

Informatique théorique : mathématiques discrètes et logique mathématique

Marc Aiguier

Pascale Le Gall

16 septembre 2024

Table des matières

I	Mathématiques discrètes	7
1	Introduction	9
Sous partie 1	Théorie de l'ordre et treillis	11
2	Introduction	13
3	Relations d'ordre, ensembles ordonnés, ensembles bien fondés	15
3.1	Relations d'ordre et ensembles ordonnés	15
3.2	Éléments remarquables : chaînes et antichaînes, majorants et minorants, et ensembles inductifs	17
3.3	Ensembles bien fondés et induction mathématique	19
	Ensembles bien fondés	19
	Induction mathématique	22
	Induction sans ordre bien fondé : lemme de Zorn et ses avatars	23
4	Treillis, définitions inductives et preuve par induction structurelle	27
4.1	Treillis et points fixes	27
	Définition	27
	Treillis complets et fonctions continues	28
4.2	Définitions inductives et induction structurelle	31
	Définitions inductives	31
	Preuve par induction structurelle	33
	Définition récursive d'une fonction	34
4.3	Application : sémantique dénotationnelle des langages de programmation	36
	Un langage de programmation simple	36
	Sémantique dénotationnelle du langage <i>WHILE</i>	38
Sous partie 2	Calculabilité	43
5	Introduction	45
6	Fonctions primitives récursives et récursives	47
6.1	Codage	47
6.2	Les fonctions primitives récursives	49
	Définition	49
	Ensemble primitif récursif	50
6.3	Les fonctions récursives	51
	Récursion primitive et algorithmique	51
	Minimisation non bornée	54

6.4	Un interpréteur de programmes	57
	Indécidabilité du problème de l'arrêt	59
6.5	Thèse de Church	61
7	Les machines de Turing	63
7.1	Définitions	63
7.2	Les fonctions T-calculables	65
7.3	Machines de Turing universelles	70
7.4	Exemples de problèmes indécidables	71
	Problème du mot dans les systèmes semi-Thuien	71
	Correspondance de Post	72
	Théorème de Rice	74
8	Complexité	77
8.1	Complexité en temps	77
8.2	Les classes P et NP	78
8.3	Structure de la classe NP	81
	Problèmes NP-complets et NP-durs	81
	Un premier problème NP-complet	84
Sous partie 3	Théorie des langages	89
9	Introduction	91
10	Langages rationnels	93
10.1	Monoïde	93
10.2	Langages réguliers et automates	94
	Définitions	94
	Lemme de pompage et langage réguliers	97
	Expressions régulières	98
10.3	Opérations sur les automates	99
	Déterminisation d'un automate	99
	Minimisation d'un automate	100
11	Langages algébriques	103
11.1	Grammaire algébrique	103
	Définitions	103
11.2	Formes normales de grammaires	105
	Algorithme CYK	108
11.3	Arbre de dérivation et itération	108
11.4	Propriétés de clôture	111
11.5	Automates à pile	112
II	Logiques mathématiques	117
12	Introduction	119
12.1	Logique mathématique et informatique	119
12.2	Structure d'une logique	120

13 Systèmes formels	123
13.1 Définition	123
13.2 Dédutions	124
13.3 Théorèmes et arbres de preuve	125
13.4 Correction et complétude des systèmes formels au regard d'une interprétation . .	127
Propriétés syntaxiques, algorithmiques	127
Propriétés sémantiques	127
Sous partie 4 Logique propositionnelle	129
14 Introduction	131
15 Logique propositionnelle : syntaxe	133
15.1 Formules propositionnelles	133
15.2 Quelques fonctions définies sur les formules propositionnelles	134
16 Sémantique	137
16.1 Validation des formules	137
16.2 Quelques propriétés remarquables	139
16.3 Conséquence sémantique	141
16.4 Théorème de la compacité	141
16.5 Décidabilité de la logique propositionnelle	142
16.6 Méthode des tableaux	144
Formules α et formules β	144
Tableaux	145
Construction d'un tableau	146
Exemple de tableau	147
Correction de la méthode des tableaux	148
Complétude de la méthode des tableaux	149
17 Calculs pour la logique propositionnelle	151
17.1 Système de preuves à la Hilbert	151
17.2 Système de la déduction naturelle	152
17.3 Calcul des séquents	153
Quelques résultats utiles.	155
17.4 Calcul des séquents (complet)	159
18 Méthodes algorithmiques	161
18.1 Mise sous forme normale	161
Rappel : Formules équivalentes	161
Formules équisatisfiables	164
18.2 Algorithme DPLL	164
Valuations partielles	165
Stratégie de choix des variables et des valeurs booléennes	166
Algorithme DPLL	167
Vers les SAT solveurs	168
18.3 Résolution	168
Quelques rappels et notations	168
Satisfiabilité des formules de Horn	170

