ((y D)) (B y))))))
(check-sat-using (then qe smt))
; (renvoie `unsat')
(exit)
```

Définir des symboles. Considérons maintenant la théorie de l'exemple 15.8. L'axiomatisation A que nous avions définie dans cet exemple permettait de modéliser fidèlement l'interprétation I donnée dans l'exemple 10.5, et pourrait être écrite en SMT-LIB. Cependant, il est plus aisé pour modéliser notre base de données d'utiliser un type énumératif pour l'ensemble des éléments du domaine D_I, ce qui crée simultanément les symboles de constantes *shining*, *player*, etc. C'est ce que nous faisons ci-dessous, en différenciant de plus trois types d'éléments dans le domaine D_I : les titres de films, les noms de réalisateurs et d'interprètes, et les noms de cinémas.

database.smt2
```
;; exemple 12.5
;; types énumératifs
(declare-datatypes () ((Titre shining player easyrider apocalypsenow)
  (Nom kubrick altman hopper nicholson robbins coppola)
```

```
  (Cinema champo odeon)))
;; la relation `Film'
(define-fun Film ((x Titre) (y Nom) (z Nom)) Bool
  (ite (and (= x shining)   (= y kubrick)  (= z nicholson)) true
  (ite (and (= x player)    (= y altman)   (= z robbins))   true
  (ite (and (= x easyrider) (= y hopper)   (= z nicholson)) true
  (ite (and (= x easyrider) (= y hopper)   (= z hopper))    true
  (ite (and (= x apocalypsenow) (= y coppola) (= z hopper)) true
    false))))))
;; la relation `Seance'
(define-fun Seance ((x Cinema) (y Titre)) Bool
  (ite (and (= x champo) (= y shining))   true
  (ite (and (= x champo) (= y easyrider)) true
  (ite (and (= x champo) (= y player))    true
  (ite (and (= x odeon)  (= y easyrider)) true
    false)))))
```

Plutôt que de déclarer un symbole de relation non interprété *Film* et d'ajouter une assertion qui garantit que *Film*(x, y, z) n'est vrai que pour les bons x, y et z, on a défini ci-dessus le symbole de relation via `(define-fun Film ...)`. La commande `(ite` *cond si sinon*`)` est un « *if-then-else* » : si *cond* s'évalue à vrai, son résultat est celui de *si*, et sinon c'est le résultat de *sinon*.

Nous pouvons maintenant tester les requêtes de l'exemple 12.5. La première requête (12) cherche un titre de film présent dans la base de données et cette requête est satisfiable.

```
database.smt2
(declare-const x Titre)
;; équation (12)
(assert (exists ((r Nom) (i Nom)) (Film x r i)))
(check-sat-using (then qe smt))
(get-value (x))
```

On demande ici à Z3 une valeur de x qui satisfait la requête par la commande `(get-value ...)` ; le résultat est le suivant.

```
sat
((x repulsion))
```

La requête (17) cherche des paires d'un réalisateur et d'un cinéma qui diffuse un de ses films ; Z3 trouve que (Altman, Le Champo) satisfait la requête.

```
database.smt2
(declare-const x Nom)
(declare-const y Cinema)
;; équation (17)
(assert (exists ((t Titre) (i Nom)) (and (Film t x i) (Seance y t))))
(check-sat-using (then qe smt))
(get-value (x y))
```

La requête (22) cherche des interprètes qui n'ont joué que dans un seul film ; Z3 trouve que Robbins satisfait cette requête.

```
database.smt2
(declare-const x Nom)
;; équation (22)
(assert (exists ((t Titre) (r Nom)) (and (Film t r x)
  (forall ((t2 Titre) (r2 Nom)) (=> (Film t2 r2 x) (= t t2))))))
```

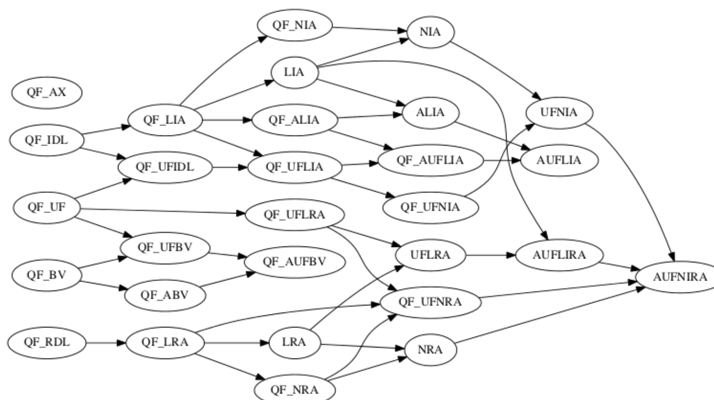


Figure 29. Les théories définies par SMT-LIB 2.6 ; voir `http://smtlib.cs.uiowa.edu/logics.shtml`.

```
(check-sat-using (then qe smt))
(get-value (x))
```

16.1.4. *Théories usuelles.* La figure 29 est tirée de la documentation de SMT-LIB et recense les théories standardisées – mais toutes ne sont pas nécessairement implémentées dans tous les solveurs SMT. Ces théories permettent différentes combinaisons de

- théories arithmétiques comme par exemple $(\mathbb{Z}, +, \times)$ (notée `NIA` dans la figure 29) ou $(\mathbb{R}, +, \times)$ (notée NRA) ou leurs restrictions à leurs fragments linéaires (par exemple `LIA` pour l'arithmétique de Presburger ou LRA pour l'arithmétique linéaire sur les réels),
- permettant des formules quantifiées ou non (les fragments sans quantificateurs sont préfixés par `QF_` dans la figure 29) , et
- permettant des symboles non interprétés (les fragments avec symboles non interprétés sont préfixés par `UF` dans la figure 29) ou non.

Par exemple, UFLRA désigne l'arithmétique linéaire sur les réels avec quantificateurs et symboles non interprétés.

Pour plusieurs des théories de la figure 29, il n'existe pas d'algorithme qui résout le problème de SATISFIABILITÉ MODULO T. Cela n'empêche pas d'implémenter une procédure, qui potentiellement répondra `(timeout)` ou `(unknown)` ou lieu de `(sat)` ou `(unsat)`.

16.2. **Exemple de modélisation : nombre de McNuggets.** Voyons maintenant un exemple complet de modélisation en logique du premier ordre et son implémentation en SMT-LIB. Le problème est le suivant : les McNuggets sont vendus en boîtes de six, de neuf, ou de vingt McNuggets [15] ; voir la figure 30.

Si on souhaite acheter un nombre $m \in \mathbb{N}$ de McNuggets, cela n'est pas forcément possible. Par exemple, on ne peut pas acheter exactement trois McNuggets ou exactement onze McNuggets. En fait, on peut acheter exactement m McNuggets si et seulement si m est une combinaison positive linéaire de six, neuf et vingt :

$$\textit{combinaison}(m) \stackrel{\text{def}}{=} (\exists c_1 \exists c_2 \exists c_3 . m = 6 \cdot c_1 + 9 \cdot c_2 + 20 \cdot c_3) \tag{53}$$

où c_1, c_2 et c_3 doivent être pris dans $\mathbb{N}$.

Cependant, si m est assez grand, de tels coefficients c_1, c_2 et c_3 existent nécessairement par le théorème combinatoire de Schur puisque 6, 9 et 20 sont premiers entre eux. Le *nombre de McNuggets* est le plus grand entier m qui ne satisfait pas la formule *combinaison*(m) définie

15. Auteur : Fritz Saalfeld, licence [CC BY-SA 2.5], via *Wikimedia Commons*.

FIGURE 30. Une boîte de vingt *chicken McNuggets*[15].

en (53). Dit autrement, on cherche m tel que

$$\neg \textit{combinaison}(m) \wedge (\forall p. p > m \Rightarrow \textit{combinaison}(p)) \ . \tag{54}$$

Calculer le nombre de McNuggets est bien une instance de SATISFIABILITÉ MODULO T où l'on peut prendre l'arithmétique de PRESBURGER comme théorie T sous-jacente. On peut implémenter tout cela en SMT-LIB comme ci-dessous.

mcnuggets.smt2

```
; nombre de McNuggets
(declare-const m Int)
(assert (>= m 0))
; équation (53) : n est-il une combinaison linéaire de 6, 9 et 20 ?
(define-fun combinaison ((n Int)) Bool
  (exists ((c1 Int)(c2 Int)(c3 Int))
    (and (>= c1 0) (>= c2 0) (>= c3 0)
      (= n (+ (* 6 c1) (* 9 c2) (* 20 c3))))))
; équation (54)
(assert (and
  (not (combinaison m))
  (forall ((p Int)) (=> (> p m) (combinaison p)))))
(check-sat)
(get-value (m))
(exit)
```

Le solveur Z3 arrive à résoudre ce problème sur ma machine de bureau, et répond correctement que 43 est le nombre de McNuggets.

16.3. Exemple de modélisation : apprentissage d'automates séparateurs. Soient deux ensembles finis disjoints I et E de mots sur un alphabet fini Σ. On souhaite apprendre un automate fini déterministe $\mathcal{A}$ qui accepte tous les mots de I et rejette tous les mots de E : $I \subseteq L(\mathcal{A})$ et $E \cap L(\mathcal{A}) = \emptyset$. Par exemple, sur $\Sigma = \{a, b\}$ pour $I = \{\varepsilon, ab\}$ (où ε dénote le mot vide) et $E = \{aa, b\}$, l'automate de la figure 31 ci-dessous répond au problème.

Cette section donne un exemple d'algorithme d'apprentissage supervisé passif d'un classifieur *– l'automate obtenu va soit accepter, soit rejeter les mots. Ce problème d'apprentissage, avec exemples positifs (à accepter) et négatifs (à rejeter) était déjà étudiée par* GOLD *en 1967.*

Modélisation du problème. Nous allons voir comment utiliser un solveur SMT pour trouver automatiquement un tel automate. Rappelons pour cela la définition d'un automate fini déterministe complet $\mathcal{A}$ sur un alphabet Σ : $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$ où Q est un ensemble fini d'états, $\delta \colon Q \times \Sigma \to Q$ est la fonction de transition, $F \subseteq Q$ est l'ensemble des états acceptants et $q_0 \in Q$

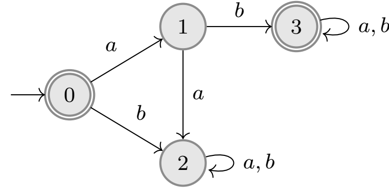


FIGURE 31. Un automate qui accepte ε et ab mais rejette b et aa.

est l'état initial. La fonction de transition peut être étendue en une fonction $\delta\colon Q \times \Sigma^* \to Q$ en posant $\delta(q, \varepsilon) \stackrel{\text{def}}{=} q$ et $\delta(q, wa) \stackrel{\text{def}}{=} \delta(\delta(q, w), a)$ pour tout $w \in \Sigma^*$ et $a \in \Sigma$. Le langage de l'automate est $L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$. Par exemple, le langage de l'automate de la figure 31 est $\{\varepsilon\} \cup \{abw \mid w \in \Sigma^*\}$.

Pour modéliser efficacement le problème, nous allons dans une première étape construire une structure d'*arbre préfixe* (aussi appelé « *trie* ») pour l'ensemble de mots $I \cup E$. Pour un ensemble de mots S sur un alphabet Σ, l'arbre préfixe de S contient un nœud pour chaque mot dans

$$\mathit{Pref}(S) \stackrel{\text{def}}{=} \{u \in \Sigma^* \mid \exists v \in \Sigma^*. uv \in S\}$$

l'ensemble des préfixes des mots de S. Un nœud u est le parent d'un nœud v s'il existe une lettre a de l'alphabet Σ telle que $v = ua$. Par exemple, l'arbre de la figure 32 est l'arbre préfixe de l'ensemble $I \cup E = \{\varepsilon, ab, aa, b\}$.

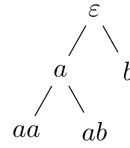


FIGURE 32. L'arbre préfixe pour l'ensemble $\{\varepsilon, ab, aa, b\}$.

L'idée de notre modélisation est qu'un automate $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$ est tel que $I \subseteq L(\mathcal{A})$ et $E \cap L(\mathcal{A}) = \emptyset$ si et seulement si il existe une fonction $f\colon \mathit{Pref}(I \cup E) \to Q$ telle que les trois contraintes ci-dessous soient vérifiées :

$$f(\varepsilon) = q_0 \tag{55}$$

$$\bigwedge_{\substack{w \in \Sigma^*, a \in \Sigma \\ wa \in \mathit{Pref}(I \cup E)}} \delta(f(w), a) = f(wa) \tag{56}$$

$$\Big(\bigwedge_{w \in I} f(w) \in F\Big) \wedge \Big(\bigwedge_{w \in E} f(w) \notin F\Big) \tag{57}$$

Proposition 16.2. *Un automate $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$ est tel que $I \subseteq L(\mathcal{A})$ et $E \cap L(\mathcal{A}) = \emptyset$ si et seulement si il existe une fonction $f\colon \mathit{Pref}(I \cup E) \to Q$ telle que* (55), (56) *et* (57) *soient vérifiées.*

Démonstration. Si $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$ est un automate tel que $I \subseteq L(\mathcal{A})$ et $E \cap L(\mathcal{A}) = \emptyset$, on définit $f(w) \stackrel{\text{def}}{=} \delta(q_0, w)$ pour tout mot $w \in \mathit{Pref}(I \cup E)$, ce qui vérifie bien (55), (56) et (57).

Inversement, si $f\colon \mathit{Pref}(I \cup E) \to Q$ est une fonction telle que (55) et (56) soient vérifiées, alors

$$\forall w \in \mathit{Pref}(I \cup E). f(w) = \delta(q_0, w)\ .$$

Cela se vérifie par induction sur le mot w. Pour le cas de base $w = \varepsilon$, par (55), on a bien $f(\varepsilon) = q_0 = \delta(q_0, \varepsilon)$. Puis pour l'étape d'induction $w = w'a$ avec $w' \in \Sigma^*$ et $a \in \Sigma$, comme nécessairement $w' \in \mathit{Pref}(I \cup E)$, l'hypothèse d'induction s'applique et donc $f(w') = \delta(q_0, w')$; de plus par (56), $f(w) = f(w'a) = \delta(f(w'), a) = \delta(\delta(q_0, w'), a) = \delta(q_0, w'a) = \delta(q_0, w)$ comme désiré.

Par suite, pour tout $w \in I$, on a $w \in \mathit{Pref}(I \cup E)$ donc $\delta(q_0, w) = f(w)$ et $f(w) \in F$ par (57), donc $I \subseteq L(\mathcal{A})$. Et pour tout $w \in E$, on a $w \in \mathit{Pref}(I \cup E)$ donc $\delta(q_0, w) = f(w)$ et $f(w) \notin F$ par (57), donc $E \cap L(\mathcal{A}) = \emptyset$. □

Implémentation en SMT-LIB. Voici maintenant comment exprimer notre problème en SMT-LIB. Tout d'abord, nous allons déclarer des types énumératifs pour notre alphabet et pour notre arbre préfixe.

automaton.smt2
```
; définition de l'alphabet A et de l'arbre préfixe T
(declare-datatypes () ((A a b) (T e ea eb eaa eab)))
```

Les symboles de constantes associés à l'arbre préfixe sont e pour ε, ea pour a, eb pour b, eaa pour aa et eab pour ab.

L'automate fini $\mathcal{A}$ utilise l'ensemble d'entiers $\{0, \ldots, n-1\}$ comme ensemble d'états Q, où 0 est l'état initial.

automaton.smt2
```
; les états de l'automate à trouver sont {0, 1, ..., n-1}
(define-sort Q () Int)
(declare-const n Q)
(assert (> n 0))
; fonction de transition de l'automate
(declare-fun delta (Q A) Q)
(assert (forall ((q Q) (a A))
  (and (>= (delta q a) 0) (< (delta q a) n))))
; ensemble d'états acceptants de l'automate
(declare-fun final (Q) Bool)
```

Il nous reste à exprimer la fonction $f\colon I \cup E \to Q$ ainsi que les contraintes des équations (55), (56) et (57).

automaton.smt2
```
; fonction des éléments de l'arbre préfixe vers les états
(declare-fun f (T) Q)
(assert (forall ((x T))
  (and (>= (f x) 0) (< (f x) n))))
; contrainte (55) sur l'état initial
(assert (= 0 (f e)))
; contraintes (56) sur les transitions
(assert (and (= (f ea)  (delta (f e)  a))
             (= (f eb)  (delta (f e)  b))
             (= (f eaa) (delta (f ea) a))
             (= (f eab) (delta (f ea) b))))
; contraintes (57) sur les états acceptants
(assert (and (final (f e))
             (final (f eab))
             (not(final (f eb)))
             (not(final (f eaa)))))
(check-sat-using (then qe smt))
```

```
(get-model)
(exit)
```

La commande `(get-model)` demande au solveur SMT de fournir l'interprétation qu'il a trouvée (si le problème était satisfiable). Voici ce que retourne Z3.

