Logique propositionnelle classique

Jean Goubault-Larrecq

January 13, 1999

Une logique est une syntaxe (une façon de construire l'ensemble des formules), une sémantique (une description de ce que ces formules signifient), et un système de preuve (qui nous permet de calculer la signification des formules en construisant des preuves). Nous exposons la syntaxe, la sémantique et quelques systèmes de preuve pour la logique propositionnelle classique, une des logiques les plus simples, mais aussi les moins expressives. Ceci nous mènera à des méthodes de démonstration automatique pour cette logique: tableaux, résolution propositionnelle, méthode de Davis-Putnam-Logemann-Loveland method, et diagrammes de décision binaire (BDD). Nous terminons ce chapitre par quelques digressions, que le lecteur pourra sauter en première lecture.

1 Syntaxe

La syntaxe de la logique propositionnelle est fondée sur des *variables de proposition* ou *atomes* que nous notons avec des lettres majuscules prises au début de l'alphabet, A, B, C, etc., possiblement avec des indices ou des primes. Ces lettres sont censées représenter des propriétés de base, qui sont soit vraies soit fausses. Ces variables sont combinées au moyens de *connecteurs logiques* pour former des *formules* ou *propositions*, que nous notons par des lettres grecques majuscules comme Φ, Ψ.

Définition 1 *Soit $\mathcal{X}$ un ensemble infini dit de* variables de proposition. *L'ensemble $\mathcal{F}$ des* formules propositionnelles *est le plus petit contenant toutes les variables, et tel que si Φ et Φ' sont des formules, alors $\Phi \wedge \Phi'$, $\Phi \vee \Phi'$, $\Phi \Rightarrow \Phi'$ et $\neg\Phi$ sont des formules.*

Ceci fait des formules des objets en forme d'arbre, où chaque nœud est appelé une sous-formule. Une sous-formule de Φ est dite *stricte* si elle n'est pas Φ elle-même. En forme textuelle, nous utilisons des parenthèses et des priorités pour éliminer les ambiguïtés: $\neg$ est plus prioritaire que $\wedge$, qui l'est plus que $\vee$, qui l'est plus que $\Rightarrow$. Donc $\Phi \wedge \Phi' \vee \Phi''$ est $(\Phi \wedge \Phi') \vee \Phi''$, $\neg\Phi \vee \Phi'$ représente $(\neg\Phi) \vee \Phi'$, et $\Phi \wedge \Phi' \Rightarrow \Phi'' \vee \Phi'''$ $(\Phi \wedge \Phi') \Rightarrow (\Phi' \vee \Phi''')$, en particulier. De plus, $\wedge$ et $\vee$ sont associatifs à gauche, i.e. $\Phi \wedge \Phi' \wedge \Phi''$ représente $(\Phi \wedge \Phi') \wedge \Phi''$ et $\Phi \vee \Phi' \vee \Phi''$ $(\Phi \vee \Phi') \vee \Phi''$. Et $\Rightarrow$ est associatif à droite, i.e. $\Phi \Rightarrow \Phi' \Rightarrow \Phi''$ représente $\Phi \Rightarrow (\Phi' \Rightarrow \Phi'')$.

Nous définissons aussi des *abréviations* pour d'autres opérations. Par exemple, l'*équivalence logique* $\Leftrightarrow$ est définie par: $\Phi \Leftrightarrow \Phi'$ égale $(\Phi \Rightarrow \Phi') \wedge (\Phi' \Rightarrow \Phi)$.

Nous aurions pu aussi définir moins de connecteurs et faire de ceux qui restent des abréviations. Par exemple, nous aurions pu définir uniquement $\neg$ et $\vee$, et poser $\Phi \wedge \Phi'$ égal à $\neg((\neg\Phi) \vee (\neg\Phi'))$, et $\Phi \Rightarrow \Phi'$ égal à $\neg\Phi \vee \Phi'$. Ceci n'aurait rien changé pour ce qui est de la logique classique, que nous développons ici. Mais cela aurait changé beaucoup de choses en logique intuitionniste, par exemple.

Définition 2 *L'ensemble des* variables libres $\mathrm{fv}(\Phi)$ *d'une formule Φ est défini par récurrence structurelle par :*

- $\mathrm{fv}(A) = \{A\}$ *pour toute variable propositionnelle* A*,*
- $\mathrm{fv}(\Phi \wedge \Phi') = \mathrm{fv}(\Phi \vee \Phi') = \mathrm{fv}(\Phi \Rightarrow \Phi') = \mathrm{fv}(\Phi) \cup \mathrm{fv}(\Phi')$*,*
- $\mathrm{fv}(\neg\Phi) = \mathrm{fv}(\Phi)$*.*

En somme, une variable est libre dans Φ si et seulement si elle apparaît à une feuille de Φ.

Une autre notion utile est :

Définition 3 *Une* substitution σ *est une application des variables propositionnelles vers les formules, telle que* $\sigma(A) = A$ *pour tout* A *sauf un nombre fini.*

Le domaine *de* σ*,* dom σ*, est l'ensemble (fini) de variables* A *telles que* $\sigma(A) \neq A$*. Nous notons* $[\Phi_1/A_1, \ldots, \Phi_n/A_n]$ *la substitution de domaine* $\{A_1, \ldots, A_n\}$ *qui envoie chaque variable* A_i *vers* Φ_i*,* $1 \leq i \leq n$*, où les* A_i *sont supposées distinctes deux à deux. En particulier,* $[]$ *est la* substitution vide*.*

Toute substitution σ *s'étend de façon unique à toutes les formules par :*

- A *a pour image* $\sigma(A)$*,*
- *si* Φ *et* Ψ *ont pour images respectives* Φ' *et* Ψ'*, alors* $\Phi \wedge \Psi$ *a pour image* $\Phi' \wedge \Psi'$*,* $\Phi \vee \Psi$ *a pour image* $\Phi' \vee \Psi'$*,* $\neg\Phi$ *a pour image* $\neg\Phi'$*, et* $\Phi \Rightarrow \Psi$ *a pour image* $\Phi' \Rightarrow \Psi'$*.*

Nous notons $\Phi\sigma$ *l'image de* Φ *par* σ *définie ci-dessus, et nous l'appelons l'*application *de* σ *à* Φ*.*

En somme, on peut voir σ comme un ensemble fini d'opérations de remplacement Φ_i/A_i, remplaçant A_i par Φ_i. $\Phi\sigma$ est la formule Φ où toutes les variables A_i sont remplacées par Φ_i. Ce remplacement doit être effectué *en parallèle*: par exemple, le fait d'appliquer $[B/A, C/B]$ à A donne B, pas C, comme ce serait le cas si nous avions appliqué B/A puis C/B.

Théorème 4 *Soit* $\sigma = [\Phi_1/A_1, \ldots, \Phi_n/A_n]$ *et* $\sigma' = [\Phi'_1/A'_1, \ldots, \Phi'_{n'}/A'_{n'}]$*.*

Il existe une substitution σ'' *unique telle que* $\Phi\sigma'' = (\Phi\sigma)\sigma'$ *pour toute formule* Φ*.*

Preuve : Par définition, si σ'' existe, elle doit envoyer chaque variable A vers $(A\sigma)\sigma'$, c'est-à-dire vers $\sigma(A)\sigma'$. Donc σ'' est unique. Maintenant, soit σ'' la substitution que nous venons de définir, alors $\Phi\sigma''$ égale $(\Phi\sigma)\sigma'$ par une récurrence structurelle facile sur Φ. □

On écrit ce σ'' $\sigma\sigma'$, ou $\sigma' \circ \sigma$. Le théorème nous permet de définir :

Définition 5 La composition de substitutions *est l'opération* $\circ$ *définie par* $\Phi(\sigma' \circ \sigma) = (\Phi\sigma)\sigma'$ *pour tout* Φ*. Nous écrivons aussi* $\sigma\sigma'$ *au lieu de* $\sigma' \circ \sigma$*, de sorte que* $\Phi(\sigma\sigma') = (\Phi\sigma)\sigma'$*.*

La composition de substitutions n'est pas la composition de fonctions ! En effet, l'image de la variable A est $\sigma(A)\sigma'$, pas $\sigma'(\sigma(A))$. En fait, ce dernier n'a aucun sens si $\sigma(A)$ n'est pas une variable.

Théorème 6 *La composition de substitutions est associative, et admet la substitution vide comme élément neutre.*

Preuve : Exercice. □

2 Sémantique

Nous voulons que $\Phi \wedge \Phi'$ soit vraie exactement quand Φ et Φ' sont toutes les deux vraies. Ce disant, nous avons tenté de préciser la *signification* des formules, et c'est le rôle de la sémantique. C'est ici essentiellement que la logique *classique* diffère des autres.

En logique propositionnelle classique, chaque formule est censée être soit vraie soit fausse. Formellement, l'ensemble des *valeurs de vérité* est l'ensemble $\mathbb{B} = \{\top, \bot\}$ des booléens, où $\top \neq \bot$. La signification des connecteurs est définie à l'aide de fonctions des booléens vers les booléens. Ces fonctions sont représentées par des *tables de vérité*. Les voici pour les connecteurs fondamentaux :

$\wedge$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$

$\vee$	$\bot$	$\top$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

$\neg$	
$\bot$	$\top$
$\top$	$\bot$

$\Rightarrow$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$

Par exemple, si Φ est supposé vraie ($\top$), et Φ' est supposé fausse ($\bot$), alors $\Phi \Rightarrow \Phi'$ doit être faux. (Le premier argument est en colonne, le second en ligne dans les tables de vérité.) Ceci définit des fonctions binaires $\overline{\wedge}$, $\overline{\vee}$ et $\overline{\Rightarrow}$ de $\mathbb{B} \times \mathbb{B}$ vers $\mathbb{B}$, et une fonction unaire $\overline{\neg}$ de $\mathbb{B}$ vers $\mathbb{B}$.

La signification d'une formule Φ dépend des valeurs de vérité supposées des variables, et nous définissons formellement :

Définition 7 *Une* affectation *or* interprétation ρ *est une application de l'ensemble* $\mathcal{X}$ *des variables propositionnelles vers* $\mathbb{B}$.

La sémantique $\llbracket \Phi \rrbracket \rho$ *d'une formule* Φ *dans l'affectation* ρ *est définie par récurrence structurelle sur* Φ *par :*

- $\llbracket A \rrbracket \rho = \rho(A)$ *si* A *est une variable propositionnelle.*
- $\llbracket \Phi \wedge \Phi' \rrbracket \rho = \llbracket \Phi \rrbracket \rho \overline{\wedge} \llbracket \Phi' \rrbracket \rho$,
- $\llbracket \Phi \vee \Phi' \rrbracket \rho = \llbracket \Phi \rrbracket \rho \overline{\vee} \llbracket \Phi' \rrbracket \rho$,
- $\llbracket \Phi \Rightarrow \Phi' \rrbracket \rho = \llbracket \Phi \rrbracket \rho \overline{\Rightarrow} \llbracket \Phi' \rrbracket \rho$,
- $\llbracket \neg \Phi \rrbracket \rho = \overline{\neg} \llbracket \Phi \rrbracket \rho$.

Nous dirons qu'une formule Φ est vraie *dans l'affectation* ρ si et seulement si $\llbracket \Phi \rrbracket \rho = \top$, and est fausse *dans* ρ si et seulement si $\llbracket \Phi \rrbracket \rho = \bot$.

Définition 8 *Soit* Φ *une formule propositionnelle, et* ρ *une affectation.*

Nous disons que ρ *est un* modèle *de* Φ*, ou que* ρ satisfait Φ*, et nous écrivons* $\rho \models \Phi$*, si et seulement si* $\llbracket \Phi \rrbracket \rho = \top$.

Nous disons qu'un ensemble Γ *de formules* entraîne Φ*, et nous écrivons* $\Gamma \models \Phi$*, si et seulement si toutes les affectations satisfaisant toutes les formules de* Γ *en même temps (les* modèles *de* Γ*) sont aussi des modèles de* Φ*, c'est-à-dire quand* $\rho \models \Psi$ *pour tout* $\Psi \in \Gamma$ *implique* $\rho \models \Phi$.

Φ *est* valide *si et seulement si* Φ *est vraie dans* toute *affectation (*$\llbracket \Phi \rrbracket \rho = \top$ *pour tout* ρ*, noté* $\models \Phi$*), et est* invalide *sinon. Une formule propositionnelle valide est aussi appelée une* tautologie.

Φ *est* satisfiable *si et seulement si elle est vraie dans* au moins une *affectation (*$\llbracket \Phi \rrbracket \rho = \top$ *pour un certain* ρ*, i.e., elle a un modèle), et est* insatisfiable *sinon.*

Toutes les formules valides sont satisfiables, et toutes les formules insatisfiables sont invalides. Ceci divise l'espace des formules en trois catégories : les valides (toujours vraies), les insatisfiables (toujours fausses), et les formules à la fois invalides et satisfiables (parfois vraies, parfois fausses). Ceci est illustré comme suit :

$$\begin{array}{c}\text{Valide}\\ |\\ \text{Invalide et satisfiable}\\ |\\ \text{Insatisfiable}\end{array}$$

Ensuite, la validité et l'insatisfiabilité se correspondent via négation : Φ est valide si et seulement si $\neg\Phi$ est insatisfiable, Φ est insatisfiable si et seulement si $\neg\Phi$ est valide. Donc la négation $\overline{\neg}$ envoie le diagramme ci-dessus au même, mais retourné. Observer que le point "invalide et satisfiable" reste à la même place.

En un sens, les affectations (au niveau sémantique) jouent le même rôle que les substitutions (au niveau syntaxique) :

Théorème 9 *Pour toute substitution σ et toute affectation ρ, soit $\sigma\rho$ l'affectation qui envoie toute variable A vers $[\![\sigma(A)]\!]\rho$.*

Pour toute formule Φ, pour toute substitution σ et toute affectation ρ, $[\![\Phi\sigma]\!]\rho = [\![\Phi]\!](\sigma\rho)$.

Preuve : Par récurrence structurelle sur Φ : laissé en exercice. □

Corollaire 10 *Soit Φ une formule propositionnelle. Si Φ est valide (resp. insatisfiable), alors $\Phi\sigma$ est valide (resp. insatisfiable) pour toute substitution σ.*

3 Systèmes de Preuve

Maintenant, c'est une chose d'avoir une sémantique pour la logique propositionnelle classique, mais elle est ardue à utiliser pour décider si une formule donnée Φ est valide ou non, satisfiable ou non. Bien qu'il soit possible d'énumérer toutes les affectations (restreintes aux variables libres de Φ), ceci peut demander un temps fabuleux. Le nombre d'affectations différentes à tester est en effet 2^n, où n est le nombre de variables libres de Φ : c'est fini, mais devient rapidement trop large en pratique lorsque n devient grand.

Nous avons donc besoin d'autres façons d'exprimer les formules valides, resp. insatisfiables. L'une des plus intéressantes est d'examiner les preuves bien formées, et de considérer leurs conclusions, que l'on appelle les *théorèmes*. Ce faisant, nous espérons (en supposant pour l'instant que les théorèmes coincident avec les tautologies) que tous, ou au moins une grande proportion des théorèmes ont des preuves courtes, ou du moins des preuves qui soient plus petites qu'une table de vérité. Ceci a deux avantages : en premier, une preuve courte est préférable à une longue liste d'affectations comme explication de la validité de la formule considérée; et si un théorème a une preuve courte, alors elle sera plus facile à trouver, que ce soit à la main ou par une machine, que de tester toutes les affectations possibles.

Le pendant du langage des propositions en matière de preuve est un langage appelé *système de preuve*. Il en existe de nombreux, tous aussi expressifs, et nous en définissons quelques-uns dans cette section.

Une question immédiate vient à l'esprit : y a-t-il un rapport entre tautologies et théorèmes propositionnels ? Dans tous les systèmes de preuve que nous présenterons, les théorèmes seront précisément les tautologies, ce qui était le but recherché. Ceci se décompose en une preuve de *correction* (tous les théorèmes sont valides) et une de *complétude* (toutes les tautologies sont prouvables).