Résolvante et preuve par résolution	171
Complétude de la réfutation par résolution	173
Sous partie 5 Logique des prédicats	177
19 Introduction	179
20 Logique des prédicats : Syntaxe	181
20.1 Signatures	181
20.2 Termes et formules	182
21 Sémantique	185
21.1 Modèles ou structures du premier ordre	185
21.2 Validation des formules	186
21.3 Vers une procédure de semi-décision	189
Formules prénexes et universelles	189
Théorème d'Herbrand	190
Skolémisation	191
21.4 Indécidabilité de la logique des prédicats du premier ordre	192
21.5 Pouvoir d'expression de la logique des prédicats	193
Logique des prédicats avec égalité	193
Théorème de Lowenheim et Skolem	194
22 Calcul pour la logique des prédicats	197
22.1 Calcul à la Hilbert	197
Correction et complétude	198
22.2 Dédution naturelle	202
22.3 Calcul des séquents	202
Correction et complétude du calcul des séquents	203
Élimination des coupures	205
22.4 Résolution	210
Problème d'unification	210
Calcul par résolution	215
Langage Prolog	217
22.5 Méthode des tableaux	217
23 Logique équationnelle et réécriture	219
23.1 Introduction	219
23.2 Logique équationnelle	219
Syntaxe	219
Sémantique	226
Le calcul équationnel	229
23.3 Réécriture	237
Systèmes de réécriture et leurs propriétés	237
Décider de la confluence	242
Complétion de Knuth-Bendix	245
24 Logique de Hoare	249
24.1 Correction totale et partielle d'un algorithme	249

Terminaison des algorithmes	249
Correction partielle	252
24.2 Logique de Hoare : définition	253
Langage While	254
Sémantique	255
Calcul de Hoare	255
Calcul de précondition la plus faible	257
Sous partie 6 Logique modale	261
25 Introduction	263
26 Logique modale : syntaxe et sémantique	265
26.1 Syntaxe	265
26.2 Sémantique : modèle et satisfaction	265
26.3 Propriété de modèle fini et décidabilité	268
Propriété de modèle fini	268
Décidabilité de la validation	270
26.4 Traduire la logique modale dans la logique des prédicats	271
26.5 Bissimulation	272
Introduction	272
Définition	273
Logique modale et bissimulation	276
27 Le μ-calcul	279
27.1 Introduction	279
27.2 Syntaxe et sémantique	280
27.3 Quelques exemples de formules à point fixe	282
28 Calculs pour la logique modale	285
28.1 Calcul à la Hilbert	285
28.2 Calcul des séquents	290
28.3 Méthode des tableaux	291
29 Application : raisonnement spatial qualitatif	293
29.1 RCC-8	293
29.2 Le modèle 9-intersection	294
Sémantique topologique	295
Formalisation du modèle 9-intersection	296

Première partie .

Mathématiques discrètes

1. Introduction

Comme toute discipline scientifique, l'informatique fait appel aux mathématiques pour formaliser et modéliser les objets qu'elle manipule tels que les machines, programmes et systèmes. À l'inverse des sciences de la nature (physique, mécanique, thermique, etc.), les mathématiques que l'informatique utilise ne sont pas exactement les mêmes. Elles se font "discrètes". Le sens de l'adjectif "*discret*" est sans rapport avec son sens courant. Il signifie la présence d'objets énumérables, c'est-à-dire que l'on peut construire par des procédés récurrents, les seules choses que savent manipuler les ordinateurs.