```
sat
(model
  (define-fun n () Int
    2)
  (define-fun k!6 ((x!0 Int)) Int
    (ite (= x!0 0) 0
      1))
  (define-fun delta!7 ((x!0 Int) (x!1 A)) Int
    (ite (and (= x!0 1) (= x!1 b)) 0
      1))
  (define-fun delta ((x!0 Int) (x!1 A)) Int
    (delta!7 (k!6 x!0) x!1))
  (define-fun final ((x!0 Int)) Bool
    (ite (= x!0 1) false
      true))
  (define-fun f ((x!0 T)) Int
    (ite (= x!0 e) 0
    (ite (= x!0 eab) 0
      1)))
)
```

L'interprétation trouvée met n à 2, donc l'automate a deux états 0 et 1. Les deux fonctions `k!6` et `delta!7` sont des fonctions auxiliaires introduites par le solveur. La fonction de transition `delta` prend en entrée un état `x!0` et une lettre de l'alphabet `x!1` et envoie sur l'état 1 sauf si `x!0` est l'état 1 et `x!1` est la lettre b, auquel cas elle envoie sur 0. La fonction `final` définit l'état 1 comme rejetant et l'état 0 comme acceptant. Tout cela décrit l'automate de la figure 33. À noter que cet automate accepte le language $((a+b)a^*b)^*$, qui n'est pas le même que celui de l'automate de la figure 31, mais qui sépare aussi I de E.

L'automate de la figure 33 est non seulement plus petit, mais souffre aussi moins de « sur-apprentissage » que celui de la figure 31 : il généralise mieux les mots de l'ensemble d'entraînement $S = I \cup E$.

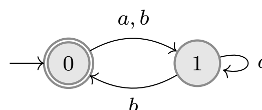


FIGURE 33. Un autre automate qui accepte ε et ab mais rejette aa et b.

16.4. **Exemple de modélisation : synthèse d'invariant de programme.** Une des applications informatiques des solveurs SMT est la *synthèse d'invariants*. Pour démontrer qu'un programme est correct, une méthode usuelle est d'écrire des annotations du programme, et en particulier des invariants de boucle : une relation entre les variables du programme qui reste vraie au cours des exécutions. Il peut cependant être assez difficile de trouver de tels invariants. L'idée est donc d'automatiser la tâche, en utilisant des solveurs SMT pour synthétiser de tels invariants.

Prenons par exemple le fragment de code Java ci-dessous.

```
int i = 0;
int j = 1;
while (i < 10) {
    i = i + 2;
```

```
    j = j + 1;
}
assert (j < 10);
```

On souhaite vérifier qu'à la sortie de la boucle `while`, c'est-à-dire quand `i >= 10`, l'assertion `j < 10` est vérifiée.

Configurations accessibles. Une manière de procéder est de calculer l'ensemble des configurations *accessibles* du programme, c'est-à-dire l'ensemble $\mathit{Reach} \subseteq \mathbb{Z}^2$ des valeurs de `i` et `j` qui peuvent apparaître au cours de l'exécution, et vérifier que $i \geq 10 \Rightarrow j < 10$ est vrai pour toutes ces configurations. C'est bien le cas :

$$\mathit{Reach} = \{(0,1),(2,2),(4,3),(6,4),(8,5),(10,6)\}\ , \tag{58}$$

qui est représenté dans la figure 34 ; on a bien $\mathit{Reach} \cap \mathit{Error} = \emptyset$, où

$$\mathit{Error} \stackrel{\text{def}}{=} \{(i,j) \in \mathbb{Z}^2 \mid i,j \geq 10\} \tag{59}$$

dénote l'ensemble des configurations interdites par l'assertion.

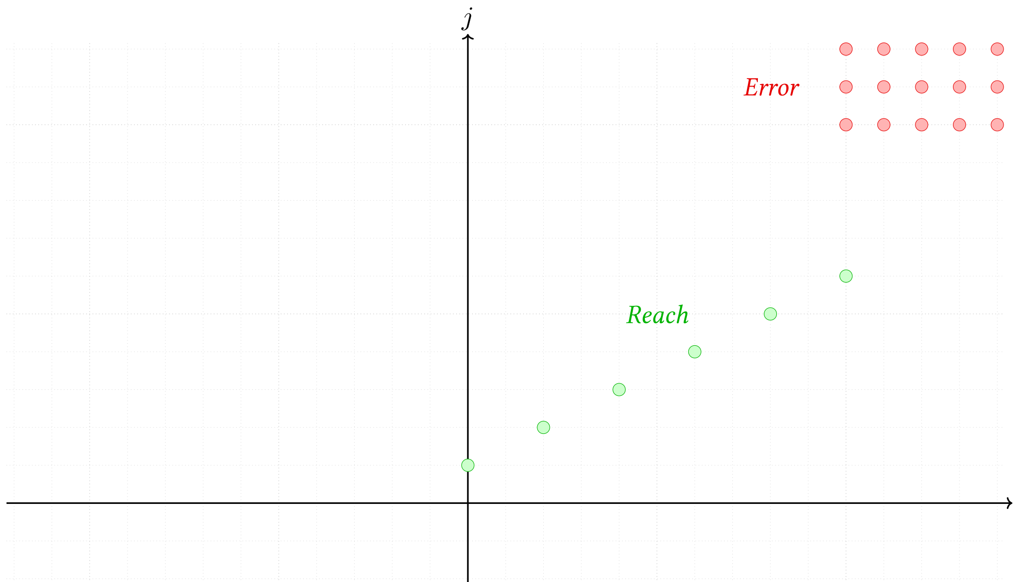


FIGURE 34. L'ensemble d'accessibilité *Reach* et les configurations d'erreur *Error* du programme.

L'ensemble des configurations accessibles était très facile à calculer pour notre exemple, mais en général ça n'est pas faisable – par exemple parce que le programme ne termine pas sous certaines conditions, or c'est justement ces situations que l'on veut éviter en vérifiant le programme sans l'exécuter !

💬 *On cherche à faire une* analyse statique *du programme, où l'on en vérifie le comportement sans l'exécuter.*

Invariants inductifs sûrs. À la place, on va chercher une *sur-approximation* de *Reach*, c'est-à-dire un ensemble de configurations $\mathit{Approx} \supseteq \mathit{Reach}$. On souhaite plus précisément trouver une sur-approximation *sûre*, c'est-à-dire telle que $\mathit{Approx} \cap \mathit{Error} = \emptyset$. Puisque $\mathit{Reach} \subseteq \mathit{Approx}$, cela garantira bien que $\mathit{Reach} \cap \mathit{Error} = \emptyset$.

Une manière d'être certain que notre ensemble *Approx* soit bien une sur-approximation, c'est-à-dire que $\mathit{Approx} \supseteq \mathit{Reach}$, est de demander à ce que ce soit un *invariant inductif* : cela veut dire d'une part, la configuration initiale est dans *Approx*, et d'autre part que, si on prend une configuration de *Approx* et si on applique une étape de calcul, alors on arrive sur une configuration qui est encore dans *Approx*. À noter que *Reach* est un invariant inductif (c'est même le plus petit :

celui qui contient le moins de configurations), mais d'autres invariants inductifs existent, comme par exemple l'ensemble

$$Safe \stackrel{\text{def}}{=} \{(i,j) \in \mathbb{Z}^2 \mid (i < 10 \land 2j \leq i + 8) \lor (i \geq 10 \land j < 10)\} \tag{60}$$

représenté dans la figure 35 (qui se trouve être le plus grand).

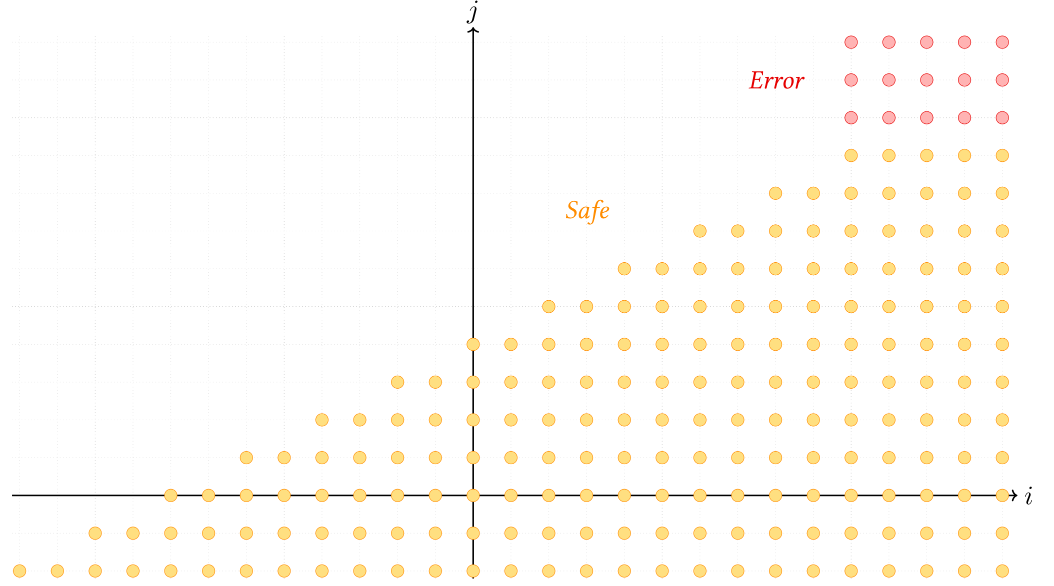


FIGURE 35. L'invariant inductif *Safe* et les configurations d'erreur *Error* du programme.

Modélisation logique. Dans l'exemple de programme ci-dessus, un invariant inductif est une relation entre les variables de la boucle (ici `i` et `j`) qui, si elle est vraie à l'entrée de la boucle, sera encore vraie à la sortie, et de plus est vraie initialement pour `i = 0` et `j = 1`. Afin que cet invariant inductif soit sûr, il faut aussi qu'il respecte l'assertion finale `j < 10` quand `i >= 10`.

On va se placer dans une signature $(\{+^{(2)}, (i^{(0)})_{i\in\mathbb{Z}}\}, \{<^{(2)}, =^{(2)}, \mathit{Invar}^{(2)}\})$ où les symboles arithmétiques, de constantes $i \in \mathbb{Z}$, d'ordre et d'égalité seront interprétés sur le domaine $D_I \stackrel{\text{def}}{=} \mathbb{Z}$, et où le symbole de relation binaire *Invar* va représenter notre invariant.

L'objectif du solveur SMT va être de fournir une interprétation $I = (\mathbb{Z}, +, (i)_{i\in\mathbb{Z}}, <, =, \mathit{Invar}^I)$ dans laquelle les interprétations de $+$, $i \in \mathbb{Z}$, $<$ et $=$ sont fixées (ce sont les interprétations habituelles dans les entiers relatifs), mais l'interprétation Invar^I du symbole $\mathit{Invar}^{(2)}$ n'est pas fixée *a priori* et devra respecter des contraintes pour qu'elle représente bien un invariant inductif sûr du programme ; on dit que $\mathit{Invar}^{(2)}$ est un symbole « non interprété ».

Pour que Invar^I représente un invariant inductif sûr du programme, il faut que ce soit bien un invariant de la boucle : si on entre dans la boucle avec certaines valeurs de `i` et `j` en respectant l'invariant, alors à la sortie de boucle l'invariant est encore respecté :

$$\forall x \forall y. \mathit{Invar}(x,y) \land x < 10 \Rightarrow \mathit{Invar}(x+2, y+1)\ . \tag{61}$$

Il faut aussi que la condition initiale `i = 0` et `j = 1` soit dans l'invariant :

$$\mathit{Invar}(0,1)\ . \tag{62}$$

Enfin, pour que l'invariant soit sûr, l'assertion finale `j < 10` doit elle aussi être vérifiée si on sort de la boucle, c'est-à-dire si `i >= 10` :

$$\forall x \forall y. \mathit{Invar}(x,y) \land (x > 10 \lor x = 10) \Rightarrow y < 10\ . \tag{63}$$

Implémentation en SMT-LIB. On peut traduire ces équations en SMT-LIB :

invariant.smt2
```
; synthèse d'invariant de programme
; on déclare le symbole non interprété de relation Invar
(declare-fun Invar (Int Int) Bool)
; équation (61) : la relation Invar est un invariant de boucle
(assert (forall (( x Int ) ( y Int ))
  (=> (and (Invar x y) (< x 10)) (Invar (+ x 2) (+ y 1)))))
; équation (62) : la relation Invar est vraie initialement
(assert (Invar 0 1))
; équation (63) : l'assertion finale est vérifiée
(assert (forall (( x Int ) ( y Int ))
  (=> (and (Invar x y) (>= x 10)) (< y 10))))
; appel au solveur
(check-sat-using (then qe smt))
(get-model)
(exit)
```

Encore une fois, la commande `(get-model)` demande au solveur SMT de fournir l'interprétation qu'il a trouvée, si le problème était satisfiable. Voici ce que retourne Z3.

```
sat
(model
  (define-fun k!43 ((x!1 Int)) Int
    (let ((a!1 (ite (>= x!1 8) (ite (>= x!1 9) (ite (>= x!1 10) 10 9) 8) 7)))
    (let ((a!2 (ite (>= x!1 5) (ite (>= x!1 6) (ite (>= x!1 7) a!1 6) 5) 4)))
    (let ((a!3 (ite (>= x!1 2) (ite (>= x!1 3) (ite (>= x!1 4) a!2 3) 2) 1)))
      (ite (>= x!1 (- 1)) (ite (>= x!1 0) (ite (>= x!1 1) a!3 0) (- 1)) (- 2))))))
  (define-fun Invar!44 ((x!1 Int) (x!2 Int)) Bool
    (ite (and (= x!1 0) (= x!2 1)) true
    (ite (and (= x!1 2) (= x!2 2)) true
    (ite (and (= x!1 4) (= x!2 3)) true
    (ite (and (= x!1 6) (= x!2 4)) true
    (ite (and (= x!1 8) (= x!2 5)) true
    (ite (and (= x!1 10) (= x!2 6)) true
      false)))))))
  (define-fun k!42 ((x!1 Int)) Int
    (let ((a!1 (ite (>= x!1 8) (ite (>= x!1 9) (ite (>= x!1 10) 10 9) 8) 7)))
    (let ((a!2 (ite (>= x!1 5) (ite (>= x!1 6) (ite (>= x!1 7) a!1 6) 5) 4)))
    (let ((a!3 (ite (>= x!1 2) (ite (>= x!1 3) (ite (>= x!1 4) a!2 3) 2) 1)))
      (ite (>= x!1 1) a!3 0)))))
  (define-fun Invar ((x!1 Int) (x!2 Int)) Bool
    (Invar!44 (k!43 x!1) (k!42 x!2)))
)
```

L'interprétation $Invar^I$ du symbole $Invar^{(2)}$ qui a été trouvée est donnée aux deux dernières lignes : c'est une fonction qui prend deux variables entières `x!1` et `x!2` et retourne une valeur de vérité. Elle appelle pour cela une fonction auxiliaire `Invar!44`, à laquelle elle passe le résultat d'un appel d'une fonction auxiliaire `k!43` sur la variable `x!1` et le résultat d'un appel d'une fonction auxiliaire `k!42` sur la variable `x!2`.

La fonction `k!43` retourne -2 si $x!1 < -1$, 10 si $x!1 \geq 10$, et `x!1` sinon. La fonction `k!42` retourne 0 si $x!2 < 1$, 10 si $x!2 \geq 10$, et `x!2` sinon. Au final, la relation $Invar^I$ trouvée est

$$Invar^I = \{(0,1),(2,2),(4,3),(6,4),(8,5)\} \cup \{(i,6) \in \mathbb{Z}^2 \mid i \geq 10\} . \tag{64}$$

Cet invariant est représenté dans la figure 36.

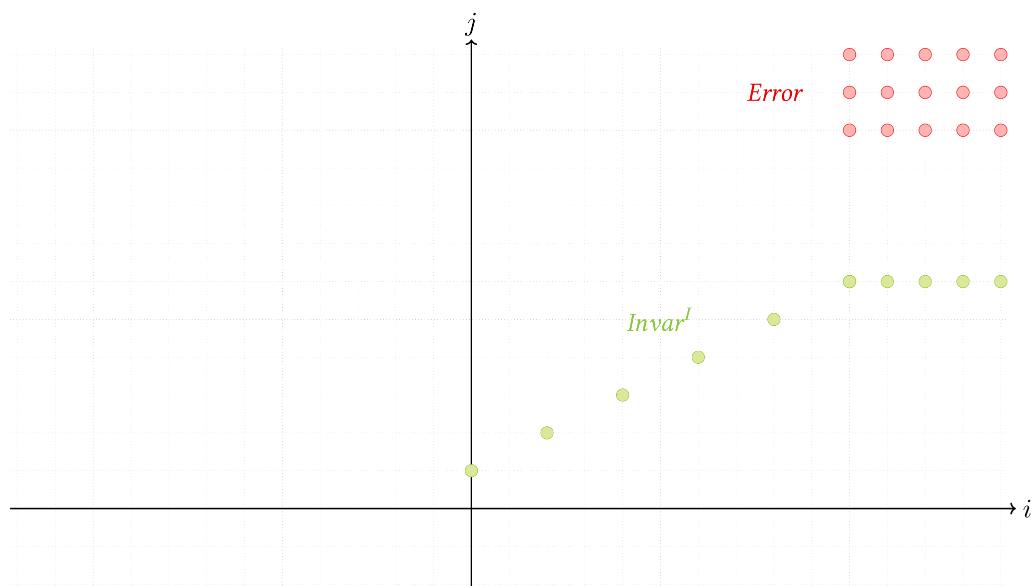


FIGURE 36. L'invariant inductif $Invar^I$ retourné par Z3 et les configurations d'erreur $Error$ du programme.

16.5. * **Exemple de modélisation : pavage du plan.** Pour finir cette section, mentionnons qu'en général, il n'existe pas d'algorithme qui résout SATISFIABILITÉ MODULO T. En effet, une formule close φ appartient à la théorie T si et seulement si sa négation $\neg\varphi$ n'est pas satisfiable modulo T. Un algorithme pour SATISFIABILITÉ MODULO T permettrait donc de décider la théorie T, or nous avons vu en section 15.3 que certaines théories comme l'arithmétique élémentaire ou $\text{Th}(\mathbb{Z}, +, \times)$ étaient indécidables.

Problème de pavage du plan. Dans cette section, nous allons en faire une preuve directe, qui sera aussi l'occasion de modéliser un problème de plus comme un problème de satisfiabilité modulo théorie. Le problème qui nous intéresse est le problème de *pavage du plan*. L'entrée du problème est un *catalogue*, qui est un ensemble fini C de *tuiles* carrées avec une couleur par côté comme celles de la figure 37.

(BÖRGER, GRÄDEL et GUREVICH, 1997, sec. 3.1)

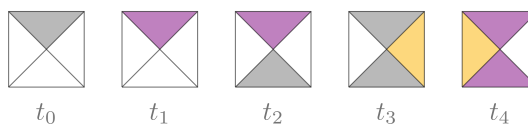


FIGURE 37. Un catalogue $C_{\text{ex}} = \{t_0, t_1, t_2, t_3, t_4\}$.

Le but pour un catalogue C donné est de déterminer s'il est possible de couvrir le plan $\mathbb{N} \times \mathbb{N}$ en respectant les couleurs. On peut pour cela réutiliser les tuiles du catalogue, mais celles-ci ne peuvent pas être tournées. La figure 38 montre deux exemples de pavages possibles avec le catalogue de la figure 37. Formellement, un catalogue C est associé à deux relations binaires $H \subseteq C \times C$ de contraintes horizontales et $V \subseteq C \times C$ de contraintes verticales, où $(t, t') \in H$ si la couleur de droite de t est la même que la couleur de gauche de t' et $(t, t') \in V$ si la couleur du haut de t est la même que la couleur du bas de t'. Par exemple, pour le catalogue de la figure 37,

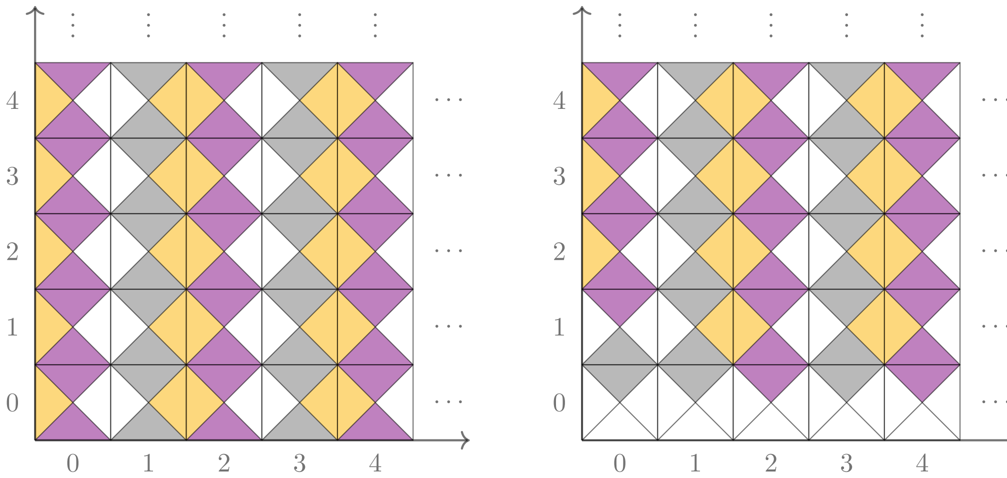


FIGURE 38. Deux pavages du plan possibles avec le catalogue C_{ex} de la figure 37.

on a les contraintes suivantes :

$$H = \{(t_0,t_0),(t_0,t_1),(t_0,t_2),(t_0,t_3),(t_1,t_0),(t_1,t_1),(t_1,t_2),(t_1,t_3),$$
$$(t_2,t_0),(t_2,t_1),(t_2,t_2),(t_2,t_3),(t_3,t_4),(t_4,t_0),(t_4,t_1),(t_4,t_2),(t_4,t_3)\}$$
$$V = \{(t_0,t_2),(t_0,t_3),(t_1,t_4),(t_2,t_4),(t_3,t_2),(t_3,t_3),(t_4,t_4)\}$$

Un *pavage du plan par* C est alors une fonction $p\colon \mathbb{N}\times\mathbb{N}\to C$ telle que

(1) si $p(i,j)=t$ et $p(i+1,j)=t'$ alors $(t,t')\in H$ et
(2) si $p(i,j)=t$ et $p(i,j+1)=t'$ alors $(t,t')\in V$.

Le problème de décision correspondant est le suivant.

Problème (PAVAGE DU PLAN)**.**
instance : un catalogue C et deux relations $H,V\subseteq C\times C$
question : est-ce qu'il existe un pavage du plan par C ?

Le problème de PAVAGE DU PLAN n'a pas de solution algorithmique.

Théorème 16.3 (BERGER–GUREVICH et KORYAKOV)**.** *Il n'existe pas d'algorithme qui prend en entrée un catalogue de tuiles C avec ses deux relations de contraintes horizontales H et de contraintes verticales V et répond s'il existe un pavage du plan.*

Modélisation en SMLT-LIB. Voyons comment modéliser le problème de PAVAGE DU PLAN comme un problème de satisfiabilité modulo une théorie. Nous commençons par déclarer un type énumératif pour le catalogue C et par définir les relations binaires H et V associées.

tiling-int.smt2
```
; déclaration du catalogue
(declare-datatypes () ((C t0 t1 t2 t3 t4)))
; les contraintes horizontales
(define-fun H ((s C)(t C)) Bool
  (ite (and (= s t0) (= t t0)) true
  (ite (and (= s t0) (= t t1)) true
  (ite (and (= s t0) (= t t2)) true
  (ite (and (= s t0) (= t t3)) true
  (ite (and (= s t1) (= t t0)) true
```