3.1 Systèmes de Hilbert

David Hilbert a été l'un des premiers mathématiciens à s'intéresser à la mécanisation des preuves et de la recherche de théorèmes. Les preuves en format de Hilbert sont des suites de propositions, qui sont des axiomes ou des conséquences de propositions précédentes, déduites par des règles de preuve bien définies.

Définition 11 *Un* système de preuve en format de Hilbert *est un couple* $(\mathcal{A}, \mathcal{R})$*, où* $\mathcal{A}$ *est un ensemble de formules appelées les* axiomes*, et* $\mathcal{R}$ *est un ensemble de* règles d'inférence *ou* règles de preuve*, c'est-à-dire des relations entre ensembles de formules (les* prémices*) et des formules (la conclusion).*

Une dérivation *à partir d'un ensemble* Γ *d'hypothèses est une suite non vide* Φ_1*,* $\Phi_2, \ldots, \Phi_n$*,* $n > 0$*, de formules telles que pour tout* i *entre* 1 *et* n*,* Φ_i *est un axiome* ($\Phi_i \in \mathcal{A}$)*, ou un élément de* Γ*, ou est déduit des formules précédentes dans la suite* ($\Delta\, r\, \Phi_i$*, pour une certaine* $r \in \mathcal{R}$ *et un certain* $\Delta \subseteq \{\Phi_1, \ldots, \Phi_{i-1}\}$)*.*

Une preuve π *d'une formule* Φ *à partir de* Γ *est une dérivation à partir de* Γ *dont la dernière formule est* Φ*. S'il existe une preuve de* Φ *à partir de* Γ*, nous disons que* Γ prouve Φ*, ou que* Φ *est* prouvable *à partir de* Γ*, et nous écrivons* $\Gamma \vdash^{(\mathcal{A},\mathcal{R})} \Phi$*.*

Un théorème *est une formule qui est prouvable à partir de l'ensemble d'hypothèses vide.*

Si Γ et Γ' sont deux ensembles de formules, alors Γ, Γ' représente l'union de Γ et Γ'; si Φ est une formule, Γ, Φ et Φ, Γ représentent $\Gamma \cup \{\Phi\}$; l'ensemble vide est représenté par une espace.

Un exemple classique de système de Hilbert pour la logique propositionnelle classique est le système $\mathcal{SKC}$ suivant. Ses axiomes sont :

(K) $\Phi \Rightarrow \Phi' \Rightarrow \Phi$,

(S) $(\Phi \Rightarrow \Phi' \Rightarrow \Phi'') \Rightarrow (\Phi \Rightarrow \Phi') \Rightarrow (\Phi \Rightarrow \Phi'')$,

(C) $\neg\neg\Phi \Rightarrow \Phi$

pour toutes formules Φ, Φ' et Φ''. Sa seule règle d'inférence est le *modus ponens* :

(MP) de Φ et $\Phi \Rightarrow \Phi'$, deduire Φ', pour toutes Φ, Φ'.

Dans ce système, il est sous-entendu que $\Rightarrow$ et $\mathbf{F}$ sont le seul connecteur et la seule constante logiques, respectivement. $\neg\Phi$ est une abréviation de $\Phi \Rightarrow \mathbf{F}$, $\Phi \wedge \Phi'$ de $\neg(\Phi \Rightarrow \neg\Phi')$, $\Phi \vee \Phi'$ de $\neg\Phi \Rightarrow \Phi'$.

Théorème 12 (Correction) *Si* $\Gamma \vdash^{\mathcal{SKC}} \Phi$*, alors* $\Gamma \models \Phi$*.*

Preuve : Par récurrence sur la longueur d'une preuve de Φ à partir de Γ : exercice. □

Le fait que le système soit correct entraîne qu'il est *cohérent*:

Théorème 13 (Cohérence) *Le système* $\mathcal{SKC}$ *est* absolument cohérent*, c'est-à-dire qu'il existe une formule* Φ *que* $\mathcal{SKC}$ *ne peut pas prouver.*

Le système $\mathcal{SKC}$ *est* cohérent par rapport à la négation*, c'est-à-dire qu'il n'existe aucune formule* Φ *telle que* Φ *et* $\neg\Phi$ *soient toutes deux prouvables.*

Preuve : Exercice. □

Théorème 14 (Complétude) *Si* $\Gamma \models \Phi$*, alors* $\Gamma \vdash^{\mathcal{SKC}} \Phi$*.*

Nous ne le démontrerons pas. Nous disons que $\mathcal{SKC}$ est *complet* par rapport à la sémantique de la logique classique propositionnelle. Il s'agit de la réciproque de la correction. Une conséquence importante (qu'il faut en fait démontrer d'abord pour pouvoir prouver la complétude) est que $\Rightarrow$ est, comme nous nous en doutions, très lié à la notion de déductibilité :

Théorème 15 (Théorème de la déduction) $\Gamma, \Phi \vdash^{\mathcal{SKC}} \Phi'$ *si et seulement si* $\Gamma \vdash^{\mathcal{SKC}} \Phi \Rightarrow \Phi'$.

Preuve : Utiliser la correction et la complétude (exercice). □

Une autre conséquence est la décidabilité de la logique propositionnelle :

Théorème 16 (Decidabilité) *Il existe un algorithme qui, étant donné un ensemble fini* Γ *de formules et une formule* Φ*, décide si* $\Gamma \vdash^{\mathcal{SKC}} \Phi$.

Preuve : Décider $\Gamma \vdash^{\mathcal{SKC}} \Phi$ signifie décider $\Gamma \models \Phi$, par correction et complétude. De plus, nous pouvons supposer que Γ est vide sans perte de généralité, par le théorème de la déduction. Mais $[\![\Phi]\!]\rho$ ne dépend que des valeurs que prend ρ sur les variables libres de Φ, par une récurrence structurelle aisée sur Φ : énumérer les restrictions des affectations à $\mathrm{fv}(\Phi)$ (il n'y en a qu'un nombre fini), et évaluer Φ dans chaque. Ceci prend un temps fini. □

Bien sûr, énumérer toutes les affectations aux variables libres peut prendre beaucoup de temps. En fait, les systèmes de Hilbert ne donnent pas d'explication raisonnable de pourquoi une formule est valide : l'argument ci-dessus ne cherche pas réellement de preuve, et en fait les preuves en $\mathcal{SKC}$ sont extrêmement illisibles.

3.2 Déduction naturelle

Le principal défaut des systèmes de Hilbert est le fait que nous ne pouvons pas poser d'hypothèse auxiliaire. Pour prouver $\Phi \Rightarrow \Phi'$, il n'y a pas de mécanisme nous permettant de poser Φ en hypothèse pour en déduire Φ'. (Quoique le théorème de la déduction nous affirme que ce serait correct de le faire.)

Les systèmes de déduction naturelle corrigent cette situation, et peuvent être présentés sous des formats différents, tous équivalents.

Une première façon de présenter les systèmes de déduction naturelle est de dire que les preuves sont des diagrammes bi-dimensionnels de formules connectées par des règles de preuve. Ces règles, pour la logique classique propositionnelle, sont classées en *règles d'introduction* et *règles d'élimination* (figure 1). Trois petits points verticaux ($\vdots$) abrègent un arbre de dérivation entier. Les règles combinent des arbres de dérivation pour en produire de nouveaux.

Par exemple, si (π) est une dérivation qui prouve Φ, et (π') est une déerivation qui prouve Φ', alors la dérivation suivante prouve $\Phi \wedge \Phi'$:

$$\dfrac{\begin{matrix}(\pi)\\ \vdots \\ \Phi\end{matrix} \qquad \begin{matrix}(\pi')\\ \vdots \\ \Phi'\end{matrix}}{\Phi \wedge \Phi'}\,(\wedge I)$$

$(\wedge I)$ est la règle d'introduction du connecteur de conjonction. De deux preuves de deux formules, elle construit une preuve de la conjonction. De façon symétrique, $(\wedge E_1)$ et $(\wedge E_2)$ prennent une dérivation d'une conjonction, et retournent une des sous-formules sous le connecteur $\wedge$: ces règles éliminent donc un signe $\wedge$ de la formule.

La règle $(\neg E)$ a comme conclusion une formule arbitraire Φ' : de Φ et $\neg\Phi$, une contradiction, nous pouvons en effet déduire n'importe quoi.

Les autres règles s'expliquent d'elles-mêmes, sauf celles qui utilisent la notation $[\ ]_i$: $(\Rightarrow I)$, $(\neg I)$ et $(\vee E)$. Les crochets représentent une hypothèse qui a été *déchargée*. Regardons par exemple la règle $(\Rightarrow I)$. Elle dit que si (π) est une preuve de Φ' utilisant Φ comme hypothèse, soit si :

Introductions	Éliminations
$\dfrac{\begin{matrix}\vdots & & \vdots \\ \Phi & & \Phi'\end{matrix}}{\Phi \wedge \Phi'}\,(\wedge I)$	$\dfrac{\begin{matrix}\vdots \\ \Phi \wedge \Phi'\end{matrix}}{\Phi}\,(\wedge E_1)$ $\dfrac{\begin{matrix}\vdots \\ \Phi \wedge \Phi'\end{matrix}}{\Phi'}\,(\wedge E_2)$
$\dfrac{\begin{matrix}[\Phi]_i \\ \vdots \\ \Phi'\end{matrix}}{\Phi \Rightarrow \Phi'}\,{}_i(\Rightarrow I)$	$\dfrac{\begin{matrix}\vdots & & \vdots \\ \Phi & & \Phi \Rightarrow \Phi'\end{matrix}}{\Phi'}\,(\Rightarrow E)$
$\dfrac{\begin{matrix}\vdots \\ \Phi\end{matrix}}{\Phi \vee \Phi'}\,(\vee I_1)$ $\dfrac{\begin{matrix}\vdots \\ \Phi'\end{matrix}}{\Phi \vee \Phi'}\,(\vee I_2)$	$\dfrac{\begin{matrix} & & [\Phi]_i & & [\Phi']_i \\ \vdots & & \vdots & & \vdots \\ \Phi \vee \Phi' & & \Phi'' & & \Phi''\end{matrix}}{\Phi''}\,{}_i(\vee E)$
$\dfrac{\begin{matrix}[\Phi]_i \\ \vdots \\ \Phi' \wedge \neg\Phi'\end{matrix}}{\neg\Phi}\,{}_i(\neg I)$	$\dfrac{\begin{matrix}\vdots & & \vdots \\ \Phi & & \neg\Phi\end{matrix}}{\Phi'}\,(\neg E)$
$\dfrac{\begin{matrix}\vdots \\ \Phi\end{matrix}}{\neg\neg\Phi}\,(\neg\neg I)$	$\dfrac{\begin{matrix}\vdots \\ \neg\neg\Phi\end{matrix}}{\Phi}\,(\neg\neg E)$

Figure 1: Déduction naturelle

$$\frac{}{\Gamma, \Phi \longrightarrow \Phi}\,(Ax)$$

$$\frac{\Gamma \longrightarrow \Phi \qquad \Gamma \longrightarrow \Phi'}{\Gamma \longrightarrow \Phi \wedge \Phi'}\,(\wedge I) \qquad \frac{\Gamma \longrightarrow \Phi_1 \wedge \Phi_2}{\Gamma \longrightarrow \Phi_i}\,(\wedge E_i) \quad (i = 1, 2)$$

$$\frac{\Gamma, \Phi \longrightarrow \Phi'}{\Gamma \longrightarrow \Phi \Rightarrow \Phi'}\,(\Rightarrow I) \qquad \frac{\Gamma \longrightarrow \Phi \Rightarrow \Phi' \qquad \Gamma \longrightarrow \Phi}{\Gamma \longrightarrow \Phi'}\,(\Rightarrow E)$$

$$\frac{\Gamma \longrightarrow \Phi_i}{\Gamma \longrightarrow \Phi_1 \vee \Phi_2}\,(\vee I_i) \quad (i = 1, 2) \qquad \frac{\Gamma \longrightarrow \Phi \vee \Phi' \qquad \begin{array}{c}\Gamma, \Phi \\ \longrightarrow \Phi''\end{array} \qquad \begin{array}{c}\Gamma, \Phi' \\ \longrightarrow \Phi''\end{array}}{\Gamma \longrightarrow \Phi''}\,(\vee E)$$

$$\frac{\Gamma, \Phi \longrightarrow \mathbf{F}}{\Gamma \longrightarrow \neg\Phi}\,(\neg I) \qquad \frac{\Gamma \longrightarrow \Phi \qquad \Gamma \longrightarrow \neg\Phi}{\Gamma \longrightarrow \Phi'}\,(\neg E)$$

$$\frac{\Gamma \longrightarrow \Phi}{\Gamma \longrightarrow \neg\neg\Phi}\,(\neg\neg I) \qquad \frac{\Gamma \longrightarrow \neg\neg\Phi}{\Gamma \longrightarrow \Phi}\,(\neg\neg E)$$

Figure 2: Déduction naturelle en format de séquents

$$\begin{array}{c} \Phi \\ \vdots \\ (\pi) \\ \vdots \\ \Phi' \end{array}$$

alors nous pouvons déduire de cette preuve une de $\Phi \Rightarrow \Phi'$ sans utiliser Φ comme hypothèse auxiliaire, de sorte que nous pouvons barrer Φ — la décharger — et ajouter une nouvelle ligne sous la preuve. La notation utilisée pour décharger Φ est $[\Phi]_i$, où i est une étiquette unique (disons un entier) dont nous nous servons pour associer l'inférence de $\Phi \Rightarrow \Phi'$ avec les occurrences de Φ que nous déchargeons. Déchargeons Φ ci-dessus :

$$\begin{array}{c} [\Phi]_1 \\ \vdots \\ (\pi) \\ \vdots \\ \Phi' \\ \hline \Phi \Rightarrow \Phi' \end{array}\ {}_1(\Rightarrow I)$$

La format bi-dimensionnel est difficile à étudier métamathématiquement, car l'identification des hypothèses auxiliaires n'est pas claire : dans une dérivation d'une formule, nous devons parcourir toute la dérivation pour chercher les hypothèses non déchargées.

Une présentation plus formelle de la déduction naturelle qui montre à la fois les formules et l'ensemble des hypothèses auxiliaires, est la *déduction naturelle en format de séquents*. Au lieu de dériver juste des formules Φ, nous dérivons des *séquents* $\Gamma \longrightarrow \Phi$, où Γ est un ensemble de formules, appelé le *contexte* du séquent.

Il s'agit précisément de l'ensemble des hypothèses auxiliaires courantes alors que nous cherchons à prouver Φ. Les règles de preuve sont données en figure 2, où $\mathbf{F}$ est n'importe quelle formule insatisfiable, par exemple $\Phi' \wedge \neg\Phi'$. Remarquer à quel point les manipulations de contextes sont moins complexes qu'avec la notation $[\ ]_i$, puisque les hypothèses auxiliaires sont décrites explicitement dans chaque séquent.

Définition 17 *Une* dérivation en déduction naturelle *est un arbre inversé dont les nœuds sont des séquents connectés par des règles de déduction de la figure 2. Les feuilles sont des instances de la règle (Ax) et sont appelées des* axiomes.

Une preuve π *d'un séquent* $\Gamma \longrightarrow \Phi$ *est une dérivation en déduction naturelle dont la racine est décorée par* $\Gamma \longrightarrow \Phi$*. Nous disons aussi que* π *est une preuve de* Φ à partir de Γ*. S'il existe une preuve de* $\Gamma \longrightarrow \Phi$*, nous disons que* $\Gamma \longrightarrow \Phi$ *est* prouvable*, ou bien que* Φ *est prouvable à partir de* Γ*, et nous écrivons* $\vdash^{\mathcal{ND}} \Gamma \longrightarrow \Phi$*, ou* $\Gamma \vdash^{\mathcal{ND}} \Phi$.