Nous allons donc aborder dans cette partie les principes fondamentaux et les outils formels (i.e. mathématiquement fondés) à la base de toutes les méthodes de conception et d'implantation des systèmes informatiques. Il sera alors abordé dans cette partie les notions fondamentales aux fondements :

- de l'induction et la récurrence (théorie des treillis, ensemble bien-fondé et équivalence avec l'induction mathématique). L'objectif est de formaliser les notions fondamentales d'induction et récursivité sous-jacentes à toutes les mathématiques discrètes.
- de l'algorithmique (fonctions récursives de Godel/Herbrand, machine de Turing et tous les théorèmes montrant l'équivalence de ces modèles de calcul ainsi que les résultats d'indécidabilité associés). L'idée est de définir formellement (i.e. mathématiquement) ce qu'est un problème de décision et donner une dénotation formelle à la notion d'algorithme (thèse de Church).
- de la théorie de la complexité (classe de complexité P et NP, et structuration de la classe NP - problèmes NP-complets et NP-durs).
- de la conception des systèmes (langages rationnels et algébriques, automates et automates à pile). L'objectif est d'étudier un outil formel à la base de toutes les méthodes de modélisation des systèmes informatiques, de l'étude des langues naturelles ou encore de la compilation.

Cette partie est composé de trois parties qui sont les suivantes :

1. Fondements de l'induction et la récursivité. On y abordera plus particulièrement les notions de :
 - Ordre et préordre ;
 - Majorants et minorants ;
 - Ensembles bien fondés et induction ;
 - Treillis et points fixes (treillis complets et fonctions continues, points fixes et fonctions monotones) avec une application à la sémantique des langages de programmation.
2. Fondements de l'algorithmique (Fonctions calculables et problèmes décidables, et théorie de la complexité). On verra plus précisément :
 - Les fonctions récursives (FR) : Fonctions primitives récursives, Fonctions récursives (Problème de la fonction d'Ackermann), définition d'un problème décidable, indécidabilité du problème de l'arrêt, Fonction récursive universelle (interpréteur) ;
 - Le calcul pas à pas (autre modèle de calcul) : Les machines de Turing (MT), théorèmes d'équivalence ($FR \equiv MT$), thèse de Church ;

- Théorie de la complexité (complexité en temps, classes P et NP, structuration de la classe NP - problèmes NP-complets et NP-durs, un premier problème NP-complet).
3. Langages rationnels et algébriques, automates et automates à pile. On abordera dans cette dernière partie les notions suivantes :
- Monoïde libre ;
 - Langages rationnels (lemme du pompage), et caractérisation par automates finis ;
 - Opérations sur les automates finis (complétion, détermination, minimalité, opérations algébriques standards) ;
 - Langages algébriques (lemme de pompage) et caractérisation par automates à pile ;
 - Automates à pile déterministes et propriétés.

SOUS PARTIE 1

Théorie de l'ordre et treillis

2. Introduction

Il y a un type de relations qui jouent un rôle très important en informatique (mais aussi en mathématiques) : Les *relations d'ordre*. La raison en est simple : les relations d'ordre sont fortement liées à la notion d'induction (terme générique regroupant à la fois la définition inductive d'un ensemble et la preuve par induction) qui comme nous le verrons dans la seconde partie de ce document fonde la notion de calculabilité. En effet, le fait d'utiliser l'induction est fortement lié à une propriété particulière des relations d'ordre : ordres bien fondés.

De plus, sur les relations d'ordre, on peut définir une structure algébrique, les *treillis*, qui ont montré leur importance dans la formalisation de l'induction. En effet, les treillis permettent de définir des endofonctions $\mathcal{F}$ (i.e. des fonctions ayant pour domaine et co-domaine le même treillis) possédant des points fixes (i.e. les solutions satisfaisant une équation de la forme $\mathcal{F}(x) = x$). Quand ils existent, l'ensemble des points fixes de $\mathcal{F}$ peut même être ordonné et ainsi dénoter le plus petit (aussi le plus grand) point fixe dont on montrera encore le lien avec l'induction.