```
  (ite (and (= s t1) (= t t1)) true
  (ite (and (= s t1) (= t t2)) true
  (ite (and (= s t1) (= t t3)) true
  (ite (and (= s t2) (= t t0)) true
  (ite (and (= s t2) (= t t1)) true
  (ite (and (= s t2) (= t t2)) true
  (ite (and (= s t2) (= t t3)) true
  (ite (and (= s t3) (= t t4)) true
  (ite (and (= s t4) (= t t0)) true
  (ite (and (= s t4) (= t t1)) true
  (ite (and (= s t4) (= t t2)) true
  (ite (and (= s t4) (= t t3)) true
  false))))))))))))))))))
; les contraintes verticales
(define-fun V ((s C)(t C)) Bool
  (ite (and (= s t0) (= t t2)) true
  (ite (and (= s t0) (= t t3)) true
  (ite (and (= s t1) (= t t4)) true
  (ite (and (= s t2) (= t t4)) true
  (ite (and (= s t3) (= t t2)) true
  (ite (and (= s t3) (= t t3)) true
  (ite (and (= s t4) (= t t4)) true
  false))))))))
```

Il reste alors à définir la fonction de pavage p qui doit respecter les contraintes horizontales et verticales.

tiling-int.smt2

```
; la fonction p
(declare-fun p (Int Int) C)
(assert
  (forall ((i Int)(j Int))
    (=> (and (>= i 0) (>= j 0))
        (and (H (p i j) (p (+ i 1) j))
             (V (p i j) (p i (+ j 1)))))))
(check-sat-using (then qe smt))
(exit)
```

Sans surprise au vu du théorème de Berger–Gurevich et Koryakov et sachant comment fonctionnent les solveurs SMT, Z3 n'arrive pas à résoudre ce problème et répond « unknown ».

Modélisation sans théorie. On peut aussi modéliser un problème de pavage comme un problème de satisfaction sans faire appel à une théorie sous-jacente. On utilise pour cela une relation binaire non interprétée P_t pour chaque tuile $t \in C$ du catalogue ; l'idée ici étant que $(x, y) \in P_t$ si $p(x, y) = t$. On écrit alors une formule close

$$\varphi_C \stackrel{\text{def}}{=} \forall x \exists x' \forall y. \varphi_1(x, y) \wedge \varphi_H(x, x', y) \wedge \varphi_V(x, x', y) \tag{65}$$

qui force les relations P_t à coder une fonction de pavage p en demandant que chaque position (x, y) soit associée à une unique tuile par

$$\varphi_1(x, y) \stackrel{\text{def}}{=} \bigwedge_{t \neq t'} \neg P_t(x, y) \vee \neg P_{t'}(x, y) , \tag{66}$$

en vérifiant les contraintes horizontales par

$$\varphi_H(x, x', y) \stackrel{\text{def}}{=} \bigvee_{(t, t') \in H} P_t(x, y) \wedge P_{t'}(x', y) , \tag{67}$$

et en vérifiant les contraintes verticales par

$$\varphi_V(x, x', y) \stackrel{\text{def}}{=} \bigvee_{(t,t')\in V} P_t(y, x) \wedge P_{t'}(y, x') \,. \tag{68}$$

Proposition 16.4. *La formule φ_C est satisfiable si et seulement si on peut paver le plan avec les tuiles du catalogue C.*

Démonstration. Notons tout d'abord que la skolémisation de φ_C est la formule

$$\varphi'_C \stackrel{\text{def}}{=} \forall x \forall y . \varphi_1(x, y) \wedge \varphi_H(x, f(x), y) \wedge \varphi_V(x, f(x), y)$$

où l'on a introduit un nouveau symbole de fonction unaire f. La formule φ'_C est équi-satisfiable avec φ, donc il suffit de montrer que φ'_C est satisfiable si et seulement si on pouvait paver le plan avec les tuiles du catalogue C.

Si on peut paver le plan par une fonction $p\colon \mathbb{N} \times \mathbb{N} \to C$, alors il existe une interprétation I telle que $I \vDash \varphi'_C$. On définit pour cela le domaine $D_I \stackrel{\text{def}}{=} \mathbb{N}$ où f est interprétée comme la fonction successeur $f^I\colon n \mapsto n+1$ et chaque P_t pour $t \in C$ est interprétée par la relation $P_t^I \stackrel{\text{def}}{=} \{(i, j) \mid p(i, j) = t\}$. La formule φ'_C est donc satisfiable dans ce cas.

Inversement, si φ'_C est satisfiable, alors il existe une interprétation I telle que $I \vDash \varphi'_C$. Par le théorème 14.15 de Herbrand, on peut supposer sans perte de généralité que I a pour domaine $D_I \stackrel{\text{def}}{=} T(\mathcal{F})$ l'ensemble des termes clos sur l'ensemble de symboles de fonctions $\mathcal{F} = \{a^{(0)}, f^{(1)}\}$ et interprète a comme le terme constitué d'une feuille étiquetée par a et f comme la fonction $f^I\colon u \mapsto f(u)$ qui rajoute une nouvelle racine étiquetée par f au-dessus du terme u. On peut alors observer que $I \vDash \varphi'_C$ implique l'existence d'un pavage du plan $p\colon \mathbb{N} \times \mathbb{N} \to C$ où $p(i, j) = t$ pour $(i, j) \in \mathbb{N} \times \mathbb{N}$ si et seulement si la paire de termes

$$(\,\underbrace{f(f(\cdots(f}_{i \text{ fois}}(a)\cdots)),\ \underbrace{f(f(\cdots(f}_{j \text{ fois}}(a)))\,)$$

appartient à l'interprétation P_t^I de P_t. □

On peut en déduire le corollaire suivant du théorème de Berger–Gurevich et Koryakov – qui pouvait aussi se déduire du théorème 15.18 d'indécidabilité de la VALIDITÉ puisque φ est valide si et seulement si $\neg\varphi$ n'est pas satisfiable.

Corollaire 16.5 (indécidabilité de la SATISFIABILITÉ)**.** *Il n'existe pas d'algorithme qui prend en entrée une formule φ et retourne si oui ou non φ est satisfiable.*

Cette autre modélisation peut aussi être écrite en SMT-LIB comme suit.

tiling.smt2
```
; on appelle notre domaine D
(declare-sort D)
; on déclare une relation binaire non interprétée par tuile du catalogue
(declare-fun P0 (D D) Bool)
(declare-fun P1 (D D) Bool)
(declare-fun P2 (D D) Bool)
(declare-fun P3 (D D) Bool)
(declare-fun P4 (D D) Bool)
; équation (66) : au plus une tuile par position
(define-fun phi1 ((x D) (y D)) Bool
  (and (not (and (P0 x y) (P1 x y)))
       (not (and (P0 x y) (P2 x y)))
       (not (and (P0 x y) (P3 x y)))
       (not (and (P0 x y) (P4 x y)))
       (not (and (P1 x y) (P0 x y)))
       (not (and (P1 x y) (P2 x y)))
```