Un théorème *est une formule prouvable à partir de l'ensemble d'hypothèses vide.*

Le théorème de la déduction ($\Gamma, \Phi \vdash^{\mathcal{ND}} \Phi'$ si et seulement si $\Gamma \vdash^{\mathcal{ND}} \Phi \Rightarrow \Phi'$) est évident dans ce formalisme. Noter aussi que la règle ($\Rightarrow E$) est précisément la règle (MP) de la section 3.1.

Théorème 18 (Correction) *Si* $\Gamma \vdash^{\mathcal{ND}} \Phi$*, alors* $\Gamma \models \Phi$.

Preuve : Exercice. □

Théorème 19 (Complétude) *Si* $\Gamma \models \Phi$*, alors* $\Gamma \vdash^{\mathcal{ND}} \Phi$.

Comme plus haut, nous ne prouverons pas ce théorème.

Théorème 20 (Decidabilité) *Il existe un algorithme qui, étant donné un ensemble fini* Γ *de formules et une formule* Φ*, décide si* $\Gamma \vdash^{\mathcal{ND}} \Phi$.

Preuve : L'algorithme teste si $\Gamma \models \Phi$, comme au théorème 16. Il est encore une fois difficile trouver une preuve directement. □

3.3 Séquents de Gentzen

Bien que la déduction naturelle soit plus commode que les systèmes de Hilbert pour écrire des preuves, il est encore malaisé de chercher une preuve d'une proposition donnée, à part en testant indirectement si elle est valide. Gerhard Gentzen a inventé ce que nous appelons maintenant des *systèmes de séquents* pour représenter les preuves en logique classique. Nous définissons maintenant la version propositionnelle du système $\mathbf{LK}_0$ de Gentzen.

Définition 21 *Un* séquent de Gentzen *est un couple* Γ, Δ *d'ensembles finis de formules, noté* $\Gamma \longrightarrow \Delta$.

Nous autorisons donc plusieurs formules, ou même aucune formule, à la droite d'un séquent. Ceci rend le calcul symétrique, en particulier, et nous permettra d'éviter les détours qu'il fallait parfois prendre pour prouver certaines formules en déduction naturelle.

En ce qui concerne la sémantique, $\Gamma \longrightarrow \Delta$ signifie que la conjonction des formules de Γ entraîne la *disjonction* des formules de Δ, c'est-à-dire qu'une formule de Δ est vraie dès que toutes les formules de Γ sont vraies.

Alors $\mathbf{LK}_0$ est le système de preuve défini par les règles de la figure 3, où Γ, Γ', Δ et Δ' sont des ensembles finis de formules et Φ, Φ' sont des formules. Nous y trouvons l'axiome Ax, comme en déduction naturelle, des règles à gauche (L)

$$\frac{}{\Gamma, \Phi \longrightarrow \Delta, \Phi}\text{Ax}$$

$$\frac{\Gamma, \Phi, \Phi' \longrightarrow \Delta}{\Gamma, \Phi \wedge \Phi' \longrightarrow \Delta}\wedge\text{L} \qquad \frac{\Gamma \longrightarrow \Delta, \Phi \qquad \Gamma \longrightarrow \Delta, \Phi'}{\Gamma \longrightarrow \Delta, \Phi \wedge \Phi'}\wedge\text{R}$$

$$\frac{\Gamma, \Phi \longrightarrow \Delta \qquad \Gamma, \Phi' \longrightarrow \Delta}{\Gamma, \Phi \vee \Phi' \longrightarrow \Delta}\vee\text{L} \qquad \frac{\Gamma \longrightarrow \Delta, \Phi, \Phi'}{\Gamma \longrightarrow \Delta, \Phi \vee \Phi'}\vee\text{R}$$

$$\frac{\Gamma \longrightarrow \Phi, \Delta \qquad \Gamma, \Phi' \longrightarrow \Delta}{\Gamma, \Phi \Rightarrow \Phi' \longrightarrow \Delta}\Rightarrow\text{L} \qquad \frac{\Gamma, \Phi \longrightarrow \Delta, \Phi'}{\Gamma \longrightarrow \Delta, \Phi \Rightarrow \Phi'}\Rightarrow\text{R}$$

$$\frac{\Gamma \longrightarrow \Delta, \Phi}{\Gamma, \neg\Phi \longrightarrow \Delta}\neg\text{L} \qquad \frac{\Gamma, \Phi \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \neg\Phi}\neg\text{R}$$

$$\frac{\Gamma \longrightarrow \Delta, \Phi \qquad \Gamma', \Phi \longrightarrow \Delta'}{\Gamma, \Gamma' \longrightarrow \Delta, \Delta'}\text{Cut}$$

Figure 3: Le système $\mathbf{LK}_0$ de Gentzen

et à droite (R) pour chaque connecteur, et la règle spéciale de *coupure* Cut. Les coupures sont des applications de lemmes : d'une preuve de Φ (en disjonction avec Δ) dans le contexte Γ, et d'une preuve d'un lemme établissant que si Φ est vraie, alors Δ' est une conséquence de Γ', nous obtenons une preuve de Δ' (en disjonction avec Δ) dans le contexte Γ, Γ'.

Définition 22 *Soit $\Gamma \longrightarrow \Delta$ un séquent de Gentzen, et ρ une affectation. Nous disons que ρ* satisfait *$\Gamma \longrightarrow \Delta$, ce que nous notons $\rho \models \Gamma \longrightarrow \Delta$, si et seulement s'il existe une formule Φ dans Δ telle que $\rho \models \Phi$, ou une formule Φ dans Γ telle que non $\rho \models \Phi$.*

Nous disons que $\Gamma \longrightarrow \Delta$ est prouvable *dans $\mathbf{LK}_0$ si et seulement s'il existe une dérivation dans $\mathbf{LK}_0$ se terminant sur $\Gamma \longrightarrow \Delta$. Nous écrivons alors $\vdash^{\mathbf{LK}_0} \Gamma \longrightarrow \Delta$, ou $\Gamma \vdash^{\mathbf{LK}_0} \Delta$.*

Théorème 23 (Correction) *Si $\vdash^{\mathbf{LK}_0} \Gamma \longrightarrow \Delta$, alors $\models \Gamma \longrightarrow \Delta$.*

Preuve : Exercice. ☐

Théorème 24 (Complétude) *Si $\models \Gamma \longrightarrow \Delta$, alors $\vdash^{\mathbf{LK}_0} \Gamma \longrightarrow \Delta$.*

De plus, si c'est le cas, alors nous pouvons en trouver une preuve dans $\mathbf{LK}_0$ qui n'utilise pas la règle Cut (une preuve sans coupure*).*

Preuve : Soit S l'ensemble des séquents $\Gamma' \longrightarrow \Delta'$ tels que $\Gamma' \cap \Delta' = \emptyset$ et $\Gamma' \cup \Delta' \not\subseteq \mathcal{X}$, et soit f une application associant à chaque séquent $\Gamma' \longrightarrow \Delta'$ dans S une formule non variable dans $\Gamma' + \Delta'$, où $+$ représente la somme directe. On appellera f la *stratégie*.

Construisons l'arbre T dont la racine est $\Gamma \longrightarrow \Delta$, et dont les nœuds sont des séquents, récursivement comme suit. Si $\Gamma' \longrightarrow \Delta'$ n'est pas dans S, nous disons que $\Gamma' \longrightarrow \Delta'$ est une *feuille* de l'arbre. Sinon, soit $\Phi = f(\Gamma' \longrightarrow \Delta')$. Alors les fils de $\Gamma' \longrightarrow \Delta'$ sont les prémices de la règle dont la conclusion est $\Gamma' \longrightarrow \Delta'$, portant sur Φ.

Plus formellement, soit $\Phi \in \Gamma'$ soit $\Phi \in \Delta'$. Si $\Phi \in \Gamma'$, soit $\Gamma'' = \Gamma' \setminus \{\Phi\}$, et définissons les fils de $\Gamma' \longrightarrow \Delta'$ comme étant $\Gamma'' \longrightarrow \Phi', \Delta'$ si Φ est de la forme $\neg\Phi'$; $\Gamma'', \Phi', \Phi'' \longrightarrow \Delta'$ si Φ vaut $\Phi' \wedge \Phi''$; $\Gamma'', \Phi' \longrightarrow \Delta'$ et $\Gamma'', \Phi'' \longrightarrow \Delta'$ si Φ vaut $\Phi' \vee \Phi''$; $\Gamma'' \longrightarrow \Phi', \Delta'$ et $\Gamma'', \Phi'' \longrightarrow \Delta'$ si Φ vaut $\Phi' \Rightarrow \Phi''$. D'autre part, si $\Phi \in \Delta'$, soit $\Delta'' = \Delta' \setminus \{\Phi\}$, et définissons les fils de $\Gamma' \longrightarrow \Delta'$ comme $\Gamma', \Phi' \longrightarrow \Delta''$ si Φ vaut $\neg\Phi'$; $\Gamma' \longrightarrow \Phi', \Delta''$ et $\Gamma' \longrightarrow \Phi'', \Delta''$ si Φ vaut $\Phi' \wedge \Phi''$; $\Gamma' \longrightarrow \Phi', \Phi'', \Delta''$ si Φ vaut $\Phi' \vee \Phi''$; $\Gamma', \Phi' \longrightarrow \Phi'', \Delta''$ si Φ vaut $\Phi' \Rightarrow \Phi''$. Remarquer que cet arbre est presque une preuve sans coupure dans $\mathbf{LK}_0$; en effet, les huit conditions définissant les fils des nœuds autres que des feuilles sont respectivement $\neg$L, $\wedge$L, $\vee$L, $\Rightarrow$L, $\neg$R, $\wedge$R, $\vee$R, $\Rightarrow$R. (Tracer l'arbre la racine en bas pour voir qu'il ressemble réellement à une preuve dans $\mathbf{LK}_0$.) Pour montrer que T est réellement une preuve, nous devons vérifier que T est un arbre *fini*, et que ses feuilles sont prouvables dans $\mathbf{LK}_0$, par la règle Ax.

Montrons que T est un arbre fini. Définissons la *longueur* d'une formule comme suit : la longueur d'une variable propositionnelle est 1, la longueur de $\neg\Phi'$ est un plus la longueur de Φ', la longueur de $\Phi' \circ \Phi''$ est un plus la longueur Φ' plus celle de Φ', pour tout connecteur binaire $\circ$. Par récurrence sur la somme L des longueurs des formules dans Γ, Δ, nous voyons que la profondeur de T ne dépasse pas L, et donc que le nombre de nœuds dans T ne dépasse pas 2^L.

Nous montrons maintenant que les feuilles sont des instances de la règle Ax. Par construction, un nœud qui n'est pas une feuille de T est valide (en tant que séquent) si et seulement si ses fils sont valides. Par récurrence structurelle sur T, nous concluons que toutes ses feuilles sont des séquents valides. Mais si $\Gamma' \longrightarrow \Delta'$ est une feuille, par définition $\Gamma' \cap \Delta' \neq \emptyset$ (donc c'est une instance de Ax), ou bien Γ' et Δ' ne contiennent que des variables propositionnelles. Dans ce dernier cas, si nous avions $\Gamma' \cap \Delta' = \emptyset$, nous pourrions définir une affectation ρ en associant $\top$ à chaque variable de Γ', et $\bot$ à chaque variable de Δ', ce qui contredirait le fait que $\Gamma' \longrightarrow \Delta'$ est valide. Dans tous les cas, les feuilles sont des instances de Ax. Ceci termine la preuve. $\Box$

Ce théorème montre non seulement que $\mathbf{LK}_0$ est complet, mais en plus que nous n'avons pas besoin de la règle Cut : si une proposition est prouvable, on peut aussi la prouver sans aucun lemme intermédiaire.

Une preuve sans coupure peut être beaucoup plus grosse qu'une preuve avec des coupures. L'intérêt principale de l'élimination des coupures en démonstration automatique, cependant, est que l'espace de recherche des preuves sans coupure est *localement plus petit* que l'espace de toutes les preuves : si nous cherchons à construire une preuve sans coupure de bas en haut, à chaque étape nous choisissons une formule non variable arbitraire dans le séquent courant, disons $\Phi \wedge \Phi'$ à gauche, et nous savons immédiatement quelle règle utiliser (ici, $\wedge$L) et quelles formules apparaîtront dans les prémices (Φ et Φ' ici). Ce ne serait pas le cas en présence de Cut, qui doit inventer une formule arbitraire Φ lors de la remontée de la conclusion vers les prémices. Ceci est formalisé par le résultat suivant :

Lemme 25 (Propriété de la sous-formule) *Dans une preuve sans coupure de $\Gamma \longrightarrow \Delta$, tous les séquents sont composés de sous-formules des formules de Γ, Δ uniquement.*

Preuve : Exercice. $\Box$

En fait, nous avons prouvé plus : nous n'avons même pas besoin de *contraction*. Expliquons-nous. Voici une instance légitime de $\wedge$L:

$$\frac{\Gamma, \Phi \wedge \Phi', \Phi, \Phi' \longrightarrow \Delta}{\Gamma, \Phi \wedge \Phi' \longrightarrow \Delta}$$

Il s'agit en effet de la règle de la figure 3, avec $\Gamma, \Phi \wedge \Phi'$ au lieu de Γ, parce que $\Phi \wedge \Phi', \Phi \wedge \Phi'$ n'est rien d'autre que $\Phi \wedge \Phi'$ (l'union ensembliste est idempotente). Mais nous aimerions chercher une preuve sans coupure en remontant à partir du but. Une instance comme ci-dessus de $\wedge$L peut être appliquée infiniment longtemps. Le théorème 24 montre que nous n'avons en fait jamais besoin de telles instances, autrement dit nous pouvons nous restreindre à n'utiliser que le cas particulier suivant de $\wedge$L :

$$\frac{(\Gamma \setminus \{\Phi \wedge \Phi'\}), \Phi, \Phi' \longrightarrow \Delta}{\Gamma, \Phi \wedge \Phi' \longrightarrow \Delta}$$

et de même pour les autres règles.

Une façon plus formelle de traiter ce problème est d'utiliser des séquents $\Gamma \longrightarrow \Delta$, où Γ et Δ ne sont plus des ensembles mais des *multi-ensembles*, c'est-à-dire des ensembles d'éléments comptés avec leurs multiplicités, ou de façon équivalente des listes où l'ordre des éléments n'a aucune importance. Les règles de $\mathbf{LK}_0$ sont alors encore celles de la figure 3, mais où la virgule dénote l'union des multi-ensembles, plus les règles de *contraction* :

$$\frac{\Gamma, \Phi, \Phi \longrightarrow \Delta}{\Gamma, \Phi \longrightarrow \Delta}\,(ContrL) \qquad \frac{\Gamma \longrightarrow \Delta, \Phi, \Phi}{\Gamma \longrightarrow \Delta, \Phi}\,(ContrR)$$

Les arguments du théorème 24 montrent alors que tout séquent valide a une preuve aussi bien sans coupure que sans contraction.

Le fait que nous puissions limiter la recherche aux preuves sans coupure et sans contraction rend l'espace de recherche beaucoup plus petit que l'espace de toutes les preuves : il n'y a qu'un nombre fini de preuves sans coupure ni contraction, alors qu'il y a une infinité de preuves (même sans coupure). En fait, étant donné une stratégie f, il n'y a qu'une preuve sans coupure ni contraction obéissant à la stratégie. La réalisation habituelle de ces algorithmes est connue sous le nom de *méthode des tableaux* (cf. section 4.1).