Dans cette sous-partie, nous allons formaliser cette notion importante d'induction, i.e. de définition inductive et de preuve par induction. Toutes ces notions bien que supposées "innées" chez la plupart des scientifiques (sans doute à tort), sont rarement (voire jamais) exposées de manière détaillée dans la plupart des livres. Nous profitons de ce cours pour les présenter formellement et permettre ainsi d'appréhender la construction récursive d'objets finis (structures de données, langages des mots finis reconnus par un automate, ensemble des chemins finis dans un graphe - plus petit point fixe) et infinis (flots de données, langages des mots infinis reconnus par un automate, ensemble des chemins infinis dans un graphe - plus grand point fixe) que l'on a l'habitude de manipuler en informatique. Nous commencerons alors par rappeler quelques notions mathématiques de base sur les relations d'ordre. Puis, nous définirons au-dessus de ces relations d'ordre la structure algébrique de treillis ainsi que les notions de complétude et de fonctions continues à partir desquels nous obtiendrons deux théorèmes de point fixe. Sur ces théorèmes de point fixe, nous aborderons les deux notions fondamentales de définition inductive et de preuve par induction. Pour illustrer les treillis, nous aborderons aussi la sémantique des langages de programmation. Nous verrons là encore, l'intérêt des théorèmes de point fixe pour dénoter le comportement des boucles et programmes récursifs, ce qui renforcera encore le lien entre récursivité et calculabilité que nous détaillerons dans la seconde sous-partie de ce partie sur les mathématiques discrètes.

3. Relations d'ordre, ensembles ordonnés, ensembles bien fondés

3.1. Relations d'ordre et ensembles ordonnés

Définition 1 (Relation binaire).

Une **relation binaire** r sur un ensemble E est un sous-ensemble de $E \times E$. Dans la suite, on utilisera la notation infixe pour les relations binaires : pour tout $(x, y) \in r$, on notera plus simplement $x r y$. De plus, $x r y$ sera appelée une **séquence de réduction**. Étant donnée une relation binaire r , son **inverse**, notée r^{-1} , est la relation binaire sur E définie par : $\forall x \in E, \forall y \in E, x r y \Leftrightarrow y r^{-1} x$.

De même, on note r^0 et r^+ les fermetures réflexive et transitive de r . Ainsi, $r^0 = r \cup \{(x, x) | x \in E\}$, et $r^+ = \bigcup_{i \in \mathbb{N}} r_i$ où :

$$r_0 = r$$

$$\forall i, i \geq 1, r_i = r_{i-1} \cup \{(x, z) | \exists y \in E, (x, y) \in r_{i-1}, (y, z) \in r_{i-1}\}.$$

On note alors $r^* = r^0 \cup r^+$.

Parmi, les relations binaires, nous sommes principalement intéressés par celles définissant un ordre sur E .

Définition 2 (Relation d'ordre).

Soit E un ensemble. Une **relation d'ordre** $\preceq$ sur E est une relation binaire sur E satisfaisant aux conditions suivantes : pour tout $e, e', e'' \in E$

- **réflexivité** : $e \preceq e$
- **anti-symétrie** : si $e \preceq e'$ et $e' \preceq e$ alors $e = e'$
- **transitivité** : si $e \preceq e'$ et $e' \preceq e''$ alors $e \preceq e''$

$\preceq$ est dite **relation d'ordre strict**, et dans ce cas-là est notée $\prec$, si, et seulement si elle est irréflexive (i.e. $\forall x \in E, \text{non}(x \prec x)$) et transitive, et elle est dite **totale** si elle satisfait la condition suivante :

$$\forall (a, b) \in E \times E, a \preceq b \text{ ou } b \preceq a$$

L'ordre est dit **partiel**, s'il n'est pas total.