```
       (not (and (P1 x y) (P3 x y)))
       (not (and (P1 x y) (P4 x y)))
       (not (and (P2 x y) (P0 x y)))
       (not (and (P2 x y) (P1 x y)))
       (not (and (P2 x y) (P3 x y)))
       (not (and (P2 x y) (P4 x y)))
       (not (and (P3 x y) (P0 x y)))
       (not (and (P3 x y) (P1 x y)))
       (not (and (P3 x y) (P2 x y)))
       (not (and (P3 x y) (P4 x y)))
       (not (and (P4 x y) (P0 x y)))
       (not (and (P4 x y) (P1 x y)))
       (not (and (P4 x y) (P2 x y)))
       (not (and (P4 x y) (P3 x y)))))
; équation (67) : les contraintes horizontales
(define-fun phiH ((x D) (xp D) (y D)) Bool
  (or (and (P0 x y) (P0 xp y))
      (and (P0 x y) (P1 xp y))
      (and (P0 x y) (P2 xp y))
      (and (P0 x y) (P3 xp y))
      (and (P1 x y) (P0 xp y))
      (and (P1 x y) (P1 xp y))
      (and (P1 x y) (P2 xp y))
      (and (P1 x y) (P3 xp y))
      (and (P2 x y) (P0 xp y))
      (and (P2 x y) (P1 xp y))
      (and (P2 x y) (P2 xp y))
      (and (P2 x y) (P3 xp y))
      (and (P3 x y) (P4 xp y))
      (and (P4 x y) (P0 xp y))
      (and (P4 x y) (P1 xp y))
      (and (P4 x y) (P2 xp y))
      (and (P4 x y) (P3 xp y))))
; équation (68) : les contraintes verticales
(define-fun phiV ((x D) (xp D) (y D)) Bool
  (or (and (P0 y x) (P2 y xp))
      (and (P0 y x) (P3 y xp))
      (and (P1 y x) (P4 y xp))
      (and (P2 y x) (P4 y xp))
      (and (P3 y x) (P2 y xp))
      (and (P3 y x) (P3 y xp))
      (and (P4 y x) (P4 y xp))))
; la formule complète (65)
(assert
  (forall ((x D))
    (exists ((xp D))
      (forall ((y D))
        (and (phi1 x y) (phiH x xp y) (phiV x xp y))))))
(check-sat-using (then qe smt))
(exit)
```

17. Calcul des séquents

Résumé. Le *calcul des séquents* est un système de déduction qui manipule des séquents $\vdash \Gamma$, où Γ est un multi-ensemble fini de formules sous forme normale négative. Un séquent $\vdash \Gamma$ pour lequel il existe une dérivation dans le système de preuve est dit *prouvable*, ce qui est noté « $\vdash_{\mathbf{LK}} \Gamma$ ».

Un séquent $\vdash \Gamma$ est *valide*, si pour toute interprétation I, il existe une formule $\vartheta \in \mathrm{dom}(\Gamma)$ du séquent telle que $I \vDash \vartheta$. Par le théorème 17.5 de *correction*, si $\vdash \Gamma$ est prouvable, alors il est valide. Inversement, par le théorème 17.17 de *complétude*, si $\vdash \Gamma$ est valide, alors il est prouvable, et même prouvable par une preuve qui n'utilise pas la règle de coupure (cut).

Le calcul des séquents peut être enrichi par des règles *admissibles*, telles que si toutes les prémisses de la règle sont prouvables, alors sa conclusion l'est aussi ; voir la figure 40. Ces résultats sont démontrés de manière « syntaxique », c'est-à-dire par manipulation des dérivations ; un résultat majeur de ce type est le théorème 17.19 d'*élimination des coupures*.

Comme dans le cas propositionnel, les règles du calcul des séquents (sauf (cut)) sont syntaxiquement *inversibles* (c.f. section 17.2.5). En revanche, au premier ordre, on n'a **pas** la propriété de *branches finies* (cela vient de la règle existentielle (∃)) : une recherche de preuve ne termine pas toujours. Comme vu avec le théorème 15.18, cela est inévitable, puisqu'il n'existe en effet pas d'algorithme qui résoud le problème de VALIDITÉ.

(Duparc, 2015, sec. 12.3), (David, Nour et Raffalli, 2003, ch. 5), (Goubault-Larrecq et Mackie, 1997, fig. 6.2) pour le calcul des séquents. Voir (ibid., sec. 6.3) pour un système de preuve à la Hilbert, et (David, Nour et Raffalli, 2003, sec 1.3) et (Goubault-Larrecq et Mackie, 1997, fig. 2.2 et 6.1) pour des systèmes de déduction naturelle.

Nous allons étendre le calcul des séquents propositionnel $\mathbf{LK}_0$ de la section 8.1 à la logique du premier ordre. Comme dans la section 8.1, nous allons travailler sur des séquents monolatères $\vdash \Gamma$ où Γ est un multi-ensemble fini de formules en forme normale négative, mais maintenant de formules du premier ordre plutôt que de formules propositionnelles.

$$\frac{}{\vdash \Gamma, \ell, \overline{\ell}}\,(\mathsf{ax}) \qquad \frac{\vdash \Gamma, \varphi \quad \vdash \Delta, \overline{\varphi}}{\vdash \Gamma, \Delta}\,(\mathsf{cut})$$

$$\frac{\vdash \Gamma, \varphi \quad \vdash \Gamma, \psi}{\vdash \Gamma, \varphi \wedge \psi}\,(\wedge) \qquad \frac{\vdash \Gamma, \varphi, \psi}{\vdash \Gamma, \varphi \vee \psi}\,(\vee)$$

$$\frac{\vdash \Gamma, \varphi[y/x]}{\vdash \Gamma, \forall x.\varphi}\,(\forall)\ \text{où } y \notin \mathrm{fv}(\Gamma, \forall x.\varphi) \qquad \frac{\vdash \Gamma, \varphi[t/x], \exists x.\varphi}{\vdash \Gamma, \exists x.\varphi}\,(\exists)\ \text{où } t \in T(\mathcal{F}, X)$$

Figure 39. Calcul des séquents monolatère.

Le calcul de la figure 39 ne contient ni la règle structurelle d'échange, qui est implicite parce que nous travaillons avec des multi-ensembles, ni la règle structurelle d'affaiblissement, qui est implicite dans la règle d'axiome (ax) (c.f. lemme 17.8), ni la règle structurelle de contraction, qui est implicite dans la règle de l'existentielle (∃) (c.f. lemme 17.12). La règle (ax) est souvent présentée sous une forme plus générale (c.f. lemme 17.9).

La notion de variable libre est étendue aux multi-ensembles de formules : $\mathrm{fv}(\Gamma) \stackrel{\mathrm{def}}{=} \bigcup_{\varphi \in \mathrm{dom}(\Gamma)} \mathrm{fv}(\varphi)$. On note le séquent vide $\vdash \bot$, où $\bot(\varphi) \stackrel{\mathrm{def}}{=} 0$ pour toute formule φ. Une règle du calcul des séquents permet de déduire un séquent *conclusion* d'un nombre fini de séquents *prémisses*. Chaque règle excepté la règle de coupure comprend une *formule principale* dans sa conclusion, indiquée en orange dans les règles de la figure 39 ; la règle (cut) élimine une *formule de coupure* indiquée en violet. Un séquent $\vdash \Gamma$ est *prouvable*, noté $\vdash_{\mathbf{LK}} \Gamma$, s'il en existe une dérivation dans le système de la figure 39.

Exemple 17.1. La formule du buveur de l'exemple 14.1 est prouvable. Une dérivation possible du séquent correspondant (avec la formule principale indiquée en orange à chaque étape) est :

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash \neg B(x), B(y), \neg B(y), \forall y.B(y), \exists x.(\neg B(x) \vee \forall y.B(y))}\text{(ax)}}{\vdash \neg B(x), B(y), \neg B(y) \vee \forall y.B(y), \exists x.(\neg B(x) \vee \forall y.B(y))}(\vee)}{\vdash \neg B(x), B(y), \exists x.(\neg B(x) \vee \forall y.B(y))}(\exists)}{\vdash \neg B(x), \forall y.B(y), \exists x.(\neg B(x) \vee \forall y.B(y))}(\forall)}{\vdash \neg B(x) \vee \forall y.B(y), \exists x.(\neg B(x) \vee \forall y.B(y))}(\vee)}{\vdash \exists x.(\neg B(x) \vee \forall y.B(y))}(\exists)$$

Remarque 17.2. La condition $y \notin \text{fv}(\Gamma)$ dans la définition de la règle $(\forall)$ est nécessaire pour la correction du calcul des séquents. Sinon, on pourrait dériver

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash B(x), \neg B(x)}\text{(ax)}}{\vdash \forall x.B(x), \neg B(x)}(\forall)\ \text{(pointillés)}}{\vdash \forall x.B(x), \forall x.\neg B(x)}(\forall)}{\vdash (\forall x.B(x)) \vee (\forall x.\neg B(x))}(\vee)$$

où l'étape en pointillés est incorrecte. Cette formule n'est cependant pas valide : par exemple $D_I \stackrel{\text{def}}{=} \{a, b\}$ avec $B^I \stackrel{\text{def}}{=} \{a\}$ en fournit un contre-modèle puisque d'une part $I, [b/x] \not\models B(x)$ et donc $I \not\models \forall x.B(x)$, et d'autre part $I, [a/x] \not\models \neg B(x)$ et donc $I \not\models \forall x.\neg B(x)$.

Remarque 17.3. La condition $y \notin \text{fv}(\forall x.\varphi)$ dans la définition de la règle $(\forall)$ est nécessaire pour la correction du calcul des séquents. Sinon, comme $(B(x) \vee \neg B(y))[y/x] = B(y) \vee \neg B(y)$, on pourrait dériver

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash B(y), \neg B(y)}\text{(ax)}}{\vdash B(y) \vee \neg B(y)}(\vee)}{\vdash \forall x.B(x) \vee \neg B(y)}(\forall)\ \text{(pointillés)}}{\vdash \forall y \forall x.B(x) \vee \neg B(y)}(\forall)$$

où l'avant-dernière étape est incorrecte. Cette formule n'est cependant pas valide : par exemple $D_I \stackrel{\text{def}}{=} \{a, b\}$ avec $B^I \stackrel{\text{def}}{=} \{a\}$ en fournit un contre-modèle puisque $I, [a/y, b/x] \not\models B(x) \vee \neg B(y)$.

Exemple 17.4. Voici un exemple plus compliqué. On souhaite montrer que la formule (25) $\forall x \exists y \forall z.x < y \wedge \neg(x < z \wedge z < y)$ n'appartient pas à $\text{Th}(A_{oldns})$ la théorie des ordres linéaires denses non bornés de la section 15.2.2. Comme $\text{Th}(A_{oldns})$ est cohérente (elle a $(\mathbb{Q}, <)$ comme modèle), nous allons montrer pour cela que la formule de densité implique la négation de la formule (25). Concrètement, cela signifie que nous allons montrer que la formule

$$\big(\forall x \forall y \exists z.(x < y) \Rightarrow (x < z \wedge z < y)\big) \Rightarrow \neg\big(\forall x \exists y \forall z.x < y \wedge \neg(x < z \wedge z < y)\big)$$

est prouvable (et donc valide par le théorème 17.5 de correction que nous verrons en section 17.1). La première étape est de mettre cette formule sous forme normale négative ; on obtient la formule équivalente :

$$\big(\exists x \exists y \forall z.x < y \wedge (\neg x < z \vee \neg z < y)\big) \vee \big(\exists x \forall y \exists z.\neg x < y \vee (x < z \wedge z < y)\big) .$$

Dans la dérivation ci-dessous, on note par souci de lisibilité :

$$\varphi_1(x) \stackrel{\text{def}}{=} \exists y \forall z.x < y \wedge (\neg x < z \vee \neg z < y) , \qquad \varphi_1'(x, y) \stackrel{\text{def}}{=} \forall z.x < y \wedge (\neg x < z \vee \neg z < y) ,$$

$$\varphi_2(x) \stackrel{\text{def}}{=} \forall y \exists z.\neg x < y \vee (x < z \wedge z < y) , \qquad \varphi_2'(x, y, z) \stackrel{\text{def}}{=} \neg x < y \vee (x < z \wedge z < y) ,$$

$$\Gamma \stackrel{\text{def}}{=} \exists x.\varphi_1(x), \exists y.\varphi_1'(x, y), \exists z.\varphi_2'(x, y, z), \exists x.\varphi_2(x) .$$

$$\dfrac{\dfrac{}{\vdash \Gamma, x < y, \neg x < y, x < z \land z < y}\text{(ax)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \Gamma, \neg x < z, \neg z < y, \neg x < y, x < z}\text{(ax)} \quad \dfrac{}{\vdash \Gamma, \neg x < z, \neg z < y, \neg x < y, z < y}\text{(ax)}}{\vdash \Gamma, \neg x < z, \neg z < y, \neg x < y, x < z \land z < y}(\land)}{\vdash \Gamma, \neg x < z \lor \neg z < y, \neg x < y, x < z \land z < y}(\lor)}{}}{\vdash \Gamma, x < y \land (\neg x < z \lor \neg z < y), \neg x < y, x < z \land z < y}(\land)$$

$$\dfrac{\vdash \Gamma, x < y \land (\neg x < z \lor \neg z < y), \neg x < y, x < z \land z < y}{\vdash \Gamma, x < y \land (\neg x < z \lor \neg z < y), \neg x < y \lor (x < z \land z < y)}(\lor)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \Gamma, x < y \land (\neg x < z \lor \neg z < y), \neg x < y \lor (x < z \land z < y)}{\vdash \exists x.\varphi_1(x), \exists y.\varphi_1'(x,y), x < y \land (\neg x < z \lor \neg z < y), \exists z.\varphi_2'(x,y,z), \exists x.\varphi_2(x)}(\exists)}{\vdash \exists x.\varphi_1(x), \exists y.\varphi_1'(x,y), \forall z.x < y \land (\neg x < z \lor \neg z < y), \exists z.\varphi_2'(x,y,z), \exists x.\varphi_2(x)}(\forall)}{\vdash \exists x.\varphi_1(x), \exists y.\varphi_1'(x,y), \exists z.\varphi_2'(x,y,z), \exists x.\varphi_2(x)}(\exists)}{\vdash \exists x.\varphi_1(x), \varphi_1(x), \forall y \exists z.\varphi_2'(x,y,z), \exists x.\varphi_2(x)}(\forall)}{\vdash \exists x.\varphi_1(x), \varphi_1(x), \exists x.\varphi_2(x)}(\exists)}{\vdash \exists x.\varphi_1(x), \exists x.\varphi_2(x)}(\exists)}{\vdash (\exists x.\varphi_1(x)) \lor (\exists x.\varphi_2(x))}(\lor)$$

17.1. **Correction.** Un séquent $\vdash \Gamma$ *satisfait* une interprétation I sous une valuation ρ, noté $I, \rho \vDash \Gamma$, s'il existe une formule $\vartheta \in \text{dom}(\Gamma)$ telle que $I, \rho \vDash \vartheta$; la formule ϑ en question est appelée *formule témoin*. Un séquent $\vdash \Gamma$ est *valide*, noté $\vDash \Gamma$, si $I, \rho \vDash \Gamma$ pour tous I et ρ. La *correction* d'un système de preuve consiste à montrer que toute formule prouvable est valide.

Théorème 17.5 (correction). *Si* $\vdash_{\mathbf{LK}} \Gamma$*, alors* $\vDash \Gamma$.

Démonstration. On procède par induction structurelle sur une dérivation de $\vdash \Gamma$, en montrant pour chaque règle du calcul des séquents que si les prémisses sont valides, alors la conclusion l'est aussi. On ne traitera ici que les règles de quantification et de coupure, les autres cas ayant été traités dans la démonstration du théorème 8.10 de correction du calcul des séquents propositionnel.

Pour $(\forall)$ **:** supposons la prémisse $\vdash \Gamma, \varphi[y/x]$ valide pour $y \notin \text{fv}(\Gamma, \forall x.\varphi)$. Prenons (I, ρ) arbitraire et montrons que $I, \rho \vDash \Gamma, \forall x.\varphi$.

Comme la prémisse est valide, pour tout $e \in D_I$, $I, \rho[e/y] \vDash \Gamma, \varphi[y/x]$, donc il existe une formule témoin $\vartheta_e \in \text{dom}(\Gamma) \cup \{\varphi[y/x]\}$ telle que $I, \rho[e/y] \vDash \vartheta_e$

S'il existe $e \in D_I$ tel que $\vartheta_e \in \text{dom}(\Gamma)$, alors comme $y \notin \text{fv}(\Gamma)$ et donc $y \notin \text{fv}(\vartheta_e)$, par la propriété 12.1, on a aussi $I, \rho \vDash \vartheta_e$. Donc $I, \rho \vDash \Gamma, \forall x.\varphi$.

On voit dans cette preuve que les deux conditions nécessaires $y \notin \text{fv}(\Gamma)$ et $y \notin \text{fv}(\forall x.\varphi)$ des remarques 17.2 et 17.3 sont aussi suffisantes pour la correction de notre calcul des séquents.

Sinon, $\vartheta_e = \varphi[y/x]$ pour tout $e \in D_I$, c'est-à-dire $I, \rho[e/y] \vDash \varphi[y/x]$ pour tout $e \in D_I$. Par le lemme 13.5 de substitution, on a ainsi $I, (\rho[e/y])[[\![y]\!]^I_{\rho[e/y]}/x] \vDash \varphi$ pour tout $e \in D_I$, ce qui n'est jamais que $I, \rho[e/y][e/x] \vDash \varphi$ pour tout $e \in D_I$. On a donc $I, \rho[e/y] \vDash \forall x.\varphi$, et comme $y \notin \text{fv}(\forall x.\varphi)$, on a même $I, \rho \vDash \forall x.\varphi$ par la propriété 12.1. Donc $I, \rho \vDash \Gamma, \forall x.\varphi$.

Pour $(\exists)$ **:** supposons la prémisse valide. Prenons (I, ρ) arbitraire et montrons que $I, \rho \vDash \Gamma, \exists x.\varphi$.

Si la formule témoin de $I, \rho \vDash \Gamma, \varphi[t/x], \exists x.\varphi$ est dans $\{\exists x.\varphi\} \cup \text{dom}(\Gamma)$, alors elle peut aussi servir de témoin pour $I, \rho \vDash \Gamma, \exists x.\varphi$. Si la formule témoin est $\varphi[t/x]$, c'est-à-dire si $I, \rho \vDash \varphi[t/x]$, alors $I, \rho[[\![t]\!]^I_\rho/x] \vDash \varphi$ par le lemme 13.5 de substitution, et donc $I, \rho \vDash \exists x.\varphi$, qui peut servir de témoin pour $I, \rho \vDash \Gamma, \exists x.\varphi$.

Pour (cut) : supposons les deux prémisses $\vdash \Gamma, \varphi$ et $\vdash \Delta, \overline{\varphi}$ valides. Prenons (I, ρ) arbitraires et montrons que $I, \rho \vDash \Gamma, \Delta$.