Le fait que $\mathbf{LK}_0$ est complet même sans la coupure peut être renforcé dans une autre direction :

Théorème 26 (Élimination des coupures) *Il existe un algorithme qui prend une preuve dans* $\mathbf{LK}_0$*, et la transforme en une preuve sans coupure du même séquent.*

Nous ne fournissons pas (encore) la preuve. Ceci dit que traduire une preuve en une preuve sans coupure du même théorème est un processus effectif (on peut le programmer sur une machine). Intuitivement, les coupures nous permettent de réutiliser des lemmes prouvés antérieurement. Si, au lieu d'utiliser ces lemmes, nous rejouons leurs preuves, nous aboutissons à une preuve sans aucun lemme intermédiaire, c'est-à-dire sans coupure. La difficulté est de montrer que ce processus termine. Ceci a des conséquences importantes dans le domaine de la conception de langages de programmation, connues sous le nom de correspondance de Curry-Howard entre preuves et programmes, et entre formules et types.

4 Méthodes de preuve automatique

La question que nous traitons maintenant est : étant donnée une formule Φ, Φ est-elle valide ? (Ou de façon équivalente, Φ, ou $\longrightarrow \Phi$, est-il prouvable ?) Traditionnellement, ce problème est présenté en niant Φ, et en demandant si $\neg\Phi$ est insatisfiable. Nous n'adopterons pas ce point de vue, que nous trouvons peu naturel. Sa justification est essentiellement historique.

(α-règles)

α	α_1	α_2
$+(X \wedge Y)$	$+X$	$+Y$
$-(X \vee Y)$	$-X$	$-Y$
$-(X \Rightarrow Y)$	$+X$	$-Y$
$+(\neg X)$	$-X$	
$-(\neg X)$	$+X$	

(β-règles)

β	β_1	β_2
$-(X \wedge Y)$	$-X$	$-Y$
$+(X \vee Y)$	$+X$	$+Y$
$+(X \Rightarrow Y)$	$-X$	$+Y$

Figure 4: Règles de tableaux

4.1 Tableaux

La méthode des tableaux est due à Beth, Hintikka et Smullyan dans les années cinquante et soixante. On peut la comprendre sémantiquement, comme un moyen systématique de recherche d'une interprétation qui ne satisfasse pas Φ par récursion structurelle sur Φ (c'est pourquoi cette méthode est parfois appelée méthode des *tableaux sémantiques*). En fait, cette méthode de recherche de ce que nous appellerons des *contre-modèles* de Φ suit exactement la preuve du théorème 24. Autrement dit, les tableaux sont plus faciles à comprendre syntaxiquement, comme une recherche systématique d'une preuve sans coupure dans $\mathbf{LK}_0$.

Si on laisse de côté la coupure et la contraction, la méthode des tableaux n'est que l'algorithme qui cherche à construire une preuve dans $\mathbf{LK}_0$ en remontant à partir du but à prouver, et terminant la preuve par des instances de Ax; cette méthode conclut à l'invalidité lorsqu'elle dérive un séquent ne contenant que des variables propositionnelles et qui n'est pas une instance de Ax. Mais la communauté des tableaux utilise souvent un vocabulaire différent, que nous définissons maintenant.

Les règles de $\mathbf{LK}_0$ peuvent être classées en deux catégories (cf. figure 4). Certaines, comme $\wedge$R, ont éventuellement plusieurs prémices, mais aucune n'est plus grosse que la conclusion : on les renomme *α-règles*, elles correspondent respectivement à $\wedge$R, $\vee$L, $\Rightarrow$L, $\neg$R et $\neg$L. Toutes les autres ont une prémice unique qui est plus grosse que leur conclusion : ce sont les *β-règles*, elles correspondent respectivement à $\wedge$L, $\vee$R, $\Rightarrow$R. Ensuite, les méthodes de tableaux travaillent sur des ensembles de chemins à travers des formules signées, c'est-à-dire des couples $(+, \Phi)$ ou $(-, \Phi)$, que l'on notera respectivement $+\Phi$ et $-\Phi$:

Définition 27 (Chemins) *L'ensemble des* chemins *à travers Φ est le plus petit ensemble d'ensembles de formules signées tel que :*

- $\{+\Phi\}$ *est un chemin;*
- *si C est un chemin, et α est une formule signée dans C de type α, alors $(C \setminus \{\alpha\}) \cup \{\alpha_1\}$ et $(C \setminus \{\alpha\}) \cup \{\alpha_2\}$ sont des chemins (où α_1 et α_2 sont définies comme dans la figure 4; dans le cas de la négation, il n'y a que le premier chemin);*
- *si C est un chemin, et β est une formule signée dans C de type β, alors $(C \setminus \{\beta\}) \cup \{\beta_1, \beta_2\}$ est un chemin (où β_1 et β_2 sont définis comme dans la figure 4.)*

Une stratégie d'expansion *f est une application envoyant chaque chemin C possédant au moins une formule signée non atomique vers une telle formule.*

Un chemin C est clos *s'il existe une formule $+\Phi'$ dans C telle que $-\Phi'$ soit aussi dans C. Un* tableau *est un ensemble de chemins. Il est* clos *si et seulement si tous ses chemins sont clos.*

Les chemins sont des séquents, où les formules négatives sont sur la gauche, et les formules positives sont sur la droite. Les règles d'expansion (α et β) correspondent aux huit règles de $\mathbf{LK}_0$ sauf Ax et Cut. Et un chemin clos est une instance de Ax. Remarquer que le choix de la formule à expanser dans un chemin est arbitraire, autrement dit toute stratégie de choix d'une telle formule est capable de prouver tous les théorèmes. (Cf. l'utilisation de la stratégie f dans la preuve de complétude de $\mathbf{LK}_0$.)

La distinction entre les deux catégories de règles est due au fait que les α-règles sont des règles de *ramification*, qui peuvent augmenter le nombre de chemins à clore, mais n'augmentent jamais le nombre de formules dans un chemin; alors que les β-règles sont des règles de *prolongation*, qui n'augmentent pas le nombre de chemins à clore, mais rallongent le chemin courant.

Une réalisation classique de la méthode des tableaux est la suivante. Nous utilisons une liste pour représenter le chemin courant : une liste est soit la liste vide $[]$ soit un couple $x :: l$ d'un objet x et d'une liste l. Soit $[x_1, \ldots, x_n]$ la liste contenant n objets $x_1, \ldots, x_n$, autrement dit $x_1 :: \ldots :: x_n :: []$. Nous définissons une fonction `prouve` prenant en arguments une liste de formules signées représentant le séquent à prouver (autrement dit, le chemin courant, et que nous voulons expanser en des chemins clos) :

- si l contient à la fois $+\Phi$ et $-\Phi$ pour un certain Φ, retourner "oui";
- sinon, si l ne contient que des variables signées, retourner "non";
- sinon, appliquer la stratégie d'expansion pour choisir une formule signée non atomique Φ dans l, et soit l' la liste l sans Φ.
  - si Φ est une formule de type α (avec n α_i-formules $\Phi_1, \ldots, \Phi_n$, où $n = 1$ ou $n = 2$), alors retourner "oui" si `prouve`$(\Phi_1 :: l'), \ldots,$ `prouve`$(\Phi_n :: l')$ retournent tous "oui"; sinon retourner "non".
  - si Φ est une formule de type β (se décomposant en formules Φ_1, Φ_2), alors retourner `prouve`$(\Phi_1 :: \Phi_2 :: l')$.

Donc, pour savoir si Φ est un théorème, nous lançons `prouve`$([+\Phi])$, et lisons la réponse.

La stratégie d'expansion est utilisée pour choisir une formule signée Φ non variable dans l au troisième point ci-dessus. C'est un léger abus, car la stratégie est maintenant une fonction des listes, non des ensembles de formules signées vers les formules signées. En pratique, l est typiquement codée par une liste `l` de formules en attente de traitement, une liste `pos` de formules atomiques positives ($+A$), et une liste `neg` de formules atomiques négatives ($-A$). En détail et en Caml, sur un langage de formules simplifié ne comprenant que le $\vee$ et le $\neg$:

```
open List;;

type atom = string;;

type form = A of atom
  | OU of form * form
  | NON of form;;

type sform = POS of form | NEG of form;;

let rec prouve1 l pos neg =
    match l with
      [] -> false
```

```
    | POS (A x)::l' -> mem x neg || prouve1 l' (x::pos) neg
    | NEG (A x)::l' -> mem x pos || prouve1 l' pos (x::neg)
    | POS (OU (f, g))::l' -> prouve1 (POS f::POS g::l') pos neg
    | NEG (OU (f, g))::l' -> prouve1 (NEG f::l') pos neg &&
                             prouve1 (NEG g::l') pos neg
    | POS (NON f)::l' -> prouve1 (NEG f::l') pos neg
    | NEG (NON f)::l' -> prouve1 (POS f::l') pos neg
      ;;

let prouve f = prouve1 [POS f] [] [];;
```

La méthode des tableaux a la propriété importante suivante :

Théorème 28 (Correction, complétude) *Les tableaux sont corrects et complets pour toute stratégie : l'expansion du tableau $+\Phi$ termine sur un tableau clos si et seulement si Φ est valide.*

Preuve : Conséquence du théorème 23 et des arguments du théorème 24, portant sur des listes plutôt que des ensembles pour ce qui est de la complétude. □

On notera bien que les termes “correct” et “complet” portent ici sur les procédures de recherche de preuve, pas sur les systèmes de preuve. Une méthode de recherche de preuve est correcte si elle ne retourne “oui” que lorsque son entrée est une tautologie. Elle est complète si elle retourne “oui” dès que son entrée est une tautologie. La distinction est subtile : une méthode de recherche de preuve est une *procédure*, alors qu'un système de preuve n'est qu'une collection de règles, sans interprétation calculatoire a priori. Plus concrètement, dire que $\mathbf{LK}_0$ sans Cut est complet dit que toute tautologie a une preuve, mais nous ne savons pas dans quel ordre appliquer les règles, c'est-à-dire quelle stratégie utiliser. Dire que `prouve` est complet nous en dit plus, à savoir que l'ordre particulier dans lequel `prouve` déplie une dérivation finira par produire une preuve si le but est une tautologie; ici, ceci est une conséquence du fait que n'importe quel ordre d'application des règles de preuve (n'importe quelle stratégie) finit par aboutir à une preuve.

Théorème 29 (Terminaison) *Soit Φ une formule propositionnelle. L'expansion du tableau $+\Phi$ termine, quelle que soit la stratégie utilisée.*

Preuve : L'expansion d'un tableau construit un nouveau nœud dans l'arbre défini au théorème 24 à chaque étape. Comme cet arbre est fini, la procédure termine. □

L'intérêt des tableaux est qu'ils sont très simples à réaliser, et que les étapes fondamentales de calcul peuvent être effectuées très rapidement. De plus, ses besoins en espace sont modestes : nous n'avons besoin que d'une pile pour gérer la récursion dans `prouve`, et sa profondeur est au plus la taille de la formule à prouver. Cependant, comme la taille d'un tableau totalement expansé est en général une exponentielle de la taille de la formule à prouver, prouver un théorème par une méthode de tableaux peut prendre quelque temps. (Ce problème est en fait général à toutes les méthodes de preuve.)

Plus spécifiquement, le principal point faible des tableaux est qu'il développe chaque chemin du tableau complètement indépendamment des autres. En particulier, si nous choisissons d'expanser la même formule dans deux chemins différents, nous faisons le même travail deux fois, et gâchons du temps.

4.2 Résolution propositionnelle

La méthode de résolution, inventée en 1965 par J. Alan Robinson dans le cadre plus général de la recherche de preuve au premier ordre, et la méthode de Davis-Putnam-Logemann-Loveland, inventée en 1960 par Martin Davis et Hillary Putnam et améliorée en 1963 par Martin Davis, George Logemann et Donald Loveland, aussi dans le cadre du premier ordre, nécessitent toutes les deux une étape préliminaire. Pour prouver Φ (ou pour réfuter $\neg\Phi$), nous mettons d'abord $\neg\Phi$ en forme normale conjonctive (voir plus bas). Une façon d'expliquer la résolution propositionnelle est alors d'interpréter la forme normale conjonctive de $\neg\Phi$ comme un ensemble de séquents, à partir duquel nous essayons de dériver le séquent absurde (vide) $\longrightarrow$ en utilisant les règles de $\mathbf{LK}_0$.

Définition 30 *Une formule Φ est en* forme négation-normale *(nnf) si et seulement si elle est construire avec les connecteurs $\wedge$, $\vee$, $\neg$ uniquement, et si toute sous-formule niée $\neg\Phi'$ est telle que Φ' est une variable propositionnelle. (Les négations ne gouvernent pas les formules composites.)*

Un atome *est un autre nom pour une variable propositionnelle. Un* littéral *est une formule de la forme A ou $\neg A$, où A est un atome.*

Une formule Φ est en forme normale disjonctive *(dnf) si et seulement si elle est en nnf et aucune disjonction n'apparaît comme argument d'une conjonction. (C'est-à-dire que Φ est une disjonction n-aire de* clauses conjonctives*, qui sont des conjonctions m-aires de littéraux.)*

De façon symétrique, une formule Φ est en forme normal conjonctive *(cnf) si et seulement si elle est en nnf et aucune conjonction n'apparaît comme argument d'une disjonction. (C'est-à-dire que Φ est une conjonction n-aire de* clauses disjonctives*, qui sont des disjonctions m-aires de littéraux.)*

La clause disjonctive vide est notée $\Box$.

Pour abréger, nous dirons *clause* au lieu de *clause disjonctive*. Une clause est donc une disjonction de littéraux. Nous pouvons interpréter les clauses comme des séquents en mettant à droite tous les littéraux positifs A, et à gauche tous les atomes A venant de littéraux négatifs $\neg A$, et en éliminant les doublons. Ceci établit une correspondance entre clauses et *séquents atomiques*, c'est-à-dire séquents ne contenant que des atomes. ($\Box$ correspond alors à $\longrightarrow$.) À partir de maintenant, *clause* voudra dire séquent, écrit avec $\vee$ et $\neg$ au lieu de la notation à flèche précédente. C'est à strictement parler un abus de langage, d'autant plus que les contractions (remplacer $\Phi \vee \Phi$ par Φ) sont sous-entendues dans les clauses. Nous utilisons cette notation parce qu'elle est traditionnelle, et parce qu'elle apporte des intuitions sémantiques utiles.

Voici un algorithme pour mettre une formule Φ en nnf : d'abord, réécrire toutes les sous-formules $\Phi' \Rightarrow \Phi''$ sous la forme $\neg\Phi' \vee \Phi''$, de sorte à obtenir une formule équivalente n'utilisant que $\wedge$, $\vee$ et $\neg$. Ensuite utiliser les identités de *de Morgan* :

$\neg(\Phi' \wedge \Phi'')$ est logiquement équivalent à $(\neg\Phi') \vee (\neg\Phi'')$

$\neg(\Phi' \vee \Phi'')$ est logiquement équivalent à $(\neg\Phi') \wedge (\neg\Phi'')$

pour pousser les négations vers l'intérieur de la formule. Maintenant, si Φ est en nnf, nous la transformons en cnf en réécrivant toutes les sous-formules $\Phi' \vee (\Phi_1 \wedge \Phi_2)$ en $(\Phi' \vee \Phi_1) \wedge (\Phi' \vee \Phi_2)$ et $(\Phi_1 \wedge \Phi_2) \vee \Phi'$ en $(\Phi_1 \vee \Phi') \wedge (\Phi_2 \vee \Phi')$. Le processus termine, mais nous ne le prouverons pas (exercice).

Mettant à profit notre compréhension des clauses comme des séquents, observons que si l'on cherche une preuve sans coupure ni contraction de $\longrightarrow \Phi$ en appliquant les règles de $\mathbf{LK}_0$ au maximum, on aboutit à un ensemble de séquents atomiques dont la conjonction est équivalente à Φ. Dans le langage des tableaux, développer un tableau pour $+\Phi$ jusqu'à ce que tous les chemins non clos restant soient atomiques : la conjonction de tous ces chemins, où chaque chemin est vu comme une disjonction de littéraux ($+A$ pour A et $-A$ pour $\neg A$), est équivalent à Φ, et est donc une cnf pour Φ.