On note $\succeq$ (resp. $\succ$) la relation inverse de $\preceq$ (resp. $\prec$).

Exemple 1.

- L'ordre habituel sur les réels est un ordre total.
- La relation de divisibilité $\leq_{div}$ sur les entiers naturels définie par :

$$\forall a, b \in \mathbb{N}, a \leq_{div} b \iff \exists c \in \mathbb{N}, b = a.c$$

est un ordre partiel.

- La relation d'inclusion est un ordre partiel sur $\mathcal{P}(E)$ ^a si $|E| > 1$, et un ordre total si $|E| \leq 1$.

a. $\mathcal{P}(E)$ est l'ensemble des parties d'un ensemble E , et $|E|$ désigne la cardinalité de l'ensemble E .

Définition 3 (Ensemble ordonné).

Un **ensemble ordonné** $(E, \preceq)$ est un ensemble E muni d'une relation d'ordre $\preceq$.

Un même ensemble peut être muni de plusieurs relations d'ordre. Par exemple, sur $\mathbb{N}$, on a la relation d'ordre usuelle $\leq$ ainsi que l'ordre de divisibilité défini plus haut.

Définition 4 (Applications monotones).

Soient $(E_1, \preceq_1)$ et $(E_2, \preceq_2)$ deux ensembles ordonnés. Une application $f : E_1 \rightarrow E_2$ est dite **monotone** si, et seulement si :

$$\forall x, y \in E_1, x \preceq_1 y \implies f(x) \preceq_2 f(y)$$

On dira de plus que f est un **mono (resp. épi)morphisme** si f est injective (resp. surjective).

Enfin, f est un **isomorphisme** si f est une bijection et f^{-1} est aussi monotone.

Exemple 2.

L'application identité de $(\mathbb{N}, \leq_{div})$ dans $(\mathbb{N}, \leq)$ est monotone.

Les applications monotones définissent ainsi les homomorphismes entre les ensembles ordonnés.

Remarque 3.

Il ne suffit pas qu'une bijection entre deux ensembles ordonnés soit monotone pour être un isomorphisme. En effet, l'application identité de $(\mathbb{N}, \leq_{div})$ dans $(\mathbb{N}, \leq)$ est une bijection monotone, mais n'est pas un isomorphisme. La raison vient du fait que $\leq$ est totale tandis que $\leq_{div}$ est partielle.

3.2. Éléments remarquables : chaînes et antichaînes, majorants et minorants, et ensembles inductifs

Définition 5 (Sous-ensemble ordonné).

Soit $(E, \preceq)$ un ensemble ordonné. Un **sous-ensemble ordonné** de $(E, \preceq)$ est un ensemble ordonné $(E', \preceq')$ tel que $E' \subseteq E$ et $\preceq' = \preceq \cap (E' \times E')$.

Définition 6 (Chaîne et antichaîne).

Soit $(E, \preceq)$ un ensemble ordonné.

- On appelle **chaîne** de $(E, \preceq)$ tout sous-ensemble totalement ordonné $(X, \preceq_X)$ de $(E, \preceq)$.
- On appelle **antichaîne** de $(E, \preceq)$ tout sous-ensemble X de E tel que $(X \times X) \cap \preceq = Id_X$ où Id_X est la relation identité sur X (i.e. $(\forall x, y \in X, x \preceq y \Rightarrow x = y)$).

Si $(X, \preceq_X)$ est une chaîne (resp. X est une antichaîne), nous dirons que $(X, \preceq_X)$ (resp. X) est **maximale** si pour toute chaîne $(Y, \preceq_Y)$ (resp. antichaîne Y) de $(E, \preceq)$, on a : $X \subset Y \Rightarrow X = Y$.

Ainsi, dans une antichaîne, deux éléments quelconques sont incomparables.

Exemple 4.

Soit $E = \mathcal{P}(\{a, b, c\})$ muni de l'inclusion.