Comme $\vdash \Gamma, \varphi$ et $\vdash \Delta, \overline{\varphi}$ sont valides, il existe une formule témoin $\vartheta \in \text{dom}(\Gamma) \cup \{\varphi\}$ et une formule témoin $\vartheta' \in \text{dom}(\Delta) \cup \{\overline{\varphi}\}$ telles que $I, \rho \vDash \vartheta$ et $I, \rho \vDash \vartheta'$. Si $\vartheta \in \text{dom}(\Gamma)$ ou si $\vartheta' \in \text{dom}(\Delta)$, alors $I, \rho \vDash \Gamma, \Delta$ comme désiré. Sinon, $\vartheta = \varphi$ et $\vartheta' = \overline{\varphi}$, mais alors $I, \rho \vDash \varphi$ et $I, \rho \vDash \overline{\varphi}$, ce qui est absurde : ce cas ne peut pas se produire. $\square$

17.2. * **Règles admissibles.** Le calcul des séquents de la figure 39 peut être enrichi par plusieurs règles *admissibles*, qui n'affectent pas la prouvabilité, et récapitulées dans la figure 40 ci-dessous. Les preuves en sont établies *syntaxiquement*, par manipulation des dérivations, et préfigurent les techniques employées pour l'élimination des coupures que nous verrons en section 17.4.

$$\frac{\vdash \Gamma}{\vdash \Delta}\,(=_\alpha) \text{ où } \Gamma =_\alpha \Delta \qquad \frac{\vdash \Gamma}{\vdash \Gamma\sigma}\,(\mathsf{S}) \text{ où } \sigma \text{ est une substitution} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \Delta}\,(\mathsf{W}) \qquad \frac{}{\vdash \Gamma, \varphi, \overline{\varphi}}\,(\mathsf{ax}')$$

$$\frac{\vdash \Gamma, \varphi, \varphi}{\vdash \Gamma, \varphi}\,(\mathsf{C}) \qquad \frac{\vdash \Gamma, \varphi \vee \psi}{\vdash \Gamma, \varphi, \psi}\,(\mathsf{E}_\vee) \qquad \frac{\vdash \Gamma, \varphi \wedge \psi}{\vdash \Gamma, \varphi}\,(\mathsf{E}^1_\wedge) \qquad \frac{\vdash \Gamma, \varphi \wedge \psi}{\vdash \Gamma, \psi}\,(\mathsf{E}^2_\wedge)$$

$$\frac{\vdash \Gamma, \forall x.\varphi}{\vdash \Gamma, \varphi[y/x]}\,(\mathsf{E}_\forall) \text{ où } y \notin \mathrm{fv}(\Gamma, \forall x.\varphi) \qquad \frac{\vdash \Gamma, \exists x.\varphi}{\vdash \Gamma, \varphi[t/x], \exists x.\varphi}\,(\mathsf{E}_\exists) \text{ où } t \in T(\mathcal{F}, X)$$

FIGURE 40. Quelques règles admissibles du calcul des séquents.

On définit la *profondeur* $p(\pi)$ d'une dérivation π dans le calcul des séquents comme celle de l'arbre sous-jacent.

17.2.1. *α-congruence syntaxique.* Commençons par étendre l'α-congruence aux multi-ensembles : si $\varphi_i =_\alpha \psi_i$ pour tout $1 \leq i \leq n$, alors $\varphi_1, \ldots, \varphi_n =_\alpha \psi_1, \ldots, \psi_n$. Le lemme suivant montre l'admissibilité de la règle $(=_\alpha)$.

Lemme 17.6 (α-congruence syntaxique)**.** *Si $\vdash_{\mathbf{LK}} \Gamma$ par une dérivation π, alors pour tout multi-ensemble $\Delta =_\alpha \Gamma$, $\vdash_{\mathbf{LK}} \Delta$ par une dérivation de profondeur $p(\pi)$ et sans coupure si π était sans coupure.*

Démonstration. On montre que si $\vdash_{\mathbf{LK}} \Gamma, \varphi$ par une dérivation π, alors pour toute formule $\psi =_\alpha \varphi$, $\vdash_{\mathbf{LK}} \Gamma, \psi$ par une dérivation de profondeur $p(\pi)$ et sans coupure si π était sans coupure, ce qui démontrera le lemme par induction sur la taille du séquent.

On procède pour cela par récurrence sur la profondeur de la dérivation π de $\vdash_{\mathbf{LK}} \Gamma, \varphi$. On opère à une distinction de cas selon la dernière règle appliquée dans la dérivation π.

Si π se termine par une règle (R) autre que (cut) où φ n'est pas principale, alors par inspection des règles, on est nécessairement dans une situation

$$\frac{\begin{matrix}\pi_1\\ \vdots\\ \vdash \Gamma_1, \varphi\end{matrix} \quad \cdots \quad \begin{matrix}\pi_k\\ \vdots\\ \vdash \Gamma_k, \varphi\end{matrix}}{\vdash \Gamma, \varphi}\,(\mathsf{R})$$

où $0 \leq k \leq 2$ ($k = 0$ correspondant au cas de la règle (ax)). Pour tout $1 \leq i \leq k$, par hypothèse de récurrence sur π_i, il existe une dérivation π'_i de $\vdash \Gamma_i, \psi$ de profondeur $p(\pi_i)$ et sans (cut) si π_i était sans (cut). On a donc la dérivation

$$\frac{\begin{matrix}\pi'_1\\ \vdots\\ \vdash \Gamma_1, \psi\end{matrix} \quad \cdots \quad \begin{matrix}\pi'_k\\ \vdots\\ \vdash \Gamma_k, \psi\end{matrix}}{\vdash \Gamma, \psi}\,(\mathsf{R})$$

Si π se termine par une règle (cut), alors sans perte de généralité, on est dans une situation

$$\dfrac{\dfrac{\pi_1}{\vdots} \quad\quad \dfrac{\pi_2}{\vdots}}{\dfrac{\vdash \Gamma, \varphi, \theta \quad\quad \vdash \Delta, \overline{\theta}}{\vdash \Gamma, \Delta, \varphi}\ (\text{cut})}$$

(l'autre situation étant celle où φ apparaissait dans la seconde prémisse). Par hypothèse de récurrence sur π_1, il existe une dérivation π_1' de $\vdash \Gamma, \psi, \theta$ de profondeur $p(\pi_1)$, et on a donc la dérivation

$$\dfrac{\dfrac{\pi_1'}{\vdots} \quad\quad \dfrac{\pi_2}{\vdots}}{\dfrac{\vdash \Gamma, \psi, \theta \quad\quad \vdash \Delta, \overline{\theta}}{\vdash \Gamma, \Delta, \psi}\ (\text{cut})}$$

Il reste à traiter les cas où φ était principale dans la dernière règle appliquée par π. On procède par induction sur la congruence $\varphi =_\alpha \psi$. Les cas de la réflexivité, de la symétrie, et de la transitivité de $=_\alpha$ sont triviaux.

Cette preuve serait beaucoup plus difficile si (ax) *était remplacé par* (ax′) *: avec* (ax), $\ell =_\alpha \ell'$ *si et seulement si* $\ell = \ell'$ *et est donc traité par le cas de la réflexivité ; avec* (ax′), *chacun des cas ci-dessous peut provenir d'une application de l'axiome étendu.*

Cas de la congruence pour $\vee$: alors $\varphi = \varphi_1 \vee \varphi_2$ et $\psi = \psi_1 \vee \psi_2$ avec $\varphi_1 =_\alpha \psi_1$ et $\varphi_2 =_\alpha \psi_2$. Comme φ est principale la dernière règle appliquée dans π était ($\vee$), et donc $\vdash_{\mathbf{LK}} \Gamma, \varphi_1, \varphi_2$ par une dérivation de profondeur $p(\pi) - 1$. Par hypothèse de récurrence, $\vdash_{\mathbf{LK}} \Gamma, \psi_1, \varphi_2$ par une dérivation de profondeur $p(\pi) - 1$, et par une seconde application de l'hypothèse de récurrence, $\vdash_{\mathbf{LK}} \Gamma, \psi_1, \psi_2$ par une dérivation de profondeur $p(\pi) - 1$. Par une application de ($\vee$), $\vdash_{\mathbf{LK}} \Gamma, \psi_1 \vee \psi_2$.

Autres cas de congruence : similaires au cas précédent.

Cas de l'α-renommage pour $\forall$: alors $\varphi = \forall x.\varphi'$ et $\psi = \forall z.\varphi'[z/x]$ où $z \notin \text{fv}(\forall x.\varphi')$ et $[z/x]$ est applicable à φ'. Comme φ est principale, la dernière règle appliquée dans π était ($\forall$), et donc $\vdash_{\mathbf{LK}} \Gamma, \varphi'[y/x]$ pour une variable $y \notin \text{fv}(\Gamma, \forall x.\varphi')$ par une dérivation π' de profondeur $p(\pi) - 1$. Par l'équation (27), $\varphi'[y/x] =_\alpha \varphi'[z/x][y/z]$ puisque $z \notin \text{fv}(\varphi')$. Par hypothèse de récurrence sur π', $\vdash_{\mathbf{LK}} \Gamma, \varphi'[z/x][y/z]$ par une dérivation de profondeur $p(\pi')$. Comme $y \notin \text{fv}(\Gamma, \varphi'[z/x][y/z])$, on peut appliquer ($\forall$) et dériver $\vdash_{\mathbf{LK}} \Gamma, \forall z.\varphi'[z/x]$.

Cas de l'α-renommage pour $\exists$: alors $\varphi = \exists x.\varphi'$ et $\psi = \exists z.\varphi'[z/x]$ où $z \notin \text{fv}(\exists x.\varphi')$ et $[z/x]$ est applicable à φ'. Comme φ est principale,la dernière règle appliquée dans π était ($\exists$), et donc $\vdash_{\mathbf{LK}} \Gamma, \varphi'[t/x]$ pour un terme $t \in T(\mathcal{F}, X)$ par une dérivation π' de profondeur $p(\pi) - 1$. Par l'équation (27), $\varphi'[t/x] =_\alpha \varphi'[z/x][t/z]$ puisque $z \notin \text{fv}(\varphi')$. Par hypothèse de récurrence sur π', $\vdash_{\mathbf{LK}} \Gamma, \varphi'[z/x][t/z]$ par une dérivation de profondeur $p(\pi')$. Par une application de ($\exists$), $\vdash_{\mathbf{LK}} \Gamma, \exists z.\varphi'[z/x]$. □

17.2.2. *Substitution syntaxique.* Une substitution σ est *applicable* à un multi-ensemble Γ si elle est applicable à chacune des formules de $\text{dom}(\Gamma)$. Si c'est le cas, on définit l'application d'une substitution σ à $\Gamma = \varphi_1, \ldots, \varphi_n$ comme une application simultanée de σ à toutes les occurrences de formules dans Γ : $\Gamma\sigma \stackrel{\text{def}}{=} \varphi_1\sigma, \ldots, \varphi_n\sigma$. Comme d'habitude, nous faisons un abus de notation et nous écrivons « $\Gamma\sigma$ » pour un multi-ensemble $\Delta\sigma$ tel que $\Delta =_\alpha \Gamma$ et que σ soit applicable à Δ. Le lemme suivant montre l'applicabilité de la règle (S).

Lemme 17.7 (substitution syntaxique)**.** *Si* $\vdash_{\mathbf{LK}} \Gamma$ *par une dérivation* π*, alors pour toute substitution* σ*,* $\vdash_{\mathbf{LK}} \Gamma\sigma$ *par une dérivation de profondeur* $p(\pi)$ *et sans coupure si* π *était sans coupure.*

Démonstration. Par le lemme 17.6 d'α-congruence syntaxique, on peut supposer σ applicable à Γ. On procède par récurrence sur la profondeur de la dérivation π de $\vdash \Gamma$. Considérons pour cela la dernière règle employée dans cette dérivation.

Cas de (ax) : alors $\Gamma = \Gamma', \ell, \overline{\ell}$. Par la propriété 14.4, $\overline{\ell\sigma} = (\overline{\ell})\sigma$ et (ax) permet aussi de dériver le séquent $\vdash \Gamma'\sigma, \ell\sigma, (\overline{\ell})\sigma$ en une étape.

Cas de (cut) : alors $\Gamma = \Gamma', \Delta$ et $\vdash_{\mathbf{LK}} \Gamma', \varphi$ et $\vdash_{\mathbf{LK}} \overline{\varphi}, \Delta$ pour une formule φ. Par hypothèse de récurrence, on obtient des dérivations de même profondeur de $\vdash \Gamma'\sigma, \varphi\sigma$ et de $\vdash (\overline{\varphi})\sigma, \Delta\sigma$, et la propriété 14.4 montre que l'on peut appliquer (cut) pour obtenir une dérivation de profondeur $p(\pi)$ de $\vdash \Gamma'\sigma, \Delta\sigma$.

Cas de $(\vee)$ **et** $(\wedge)$ **:** similairement par hypothèse de récurrence.

Cas de $(\forall)$ **:** alors $\Gamma = \Gamma', \forall x.\varphi$ et $\vdash_{\mathbf{LK}} \Gamma', \varphi[y/x]$ où $y \notin \text{fv}(\Gamma', \forall x.\varphi)$ par une dérivation π' de profondeur $p(\pi) - 1$.

Par hypothèse de récurrence sur π', on a une dérivation de même profondeur $p(\pi)-1$ du séquent $\vdash \Gamma'[z/y]\sigma[u/z], \varphi[y/x][z/y]\sigma[u/z]$ où on a choisi $z \notin \text{fv}(\Gamma'\sigma, \forall x.\varphi) \cup \text{dom}(\sigma) \cup \text{rv}(\sigma)$ et $u \notin \text{fv}(\Gamma'\sigma, \forall z.\varphi[z/x]\sigma)$ tels que les substitutions soient applicables. Comme $y \notin \text{fv}(\Gamma')$, $\Gamma'[z/y] =_\alpha \Gamma'$ par l'équation (26), et comme $z \notin \text{fv}(\Gamma'\sigma)$, de même $\Gamma'\sigma[u/z] =_\alpha \Gamma'\sigma$. Comme $y \notin \text{fv}(\forall x.\varphi)$, par l'équation (27), $\varphi[y/x][z/y]\sigma[u/z] =_\alpha \varphi[z/x]\sigma[u/z]$. Par le lemme d'$\alpha$-congruence syntaxique, on a donc une dérivation de profondeur $p(\pi) - 1$ de $\vdash \Gamma'\sigma, \varphi[z/x]\sigma[u/z]$. Comme $u \notin \text{fv}(\Gamma'\sigma, \forall z.\varphi[y/x][z/y]\sigma)$, on peut appliquer $(\forall)$ pour obtenir une dérivation de profondeur $p(\pi)$ de $\vdash \Gamma'\sigma, \forall z.(\varphi[z/x]\sigma)$.

Finalement, comme $z \notin \text{fv}(\forall x.\varphi)$, par α-renommage $(\forall x.\varphi)\sigma =_\alpha (\forall z.\varphi[z/x])\sigma = \forall z.(\varphi[z/x]\sigma)$ où σ est applicable puisque $z \notin \text{dom}(\sigma) \cup \text{rv}(\sigma)$. Par le lemme 17.6 d'$\alpha$-congruence syntaxique, on a donc une dérivation de profondeur $p(\pi)$ de $\vdash \Gamma'\sigma, (\forall x.\varphi)\sigma$.

Cas de $(\exists)$ **:** alors $\Gamma = \Gamma', \exists x.\varphi$ et $\vdash_{\mathbf{LK}} \Gamma', \varphi[t/x]$ par π' de profondeur $p(\pi) - 1$.

Par hypothèse de récurrence sur π', on a une dérivation de profondeur $p(\pi) - 1$ du séquent $\vdash \Gamma'\sigma, \varphi[t/x]\sigma$. Observons que si on choisit $z \notin \text{fv}(\exists x.\varphi) \cup \text{dom}(\sigma) \cup \text{rv}(\sigma)$, alors $\varphi[z/x]\sigma[t\sigma/z] =_\alpha \varphi[t/x]\sigma$. Par le lemme d'$\alpha$-congruence syntaxique, on a une dérivation de profondeur $p(\pi) - 1$ de $\vdash \Gamma'\sigma, \varphi[z/x]\sigma[t\sigma/z]$. On peut alors appliquer $(\exists)$ pour dériver $\vdash \Gamma'\sigma, \exists z.(\varphi[z/x]\sigma)$. Comme $z \notin \text{fv}(\exists x.\varphi)$, $\exists z.(\varphi[z/x]\sigma) =_\alpha (\exists x.\varphi)\sigma$ et donc $\vdash_{\mathbf{LK}} \Gamma'\sigma, (\exists x.\varphi)\sigma$ par une dérivation de profondeur $p(\pi)$ par une autre application du lemme d'α-congruence syntaxique. □

17.2.3. *Affaiblissement.* Bien que le système de la figure 39 ne comporte pas de manière explicite la règle d'affaiblissement (notée (W) pour « *weakening* »), celle-ci est admissible.

(David, Nour et Raffalli, 2003, lem. 7.3.1), (Goubault-Larrecq et Mackie, 1997, lem. 2.31).

Lemme 17.8 (affaiblissement)**.** *Si* $\vdash_{\mathbf{LK}} \Gamma$ *par une dérivation* π*, alors pour tout multi-ensemble* Δ*,* $\vdash_{\mathbf{LK}} \Gamma, \Delta$ *par une dérivation de profondeur* $p(\pi)$ *et sans coupure si* π *était sans coupure.*

Démonstration. Par récurrence sur la profondeur de la dérivation de $\vdash \Gamma$, on ajoute systématiquement Δ à tous les séquents. On fait pour cela une distinction de cas selon la dernière règle employée dans la dérivation π.

Cas de (ax) : alors $\Gamma = \Gamma', \ell, \overline{\ell}$ et on a aussi une dérivation de $\vdash \Gamma', \ell, \overline{\ell}, \Delta$ par (ax).

Cas de $(\forall)$ **:** alors $\Gamma = \Gamma', \forall x.\varphi$ et $\vdash_{\mathbf{LK}} \Gamma', \varphi[y/x]$ par une dérivation de profondeur $p(\pi) - 1$ où $y \notin \text{fv}(\Gamma', \forall x.\varphi)$.

Soit $z \notin \text{fv}(\Gamma', \forall x.\varphi, \Delta)$. Par le lemme 17.7, $\vdash_{\mathbf{LK}} \Gamma'[z/y], \varphi[y/x][z/y]$ par une dérivation de profondeur $p(\pi) - 1$. Comme $y \notin \text{fv}(\Gamma')$, par l'équation (26), $\Gamma'[z/y] =_\alpha \Gamma'$, et comme $y \notin \text{fv}(\forall x.\varphi)$, par l'équation (27), $\varphi[y/x][z/y] =_\alpha \varphi[z/x]$. Par le lemme 17.6 d'$\alpha$-congruence syntaxique on peut donc dériver $\vdash \Gamma', \varphi[z/x]$. Par hypothèse de récurrence, $\vdash_{\mathbf{LK}} \Gamma', \varphi[z/x], \Delta$ par une dérivation de profondeur $p(\pi)-1$, et comme $z \notin \text{fv}(\Gamma, \forall x.\varphi, \Delta)$, une application de $(\forall)$ permet de dériver $\vdash \Gamma', \forall x.\varphi, \Delta$.

Les autres cas découlent aisément de l'hypothèse de récurrence. □

17.2.4. *Axiome étendu.* Nous sommes maintenant en mesure de démontrer l'admissibilité de la règle d'axiome étendu (ax′). Cet axiome étendu est usuellement employé à la place de (ax) – qu'il subsume – dans les calculs des séquents de la littérature (David, Nour et Raffalli, 2003 ; Goubault-Larrecq et Mackie, 1997).

Lemme 17.9 (axiome étendu)**.** *Pour tout multi-ensemble* Γ *et toute formule* φ, $\vdash_{\mathbf{LK}} \Gamma, \varphi, \overline{\varphi}$.

Démonstration. On procède par récurrence sur $p(\varphi)$.

Cas de base ℓ : C'est immédiat par une application de l'axiome (ax).

Cas de $\varphi \wedge \psi$: Alors $\overline{\varphi \wedge \psi} = \overline{\varphi} \vee \overline{\psi}$. Par hypothèse de récurrence, $\vdash_{\mathbf{LK}} \Gamma, \varphi, \overline{\varphi}$ par une dérivation π_1 et $\vdash_{\mathbf{LK}} \Gamma, \psi, \overline{\psi}$ par une dérivation π_2. On a donc la dérivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_1}{\vdash \Gamma, \varphi, \overline{\varphi}}}{\vdash \Gamma, \varphi, \overline{\varphi}, \overline{\psi}}\,(\mathrm{W}) \qquad \dfrac{\dfrac{\pi_2}{\vdash \Gamma, \psi, \overline{\psi}}}{\vdash \Gamma, \psi, \overline{\psi}, \overline{\varphi}}\,(\mathrm{W})}{\vdash \Gamma, \varphi \wedge \psi, \overline{\varphi}, \overline{\psi}}\,(\wedge)}{\vdash \Gamma, \varphi \wedge \psi, \overline{\varphi} \vee \overline{\psi}}\,(\vee)$$