Nous avons ensuite :

Théorème 31 (Résolution) *Soit Φ une formule en cnf, et considérons Φ comme un ensemble S de séquents atomiques. Alors Φ est insatisfiable si et seulement si le séquent vide $\longrightarrow$ est déductible de S et de la règle Cut seulement.*

Preuve : La correction (si $\longrightarrow$ est déductible par Cut de S, alors Φ est insatisfiable) est due à l'observation suivante : si $\Gamma \longrightarrow \Phi', \Delta$ et $\Gamma', \Phi' \longrightarrow \Delta'$ sont coupés sur Φ' pour donner $\Gamma, \Gamma' \longrightarrow \Delta, \Delta'$, alors toute interprétation ρ qui satisfait les deux premiers doit aussi satisfaire la dernière. Donc si Φ était satisfiable, par récurrence sur la longueur de la dérivation de $\longrightarrow$ à partir de S, nous montrerions que $\longrightarrow$ est satisfiable, ce qui est absurde.

Montrons maintenant la complétude, c'est-à-dire la réciproque.

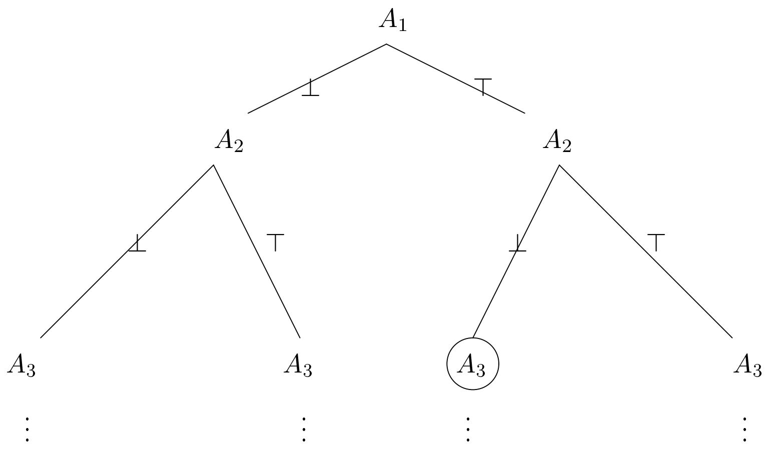


Figure 5: Un arbre de décision

Nous construisons un arbre de décision (figure 5). Soient $A_1, \ldots, A_n$ les variables libres de Φ. Nous définissons les nœuds de l'arbre à profondeur i (à partir de la racine), $0 \leq i \leq n$, comme toutes les suites $A_1^\rho, \ldots, A_i^\rho$, où ρ est une affectation quelconque, et A^ρ dénote A si $\rho(A) = \top$ et $\neg A$ sinon. Si $A_1^\rho, \ldots, A_i^\rho$ est un nœud, soit $i = n$ et il s'agit d'une *feuille*, soit $i < n$, et il a deux *fils* $A_1^\rho, \ldots, A_i^\rho, \neg A_{i+1}$ et $A_1^\rho, \ldots, A_i^\rho, A_{i+1}$. Appelons une telle suite $A_1^\rho, \ldots, A_i^\rho$ une *interprétation partielle.* Si $i = n$, il s'agit d'une *interprétation totale.* (Ces dernières sont les interprétations au sens intuitif, envoyant A_i vers $\top$ si $A_i^\rho = A_i$, vers $\perp$ si $A_i^\rho = \neg A_i$.)

Supposons que S, en tant que conjonction des séquents qui lui appartiennent, est insatisfiable, c'est-à-dire que pour toute interprétation ρ, il existe un $C \in S$ tel que non $\rho \models C$. Pour toute interprétation totale, il y a au moins un séquent dans S qui n'est pas satisfait par l'interprétation correspondante ρ. Nous appelons *nœud d'échec* pour un séquent C dans S tout nœud le plus haut possible dans l'arbre tel qu'aucune interprétation totale passant par ce nœud ne satisfasse C. Nous définissons ensuite l'*arbre de décision clos* de S comme étant l'arbre de décision de départ, coupé à chaque nœud d'échec (autrement dit, les nœuds d'échec deviennent des feuilles). Remarquons que toute interprétation totale doit traverser au moins un nœud d'échec, et donc les feuilles de l'arbre clos sont exactement les nœuds d'échec de l'arbre de départ.

Par exemple, considérons la figure 6, où S consiste en $\longrightarrow B$, $B \longrightarrow A$, $A \longrightarrow C$, $C \longrightarrow$ (en tant que clauses, B, $A \vee \neg B$, $\neg A \vee C$, $\neg C$). Notre énumération des atomes est $A_1 = A$, $A_2 = B$, $A_3 = C$. Le nœud dans un rond le plus à gauche (l'interprétation partielle $\neg A, \neg B$) est un nœud d'échec pour le séquent $\longrightarrow B$ (la

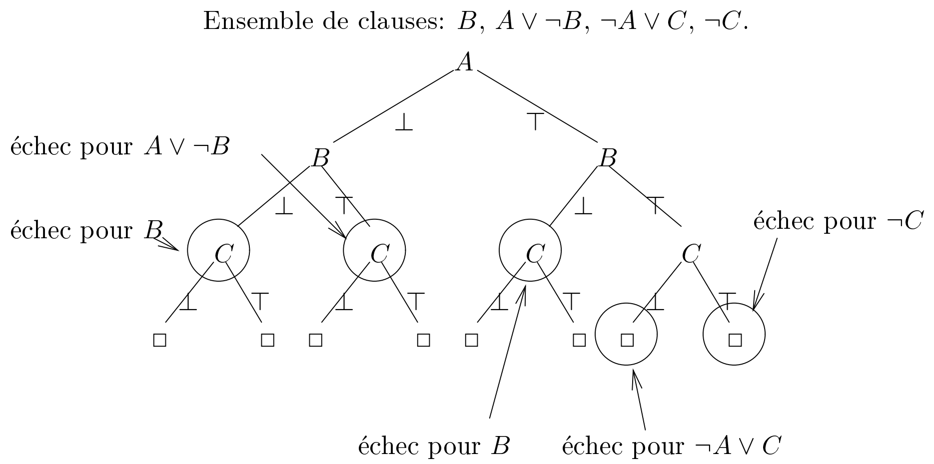


Figure 6: Un arbre de décision, avec nœuds d'échec

clause B) parce que B est rendue fausse à ce nœud, mais à aucun nœud plus haut. Les nœuds d'échec sur cet exemple sont précisément ceux entourés d'un rond dans la figure 6, et l'arbre clos est le sous-arbre situé au-dessus de ces nœuds d'échec.

Nous définissons maintenant les *nœuds d'inférence* comme étant les nœuds de l'arbre clos qui ont deux nœuds d'échec comme fils. Nous affirmons que si l'arbre clos a plus d'un nœud (si la racine n'est pas un nœud d'échec), alors il y a au moins un nœud d'inférence.

En effet, soit n le nombre de nœuds dans l'arbre clos (en incluant les feuilles), et prouvons l'affirmation par récurrence sur n. Comme la racine n'est pas un nœud d'échec, $n \geq 3$. Si $n = 3$, alors la racine est un nœud d'inférence. Sinon, $n > 3$ et supposons l'affirmation prouvée pour tous les arbres clos avec au moins 3 nœuds, et au plus $n - 1$ nœuds. Notre arbre clos a n nœuds, $n > 3$: la racine n'est pas un nœud d'échec, et un de ses fils n'est pas non plus un nœud d'échec; le sous-arbre dont la racine est ce fils a donc au moins 3 nœuds, et strictement moins de n nœuds, donc par hypothèse de récurrence il contient un nœud d'inférence. Donc l'arbre clos total contient un nœud d'inférence.

Par définition, si un nœud d'inférence N est étiqueté par une variable propositionnelle A (c'est-à-dire A égale A_i et N est une interprétation partielle $A_1^\rho, \ldots, A_{i-1}^\rho$), alors il existe un séquent dans S pour lequel le fils de gauche du nœud est un nœud d'échec (et ce séquent doit être de la forme $\Gamma \longrightarrow A, \Delta$), et de même il existe un séquent dans S pour lequel le fils de droit du nœud est un nœud d'échec (et il doit être de la forme $\Gamma', A \longrightarrow \Delta'$).

Par exemple, sur la figure 6, les nœuds d'inférence sont le nœud B le plus à gauche (clause de gauche B, clause de droite $A \vee \neg B$), et le nœud C le plus à droite (clause de gauche $\neg A \vee C$, clause de droite $\neg C$).

Produisons alors la clause $\Gamma, \Gamma' \longrightarrow \Delta, \Delta'$, qui est le résultat de l'application de Cut sur ces séquents, et ajoutons le à S. Ce faisant, nous obtenons un nouvel ensemble S' de séquents, qui est insatisfiable. Remarquons que tout nœud d'échec pour S est un nœud d'échec ou est en-dessous d'un nœud d'échec de S'. Remarquons aussi que le nœud d'inférence N choisi dans S est maintenant nœud d'échec pour S' (prouvez-le en exercice), de sorte que l'arbre clos pour S' est strictement plus petit que celui pour S. (Dans l'exemple, si l'on coupe entre B et $A \vee \neg B$ sur le nœud d'inférence de gauche, ceci produit A comme nouvelle clause, et produit l'arbre de décision de la figure 7.)

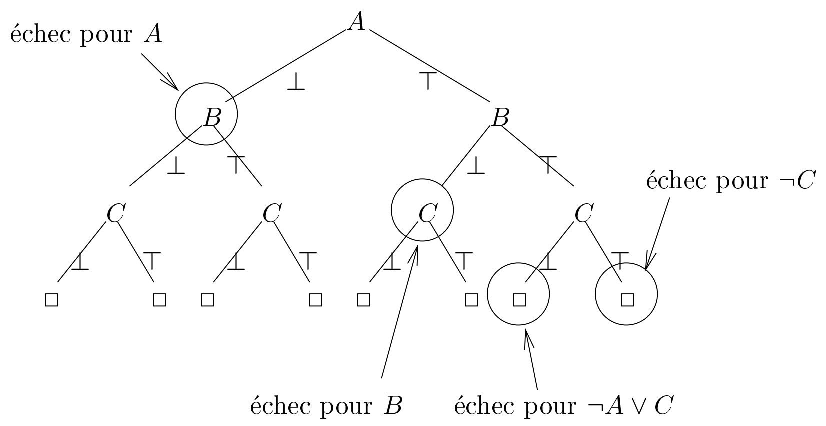


Figure 7: L'arbre de décision, après application de Cut

En somme, si nous prenons comme valeur initiale de S l'ensemble des clauses représentant Φ, nous fabriquons, par l'application de Cut entre des clauses aux nœuds d'inférence, des ensembles insatisfiables de clauses de plus en plus gros et ayant des arbres clos de plus en plus petits. Après un nombre fini de telles opérations, l'arbre clos sera donc réduit à sa racine. Mais si la racine est un nœud d'échec, c'est qu'une clause doit être rendue fausse par l'interprétation partielle vide. Comme toutes les clauses que nous produisons sont atomiques, une telle clause ne peut être que la clause vide. Ainsi, la clause vide a dû être engendrée en un nombre fini d'application de Cut à partir de S. $\square$

Ce théorème est surprenant : si une formule Φ est prouvable, alors soit S l'ensemble des séquents atomiques correspondant à une cnf pour $\neg\Phi$. En utilisant la règle Cut, nous pouvons montrer que Φ est prouvable à partir de $\mathbf{LK}_0$ si et seulement si $\longrightarrow$ est dérivable à partir de $\mathbf{LK}_0 + S$ (le système de preuve contenant les règles de $\mathbf{LK}_0$plus tous les séquents de S vus comme de nouveaux axiomes). Le théorème 31 est ainsi en quelque sort le contraire du théorème 26 : au lieu d'éliminer les coupures, nous éliminons toutes les règles de $\mathbf{LK}_0$ sauf Cut. Nous expliquerons cette énigme lorsque nous examinerons le processus d'élimination des coupures.

Pour appliquer le théorème 31, Robinson avait proposé une variante de l'algorithme suivant. Étant donné un ensemble S de clauses pour $\neg\Phi$, où Φ est la formule à prouver :

1. Soit $S_0 = S$, et $n = 0$ (n sera appelé le *niveau*, et S_n l'ensemble des clauses au niveau n).

2. Si S_n contient la clause vide, retourner "oui".

3. Sinon, calculer l'ensemble des *resolvants* de clauses de S_n (un résolvant est le résultat d'une coupure sur deux clauses), telles que les clauses coupées (les prémices) ne sont pas toutes deux dans S_{n-1} si $n \geq 1$; (sinon, on refabriquerait les clauses qui sont déjà dans S_n;) les ajouter à S_n pour obtenir un nouvel ensemble S_{n+1}.

4. Si $S_{n+1} = S_n$, alors retourner "non".

5. Sinon, poser n égale $n + 1$, et retourner à l'étape 2.

Cet algorithme est appelé résolution par *saturation de niveaux*, parce que S_n est l'ensemble de toutes les conséquences de S déductibles en appliquant Cut au plus n fois.

Cet algorithme est très inefficace. Le problème ne tient pas tant à la comparaison entre les ensembles S_n et S_{n+1}, mais dans le fait que les résolvants qui sont calculés à l'étape 3 peuvent contenir énormément de redondance : de nombreuses tautologies (séquents Ax) et des clauses qui avaient déjà été fabriquées sont refabriquées constamment. En fait, la résolution est une très mauvaise méthode de recherche de preuve propositionnelle, mais elle est plus intéressante au premier ordre.

4.3 La méthode de Davis-Putnam-Logemann-Loveland

Nous arrivons aux méthodes sémantiques. Ces méthodes sont, en pratique, parmi les procédures correctes, complètes et terminantes les plus rapides que l'on connaît aujourd'hui.

L'idée de la méthode de Davis-Putnam-Logemann-Loveland est la suivante. Elle prend encore en entrée un ensemble S de clauses représentant $\neg\Phi$, où Φ est la formule à prouver. Φ est valide si et seulement si S est insatisfiable. Si S est vide, alors S est satisfiable, donc Φ est invalide. Si S contient la clause vide, il est insatisfiable, donc Φ est valide. Sinon, nous choisissons une variable propositionnelle libre A dans S, et essentiellement nous remplaçons A par vrai (resp. faux) dans S, et simplifions le tout, ce qui donne un ensemble S_1 (resp. S_2) de clauses : S est alors satisfiable si et seulement si S_1 ou S_2 est satisfiable.

Ce qui fait que la méthode de Davis-Putnam-Logemann-Loveland est efficace est qu'elle identifie un certain nombre de cas particuliers où nous n'avons pas à séparer S en S_1 et S_2 selon la valeur de la variable A. En effet, cette séparation multiplie le nombre de sous-problèmes à résoudre par 2 pour chaque variable propositionnelle, et c'est la cause de l'explosion exponentielle du nombre de sous-problèmes à résoudre.

Les règles sont les suivantes :

Définition 32 (Tautologies) *Une clause est appelée une* tautologie *si et seulement si elle contient A et $\neg A$, où A est une variable.*

Remarquer en effet qu'une clause est valide si et seulement si elle contient à la fois A et $\neg A$, pour une certaine variable A.

Définition 33 (Séparation (splitting)) *Soit S un ensemble de clauses non tautologiques, et A une variable propositionnelle.*

Définissons $S[\mathbf{T}/A]$ comme l'ensemble S d'où toutes les clauses de la forme $C \vee A$ (contenant A comme littéral positif) ont été supprimées, et où toutes les clauses $C \vee \neg A$ ont été remplacées par C.

Définissons $S[\mathbf{F}/A]$ comme l'ensemble S où toutes les clauses de la forme $C \vee A$ ont été remplacées par C, et d'où toutes les clauses $C \vee \neg A$ ont été supprimées.