- $X = \{\{a\}, \{a, b\}, \{a, b, c\}\}$ est une chaîne non maximale.
- $X = \{\{a\}, \{b, c\}\}$ est une antichaîne.
- $X = \{\emptyset, \{a\}, \{a, b\}, \{a, b, c\}\}$ est une chaîne maximale.
- $X = \{\{a\}, \{b\}, \{c\}\}$ est une antichaîne maximale.
- $X = \{\{a\}\}$ est à la fois une chaîne et une antichaîne non maximale.

Définition 7 (Majorant, minorant, plus grand et plus petit élément).

Soient $(E, \preceq)$ un ensemble ordonné, $X \subseteq E$, et $\alpha \in E$. On dit que α est :

- un **majorant** (resp. **minorant**) de X si :

$$\forall x \in X, x \preceq \alpha \text{ (resp. } \alpha \preceq x)$$

Notons $Maj(X)$ (resp. $Min(X)$) l'ensemble des majorants (resp. des minorants) de X .

- un élément **maximal** ou **plus grand élément** (resp. **minimal** ou **plus petit élément**) de X , si $\alpha \in X$ et :

$$\forall x \in X, \alpha \preceq x \Rightarrow \alpha = x \text{ (resp. } x \preceq \alpha \Rightarrow \alpha = x)$$

- une **borne supérieure** (resp. **inférieure**) de X s'il est le plus petit des majorants (resp. le plus grand des minorants), i.e. si m est un majorant (resp. un minorant) de X , alors on a $\alpha \preceq m$ (resp. $m \preceq \alpha$).

Si X admet une borne supérieure (resp. inférieure), cette dernière est unique et on la note $Sup(X)$ (resp. $Inf(X)$).

Proposition 5.

Soit $(E, \preceq)$ un ensemble ordonné. Soit $X \subseteq E$. $Maj(X) \cap X$ et $Min(X) \cap X$ ont au plus un élément.

Démonstration : Supposons que $Maj(X) \cap X$ a deux éléments x et y . Par définition, on a alors $x \preceq y$ et $y \preceq x$, et donc par anti-symétrie, $x = y$.

La démonstration est analogue pour Min . ■

Si $Maj(X) \cap X$ n'est pas vide, l'unique élément est appelé **maximum** de X . De même, si $Min(X) \cap X$ n'est pas vide, l'unique élément est appelé **minimum** de X .

Remarque 6.

Tout ordre totalement ordonné, quand il possède un élément maximal (resp. minimal), ce dernier est unique, et donc est maximum (resp. minimum).

Comme simple conséquence de l'anti-symétrie des relations d'ordre, si un sous-ensemble X de $(E, \preceq)$ admet un plus grand ou un plus petit élément, il est unique. De même si α est une borne supérieure (resp. inférieure), elle est unique.

Définition 8 (Ensemble inductif).

Un ensemble ordonné $(E, \preceq)$ est dit **inductif** si chacune de ses chaînes admet un majorant.

Les ensembles inductifs ne doivent pas être confondus avec les ensembles définis de façon explicite inductivement et dont on verra les fondements mathématiques dans le chapitre sur les treillis. Néanmoins, un lien unit tout ça, l'induction mathématique. Nous verrons alors que pour les ensembles définis inductivement, ce principe de preuve peut être explicitement appliqué, tandis que dans le cas des ensembles inductifs, on peut contourner ce principe pour démontrer une propriété sur tous les éléments de l'ensemble, en utilisant un lemme puissant de la théorie des ensembles, le lemme de Zorn (cf. Section 3.3).

3.3. Ensembles bien fondés et induction mathématique

Ensembles bien fondés

Certaines relations d'ordre possèdent une propriété caractéristique, celle d'être bien-fondée ou encore Nothérienne. L'intérêt de telles relations est qu'elles permettent de raisonner par induction généralisée sur les éléments de l'ensemble.

Définition 9 (Ordre bien-fondé et bon ordre).