Cas de $\varphi \vee \psi$: Similaire au cas précédent.

Cas de $\forall x.\varphi$: Alors $\overline{\forall x.\varphi} = \exists x.\overline{\varphi}$. Par hypothèse de récurrence et la propriété 14.4, on a $\vdash_{\mathbf{LK}} \Gamma, \varphi[y/x], \overline{\varphi}[y/x]$ pour $y \notin \mathrm{fv}(\Gamma, \forall x.\varphi)$ par une dérivation π. On a donc la dérivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi}{\vdash \Gamma, \varphi[y/x], \overline{\varphi}[y/x]}}{\vdash \Gamma, \varphi[y/x], \overline{\varphi}[y/x], \exists x.\overline{\varphi}}\,(\mathrm{W})}{\vdash \Gamma, \varphi[y/x], \exists x.\overline{\varphi}}\,(\exists)}{\vdash \Gamma, \forall x.\varphi, \exists x.\overline{\varphi}}\,(\forall)$$

Cas de $\exists x.\varphi$: Similaire au cas précédent. □

Exemple 17.10. Reprenons la dérivation de l'exemple 17.4. Celle-ci devient très simple quand on utilise la règle (ax′) puisque $\varphi_1(x) = \overline{\varphi_2(x)}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \exists x.\varphi_1(x), \varphi_1(x), \varphi_2(x), \exists x.\varphi_2(x)}\,(\mathrm{ax}')}{\vdash \exists x.\varphi_1(x), \varphi_1(x), \exists x.\varphi_2(x)}\,(\exists)}{\vdash \exists x.\varphi_1(x), \exists x.\varphi_2(x)}\,(\exists)}{\vdash (\exists x.\varphi_1(x)) \vee (\exists x.\varphi_2(x))}\,(\vee)$$

17.2.5. *Inversibilité syntaxique.* Une règle de déduction est *syntaxiquement inversible* si, quand il existe une dérivation (que l'on supposera sans coupure grâce au théorème 17.19 d'élimination des coupures) de son séquent conclusion, alors il existe une dérivation sans coupure de chacun de ses séquents prémisses. La règle (ax) est bien sûr inversible, mais ce qui est plus intéressant, c'est que toutes les autres règles sauf (cut) le sont aussi. Cela signifie que dans une recherche de preuve sans coupure, on peut appliquer ces règles de manière « gloutonne » sans faire de *backtrack* – donc sans avoir à mémoriser de points de choix – et que si l'on arrive à un séquent pour lequel aucune règle ne s'applique, alors c'est qu'il n'y a *pas* de preuve. Cela démontre aussi l'admissibilité des règles d'élimination ($\mathrm{E}_\vee$), ($\mathrm{E}^1_\wedge$), ($\mathrm{E}^2_\wedge$), ($\mathrm{E}_\forall$) et ($\mathrm{E}_\exists$) de la figure 40.

Lemme 17.11 (inversibilité syntaxique)**.** *Les règles du calcul des séquents sauf* (cut) *sont syntaxiquement inversibles.*

Démonstration. On ne traite ici que les règles ($\exists$) et ($\forall$), les autres étant traitées dans la démonstration du lemme 8.8 d'inversibilité syntaxique du calcul des séquents propositionnel.

Pour la règle ($\exists$) : s'il y a une preuve sans coupure de $\vdash \Gamma, \exists x.\varphi$, alors par affaiblissement il en existe aussi une de $\vdash \Gamma, \exists x.\varphi, \varphi[t/x]$.

Pour la règle ($\forall$) : soit π une dérivation sans coupure du séquent $\vdash \Gamma, \forall x.\varphi$. On montre par récurrence sur la profondeur de π que pour toute variable $y \notin \text{fv}(\Gamma, \forall x.\varphi)$, $\vdash_{\mathbf{LK}} \Gamma, \varphi[y/x]$.

- Si π se termine par ($\forall$) où $\forall x.\varphi$ est principale, alors évidemment il existe une sous-dérivation de π de la prémisse $\vdash \Gamma, \varphi[z/x]$ pour une certaine variable $z \notin \text{fv}(\Gamma, \forall x.\varphi)$. Par le lemme 17.7 de substitution syntaxique, il existe aussi une dérivation du séquent $\vdash \Gamma[y/z], \varphi[z/x][y/z]$ pour tout $y \notin \text{fv}(\Gamma, \forall x.\varphi)$, et comme $z \notin \text{fv}(\Gamma, \forall x.\varphi)$, par les équations (26) et (27) ce dernier séquent est α-congruent à $\vdash \Gamma, \varphi[y/x]$ et est dérivable par le lemme d'α-congruence syntaxique.
- Si π se termine par ($\forall$) où $\forall x.\varphi$ n'est pas principale, alors $\Gamma = \Gamma', \forall z.\psi$ et on a la dérivation

$$\dfrac{\begin{array}{c}\pi' \\ \vdots \\ \vdash \Gamma', \psi[v/z], \forall x.\varphi\end{array}}{\vdash \Gamma', \forall z.\psi, \forall x.\varphi}\ (\forall)$$

  où $v \notin \text{fv}(\Gamma', \forall z.\psi, \forall x.\varphi)$.
  On prend une variable fraîche $w \notin \text{fv}(\Gamma', \psi[v/z], \forall x.\varphi)$; par hypothèse de récurrence appliquée à π', il existe une dérivation de $\vdash \Gamma', \psi[v/z], \varphi[w/x]$.
  Comme $v \notin \text{fv}(\Gamma', \forall z.\psi, \varphi[w/x])$, on peut appliquer ($\forall$) avec $\forall z.\psi$ comme formule principale pour obtenir $\vdash \Gamma', \forall z.\psi, \varphi[w/x]$.
  Par le lemme de substitution syntaxique, pour toute variable $y \notin \text{fv}(\Gamma', \forall z.\psi, \forall x.\varphi)$ on a donc aussi une dérivation du séquent $\vdash \Gamma'[y/w], (\forall z.\psi)[y/w], \varphi[w/x][y/w]$. Comme $w \notin \text{fv}(\Gamma', \psi[v/z], \forall x.\varphi)$, $w \notin \text{fv}(\Gamma', \forall z.\psi, \forall x.\varphi)$ et donc par les équations (26) et (27) ce dernier séquent est α-congruent à $\vdash \Gamma', \forall z.\psi, \varphi[y/x]$ et est dérivable par le lemme d'α-congruence syntaxique.
- Sinon π se termine par une règle (R) autre que ($\forall$) où $\forall x.\varphi$ n'est pas principale. Par inspection des règles, on est nécessairement dans une situation

$$\dfrac{\begin{array}{c}\pi_1 \\ \vdots \\ \vdash \Gamma_1, \forall x.\varphi\end{array} \quad \cdots \quad \begin{array}{c}\pi_k \\ \vdots \\ \vdash \Gamma_k, \forall x.\varphi\end{array}}{\vdash \Gamma, \forall x.\varphi}\ (\text{R})$$

  où $0 \leq k \leq 2$ ($k = 0$ correspondant au cas de la règle (ax)). Pour tout $y \notin \text{fv}(\Gamma, \forall x.\varphi)$ et tout $1 \leq i \leq k$, par hypothèse de récurrence sur π_i, il existe une dérivation sans coupure π'_i de $\vdash \Gamma_i, \varphi[y/x]$. On a donc la dérivation

$$\dfrac{\begin{array}{c}\pi'_1 \\ \vdots \\ \vdash \Gamma_1, \varphi[y/x]\end{array} \quad \cdots \quad \begin{array}{c}\pi'_k \\ \vdots \\ \vdash \Gamma_k, \varphi[y/x]\end{array}}{\vdash \Gamma, \varphi[y/x]}\ (\text{R})$$

□

L'intérêt de définir le calcul des séquents sans la règle de contraction est bien sûr de diminuer le nombre de règles. Mais cela simplifie aussi la preuve de l'élimination des coupures, qui nécessite sinon l'ajout d'une règle (mix).

17.2.6. *Contraction.* L'admissibilité de la règle (C) est un peu plus délicate à démontrer, et nous allons utiliser une stratégie de démonstration similaire à celle de l'élimination des coupures. Le lemme suivant, combiné au théorème 17.19, montre son admissibilité.

Lemme 17.12 (contraction). *Si $\vdash_{\mathbf{LK}} \Gamma, \varphi, \varphi$ par une dérivation sans coupure, alors $\vdash_{\mathbf{LK}} \Gamma, \varphi$ par une dérivation sans coupure.*

Démonstration. Le *rang de contraction* d'une dérivation π d'un séquent $\vdash \Gamma, \varphi, \varphi$ est le couple $(p(\varphi), p(\pi))$ dans $\mathbb{N}^2$. On ordonne les rangs de contraction par l'ordre lexicographique $<_{\text{lex}}$ sur $\mathbb{N}^2$; il s'agit d'un ordre bien fondé.

On procède par induction bien fondée sur le rang de contraction de la dérivation sans coupure π de $\vdash \Gamma, \varphi, \varphi$. Si φ n'est pas principale dans la dernière règle (R) appliquée dans π, alors par inspection des règles, cette dérivation π est nécessairement de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \vdash \Gamma_1, \varphi, \varphi\end{array} \quad \cdots \quad \begin{array}{c}\pi_k\\ \vdots\\ \vdash \Gamma_k, \varphi, \varphi\end{array}}{\vdash \Gamma, \varphi, \varphi}\,\text{(R)}$$

où $0 \leq k \leq 2$ ($k = 0$ correspondant à une règle (ax)). Comme $(p(\varphi), p(\pi_i)) <_{\text{lex}} (p(\varphi), p(\pi))$ pour tout $1 \leq i \leq k$, on peut appliquer l'hypothèse d'induction à chaque π_i pour obtenir des dérivations π'_i de $\vdash \Gamma_i, \varphi$. Encore par inspection des règles, on obtient une nouvelle dérivation

$$\dfrac{\begin{array}{c}\pi'_1\\ \vdots\\ \vdash \Gamma_1, \varphi\end{array} \quad \cdots \quad \begin{array}{c}\pi'_k\\ \vdots\\ \vdash \Gamma_k, \varphi\end{array}}{\vdash \Gamma, \varphi}\,\text{(R)}$$

Sinon, si φ est principale dans la dernière règle appliquée dans π, on fait une distinction de cas selon cette règle.

Cas de (ax) : alors $\varphi = \ell$ et $\Gamma = \Gamma', \bar{\ell}$, et on a aussi une dérivation de $\vdash \Gamma', \bar{\ell}, \ell$ par (ax).

Cas de $(\vee)$ **:** alors $\varphi = \varphi' \vee \psi$ et π est de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \vdash \Gamma, \varphi', \psi, \varphi' \vee \psi\end{array}}{\vdash \Gamma, \varphi' \vee \psi, \varphi' \vee \psi}\,(\vee)$$

Par le lemme 17.11 d'inversibilité syntaxique appliqué à π_1, on a aussi une dérivation π'_1 de $\vdash \Gamma, \varphi', \psi, \varphi', \psi$. Comme $(p(\varphi'), p(\pi'_1)) <_{\text{lex}} (p(\varphi' \vee \psi), p(\pi))$, par hypothèse d'induction, on a une dérivation π''_1 de $\vdash \Gamma, \varphi', \psi, \psi$. Comme $(p(\psi), p(\pi''_1)) <_{\text{lex}} (p(\varphi' \vee \psi), p(\pi))$, par hypothèse d'induction, on a une dérivation de $\vdash \Gamma, \varphi', \psi$. Par une application de $(\vee)$, on obtient alors une dérivation de $\vdash \Gamma, \varphi' \vee \psi$.

Cas de $(\wedge)$ **:** alors $\varphi = \varphi' \wedge \psi$ et π est de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \vdash \Gamma, \varphi', \varphi' \wedge \psi\end{array} \quad \begin{array}{c}\pi_2\\ \vdots\\ \vdash \Gamma, \psi, \varphi' \wedge \psi\end{array}}{\vdash \Gamma, \varphi' \wedge \psi, \varphi' \wedge \psi}\,(\wedge)$$

Par le lemme 17.11 d'inversibilité syntaxique appliqué à π_1 et π_2, on a aussi une dérivation π'_1 de $\vdash \Gamma, \varphi', \varphi'$ et une dérivation π'_2 de $\vdash \Gamma, \psi, \psi$. Comme $(p(\varphi'), p(\pi'_1)) <_{\text{lex}} (p(\varphi' \wedge \psi), p(\pi))$ et $(p(\psi), p(\pi'_2)) <_{\text{lex}} (p(\varphi' \wedge \psi), p(\pi))$, par hypothèse d'induction, on a des dérivations de $\vdash \Gamma, \varphi'$ et de $\vdash \Gamma, \psi$. Par une application de $(\wedge)$, on obtient alors une dérivation de $\vdash \Gamma, \varphi' \wedge \psi$.

Cas de $(\forall)$ **:** alors $\varphi = \forall x.\psi$ et π est de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \vdash \Gamma, \psi[y/x], \forall x.\psi\end{array}}{\vdash \Gamma, \forall x.\psi, \forall x.\psi}\,(\forall)$$

où $y \notin \text{fv}(\Gamma, \forall x.\psi)$. Par le lemme 17.11 d'inversibilité syntaxique appliqué à π_1, on a aussi une dérivation π_1' de $\vdash \Gamma, \psi[y/x], \psi[z/x]$ où $z \notin \{y\} \cup \text{fv}(\Gamma, \forall x.\psi)$. Par le lemme 17.7, on a aussi une dérivation π_1'' de $\vdash \Gamma[y/z], \psi[y/x][y/z], \psi[z/x][y/z]$. Comme $z \notin \{y\} \cup \text{fv}(\Gamma, \forall x.\psi)$, par les équations (26) et (27), ce dernier séquent est α-congruent au séquent $\vdash \Gamma, \psi[y/x], \psi[y/x]$ et donc dérivable par le lemme 17.6 d'α-congruence syntaxique. Comme $(p(\psi[y/x]), p(\pi_1'')) <_{\text{lex}} (p(\forall x.\psi), p(\pi))$, par hypothèse d'induction on obtient une dérivation de $\vdash \Gamma, \psi[y/x]$. Comme $y \notin \text{fv}(\Gamma, \forall x.\psi)$, on peut appliquer ($\forall$) et on obtient une dérivation de $\vdash \Gamma, \forall x.\psi$.

Cas de ($\exists$) : alors $\varphi = \exists x.\psi$ et π est de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots \\ \vdash \Gamma, \psi[t/x], \exists x.\psi, \exists x.\psi\end{array}}{\vdash \Gamma, \exists x.\psi, \exists x.\psi}\ (\forall)$$

où $t \in T(\mathcal{F}, X)$. Comme $(p(\exists x.\psi), p(\pi_1)) <_{\text{lex}} (p(\exists x.\psi), p(\pi))$, par hypothèse d'induction sur π_1 on obtient une dérivation de $\vdash \Gamma, \psi[t/x], \exists x.\psi$ à laquelle on applique ($\exists$) pour obtenir $\vdash \Gamma, \exists x.\psi$. □

La première démonstration de la complétude d'un système de preuve pour la logique du premier ordre est le théorème de complétude de Gödel, *qui démontre la complétude d'un système de preuve à la Hilbert.*

C.f. (Ebbinghaus, Flum et Thomas, 1994, ch. V) pour la construction de Henkin pour le calcul des séquents, et (David, Nour et Raffalli, 2003, ch. 2) et (Lassaigne et Rougemont, 2004, sec. 4.3) dans le cadre de la déduction naturelle.

C.f. (Fitting, 1996, sec. 3.5 et 5.6) pour la construction de Hintikka

17.3. * **Complétude.** La complétude des systèmes de preuves en logique du premier ordre, c'est-à-dire que toute formule valide est dérivable dans le système de preuve en question, se montre par contraposée : un séquent $\vdash \Gamma$ non prouvable n'est pas valide, ce qui par définition signifie qu'il a un *contre-modèle*, c'est-à-dire une interprétation I et une valuation ρ telles que $I, \rho \not\models \varphi$ pour toute formule $\varphi \in \text{dom}(\Gamma)$.

Ce contre-modèle est habituellement obtenu par la « construction de Henkin ». La construction présentée ici en est une variante, qui repose plutôt sur les travaux de Hintikka sur les tableaux en logique du premier ordre. Nous la voyons ici dans le cas du calcul des séquents monolatère de la figure 39, mais l'approche se généralise aisément à d'autres systèmes de preuve.

17.3.1. *Lemme de Hintikka.* Le principe de la construction est de considérer des ensembles de formules suffisamment « saturés ». Comme l'objectif est de construire un contre-modèle plutôt qu'un modèle, la définition qui suit est duale de la définition usuelle pour les tableaux.

Définition 17.13 (ensemble de Hintikka)**.** Un ensemble E de formules en forme normale négative est *dualement saturé* si

(1) si E contient $\varphi \vee \psi$, alors E contient φ et ψ,
(2) si E contient $\varphi \wedge \psi$, alors E contient φ ou ψ,
(3) si E contient $\exists x.\varphi$, alors E contient $\varphi[t/x]$ pour tout terme $t \in T(\mathcal{F}, X)$, et
(4) si E contient $\forall x.\varphi$, alors E contient $\varphi[y/x]$ pour une variable y.

Un ensemble est *cohérent* s'il ne contient pas une formule atomique α et sa négation $\neg\alpha$. Un *ensemble de Hintikka* est un ensemble dualement saturé cohérent.

Lemme 17.14 (Hintikka)**.** *Si H est un ensemble de Hintikka, alors il existe une interprétation I et une valuation ρ telles que, pour toute formule $\varphi \in H$, $I, \rho \not\models \varphi$.*

On peut remarquer que ce modèle est presque une structure de Herbrand (voir section 14.5), si ce n'est que son domaine est $T(\mathcal{F} \cup X)$ et non $T(\mathcal{F})$.

Démonstration. L'interprétation I a pour domaine $T(\mathcal{F} \cup X)$ l'ensemble des termes clos sur l'alphabet $\mathcal{F} \cup X$, où les variables de X sont considérées comme des constantes d'arité zéro. L'interprétation d'un symbole de fonction $f \in \mathcal{F}_m$ est $f^I(t_1, \ldots, t_m) \stackrel{\text{def}}{=} f(t_1, \ldots, t_m)$. L'interprétation d'un symbole de relation $R \in \mathcal{P}_m$ est $R^I \stackrel{\text{def}}{=} \{(t_1, \ldots, t_m) \mid \neg R(t_1, \ldots, t_m) \in H\}$. La valuation ρ est l'identité sur X.