Remarquer qu'une affectation ρ telle que $\rho(A) = \top$ (resp. $\perp$) satisfait S si et seulement si elle satisfait $S[\mathbf{T}/A]$ (resp. $S[\mathbf{F}/A]$), et que ces derniers n'ont plus A comme variable libre.

Définition 34 (Clause unitaire) *Une clause C est dite* unitaire *si elle contient exactement un littéral.*

Les clauses unitaires sont intéressantes parce que si S contient une clause unitaire, disons A, alors S est satisfiable si et seulement si A est satisfiable. Toute affectation satisfaisant S doit attribuer vrai à A, donc S est satisfiable si et seulement si $S[\mathbf{T}/A]$ l'est. (Si la clause unitaire est de la forme $\neg A$, S est satisfiable si et seulement si $S[\mathbf{F}/A]$ l'est.) En particulier, si S contient une clause unitaire, nous n'avons pas besoin d'effectuer une séparation sur A, car il n'y a qu'un choix possible.

Définition 35 (Littéraux purs) *Un littéral A (resp. $\neg A$) est dit* pur *dans un ensemble de clauses S si et seulement si $\neg A$ (resp. A) n'apparaît dans aucune clause de S.*

Une clause est pure *dans S si elle contient un littéral pur dans S.*

Supposons que A (resp. $\neg A$) est pur dans S. $S[\mathbf{T}/A]$ (resp. $S[\mathbf{F}/A]$) est alors un sous-ensemble de S, donc si S est satisfiable, alors $S[\mathbf{T}/A]$ (resp. $S[\mathbf{F}/A]$) l'est aussi. Réciproquement, si $S[\mathbf{T}/A]$ (resp. $S[\mathbf{F}/A]$) est satisfiable, soit ρ une affectation satisfaisant toutes ses clauses, et ρ' l'affectation envoyant A vers $\top$ (resp. $\bot$) et toutes les autres variables B vers $\rho(B)$. Alors ρ' satisfait toutes les clauses de S; en effet, pour chaque clause dans C, soit A n'apparaît pas dans C et C est dans $S[\mathbf{T}/A]$ (resp. $S[\mathbf{F}/A]$), donc ρ' satisfait C, ou C est de la forme $C' \vee A$ (resp. $C' \vee \neg A$), et par déefinition de ρ' en A, ρ' satisfait C de nouveau.

Une autre façon de le dire est que si S contient une clause pure, alors S sans cette clause pure est satisfiable si et seulement si S est satisfiable. Nous pouvons donc éliminer les clauses pures de S sans changer sa satisfiabilité. On notera de plus qu'éliminer des clauses pures peut rendre d'autres clauses pures: par exemple, parmi les trois clauses $A \vee \neg B$, $\neg A \vee \neg B$, $B \vee C$, la troisième est pure car $\neg C$ n'apparaît pas dans l'ensemble de clauses: on peut donc l'éliminer; mais alors les deux premières clauses deviennent pures elles-mêmes, car B n'apparaî plus nulle part; on conclut donc immédiatement que ces trois clauses forment un ensemble satisfiable. (Exercice: reconstruire une affectation satisfaisant ces trois clauses.)

La procédure de Davis-Putnam-Logemann-Loveland fonctionne comme suit. Au départ, S est un ensemble fini de clauses :

1. si S est vide, retourner "satisfiable".
2. si S contient la clause vide, retourner "insatisfiable".
3. si S contient une tautologie (definition 32) ou une clause pure, l'éliminer de S et aller en étape 1.
4. si S contient une clause unitaire A (resp. $\neg A$), remplacer S par $S[\mathbf{T}/A]$ (resp. $S[\mathbf{F}/A]$), et aller en étape 1.
5. sinon, choisir une variable libre A dans S. Appliquer la procédure de Davis-Putnam-Logemann-Loveland récursivement sur $S[\mathbf{T}/A]$ et $S[\mathbf{F}/A]$. Retourner "satisfiable" si l'un des résultats était "satisfiable", et retourner "insatisfiable" sinon.

On peut beaucoup améliorer la dernière étape. Tout d'abord, nous pouvons déjà appliquer la procédure de Davis-Putnam-Logemann-Loveland sur $S[\mathbf{T}/A]$, et retourner "satisfiable" immédiatement si le résultat est "satisfiable", sinon nous appliquons la procédure de Davis-Putnam-Logemann-Loveland sur $S[\mathbf{F}/A]$. Nous pouvons aussi décider d'appliquer Davis-Putnam-Logemann-Loveland sur $S[\mathbf{F}/A]$ en premier, et ensuite éventuellement sur $S[\mathbf{T}/A]$, ou lancer les deux en parallèle...

Sur un ordinateur séquentiel, nous avons besoin d'une stratégie pour savoir lequel des deux tester en premier. Nous avons aussi besoin d'une stratégie pour choisir la variable A dans la dernière étape. Les choix de stratégie peuvent avoir beaucoup d'influence sur la rapidité de l'algorithme à conclure. Les bonnes heuristiques sont typiquement de choisir A de sorte que le nombre de clauses dans lesquelles A apparaît multiplié par le nombre de clauses dans lesquelles $\neg A$ apparaît est maximal (si nous devons effectuer une séparation, séparons au moins sur une variable qui a le plus d'influence sur l'ensemble de clauses), et de tester $S[\mathbf{T}/A]$ d'abord s'il y a plus de clauses contenant A que de clauses contenant $\neg A$, et $S[\mathbf{F}/A]$ sinon. (Si nous devons effectuer une séparation, testons d'abord l'ensemble qui a le moins de clauses.) Une stratégie plus évoluée est la règle de Jeroslow-Wang, qui consiste à choisir un littéral L qui maximise $\sum_i 2^{-n_i}$, où i parcourt les clauses contenant L et n_i est la longueur de la clause i. Consulter [HV95] pour une discussion.

Un autre point important est comment coder la méthode de Davis-Putnam-Logemann-Loveland efficacement. La réalisation traditionnelle n'efface ni ne modifie aucun clause, mais cherche une affectation qui satisfasse l'ensemble des clauses. Soit ρ une *affectation partielle*, à savoir un ensemble de littéraux tels que A et $\neg A$ ne sont jamais simultanément dans ρ. (On retrouve la notion habituelle d'affectation quand, pour chaque A, soit A soit $\neg A$ est dans ρ.) Un littéral L est satisfait par une affectation partielle ρ si $L \in \rho$; une clause C est satisfaite par ρ si $C \cap \rho \neq \emptyset$; et un ensemble de clauses S est satisfait par ρ si toutes ses clauses sont satisfaites par ρ. L'algorithme tente ensuite de trouver une affectation (partielle) ρ satisfaisant S.

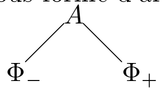
Il organise les clauses sous forme d'une matrice creuse, où les lignes sont des clauses et les colonnes sont repérées par les variables propositionnelles : si A est une variable propositionnelle, la colonne A est la liste de toutes les clauses où A ou $\neg A$ apparaît. En plus, chaque clause est munie d'attributs entiers comptant le nombre de littéraux qui sont vrais (satisfaits), resp. faux (dont la négation est satisfaite), resp. non affectées par l'affectation partielle courante ρ. Pour ajouter un littéral L (A or $\neg A$) à ρ, comme lors des étapes de séparation, de propagation des littéraux unitaires ou d'élimination de clauses pures, on parcourt la colonne A, et on met à jour les compteurs. Si le compteur de littéraux vrais et le compteur de littéraux non affectés sont tous les deux nuls dans une clause, alors échouer, et revenir (*backtracker*) au dernier point où une séparation a été effectuée. Pour gérer les clauses unitaires, on maintient une pile de littéraux unitaires : chaque fois que le nombre de littéraux non affectés dans une clause devient un et que le nombre de littéraux vrais est toujours nul, on empile le seul littéral restant sur la pile, à moins que sa négation ne soit déjà sur la pile. (On backtracke dans ce dernier cas.) On peut aussi gérer les clauses pures en conservant dans chaque colonne le nombre d'occurrences positives, resp. négatives de variables dans chaque clause.

4.4 Diagrammes de décision binaire

Une autre idée qui fonctionne pour prouver des formules propositionnelles et qui vient d'idées sémantiques est celle des *diagrammes de décision binaire*, ou *BDD*. Le créateur des BDD tels que nous les connaissons aujourd'hui est Randall E. Bryant en 1986. Cependant, les BDD sont juste des arbres de décision avec quelques astuces bien connues en plus, et les arbres de décision remontent à George Boole (1854), si pas plus tôt. Les idées sont très simples, mais les réalisations informatiques sont usuellement plus complexes qu'avec les méthodes précédentes.

En gros, les astuces qui font que les BDD fonctionnent sont : d'abord, au lieu de représenter les arbres de décision comme des arbres en mémoire (où il y a un unique chemin de la racine à n'importe quel nœud), nous les représentons comme des *graphes orientés acycliques* ou *DAG*, autrement dit nous partageons tous les sous-arbres identiques. Ensuite, nous utilisons la règle de simplification suivante : si un sous-arbre a deux fils identiques, alors remplacer le sous-arbre par ce fils; essentiellement, ce sous-arbre signifie “si A est vrai, alors utilise le fils de droite; si A est faux, utilise le fils de gauche”: comme les fils de gauche et de droite coincident, il n'y a pas lieu d'effectuer une sélection fondée sur la valeur de A. Enfin, nous ordonnons les variables dans un ordre total donné $<$, et exigeons que, si nous descendons dans le BDD sur n'importe quel chemin, nous rencontrons les variables en ordre croissant. Cette dernière propriété assurera que les BDD sont des *représentants canoniques* de formules à équivalence logique près, c'est-à-dire que si Φ et Φ' sont deux formules équivalentes, alors leurs BDD construits sur le même ordre sont identiques.

À la base, les BDD sont soit $\mathbf{T}$ (vrai), $\mathbf{F}$ (faux) ou "si A alors Φ_+ sinon Φ_-", où Φ_+ et Φ_- sont des BDD distincts. Nous notons cette dernière forme $A \longrightarrow \Phi_+; \Phi_-$, ou sous forme d'arbre :



Cette représentation simple nous permet de construire des BDD morceau par morceau. Par exemple, pour construire le BDD de $\Phi \wedge \Phi'$, il suffit de combiner les BDD de Φ et de Φ'. Si le BDD de Φ est "si A alors Φ_+ sinon Φ_-", et celui de Φ' est "si A alors Φ'_+ sinon Φ'_-"; alors le BDD de $\Phi \wedge \Phi'$ est simplement "si A alors $\Phi_+ \wedge \Phi'_+$ sinon $\Phi_- \wedge \Phi'_-$", où les deux conjonctions $\Phi_+ \wedge \Phi'_+$ et $\Phi_- \wedge \Phi'_-$ sont calculées récursivement. Le même principe s'applique aux disjonctions, implications, négations, et ainsi de suite, et est appelé le principe de décomposition de Shannon (cf. plus bas).

On peut donc calculer les BDD de formules de bas en haut : par exemple, pour calculer le BDD de $(A \wedge B) \vee C$, nous calculons celui de A, à savoir $A \longrightarrow \mathbf{T}; \mathbf{F}$, celui de B, à savoir $B \longrightarrow \mathbf{T}; \mathbf{F}$, puis nous calculons leur conjonction, puis la disjonction de ce dernier avec le BDD de C. Le fait que les BDD sont des formes canoniques nous assure que la formule de départ est valide si et seulement si son BDD est exactement $\mathbf{T}$.

Définissons maintenant les BDD plus formellement. Soit $\mathbf{T}$ (vrai) et $\mathbf{F}$ (faux) deux nouveaux symboles (à ne pas confondre avec les valeurs de vérité $\top$ et $\bot$, bien que ce soit ce qu'elles représentent.)

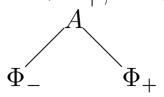
Lemme 36 (Shannon) *La sémantique de* $\mathbf{T}$ *est* $[\![\mathbf{T}]\!] = \top$*, celle de* $\mathbf{F}$ *est* $[\![\mathbf{F}]\!] = \bot$.

Pour toute formule Φ*, et toute variable propositionnelle* A*,* Φ *est équivalente à* $(A \Rightarrow \Phi[\mathbf{T}/A]) \wedge (\neg A \Rightarrow \Phi[\mathbf{F}/A])$*, où* A *n'apparaît libre ni dans* $\Phi[\mathbf{T}/A]$ *ni dans* $\Phi[\mathbf{F}/A]$*. Ceci est appelé* le principe de décomposition de Shannon.

Preuve : exercice. □

Définition 37 (Graphes de Shannon) *Un* graphe de Shannon *est une formule construite sur les deux constantes* $\mathbf{T}$*,* $\mathbf{F}$ *au moyen du seul connecteur ternaire* $\_ \longrightarrow \_; \_$*. Plus précisément, l'ensemble des graphes de Shannon est le plus petit ensemble tel que* $\mathbf{T}$ *et* $\mathbf{F}$ *sont des graphes de Shannon, et tel que, si* A *est une variable, et* Φ_+*,* Φ_- *sont deux graphes de Shannon, alors* $A \longrightarrow \Phi_+; \Phi_-$ *("si* A *alors* Φ_+*, sinon* Φ_-*") est un graphe de Shannon.*

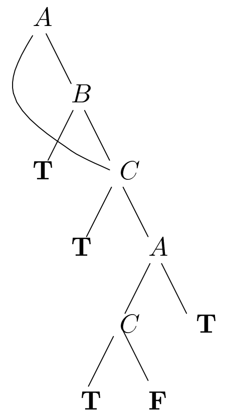
$A \longrightarrow \Phi_+; \Phi_-$ *sera aussi représenté graphiquement par*



(Remarquer que la branche "A vrai" est à droite, alors que la branche "A faux" est à gauche, ce qui est l'ordre inverse de Φ_+ *et* Φ_-*.)*

Nous devons définir la sémantique. Nous avons déjà défini $[\![\mathbf{T}]\!]\rho$ comme $\top$, $[\![\mathbf{F}]\!]\rho$ comme $\bot$. La formule $A \longrightarrow \Phi_+; \Phi_-$ est censée représenter la formula $(A \Rightarrow \Phi_+) \wedge (\neg A \Rightarrow \Phi_-)$ du principe de décomposition de Shannon, donc nous posons $[\![A \longrightarrow \Phi_+; \Phi_-]\!]\rho = [\![\Phi_+]\!]\rho$ si $\rho(A) = \top$, et $= [\![\Phi_-]\!]\rho$ si $\rho(A) = \bot$.

Nous représentons alors les formules construites avec $\wedge$, $\vee$, $\neg$, $\Rightarrow$, etc. par des graphes de Shannon logiquement équivalents. Par exemple, la formule $(((A \Rightarrow B) \wedge C) \Rightarrow A) \vee \neg C$ peut être représentée par le graphe de Shannon suivant :



où nous avons dessiné plusieurs nœuds **T** au lieu d'un dans l'intérêt de la lisibilité. (Vérifier que ceci est effectivement équivalent à la formule de départ, en énumérant toutes les affectations possibles sur A, B, C.) Ce n'est pas la seule représentation de la formule sous forme de graphe de Shannon, cependant.

Les graphes (ou arbres) de Shannon ont la propriété suivante, qui rend le calcul des opérations logiques aisé :

Théorème 38 (Orthogonalité) *Soit* $\Phi = A \longrightarrow \Phi_+; \Phi_-$, $\Phi' = A \longrightarrow \Phi'_+; \Phi'_-$.

La négation de Φ*, que nous notons* $\neg\Phi$*, est* $A \longrightarrow \neg\Phi_+; \neg\Phi_-$*. Plus formellement, ce dernier graphe est satisfait exactement par les affectations qui ne satisfont pas* Φ.