Soit E un ensemble. Une relation d'ordre $\preceq$ sur E est dite **bien-fondée** si, et seulement si toute suite $(x_i)_{i \in \mathbb{N}}$ d'éléments de E décroissante selon $\preceq$ (c-à-d., $x_{i+1} \preceq x_i$) est **stationnaire**, i.e. il existe un $j \in \mathbb{N}$ tel que pour tout $i > j$, nous avons $x_i = x_j$.
Un ensemble E muni d'une relation d'ordre bien fondée est dit **bien fondé**.
 $\preceq$ est un **bon ordre** si il est total et bien fondé.

Une définition équivalente à celle ci-dessus peut aussi être trouvée dans les livres traitant de ce sujet comme l'établit le résultat suivant :

Proposition 7.

$(E, \preceq)$ est bien fondé si, et seulement si tout sous-ensemble non vide de E admet un élément minimal vis-à-vis de $\preceq$.

Démonstration : Pour les deux sens de l'équivalence, la preuve se fait par l'absurde.

($\Rightarrow$) Hypothèse : $(E, \preceq)$ est bien fondé et S est un sous-ensemble non vide de E qui n'admet pas d'élément minimal. Soit $x_0 \in S$ un élément quelconque (cet élément existe car $S \neq \emptyset$). Par hypothèse, il existe $x_1 \prec x_0$. On peut recommencer sur x_1 et construire par récurrence une suite décroissante $(x_i)_{i \in \mathbb{N}}$ à partir des éléments de S . Puisque S n'admet pas d'élément minimal, cette suite est infinie et non stationnaire, ce qui contredit l'hypothèse de E d'être bien fondé.

Soit ($\Leftarrow$) Hypothèse : Tout sous-ensemble non vide S de E admet un élément minimal, et il existe une suite $(x_i)_{i \in \mathbb{N}}$ décroissante selon l'ordre $\preceq$ non stationnaire.
Par définition, l'ensemble des x_i de la suite est un sous-ensemble de E . Il admet donc un élément minimal ce qui contredit le fait que la suite n'est pas stationnaire. ■

Exemple 8.

- L'ordre naturel est un bon ordre sur $\mathbb{N}$ mais pas sur $\mathbb{Z}$.
- L'ordre lexicographique sur $\mathbb{N} \times \mathbb{N}$ est défini par $(n, m) \preceq (n', m')$ si et seulement si : $(n < n')$ ou $(n = n' \text{ et } m \leq m')$. On remarque que si $n > 0$ alors (n, m) a une infinité de minorants (tous les éléments (i, p) avec $0 \leq i \leq n - 1$ et $p \in \mathbb{N}$). Néanmoins, l'ordre lexicographique est un bon ordre sur $\mathbb{N} \times \mathbb{N}$. En effet, soit X un sous-ensemble non vide de $\mathbb{N} \times \mathbb{N}$. Soit $n = \min\{p \mid \exists q \in \mathbb{N}, (p, q) \in X\}$ et $m = \min\{q \mid (n, q) \in X\}$. On vérifie aisément que (n, m) est le plus petit élément de X .
- Soit $(A, \preceq)$ un ensemble ordonné. Notons A^* l'ensemble de toutes les suites finies sur A , i.e., $A^* = \{(a_1, \dots, a_n) \mid n \in \mathbb{N}, \forall i, 1 \leq i \leq n, a_i \in A\}$. A^* est appelé le *monoïde libre* sur A et sera étudié plus en détail dans la troisième partie de ce document. Pour simplifier les notations, une suite $(a_1, \dots, a_n)$ de A^* sera notée $a_1 \dots a_n$.
Là encore, on peut définir l'ordre lexicographique $\preceq_{lex}$ sur A^* de la façon suivante : soient $l = a_1 \dots a_n$ et $l' = b_1 \dots b_p$ deux mots de A^* . $l \preceq_{lex} l'$ si, et seulement si pour $r = \min\{n, p\}$, il existe $i \leq r$ tel que $a_i < b_i$ et pour tout $j < i$, $a_j = b_j$. $\preceq_{lex}$ n'est pas bien fondé. En effet, considérons l'alphabet $\{a, b\}$ avec $a \preceq b$, et considérons la suite $(a^n.b)_{n \in \mathbb{N}}$ où $a^n = \underbrace{a \dots a}_n$. Cette suite est selon l'ordre lexicographique décroissante, et pourtant elle n'est pas stationnaire.