On vérifie aisément que pour tout terme $t \in T(\mathcal{F}, X)$, $[\![t]\!]^I_\rho = t$. Cela implique que pour une substitution $[t/x]$ où t est vu comme un terme de $T(\mathcal{F}, X)$, la valuation $[t/x]\rho$ n'est autre que $\rho[t/x]$ où t est vu comme un élément du domaine D_I.

Montrons par induction structurelle sur $\varphi \in H$ que $I, \rho \not\models \varphi$.

- Pour un littéral positif $R(t_1, \ldots, t_m) \in H$, comme H est cohérent, $\neg R(t_1, \ldots, t_m) \notin H$ et donc $([\![t_1]\!]^I_\rho, \ldots, [\![t_m]\!]^I_\rho) = (t_1, \ldots, t_m) \notin R^I$: on a bien $I, \rho \not\models R(t_1, \ldots, t_m)$.
- Pour un littéral négatif $\neg R(t_1, \ldots, t_m) \in H$, $([\![t_1]\!]^I_\rho, \ldots, [\![t_m]\!]^I_\rho) = (t_1, \ldots, t_m) \in R^I$: on a bien $I, \rho \not\models \neg R(t_1, \ldots, t_m)$.

(1) Pour une formule $\varphi \vee \psi \in H$, comme H est dualement saturé il contient φ et ψ, donc par hypothèse d'induction $I, \rho \not\models \varphi$ et $I, \rho \not\models \psi$: on a bien $I, \rho \not\models \varphi \vee \psi$.

(2) Pour une formule $\varphi \wedge \psi \in H$, comme H est dualement saturé il contient φ ou ψ, donc par hypothèse d'induction $I, \rho \not\models \varphi$ ou $I, \rho \not\models \psi$: on a bien $I, \rho \not\models \varphi \wedge \psi$.

(3) Pour une formule $\exists x.\varphi \in H$, comme H est dualement saturé il contient $\varphi[t/x]$ pour tout terme t, donc par hypothèse d'induction $I, \rho \not\models \varphi[t/x]$ pour tout terme t et par le lemme 13.5, $I, \rho[t/x] \not\models \varphi$ pour tout $t \in D_I$: on a bien $I, \rho \not\models \exists x.\varphi$.

(4) Pour une formule $\forall x.\varphi \in H$, comme H est dualement saturé il contient $\varphi[y/x]$ pour une variable y, donc par hypothèse d'induction $I, \rho \not\models \varphi[y/x]$ et par le lemme 13.5, $I, \rho[y/x] \not\models \varphi$ où $y \in D_I$: on a bien $I, \rho \not\models \forall x.\varphi$. □

17.3.2. *Théorème de complétude.* Un ensemble E de formules en forme normale négative est *prouvable* s'il existe un multi-ensemble fini Γ avec $\mathrm{dom}(\Gamma) \subseteq E$ tel que $\vdash_{\mathbf{LK}} \Gamma$ par une dérivation sans coupure, et *non prouvable* sinon.

Lemme 17.15 (propagation)**.** *Soit E un ensemble non prouvable :*

(1) si E contient $\varphi \vee \psi$, alors $E \cup \{\varphi, \psi\}$ est non prouvable,

(2) si E contient $\varphi \wedge \psi$, alors $E \cup \{\varphi\}$ est non prouvable ou $E \cup \{\psi\}$ est non prouvable,

(3) si E contient $\exists x.\varphi$, alors $E \cup \{\varphi[t/x]\}$ est non prouvable pour tout terme $t \in T(\mathcal{F}, X)$, et

(4) si E contient $\forall x.\varphi$, alors $E \cup \{\varphi[y/x]\}$ est non prouvable pour toute variable $y \notin \mathrm{fv}(E)$.

*Le lemme de propagation est une forme d'*inversibilité sémantique *du calcul des séquents du premier ordre au sens du lemme 8.12.*

Démonstration. On procède dans chaque cas par l'absurde.

(1) Si $E \cup \{\varphi, \psi\}$ est prouvable mais E n'est pas prouvable, alors nécessairement $\{\varphi, \psi\} \not\subseteq E$ et cette dérivation doit faire intervenir φ ou ψ. Sans perte de généralité, supposons $\vdash_{\mathbf{LK}} \Gamma, \varphi^m, \psi^n$ pour un multi-ensemble fini Γ avec $\mathrm{dom}(\Gamma) \subseteq E$ et $m + n > 0$. Par affaiblissement, on a aussi $\vdash_{\mathbf{LK}} \Gamma, \varphi^{\max(m,n)}, \psi^{\max(m,n)}$. Par $\max(m, n)$ applications de ($\vee$), on a donc $\vdash_{\mathbf{LK}} \Gamma, (\varphi \vee \psi)^{\max(m,n)}$ et E prouvable, contradiction.

(2) Si $E \cup \{\varphi\}$ et $E \cup \{\psi\}$ sont tous les deux prouvables mais E n'est pas prouvable, alors nécessairement $\varphi \notin E$ et $\psi \notin E$ et ces dérivations doivent faire intervenir φ et ψ : $\vdash_{\mathbf{LK}} \Gamma, \varphi^m$ et $\vdash_{\mathbf{LK}} \Delta, \psi^n$ pour des multi-ensemble finis Γ et Δ avec $\mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta) \subseteq E$ et $m, n > 0$. Par affaiblissement et contraction, on a aussi $\vdash_{\mathbf{LK}} \Gamma, \Delta, \varphi$ et $\vdash_{\mathbf{LK}} \Gamma, \Delta, \psi$. Par une application de ($\wedge$), on a donc $\vdash_{\mathbf{LK}} \Gamma, \Delta, \varphi \wedge \psi$ et E prouvable, contradiction.

(3) Si $E \cup \{\varphi[t/x]\}$ est prouvable pour un terme t mais E n'est pas prouvable, alors $\varphi[t/x] \notin E$ et on a $\vdash_{\mathbf{LK}} \Gamma, (\varphi[t/x])^m$ pour un multi-ensemble fini Γ avec $\mathrm{dom}(\Gamma) \subseteq E$ et $m > 0$. Par affaiblissement, $\vdash_{\mathbf{LK}} \Gamma, (\varphi[t/x])^m, (\exists x.\varphi)^m$. Par m applications de ($\exists$), on a donc $\vdash_{\mathbf{LK}} \Gamma, (\exists x.\varphi)^m$ et E prouvable, contradiction.

(4) Si $E \cup \{\varphi[y/x]\}$ est prouvable pour une variable $y \notin \mathrm{fv}(E)$, alors $\varphi[y/x] \notin E$ et $\vdash_{\mathbf{LK}} \Gamma, (\varphi[y/x])^m$ pour un multi-ensemble fini Γ avec $\mathrm{dom}(\Gamma) \subseteq E$ et $m > 0$. Par contraction, on a aussi $\vdash_{\mathbf{LK}} \Gamma, \varphi[y/x]$, et comme $\mathrm{dom}(\Gamma) \subseteq E$ et $\forall x.\varphi \in E$, $y \notin \mathrm{fv}(\Gamma, \forall x.\varphi)$, donc on peut appliquer ($\forall$) pour dériver $\vdash_{\mathbf{LK}} \Gamma, \forall x.\varphi$: E est prouvable, contradiction. □

Le cas (4) *est le seul où l'admissibilité de la contraction soit vraiment utile.*

Le cœur de la preuve du théorème de complétude est le lemme suivant :

Lemme 17.16 (saturation)**.** *Tout ensemble fini non prouvable est inclus dans un ensemble de* Hintikka*.*

La preuve du lemme de saturation correspond intuitivement à la construction d'une branche infinie d'une recherche de preuve pour un séquent non prouvable.

Démonstration. Appelons une *tâche* une formule qui n'est pas de la forme $\exists x.\varphi$ ou une paire $(\exists x.\varphi, t)$ où $t \in T(\mathcal{F}, X)$. Comme il n'y a qu'un nombre dénombrable de formules φ et de termes $t \in T(\mathcal{F}, X)$, il y a un nombre dénombrable de tâches. Fixons une énumération $\theta_0, \theta_1, \ldots$ de toutes les tâches, telle que chaque tâche apparaisse infiniment souvent.

Soit E_0 un ensemble fini non prouvable. On construit une séquence croissante d'ensembles finis non prouvables $E_0 \subseteq E_1 \subseteq \cdots$. Étant donné E_n, on construit E_{n+1} en utilisant θ_n la $n$ième tâche.

(1) Si $\theta_n = \varphi \vee \psi$: si $\varphi \vee \psi \in E_n$, alors par le lemme 17.15.(1), $E_{n+1} \stackrel{\text{def}}{=} E_n \cup \{\varphi, \psi\}$ est non prouvable ; sinon on pose $E_{n+1} \stackrel{\text{def}}{=} E_n$.

(2) Si $\theta_n = \varphi \wedge \psi$: si $\varphi \wedge \psi \in E_n$, alors par le lemme 17.15.(2), $E_n \cup \{\varphi\}$ ou $E_n \cup \{\psi\}$ est non prouvable et on pose $E_{n+1} \stackrel{\text{def}}{=} E_n \cup \{\varphi\}$ dans le premier cas et $E_{n+1} \stackrel{\text{def}}{=} E_n \cup \{\psi\}$ dans le second ; sinon on pose $E_{n+1} \stackrel{\text{def}}{=} E_n$.

(3) Si $\theta_n = (\exists x.\varphi, t)$: si $\exists x.\varphi \in E_n$, alors par le lemme 17.15.(3), $E_{n+1} \stackrel{\text{def}}{=} E_n \cup \{\varphi[t/x]\}$ est non prouvable ; sinon on pose $E_{n+1} \stackrel{\text{def}}{=} E_n$.

(4) Si $\theta_n = \forall x.\varphi$: si $\forall x.\varphi \in E_n$, alors il existe $y \notin \text{fv}(E_n)$ puisque ce dernier est fini, et par le lemme 17.15.(4) $E_{n+1} \stackrel{\text{def}}{=} E_n \cup \{\varphi[y/x]\}$ est non prouvable ; sinon on pose $E_{n+1} \stackrel{\text{def}}{=} E_n$.

La restriction à un ensemble initial fini est importante pour ce cas (4).

Définissons $H \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} E_n$. Il reste à montrer que H est dualement saturé et cohérent. Commençons par vérifier qu'il est dualement saturé :

(1) Si H contient $\varphi \vee \psi$, alors il existe $m \in \mathbb{N}$ tel que $\varphi \vee \psi \in E_m$ et $n > m$ tel que $\theta_n = \varphi \vee \psi$, donc E_{n+1} et donc H contiennent φ et ψ.

(2) Si H contient $\varphi \wedge \psi$, alors il existe $m \in \mathbb{N}$ tel que $\varphi \wedge \psi \in E_m$ et $n > m$ tel que $\theta_n = \varphi \wedge \psi$, donc E_{n+1} et donc H contiennent φ ou ψ.

(3) Si H contient $\exists x.\varphi$ et $t \in T(\mathcal{F}, X)$, alors il existe $m \in \mathbb{N}$ tel que $\exists x.\varphi \in E_m$ et $n > m$ tel que $\theta_n = (\exists x.\varphi, t)$, donc E_{n+1} et donc H contiennent $\varphi[t/x]$.

(4) Si H contient $\forall x.\varphi$, alors il existe $m \in \mathbb{N}$ tel que $\forall x.\varphi \in E_m$ et $n > m$ tel que $\theta_n = \forall x.\varphi$, donc il existe $y \in X$ telle que E_{n+1} et donc H contiennent $\varphi[y/x]$.

L'ensemble H est de plus cohérent : si α et $\neg\alpha$ appartenaient à H, alors il existerait $n \in \mathbb{N}$ tel que $\{\alpha, \neg\alpha\} \subseteq E_n$, mais alors E_n serait prouvable par une application de la règle d'axiome. □

Théorème 17.17 (complétude). *Si* $\models \Gamma$, *alors* $\vdash_{\mathbf{LK}} \Gamma$ *par une dérivation sans coupure.*

Démonstration. Par contraposée, supposons que le séquent $\vdash \Gamma$ ne soit pas dérivable sans coupure. Alors l'ensemble $\text{dom}(\Gamma)$ n'est pas prouvable : si Δ est un multi-ensemble fini tel que $\text{dom}(\Delta) \subseteq \text{dom}(\Gamma)$, alors $\vdash \Delta$ n'est pas dérivable sans coupure, sans quoi on pourrait aussi dériver $\vdash \Gamma$ par affaiblissement et contraction.

Par le lemme 17.16 de saturation, $\text{dom}(\Gamma)$ est inclus dans H un ensemble de Hintikka. Par le lemme 17.14 de Hintikka, il existe une interprétation I et une valuation ρ telles que, pour toute formule φ de H (et donc en particulier pour toute formule φ de $\text{dom}(\Gamma)$), $I, \rho \not\models \varphi$: on a bien $\not\models \Gamma$. □

(David, Nour et Raffalli, 2003, sec. 5.4). Le théorème d'élimination des coupures est originellement dû à Gentzen.

17.4. * **Élimination des coupures.** Les théorèmes 17.5 et 17.17 montrent que la règle de coupure est inutile dans le calcul des séquents : si $\vdash_{\mathbf{LK}} \Gamma$ en utilisant (cut), alors $\models \Gamma$ par le théorème de correction, et donc $\vdash_{\mathbf{LK}} \Gamma$ par une dérivation sans coupure par le théorème de complétude.

On peut cependant faire une preuve directe de ce résultat, sans faireappel à des notions sémantiques. La stratégie générale est de réécrire les dérivations du calcul des séquents avec coupure en des dérivations sans coupure, en faisant « remonter » les instances de (cut) vers les axiomes. L'intérêt de cette preuve syntaxique est qu'étant donné une preuve avec coupure, elle *construit* une preuve sans coupure. La difficulté principale est de montrer que ce processus termine.

Une *coupure maximale* est une dérivation π finissant par une règle (cut) de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \vdash \Gamma, \varphi\end{array} \qquad \begin{array}{c}\pi_2\\ \vdots\\ \vdash \overline{\varphi}, \Delta\end{array}}{\vdash \Gamma, \Delta}\ (\text{cut})$$

où π_1 et π_2 sont sans coupure. Son *rang de coupure* est $r(\pi) \stackrel{\text{def}}{=} (p(\varphi), p(\pi_1)+p(\pi_2))$. On ordonne les rangs de coupure dans $\mathbb{N}^2$ lexicographiquement ; il s'agit d'un ordre bien fondé.

Lemme 17.18. *Soit π une coupure maximale. Alors il existe une dérivation π' sans coupure du même séquent.*

Démonstration. On procède par induction bien fondée sur $r(\pi)$. Rappelons qu'une telle dérivation π est de la forme

$$\dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \vdash \Gamma, \varphi\end{array} \qquad \begin{array}{c}\pi_2\\ \vdots\\ \vdash \overline{\varphi}, \Delta\end{array}}{\vdash \Gamma, \Delta}\ (\text{cut})$$

où π_1 et π_2 sont sans coupure.

On procède à une première analyse par cas, selon que φ soit principale dans la dernière règle de π_1 ou $\overline{\varphi}$ dans la dernière règle de π_2.

cas « np1 » : si φ n'est pas principale dans la dernière règle (R) de π_1. Alors π_1 était de la forme

$$\dfrac{\begin{array}{c}\pi'_1\\ \vdots\\ \vdash \Gamma_1, \varphi\end{array} \qquad \cdots \qquad \begin{array}{c}\pi'_k\\ \vdots\\ \vdash \Gamma_k, \varphi\end{array}}{\vdash \Gamma, \varphi}\ (\text{R})$$

pour un certain $0 \leq k \leq 2$, où $k = 0$ correspond au cas de la règle (ax). Par inspection des règles (R) du calcul des séquents, comme φ n'est pas principale, elle apparaît en effet dans toutes les prémisses de (R). Encore par inspection des règles, on observe que l'on peut transformer π en

$$\dfrac{\dfrac{\begin{array}{c}\pi'_1\\ \vdots\\ \vdash \Gamma_1, \varphi\end{array} \qquad \begin{array}{c}\pi_2\\ \vdots\\ \vdash \overline{\varphi}, \Delta\end{array}}{\vdash \Gamma_1, \Delta}\ (\text{cut}) \qquad \cdots \qquad \dfrac{\begin{array}{c}\pi'_k\\ \vdots\\ \vdash \Gamma_k, \varphi\end{array} \qquad \begin{array}{c}\pi_2\\ \vdots\\ \vdash \overline{\varphi}, \Delta\end{array}}{\vdash \Gamma_k, \Delta}\ (\text{cut})}{\vdash \Gamma, \Delta}\ (\text{R})$$

Les k nouvelles instances de (cut) entre π'_i et π_2 sont maximales et de rangs respectifs $(p(\varphi), p(\pi'_i) + p(\pi_2))$ pour $1 \leq i \leq k$, où $p(\pi'_i) < p(\pi_1)$. On peut donc appliquer l'hypothèse d'induction pour obtenir des dérivations sans coupure de chaque $\vdash \Gamma_i, \Delta$. La dérivation résultante de $\vdash \Gamma, \Delta$ est sans coupure.

cas « np2 » : si $\overline{\varphi}$ n'est pas principale dans la dernière règle de π_2. Similaire au cas précédent.

On peut donc supposer maintenant que φ est principale dans la dernière règle de π_1 et $\overline{\varphi}$ dans la dernière de π_2. Nous faisons une nouvelle distinction de cas selon que ces dernières règles soient (ax) ou non.

cas « ax1 » : si φ est principale dans π_1 finissant par (ax). Alors $\varphi = \ell$, $\Gamma = \Gamma', \overline{\ell}$ et π était de la forme

$$\dfrac{\dfrac{}{\vdash \Gamma', \ell, \overline{\ell}}\ (\text{ax}) \qquad \begin{array}{c}\pi_2\\ \vdots\\ \vdash \overline{\ell}, \Delta\end{array}}{\vdash \Gamma', \overline{\ell}, \Delta}\ (\text{cut})$$

Par le lemme 17.8, on aurait pu obtenir le même résultat par affaiblissement de π_2. On obtient alors directement une preuve sans coupure de $\vdash \Gamma, \Delta$.

cas « ax2 » : si $\overline{\varphi}$ est principale dans π_2 finissant par (ax). Similaire au cas précédent.

Il reste maintenant le cas où π_1 et π_2 ont pour dernières règles $(\vee)$ et $(\wedge)$, ou $(\exists)$ et $(\forall)$: ce sont les seules paires possibles puisque φ et $\overline{\varphi}$ sont principales et que (ax) a déjà été traité.

cas « $\vee\wedge$ » : $\varphi = \varphi' \vee \psi$ est principale dans la dernière règle $(\vee)$ de π_1 et $\overline{\varphi} = \overline{\varphi'} \wedge \overline{\psi}$ dans la dernière règle $(\wedge)$ de π_2. La dérivation π est donc de la forme

$$\dfrac{\dfrac{\dfrac{\pi_1'}{\vdash \Gamma, \varphi', \psi}}{\vdash \Gamma, \varphi' \vee \psi}(\vee) \qquad \dfrac{\dfrac{\pi_2'}{\vdash \overline{\varphi'}, \Delta} \quad \dfrac{\pi_3'}{\vdash \overline{\psi}, \Delta}}{\vdash \overline{\varphi'} \wedge \overline{\psi}, \Delta}(\wedge)}{\vdash \Gamma, \Delta}(\text{cut})$$