Pour tout connecteur binaire $\circ$ *(*$\wedge$*,* $\vee$*,* $\Rightarrow$*,* $\Leftrightarrow$*, respectivement),* $\Phi \circ \Phi' = A \longrightarrow (\Phi_+ \circ \Phi'_+); (\Phi_- \circ \Phi'_-)$*; plus formellement, ce dernier est satisfait exactement par les affectations qui satisfont à la fois* Φ *et* Φ'*, resp.* Φ *ou* Φ'*, resp.* Φ' *ou* $\neg\Phi$*, resp. à la fois* Φ *et* Φ' *ou ni* Φ *ni* Φ'.

Preuve : Immédiat. $\square$

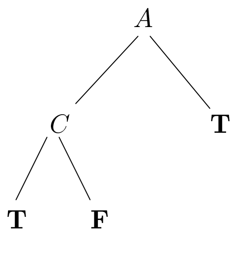
Nous pouvons faire mieux que les graphes de Shannon, et définir les BDD :

Définition 39 (BDD) *Soit* Φ *un graphe de Shannon, et* $<$ *un ordre strict total des variables de* $\mathrm{fv}(\Phi)$.

Nous disons que Φ *est un* diagramme de décision binaire*, ou* BDD*, ordonné par* $<$ *si et seulement si:*

- *(Réduit)* Φ *ne contient aucun sous-graphe de la forme* $A \longrightarrow \Phi'; \Phi'$ *(avec deux fils identiques);*
- *(Ordonné) tous les sous-graphes de* Φ *de la forme* $A \longrightarrow \Phi_+; \Phi_-$ *sont tels que* A *est strictement inférieur (pour* $<$*) à toutes les variables dans* $\mathrm{fv}(\Phi_+) \cup \mathrm{fv}(\Phi_-)$.

Le BDD de $(((A \Rightarrow B) \wedge C) \Rightarrow A) \vee \neg C$, avec $A < B < C$, est par exemple :



On peut voir un BDD Φ comme un automate qui décide si une affectation ρ donnée satisfait Φ. En effet, partons de la racine du BDD et descendons comme suit : lorsque nous sommes à un nœud étiqueté A (nous considérons un sous-graphe $A \longrightarrow \Phi_+; \Phi_-$), prenons la branche de droite (Φ_+ dans l'exemple) si $\rho(A) = \top$, et la branche de gauche (Φ_-) si $\rho(A) = \perp$. Ce processus termine lorsque nous arrivons à une feuille : si cette feuille est $\mathbf{T}$, alors ρ satisfait Φ, alors que si elle est $\mathbf{F}$, ρ ne satisfait pas Φ. Tout ceci est vrai parce qu'essentiellement, les BDD sont des représentations compactes d'arbres de décision.

Les BDD sont des formes canoniques des formules modulo équivalence logique (ce que les graphes de Shannon n'étaient pas) :

Théorème 40 (Canonicité) *Soient Φ et Φ' deux BDD construits sur le même ordre $<$. Alors Φ et Φ' sont logiquement équivalents si et seulement s'ils sont égaux.*

Preuve : S'ils sont égaux, leur équivalence logique est claire.

Réciproquement, supposons Φ et Φ' logiquement équivalents. Nous montrons qu'ils sont égaux par récurrence sur le cardinal de $\mathrm{fv}(\Phi) \cup \mathrm{fv}(\Phi')$.

Si $n = 0$, alors Φ et Φ' sont dans $\{\mathbf{T}, \mathbf{F}\}$; comme Φ et Φ' sont logiquement équivalents, ils sont soit tous les deux $\mathbf{T}$ soit tous les deux $\mathbf{F}$.

Si $n \geq 1$, soit A la plus petite variable de $\mathrm{fv}(\Phi) \cup \mathrm{fv}(\Phi')$ pour l'ordre $<$ (autrement dit, la variable la plus haute). Sans perte de généralité, supposons que A est libre dans Φ. Comme A est la plus petite variable dans $\mathrm{fv}(\Phi)$, nécessairement $\Phi = A \longrightarrow \Phi_+; \Phi_-$ pour deux BDD Φ_+ et Φ_-.

Si A n'est pas libre dans Φ', alors Φ' est logiquement équivalent à Φ_+ et Φ_-. En effet, si $\rho \models \Phi'$, alors $\rho' \models \Phi'$, où ρ' envoie A vers $\top$ et toutes les autres variables B vers $\rho(B)$; comme Φ' est logiquement équivalent à Φ, $\rho' \models \Phi$; comme $\rho'(A) = \top$, $\rho' \models \Phi_+$. Réciproquement, si $\rho \models \Phi_+$, alors $\rho' \models \Phi_+$ où ρ' est comme ci-dessus, donc $\rho' \models \Phi$, et par équivalence logique $\rho' \models \Phi'$; comme A n'est pas libre dans Φ', $\rho \models \Phi'$. Donc Φ' est logiquement équivalent à Φ_+; et de même, Φ' est logiquement équivalent à Φ_-. Par hypothèse de récurrence (les cardinaux de $\mathrm{fv}(\Phi_+) \cup \mathrm{fv}(\Phi')$ et de $\mathrm{fv}(\Phi_-) \cup \mathrm{fv}(\Phi')$ sont en effet strictement inférieurs à n), Φ' égale à la fois Φ_+ et Φ_-. Mais c'est impossible, parce que Φ est réduit ($\Phi_+ \neq \Phi_-$).

Donc A est libre aussi dans Φ', et en fait $\Phi' = A \longrightarrow \Phi'_+; \Phi'_-$ pour deux BDD Φ'_+ et Φ'_-. Par des arguments similaires à ceux ci-dessus, Φ_+ est logiquement équivalent à Φ'_+ et Φ_- à Φ'_-, donc par hypothèse de récurrence $\Phi_+ = \Phi'_+$ et $\Phi_- = \Phi'_-$. Donc $\Phi = \Phi'$. □

En plus, les BDD de formules sont usuellement compacts. Si une formule est valide ou insatisfiable, son BDD est $\mathbf{T}$ ou $\mathbf{F}$, qui est évidemment très compact. (Mais les BDD de ses sous-formules, que nous devons construire pour calculer le BDD de la formule toute entière, peuvent être bien plus gros.) Mais même lorsqu'ils sont invalides et satisfiables, les BDD restent généralement petits. Il y a des exceptions, et dans le pire des cas les BDD peuvent avoir une taille exponentielle en le nombre de variables libres dans la formule de départ. (Pour être précis, si n est ce nombre, le nombre de nœuds dans un BDD ne dépasse pas $2^n/n$.)

La méthode usuelle (et la mieux connue) pour construire un BDD pour une formule donnée Φ est d'utiliser le théorème 38 pour le construire récursivement à partir des BDD de ses sous-formules.

Nous réalisons d'abord une fonction qui alloue et partage des triplets $A \longrightarrow \Phi'; \Phi''$ comme des enregistrements ("records") en mémoire avec trois champs pour A, Φ' et Φ'' respectivement. Pour ce faire, nous utilisons une table de hachage globale ("hash-table", cf. [Knu73]) qui mémorise tous les triplets construits antérieurement. Les tables de hachage sont faites de sorte que, étant donnés A, Φ', Φ'', soit un triplet $A \longrightarrow \Phi'; \Phi''$ est déjà dans la table et on peut retourner son adresse rapidement, ou bien on annonce son absence rapidement. Un algorithme simple de table de hachage qui fonctionne bien en pratique consiste à fixer un entier N suffisamment grand (et premier pour de meilleurs résultats), et d'allouer un tableau global à N entrées, vides au départ. Ces entrées contiendront des listes chaînées de triplets déjà construits.

Pour découvrir si $A \longrightarrow \Phi'; \Phi''$ est dans la table, nous calculons d'abord une *fonction de hachage* $h(A, \Phi', \Phi'')$ sur A, Φ', Φ'', qui retourne un entier entre 0 et $N-1$ (i.e., un index dans la table). Comme tous les BDD identiques sont partagés, ils sont décrits de façon unique par leur adresse en mémoire : un bon choix pour $h(A, \Phi', \Phi'')$ est la somme des adresses de A, Φ' et Φ'', modulo N. L'invariant de la table est que si $A \longrightarrow \Phi'; \Phi''$ est dans la table, il est dans la liste à l'entrée numéro i de la table, où $i = h(A, \Phi', \Phi'')$.

Pour trouver si $A \longrightarrow \Phi'; \Phi''$ est déjà dans la table, alors, calculer $i = h(A, \Phi', \Phi'')$; parcourir la liste dans l'entrée i, et comparer chaque élément avec l'enregistrement de composantes A, Φ' et Φ''. Si nous rencontrons le triplet attendu, nous retournons son adresse. Sinon, nous annonçons son absence. De même, pour allouer et partager $A \longrightarrow \Phi'; \Phi''$, nous le recherchons dans la table, et le retournons si nous l'avons trouvé; sinon, nous allouons un nouvel enregistrement contenant A, Φ', Φ'', le rajoutons en tête de la liste à l'entrée numéro i, et retournons son adresse.

Le temps moyen pour retrouver ou construire et partager un triplet $A \longrightarrow \Phi'; \Phi''$ est de l'ordre de n/N, où n est le nombre de triplets antérieurement alloués. Si n n'est pas plus large que N dans de trop grandes proportions, ceci est quasi-instantané.

Pour construire des BDD, nous définissons d'abord la fonction `BDDmake` comme suit : si $\Phi' = \Phi''$ (noter que nous comparons les BDD par adresse et non récursivement par contenu, puisqu'ils sont partagés), alors $\texttt{BDDmake}(A, \Phi', \Phi'')$ retourne Φ'; sinon, $\texttt{BDDmake}(A, \Phi', \Phi'')$ alloue et partage $A \longrightarrow \Phi'; \Phi''$ et retourne son adresse.

La négation $\texttt{BDDneg}(\Phi)$ d'un BDD Φ est définie par récursion structurelle comme suit : $\texttt{BDDneg}(\mathbf{T}) = \mathbf{F}$, $\texttt{BDDneg}(\mathbf{F}) = \mathbf{T}$, $\texttt{BDDneg}(A \longrightarrow \Phi_+; \Phi_-) = \texttt{BDDmake}(A, \texttt{BDDneg}(\Phi_+), \texttt{BDDneg}(\Phi_-))$. Si $\circ$ est un opérateur binaire quelconque, nous définissons l'opération correspondante comme suit. Prenons l'exemple de la disjonction, réalisée par la fonction `BDDor` :

- $\texttt{BDDor}(\mathbf{T}, \Phi) = \mathbf{T}$, $\texttt{BDDor}(\mathbf{F}, \Phi) = \Phi$;
- $\texttt{BDDor}(\Phi, \mathbf{T}) = \mathbf{T}$, $\texttt{BDDor}(\Phi, \mathbf{F}) = \Phi$;
- si $\Phi = A \longrightarrow \Phi_+; \Phi_-$, et $\Phi' = A' \longrightarrow \Phi'_+; \Phi'_-$, alors:
  - si $A < A'$, alors
    $\texttt{BDDor}(\Phi, \Phi') = \texttt{BDDmake}(A, \texttt{BDDor}(\Phi_+, \Phi'), \texttt{BDDor}(\Phi_-, \Phi'))$;
  - si $A > A'$, alors
    $\texttt{BDDor}(\Phi, \Phi') = \texttt{BDDmake}(A', \texttt{BDDor}(\Phi, \Phi'_+), \texttt{BDDor}(\Phi, \Phi'_-))$;
  - si $A = A'$, alors
    $\texttt{BDDor}(\Phi, \Phi') = \texttt{BDDmake}(A, \texttt{BDDor}(\Phi_+, \Phi'_+), \texttt{BDDor}(\Phi_-, \Phi'_-))$.

Le seul problème si nous codons ces fonctions naïvement, est qu'elles tournent en temps environ proportionnel au nombre de *chemins* dans le BDD, et non son nombre de *nœuds*. C'est un point important, parce qu'à cause du partage, les BDD peuvent avoir moins de nœuds qu'ils n'ont de chemins, et ce dans des proportions astronomiques.

Nous codons donc `BDDneg` et `BDDor` en utilisant des *mémo-fonctions*: une mémo-fonction est une fonction f associée à une table de hachage T qui enregistre tous les résultats précédemment calculés par f. Pour être plus précis, T envoie des arguments de f vers les résultats correspondants de f. La mémo-fonction consulte d'abord la table de hachage T, et recherche une entrée correspondant à ses arguments. Si elle en trouve une, elle retourne le résultat enregistré. Sinon, elle appelle la fonction ordinaire f, enregistre le couple (argument, résultat) dans la table T et retourne le résultat. Cette astuce fait tourner `BDDneg` et `BDDor` en temps quasi-linéaire et quasi-quadratique respectivement.

5 Digressions

Les sections suivantes sont techniques et demandent quelques connaissances en théorie de la complexité et en théorie des treillis. Le lecteur peu curieux pourra les laisser de côté en première lecture.

5.1 Pouvoir d'expression de la logique propositionnelle classique

Discutons du pouvoir d'expression de la logique propositionnelle classique.

Tout d'abord, la logique propositionnelle est un formalisme que l'on peut coder sous forme de matériel : on peut construire des circuits en reliant des portes les unes aux autres, chaque porte réalisant un connecteur logique. (Par exemple, une porte NAND est un bout de circuit à deux entrées et une sortie; dans les réalisations positives de la logique, lorsque la tension des deux entrées est au niveau bas, représentant $\bot$, la sortie est haute, et si une entrée est haute, alors la sortie est basse.) Elle ne peut pas décrire des circuits avec des connexions cycliques (comme l'on fait usuellement pour créer des dispositifs instables comme des horloges), mais presque tous les circuits dans un ordinateur contemporain sont des superpositions de portes logiques. En simplifiant, un ordinateur réel est une grosse formule propositionnelle codée sous forme de matériel, plus une horloge (qui envoie des suites de $\top$ et $\bot$ à une vitesse prédéfinie) et des composants d'interface. En somme, la logique propositionnelle est essentiellement suffisamment expressive pour décrire n'importe quel ordinateur réel.

Cet argument est légèrement trompeur, non seulement parce que la formule propositionnelle représentant un ordinateur est gigantesque, mais aussi parce que les ordinateurs réels sont des approximations de ce que nous aimerions qu'un ordinateur soit. Un ordinateur idéal aurait une quantité de mémoire infinie, et en effet la mémoire virtuelle et les appareils de stockage secondaire sont des moyens pour approcher les ordinateurs de cet idéal.

Il y a plusieurs modèles mathématiques des ordinateurs idéaux, et clairement, plus simples ils sont, plus faciles ils seront à manipuler. Un des modèles d'ordinateur idéal les plus simples est la *machine de Turing*. En quelques mots, une machine de Turing est une machine d'états finis (un automate, avec un nombre fini d'états et de transitions entre ces états), plus une tête de lecture/écriture voyageant sur une bande infiniment longue. La bande est une suite linéaire de symboles pris dans un alphabet à au moins deux lettres, et s'étend à l'infini dans les deux directions. La machine de Turing calcule en lisant la lettre sous la tête de lecture/écriture, l'utilise pour décider d'une transition à suivre depuis son état courant jusqu'à l'état suivant, d'une lettre à écrire à la place de la lettre lue, et de la direction du mouvement de la tête (une case en avant ou une case en arrière). La machine procède ainsi jusqu'à ce qu'elle atteigne une case spéciale appelée *état final*. Les machines de Turing représentent des fonctions partielles des mots vers les mots : écrivez le mot de départ sur la bande, entre des délimiteurs spéciaux, le premier délimiteur se situant juste sous la tête; lancer la machine de Turing, et si elle termine, lire le mot écrit sur la bande à la fin de l'exécution.

Les machines de Turing sont universelles, en ce sens que l'ensemble des fonctions qu'elles calculent est exactement l'ensemble de toutes les fonctions calculables. En pratique, on est plus intéressé par savoir quelles fonctions peuvent être calculées sur un ordinateur idéalisé en un temps réaliste, autrement dit quelles fonctions sont *tractables*. Des recherches ont montré qu'un critère suffisamment bon pour définir un "temps réaliste" était un "temps majoré par un polynôme de degré bas en la taille de l'argument d'entrée". En particulier, si la borne supérieure des temps nécessaires à la résolution d'un problème est une exponentielle de la taille de l'entrée, on considère le problème comme intractable. Un résultat fondamental est qu'un problème est tractable sur une machine de Turing si et seulement si il est tractable sur la plupart des autres ordinateurs idéalisés, y compris notre ordinateur avec une mémoire infinie (voir plus haut). L'étude de la tractabilité et de l'intractabilité des problèmes est le sujet de la théorie abstraite de la complexité [GJ79].

Le pouvoir expressif de la logique propositionnelle est alors capturée par le théorème suivant, dû à Stephen Cook en 1971. Supposons que nos machines de Turing ont deux états finaux "oui" et "non"; nous lui soumettrons des questions de la forme "la formule Φ est-elle satisfiable ?", et nous lirons la réponse dans l'état dans lequel la machine termine. De plus, les machines de Turing peuvent fonctionner soit de façon déterministe (la donnée de l'état courant et de la lettre lue détermine complètement l'état suivant, la lettre à écrire et la direction du mouvement) ou de façon non déterministe (l'état courant et la lettre lue déterminent plusieurs actions possibles différentes). Une machine de Turing déterministe accepte son argument d'entrée si elle termine dans l'état "oui"; une machine de Turing non déterministe accepte son entrée s'il existe un chemin d'exécution qui termine dans l'état "oui". Remarquer que l'on définit les problèmes de décision comme étant des fonctions à valeurs booléennes :

Théorème 41 (Cook) *Soit SAT le problème de décision suivant :*
ENTRÉE : une formule propositionnelle Φ,
QUESTION : Φ est-elle satisfiable ?
De façon équivalente, SAT est une fonction qui associe vrai à Φ si et seulement si Φ est satisfiable.

Soit NP la classe des problèmes de décision f tels que $f(x)$ est vrai si et seulement si il existe une machine de Turing non déterministe qui accepte l'entrée x en temps polynomial en la taille de x. (En bref, NP est la classe des problèmes résolubles "en temps polynomial non déterministe".)

Alors SAT est NP-complet, c'est-à-dire :

- *SAT est dans NP,*
- *et pour tout problème f dans NP, il existe un algorithme en temps polynomial qui traduit l'entrée x de f en une entrée Φ de SAT, de sorte que Φ soit satisfiable si et seulement si $f(x)$ est vrai.*

Ce théorème peut être lu de deux façons. La première est que NP est (probablement, cela n'a été ni prouvé ni contredit) beaucoup plus grande que la classe P des problèmes tractables. En particulier, NP contient de nombreux problèmes que l'on ne sait pas, à l'heure actuelle, résoudre en un temps moins qu'exponentiel en général. Le fait que SAT soit NP-difficile (la deuxième condition dans la définition ci-dessus de la NP-complétude) signifie que SAT est au moins aussi difficile que tous ces problèmes.

La deuxième lecture du théorème est la suivante. Étant donnée la représentation d'une machine de Turing non déterministe M avec son argument d'entrée déjà codé sur sa bande, nous voulons savoir si M accepte son argument. Nous pourrions exécuter M pour le découvrir, mais la preuve du théorème de Cook construit en fait une formule propositionnelle Φ qui est satisfiable si et seulement si M accepte. En somme, le problème de la satisfiabilité en logique propositionnelle a exactement le même pouvoir d'expression que les machines de Turing non déterministes en temps polynomial.

Ceci nous permet de quantifier le pouvoir d'expression de la logique propositionnelle de façon plus précise : NP est très certainement plus expressive que P (autrement dit, nous pouvons exprimer des problèmes, et même des problèmes non tractables en logique propositionnelle); d'un autre côté, NP est aussi très certainement beaucoup plus petite que DEXPTIME, la classe de tous les problèmes que l'on peut résoudre en temps exponentiel d'un polynôme de la taille de l'entrée sur une machine déterministe (et donc, il y a de nombreux problèmes intractables, solubles en temps exponentiel comme SAT, mais qui ne peuvent très certainement pas être exprimés comme la satisfiabilité d'une formule propositionnelle fabricable en temps polynomial).

5.2 Algèbres booléennes

Au lieu d'utiliser des systèmes de déduction pour déterminer si une formule propositionnelle est valide (ou satisfiable), il est parfois pratique d'utiliser une théorie équationnelle de l'équivalence logique, et d'essayer de dériver **T** en remplaçant les équivalents dans cette théorie.

Le langage des formules à équivalence propositionnelle classique près est en fait celui des algèbres booléennes. D'abord, nous définisson les treillis booléens :

Définition 42 *Un* treillis *est un ensemble non vide L muni d'un ordre partiel $\leq$ tel que tout sous-ensemble fini S de L a une borne supérieure $\bigvee S$ et une borne inférieure $\bigwedge S$.*

Nous notons $\top = \bigwedge \emptyset$ (le plus grand élément de L), $\bot = \bigvee \emptyset$ (le plus petit élément de L), $a \wedge b$ au lieu de $\bigwedge\{a, b\}$, $a \vee b$ au lieu de $\bigvee\{a, b\}$.

Nous disons que L est un treillis distributif *si et seulement si $\wedge$ distribue par rapport à $\vee$, et $\vee$ distribue par rapport à $\wedge$, c'est-à-dire $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ et $a \wedge (b \vee c) = (a \wedge b) \wedge (a \wedge c)$.*

Nous disons qu'un treillis distributif est un treillis booléen *s'il est muni d'une opération de* complémentation *$\neg$ telle que $a \vee \neg a = \top$, $a \wedge \neg a = \bot$, et $\neg\neg a = a$.*

Johnstone ([Joh92], p.7) résume élégamment l'essence des algèbres booléennes en disant que, en somme, leurs axiomes sont "tout ce que vous pouvez dire d'un ensemble à deux éléments [dans le langage de l'algèbre universelle]". En effet :

Définition 43 *Une* algèbre booléenne *est un ensemble non vide A contenant deux éléments $\top$, $\bot$, et muni d'opérations $\wedge$, $\vee$, $\neg$ obéissant aux équations suivantes :*

- $a \wedge b = b \wedge a$, $a \wedge (b \wedge c) = (a \wedge b) \wedge c$, $a \wedge a = a$, $a \wedge \top = a$, $a \wedge \bot = \bot$,
- $a \vee b = b \vee a$, $a \vee (b \vee c) = (a \vee b) \vee c$, $a \vee a = a$, $a \vee \top = \top$, $a \vee \bot = a$,
- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$, $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$,
- $a \wedge \neg a = \bot$, $a \vee \neg a = \top$, $\neg\neg a = a$,
- $\neg(a \vee b) = \neg a \wedge \neg b$, $\neg(a \wedge b) = \neg a \vee \neg b$.

Tout treillis booléen définit clairement une algèbre booléenne. Réciproquement, toute algèbre booléenne décrit un treillis booléen si l'on définit $a \leq b$ comme $\neg a \vee b = \top$ (ou de façon équivalente, $a \wedge b = a$, ou encore $a \vee b = b$).

Le résultat de complétude (que nous ne prouverons pas) est qu'une formule Φ est valide si et seulement si l'on peut transformer Φ en $\top$ en un nombre fini d'étapes de réécriture utilisant les équations ci-dessus.

Comme les algèbres booléennes sont équivalentes aux treillis booléens, nous pouvons aussi prouver des formules en appliquant les lois des treillis booléens à la place. Ceci fournit parfois des preuves plus courtes à la main. À notre connaissance, ceci n'a pas été approfondi dans le domaine de la démonstration automatique.

Une autre idée est d'utiliser des *anneaux booléens* au lieu d'algèbres booléennes. Ceci revient à axiomatiser $\wedge$, $\oplus$ (le ou exclusif, c'est-à-dire la négation de $\Leftrightarrow$), $\mathbf{T}$ (vrai) et $\mathbf{F}$ (faux), et à les utiliser au lieu de $\wedge$, $\vee$ et $\neg$. Les axiomes peuvent alors être orientés sous forme de règles de réécriture [Hsi85], modulo associativité et commutativité de $\wedge$ et $\oplus$. Les règles de traduction sont :

- $\neg\Phi \longrightarrow \Phi \oplus \mathbf{T}$,
- $\Phi \vee \Phi' \longrightarrow (\Phi \wedge \Phi') \oplus \Phi \oplus \Phi'$,
- $\Phi \Rightarrow \Phi' \longrightarrow (\Phi \wedge \Phi') \oplus \Phi \oplus \mathbf{T}$,
- $\Phi \Leftrightarrow \Phi' \longrightarrow \Phi \oplus \Phi' \oplus \mathbf{T}$.

Et les règles de simplification sont :

- $\Phi \oplus \mathbf{F} \longrightarrow \Phi$,
- $\Phi \oplus \Phi \longrightarrow \mathbf{F}$,
- $\Phi \wedge (\Phi' \oplus \Phi'') \longrightarrow (\Phi \wedge \Phi') \oplus (\Phi \wedge \Phi'')$,
- $(\Phi \oplus \Phi') \wedge \Phi'' \longrightarrow (\Phi \wedge \Phi'') \oplus (\Phi' \wedge \Phi'')$,
- $\Phi \wedge \mathbf{T} \longrightarrow \Phi$,
- $\Phi \wedge \mathbf{F} \longrightarrow \mathbf{F}$,
- $\Phi \wedge \Phi \longrightarrow \Phi$.

La forme normale qui en résulte (modulo associativité et commutativité de $\wedge$ et $\oplus$) est appelée *forme normale de Reed-Muller* de la formule. Comme pour les BDD, elle est canonique, et peut être construite depuis le bas, comme suit.

On code la forme normale de Reed-Muller d'une formule sous forme d'un ensemble d'ensembles de variables. (Si on se donne un ordre sur les variables, on peut représenter un ensemble de variables sous forme d'une liste ordonnée de ces variables sans duplication, et un ensemble d'ensembles de variables sous forme d'une liste de ces listes, ordonnée lexicographiquement par exemple.) Les ensembles internes de variables sont vus comme des conjonctions de leurs variables, et l'ensemble externe est vu comme le ou exclusif de ses éléments.

$\mathbf{F}$ est alors l'ensemble vide $\emptyset$, et $\mathbf{T}$ est le singleton $\{\emptyset\}$. Une variable A est codée sous forme de $\{\{A\}\}$. Le ou exclusif $\Phi \oplus \Phi'$ de deux formes de Reed-Muller Φ et Φ' est calculé comme la différence symétrique $(\Phi \setminus \Phi') \cup (\Phi' \setminus \Phi)$ des deux ensembles. La conjonction $\Phi \wedge \Phi'$ de deux formes de Reed-Muller Φ et Φ' est la différence symétrique de tous les $\{a \cup a'\}$, où $a \in \Phi$ et $a' \in \Phi'$. Notons finalement que la forme normale de Reed-Muller peut se coder sous une forme très proche des BDD, en utilisant les *ZBDD* (*zero-suppressed BDD*) de Shin-Ichi Minato [Min93], qui sont des représentations compactes d'ensembles d'ensembles.

5.3 Logique propositionnelle quantifiée

Il est parfois intéressant d'ajouter des quantificateurs au langage de la logique propositionnelle. Ceci n'ajoute pas de pouvoir d'expression, mais peut raccourcir de façon significative certaines formules propositionnelles.

Si Φ est une formule, et A est une variable propositionnelle, alors nous définissons $\forall A \cdot \Phi$ ("pour tout A, Φ") comme $\Phi[\mathbf{T}/A] \wedge \Phi[\mathbf{F}/A]$, où $\mathbf{T}$ et $\mathbf{F}$ sont deux constantes nouvelles telles que $[\![\mathbf{T}]\!]\rho = \top$ et $[\![\mathbf{F}]\!]\rho = \bot$, comme dans les BDD. De même, $\exists A \cdot \Phi$ ("il existe A tel que Φ") se définit comme $\Phi[\mathbf{T}/A] \vee \Phi[\mathbf{F}/A]$.

Nous pourrions aussi bien définir $\forall A \cdot \Phi$ et $\exists A \cdot \Phi$ directement en leur donnant une sémantique (dérivée des formules ci-dessus), et en étendant les systèmes de déduction pour traiter les nouveaux opérateurs. Par exemple, nous pourrions étendre $\mathbf{LK}_0$ en ajoutant les règles de déduction suivantes :

$$\frac{\Gamma, \Phi[\Phi'/A] \longrightarrow \Delta}{\Gamma, \forall A \cdot \Phi \longrightarrow \Delta}\,\forall\mathrm{L} \qquad \frac{\Gamma \longrightarrow \Delta, \Phi}{\Gamma \longrightarrow \Delta, \forall A \cdot \Phi}\,\forall\mathrm{R}\ (\text{si } A \notin \mathrm{fv}(\Gamma) \cup \mathrm{fv}(\Delta))$$

$$\frac{\Gamma, \Phi \longrightarrow \Delta}{\Gamma, \exists A \cdot \Phi \longrightarrow \Delta}\,\exists\mathrm{L}\ (\text{si } A \notin \mathrm{fv}(\Gamma) \cup \mathrm{fv}(\Delta)) \qquad \frac{\Gamma \longrightarrow \Delta, \Phi[\Phi'/A]}{\Gamma \longrightarrow \Delta, \exists A \cdot \Phi}\,\exists\mathrm{R}$$

et ceci produirait exactement la même logique.

La logique propositionnelle quantifiée se code facilement dans les BDD : remplacer A par $\mathbf{T}$ (resp. $\mathbf{F}$) revient à remplacer tous les sous-BDD de la forme $A \longrightarrow \Phi_+; \Phi_-$ par Φ_+ (resp. Φ_-), et à réduire le BDD. On code alors $\forall$ et $\exists$ en prenant la conjonction ou la disjonction des BDD résultants.

L'utilisation de la logique propositionnelle quantifiée est commune en vérification de matériel, par exemple. Elle est en fait un ingrédient essentiel de la définition des procédures de vérification de modèle ("model-checking" en anglais), que nous examinerons plus tard.

Noter que nous ne sommes pas limités à une seule alternance de quantificateurs, et que nous pouvons exprimer des propositions de la forme $Q_1 A_1 \cdot \ldots Q_n A_n \cdot \Phi$, où les Q_i sont soit $\forall$ soit $\exists$. Si nous devions exprimer les quantifications sous forme de conjonctions ou de disjonctions, ceci expanserait la formule Φ exponentiellement en général, ce qui veut dire qu'il serait déjà infaisable d'*écrire* la formule sans utiliser de quantificateurs. Sur les BDD, on peut le faire en général, parce que dans les bons cas, le partage et la réduction contribuent à faire que le BDD reste suffisamment petit, bien que ce ne soit pas toujours le cas.

En ce qui concerne la notion de pouvoir d'expression esquissée en section 5.1, la logique propositionnelle quantifiée est aussi expressive que les *machines de Turing en espace polynomial.* Plus précisément, vérifier une formule propositionnelle quantifiée est un problème PSPACE-complet, où PSPACE est la classe des problèmes de décision que l'on peut vérifier en temps fini sur une machine de Turing (déterministe ou non, ici cela n'a pas d'importance) en espace polynomial (c'est-à-dire en n'utilisant qu'un nombre polynomial de cases sur la bande). Il y a (très probablement) beaucoup plus de problèmes dans PSPACE que dans NP, ce qui suggère que les problèmes PSPACE-complets présentent davantage de difficultés que les problèmes de NP. Cependant, PSPACE est encore à l'intérieur de DEXPTIME, ce qui veut dire que nous pouvons encore les résoudre en temps exponentiel.

References

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of* NP*-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[Hsi85] Jieh Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25(1):255–300, 1985.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[Joh92] Peter T. Johnstone. *Notes on Logic and Set Theory.* Cambridge University Press, 1992.

[Knu73] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[Min93] Shin-Ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, Dallas, TX, June 1993. ACM Press.