On transforme cette dérivation en

$$\dfrac{\dfrac{\dfrac{\pi_1'}{\vdash \Gamma, \varphi', \psi} \quad \dfrac{\pi_2'}{\vdash \overline{\varphi'}, \Delta}}{\vdash \Gamma, \psi, \Delta}(\text{cut}) \qquad \dfrac{\pi_3'}{\vdash \overline{\psi}, \Delta}}{\vdash \Gamma, \Delta, \Delta}(\text{cut})$$

La coupure entre π_1' et π_2' est maximale et de rang $(p(\varphi'), p(\pi_1')+p(\pi_2'))$ où $p(\varphi') < p(\varphi)$; on peut donc appliquer l'hypothèse d'induction pour obtenir une dérivation π' sans coupure de $\vdash \Gamma, \psi, \Delta$. La dérivation résultante est maintenant

$$\dfrac{\dfrac{\pi'}{\vdash \Gamma, \psi, \Delta} \qquad \dfrac{\pi_3'}{\vdash \overline{\psi}, \Delta}}{\vdash \Gamma, \Delta, \Delta}(\text{cut})$$

La coupure y est maximale et de rang $(p(\psi), p(\pi')+p(\pi_3'))$ où $p(\psi) < p(\varphi)$; on peut encore appliquer l'hypothèse d'induction pour obtenir une dérivation sans coupure de $\vdash \Gamma, \Delta, \Delta$. Enfin, par le lemme 17.12 de contraction, puisque $\vdash \Gamma, \Delta, \Delta$ est dérivable sans coupure, $\vdash \Gamma, \Delta$ l'est aussi.

cas « $\wedge\vee$ » : $\varphi = \psi \wedge \psi'$ est principale dans la dernière règle $(\wedge)$ de π_1 et $\overline{\varphi} = \overline{\psi} \vee \overline{\psi'}$ dans la dernière règle $(\vee)$ de π_2. Similaire au cas précédent.

cas « $\exists\forall$ » : $\varphi = \exists x.\psi$ est principale dans la dernière règle $(\exists)$ de π_1 et $\overline{\varphi} = \forall x.\overline{\psi}$ dans la dernière règle $(\forall)$ de π_2. La dérivation π est donc de la forme

$$\dfrac{\dfrac{\dfrac{\pi_1'}{\vdash \Gamma, \psi[t/x], \exists x.\psi}}{\vdash \Gamma, \exists x.\psi}(\exists) \qquad \dfrac{\dfrac{\pi_2'}{\vdash (\overline{\psi})[y/x], \Delta}}{\vdash \forall x.\overline{\psi}, \Delta}(\forall)}{\vdash \Gamma, \Delta}(\text{cut})$$

où $y \notin \text{fv}(\forall x.\overline{\psi}, \Delta)$. Par le lemme 17.7, il existe une dérivation sans coupure π_2'' du séquent $\vdash (\overline{\psi})[y/x][t/y], \Delta[t/y]$. Comme $y \notin \text{fv}(\forall x.\overline{\psi})$, par l'équation (27), $(\overline{\psi})[y/x][t/y] =_\alpha (\overline{\psi})[t/x]$ et comme $y \notin \text{fv}(\Delta)$, par l'équation (26), $\Delta[t/y] =_\alpha \Delta$. De plus, par la propriété 14.4, $(\overline{\psi})[t/x] = \overline{(\psi[t/x])}$. On obtient donc une nouvelle dérivation par le lemme 17.6 d'α-congruence syntaxique

$$\dfrac{\dfrac{\begin{matrix}\pi_1' \\ \vdots \\ \vdash \Gamma, \psi[t/x], \exists x.\psi\end{matrix} \qquad \begin{matrix}\pi_2 \\ \vdots \\ \vdash \forall x.\overline{\psi}, \Delta\end{matrix}}{\vdash \Gamma, \psi[t/x], \Delta}\,(\text{cut}) \qquad \begin{matrix}\pi_2'' \\ \vdots \\ \vdash \overline{(\psi[t/x])}, \Delta\end{matrix}}{\vdash \Gamma, \Delta, \Delta}\,(\text{cut})$$

La coupure entre π_1' et π_2 est maximale et de rang $(p(\varphi), p(\pi_1')+p(\pi_2))$ où $p(\pi_1') < p(\pi_1)$; on peut donc appliquer l'hypothèse d'induction pour obtenir une dérivation π' sans coupure de $\vdash \Gamma, \psi[t/x], \Delta$. La dérivation résultante est

$$\dfrac{\begin{matrix}\pi' \\ \vdots \\ \vdash \Gamma, \psi[t/x], \Delta\end{matrix} \qquad \begin{matrix}\pi_2'' \\ \vdots \\ \vdash \overline{(\psi[t/x])}, \Delta\end{matrix}}{\vdash \Gamma, \Delta, \Delta}\,(\text{cut})$$

La coupure y est maximale et de rang $(p(\psi[t/x]), p(\pi') + p(\pi_2''))$ où $p(\psi[t/x]) = p(\psi) < p(\varphi)$; on peut encore appliquer l'hypothèse d'induction pour obtenir une dérivation sans coupure de $\vdash \Gamma, \Delta, \Delta$. Enfin, par le lemme 17.12 de contraction, $\vdash \Gamma, \Delta$ est dérivable sans coupure.

cas « ∀∃ » : $\varphi = \forall x.\psi$ est principale dans la dernière règle (∀) de π_1 et $\overline{\varphi} = \exists x.\overline{\psi}$ dans la dernière règle (∃) de π_2. Similaire au cas précédent. □

Théorème 17.19 (élimination des coupures)**.** *Soit π une dérivation du calcul des séquent. Alors il existe une dérivation sans coupure du même séquent.*

Démonstration. On procède par récurrence sur le nombre n d'instances de la règle (cut) dans π. Pour le cas de base $n = 0$, π est déjà une dérivation sans coupure. Supposons donc $n > 0$. Il existe donc une coupure maximale π' qui est une sous-dérivation de π. Par le lemme 17.18, il existe une dérivation sans coupure équivalente à π' ; on remplace π' par celle-ci dans π pour obtenir une dérivation équivalente à π avec $n-1$ instances de (cut). Par hypothèse de récurrence, cette nouvelle dérivation a une dérivation sans coupure équivalente. □

L'élimination des coupures est souvent combinée avec la propriété de sous-formule : pour une formule φ en forme normale négative, son ensemble de *sous-formules* $\text{sub}(\varphi)$ est défini inductivement par

$$\text{sub}(\ell) \stackrel{\text{def}}{=} \{\ell\} ,$$

$$\text{sub}(\varphi \vee \psi) \stackrel{\text{def}}{=} \{\varphi \vee \psi\} \cup \text{sub}(\varphi) \cup \text{sub}(\psi) , \qquad \text{sub}(\varphi \wedge \psi) \stackrel{\text{def}}{=} \{\varphi \wedge \psi\} \cup \text{sub}(\varphi) \cup \text{sub}(\psi) ,$$

$$\text{sub}(\exists x.\varphi) \stackrel{\text{def}}{=} \{\exists x.\varphi\} \cup \{\varphi[t/x] \mid t \in T(\mathcal{F}, X)\}, \; \text{sub}(\forall x.\varphi) \stackrel{\text{def}}{=} \{\forall x.\varphi\} \cup \{\varphi[y/x] \mid y \in X\} .$$

Propriété 17.20 (sous-formule)**.** *Dans toutes les règles du calcul des séquents sauf* (cut)*, toutes les formules apparaissant dans les prémisses sont des sous-formules de formules apparaissant dans la conclusion.*

Références

Quelques ouvrages sur la logique.

BEN-ARI, Mordechai (2012). *Mathematical Logic for Computer Science.* 3e édition. Springer. DOI : 10.1007/978-1-4471-4129-7.

BÖRGER, Egon, Erich GRÄDEL et Yuri Sh. GUREVICH (1997). *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer-Verlag.

CHANG, Chen C. et Howard J. KEISLER (1990). *Model Theory.* 3e édition. Studies in Logic and the Foundations of Mathematics 73. North Holland. DOI : 10.1016/S0049-237X(08)70077-6.

CHANG, Chin-Liang et Richard Char-Tung LEE (1973). *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics. Academic Press. DOI : 10.1016/B978-0-08-091728-3.50001-1.

CONCHON, Sylvain et Laurent SIMON (2018). « Satisfaisabilité propositionnelle (SAT) et modulo théories (SMT) ». Dans : *Informatique Mathématique. Une photographie en 2018.* Sous la dir. d'Emmanuel JEANDEL et Laurent VIGNERON. CNRS Éditions. Chap. 2, pp. 43-86. URL : https://ejcim2018.sciencesconf.org/data/pages/ejcim2018-2.pdf.

CORI, René et Daniel LASCAR (2003). *Logique mathématique.* 2e édition. Volume II. Fonctions récursives, théorème de GÖDEL, théorie des ensembles, théorie des modèles. Dunod.

DAVID, René, Karim NOUR et Christophe RAFFALLI (2003). *Introduction à la logique.* 2e édition. Dunod.

DUPARC, Jacques (2015). *La logique pas à pas.* Presses polytechniques et universitaires romandes.

EBBINGHAUS, Heinz-Dieter, Jörg FLUM et Wolfgang THOMAS (1994). *Mathematical Logic.* 2e édition. Undergraduate Texts in Mathematics. Springer. DOI : 10.1007/978-1-4757-2355-7.

FITTING, Melvin (1996). *First-Order Logic and Automated Theorem Proving.* 2e édition. Graduate Texts in Computer Science. Springer. DOI : 10.1007/978-1-4612-2360-3.

GOUBAULT-LARRECQ, Jean et Ian MACKIE (1997). *Proof Theory and Automated Deduction.* Applied Logic Series 6. Kluwer Academic Publishers.

HARRISSON, John (2009). *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press.

KNUTH, Donald E. (2008). *The Art of Computer Programming.* Volume 4, fascicule 0 : *Introduction to Combinatorial Algorithms and Boolean Functions.* Addison-Wesley. URL : https://cs.stanford.edu/~knuth/fasc0b.ps.gz.

——— (2015). *The Art of Computer Programming.* Volume 4, fascicule 6 : *Satisfiability.* Addison-Wesley. URL : https://cs.stanford.edu/~knuth/fasc6a.ps.gz.

LASSAIGNE, Richard et Michel de ROUGEMONT (2004). *Logic and Complexity.* Discrete Mathematics and Theoretical Computer Science. Springer. DOI : 10.1007/978-0-85729-392-3.

LIBKIN, Leonid (2004). *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. Springer. DOI : 10.1007/978-3-662-07003-1.

Quelques textes fondateurs.

BERGER, Robert (1966). *The Undecidability of the Domino Problem.* Memoirs of the American Mathematical Society 66. American Mathematical Society. DOI : 10.1090/memo/0066.

CHURCH, Alonzo (1936a). « A note on the Entscheidungsproblem ». Dans : *Journal of Symbolic Logic* 1(1), pp. 40-41. DOI : 10.2307/2269326. Correction en 1936 dans : *Journal of Symbolic Logic* 1(3), pp. 101–102. DOI : 10.2307/2269030.

——— (1936b). « An unsolvable problem of elementary number theory ». Dans : *American Journal of Mathematics* 58(2), pp. 345-363. DOI : 10.2307/2371045.

COOK, Stephen A. (1971). « The complexity of theorem proving procedures ». Dans : *Actes de STOC '71.* ACM, pp. 151-158. DOI : 10.1145/800157.805047.

Cook, Stephen A. et Robert A. Reckhow (1979). « The relative efficiency of propositional proof systems ». Dans : *Journal of Symbolic Logic* 44(1), pp. 36-50. doi : 10.2307/2273702.

Craig, William (1957). « Three uses of the Herbrand-Gentzen Theorem in relating model theory and proof theory ». Dans : *Journal of Symbolic Logic* 22(3), pp. 269-285. doi : 10.2307/2963594.

Davis, Martin, George Logemann et Donald Loveland (1961). « A machine program for theorem proving ». Dans : *Communications of the ACM* 5(7), pp. 394-397. doi : 10.1145/368273.368557.

Davis, Martin et Hilary Putnam (1960). « A computing procedure for quantification theory ». Dans : *Journal of the ACM* 7(3), pp. 201-215. doi : 10.1145/321033.321034.

Gentzen, Gerhard (1935). « Untersuchungen über das logische Schließen. I ». Dans : *Mathematische Zeitschrift* 39, pp. 176-210. doi : 10.1007/BF01201353. Suivi par « Untersuchungen über das logische Schließen. II », pp. 405–431. doi : 10.1007/BF01201363.

Gödel, Kurt (1929). « Über die Vollständigkeit des Logikkalküls ». Thèse de doctorat. Universität Wien.

——— (1931). « Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I ». Dans : *Monatshefte für Mathematik und Physik* 38, pp. 173-198. doi : 10.1007/BF01700692.

Henkin, Leon (1949). « The completeness of the first-order functional calculus ». Dans : *Journal of Symbolic Logic* 14(3), pp. 159-166. doi : 10.2307/2267044.

Herbrand, Jacques (1930). « Recherches sur la théorie de la démonstration ». Dans : *Travaux de la société des Sciences et des Lettres de Varsovie. Classe III, Sciences Mathématiques et Physiques* 33.

Horn, Alfred (1951). « On sentences which are true of direct unions of algebras ». Dans : *Journal of Symbolic Logic* 16(1), pp. 14-21. doi : 10.2307/2268661.

Levin, Leonid (1973). « Universal sequential search problems ». Dans : *Problems of Information Transmission* 9(3), pp. 115-116. url : http://mi.mathnet.ru/eng/ppi914.

Matiassevitch, Iouri V. (1970). « Les ensembles énumerables sont diophantiens ». Dans : *Comptes rendus de l'Académie des sciences de l'URSS* 191(2). En russe orginal : Матиясевич, Юрий В. (1970). « Диофантовость перечислимых множеств ». Dans : *Доклады Академии наук СССР* 191(2), pp. 279-282. url : http://mi.mathnet.ru/dan35274.

Mostowski, Andrzej, Raphael M. Robinson et Alfred Tarski (1953). « Undecidability and Essential Undecidability in Arithmetic ». Dans : *Undecidable Theories*. Sous la dir. d'Alfred Tarski. Studies in Logic and the Foundations of Mathematics 13. North-Holland. Chap. II, pp. 37-73. doi : 10.1016/S0049-237X(09)70293-9.

Robinson, J. Alan (1965). « A machine-oriented logic based on the resolution principle ». Dans : *Journal of the ACM* 12(1), pp. 23-41. doi : 10.1145/321250.321253.

Rosser, J. Barkley (1936). « Extensions of some theorems of Gödel and Church ». Dans : *Journal of Symbolic Logic* 1(3), pp. 87-91. doi : 10.2307/2269028.

Skolem, Thoralf (1934). « Über die Nicht-charakterisierbarkeit der Zahlenreihe mittels endlich oder abzählbar unendlich vieler Aussagen mit ausschliesslich Zahlenvariablen ». Dans : *Fundamenta Mathematicae* 23, pp. 150-161. doi : 10.4064/fm-23-1-150-161.

Tseitin, Grigori S. (1966). « On the complexity of derivation in propositional calculus ». Dans : *Actes du Séminaire de Leningrad sur la logique mathématique*. url : http://www.decision-procedures.org/handouts/Tseitin70.pdf.

Turing, Alan M. (1937). « On computable numbers, with an application to the Entscheidungsproblem ». Dans : *Proceedings of the London Mathematical Society*. 2e sér. 42(1), pp. 230-265.

DOI : 10.1112/plms/s2-42.1.230. Correction en 1938 dans : *Proceedings of the London Mathematical Society* 2e sér. 43(1), pp. 544–546. DOI : 10.1112/plms/s2-43.6.544.

Autres ouvrages.

ARORA, Sanjeev et Boaz BARAK (2009). *Computational Complexity: A Modern Approach.* Cambridge University Press. DOI : 10.1017/CBO9780511804090.

CARTON, Olivier (2008). *Langages formels, calculabilité et complexité.* Vuibert.

HARRISON, Michael A. (1978). *Introduction to Formal Language Theory.* Addison-Wesley.

KROENING, Daniel et Ofer STRICHMAN (2016). *Decision Procedures. An Algorithmic Point of View.* 2e édition. Texts in Theoretical Computer Science. Springer. DOI : 10.1007/978-3-662-50497-0.

PAPADIMITRIOU, Christos (1993). *Computational Complexity.* Addison-Wesley.

PERIFEL, Sylvain (2014). *Complexité algorithmique.* Ellipses. URL : https://www.irif.fr/~sperifel/complexite.pdf.

SIPPU, Seppo et Eljas SOISALON-SOININEN (1988). *Parsing Theory.* Volume I : Languages and Parsing. Springer-Verlag.

Autres références.

BARRETT, Clark, Pascal FONTAINE et Cesare TINELLI (2017). *The SMT-LIB Standard: Version 2.6.* Rapp. tech. Department of Computer Science, The University of Iowa. URL : http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

DOWLING, William F. et Jean H. GALLIER (1984). « Linear-time algorithms for testing the satisfiability of propositional Horn formulæ ». Dans : *The Journal of Logic Programming* 1(3), pp. 267-284. DOI : 10.1016/0743-1066(84)90014-1.

GOLD, E. Mark (1967). « Language identification in the limit ». Dans : *Information and Control* 10(5), pp. 447-474. DOI : 10.1016/S0019-9958(67)91165-5.

GOLDBERG, Evgueni et Yakov NOVIKOV (2003). « Verification of proofs of unsatisfiability for CNF formulas ». Dans : *Actes de DATE '03*, pp. 886-891.

GUREVICH, Yuri Sh. et I. O. KORYAKOV (1972). « Remarks on Berger's paper on the domino problem ». Dans : *Siberian Mathematics Journal* 13, pp. 319-321. DOI : 10.1007/BF00971620.

MANCINELLI, Fabio, Jaap BOENDER, Roberto DI COSMO, Jérôme VOUILLON, Berke DURAK, Xavier LEROY et Ralf TREINEN (2006). « Managing the complexity of large free and open source package-based software distributions ». Dans : *Actes de ASE '06.* IEEE, pp. 199-208. DOI : 10.1109/ASE.2006.49. URL : https://hal.archives-ouvertes.fr/hal-00149566/.

WETZLER, Nathan, Marijn J. H. HEULE et Warren A. HUNT (2014). « DRAT-trim: Efficient checking and trimming using expressive clausal proofs ». Dans : *Actes de SAT '14.* Lecture Notes in Computer Science 8561. Springer, pp. 422-429. DOI : 10.1007/978-3-319-09284-3_31.

ZHANG, Lintao, C.F. MADIGAN, M.H. MOSKEWICZ et S. MALIK (2001). « Efficient conflict driven learning in a Boolean satisfiability solver ». Dans : *Actes de ICCAD '01*, pp. 279-285. DOI : 10.1109/ICCAD.2001.968634.