Les bons ordres ont une propriété remarquable qui signifie que l'on peut ordonner à isomorphisme près et selon la relation "être un début de" tous les bons ordres, où la notion d'"être un début de" est définie par :

Définition 10.

Soit $(E, \preceq)$ un ensemble ordonné. Soit X une chaîne de E . On dit qu'un sous-ensemble strict I de X est un **début** de X si tous les éléments de $X \setminus I$ sont supérieurs strictement à tous les éléments de I .

Lemme 9.

Un ensemble ordonné $(E, \preceq)$ tel que $\preceq$ est un bon ordre ne peut pas être mis en correspondance par un homomorphisme (i.e. une fonction monotone) avec un de ses débuts.

Démonstration : Soit X un début non vide. Supposons un homomorphisme $f : E \rightarrow X$. Par définition des bons ordres, $E \setminus X$ possède un plus petit élément e , et donc $X = \{x \in E \mid x < e\}$. On a alors $f(e) < e$. Posons alors $Y = \{x \in E \mid f(x) < x\}$. Y est non vide, il contient au moins e . Y possède alors un élément minimal m . Puisque que $m \in Y$, on a $f(m) < m$. f étant un homomorphisme, on a alors $f(f(m)) < f(m)$. Maintenant, $f(m) \notin Y$ puisque m est minimal dans Y . Donc, par définition

de Y , on a $f(f(m)) \succeq f(m)$ contredisant ce qui précède. C'est donc que Y est vide, ce qui interdit que f soit monotone. ■

A fortiori, $(E, \preceq)$ ne peut pas être isomorphe à un de ses débuts. La propriété en question s'énonce ainsi :

Proposition 10.

Soient $(E, \preceq)$ et $(E', \preceq')$ deux ensembles bien ordonnés tels que $\preceq$ et $\preceq'$ sont des bons ordres. Alors, soit E et E' sont isomorphes, soit l'un est début de l'autre à isomorphisme près.

Démonstration : Définissons la correspondance f de E dans E' par :

$$f(e) = e' \text{ ssi } \{x \in E \mid x \prec e\} \text{ est isomorphe à } \{x' \in E' \mid x' \prec' e'\}$$

Tout d'abord montrons que f est une fonctionnelle, i.e. que pour tout $e \in E$ quand $f(e)$ est définie, alors elle est unique. Supposons alors qu'il existe $e'_1, e'_2 \in E'$ tels que $f(e) = e'_1$ et $f(e) = e'_2$. Par définition, ceci veut dire que l'ensemble $\{x \in E \mid x \prec e\}$ est à la fois isomorphe à $\{x' \in E' \mid x' \prec' e'_1\}$ et à $\{x' \in E' \mid x' \prec' e'_2\}$, et donc $\{x' \in E' \mid x' \prec' e'_1\}$ et $\{x' \in E' \mid x' \prec' e'_2\}$ sont aussi isomorphes. Maintenant, ces deux ensembles sont des sous-ensembles de E' totalement ordonnés par $\preceq'$. Ainsi, tous les éléments de $\{x' \in E' \mid x' \prec' e'_1\} \cup \{x' \in E' \mid x' \prec' e'_2\}$ sont comparables entre eux. Supposons alors que $e'_1 \prec' e'_2$, ceci veut dire que $e'_1 \in \{x' \in E' \mid x' \prec' e'_2\}$, ce qui implique que $\{x' \in E' \mid x' \prec' e'_1\}$ est un début de $\{x' \in E' \mid x' \prec' e'_2\}$. Or par le lemme 9, on en déduit une contradiction. On peut faire le même raisonnement et arriver à la même conclusion si l'on suppose que $e'_2 \prec' e'_1$.

Montrons maintenant que f est injective, i.e. pour tout $e' \in E'$ s'il existe $e \in E$ tel que $f(e) = e'$ alors e est l'unique élément satisfaisant cette équation. Supposons deux éléments $e_1, e_2 \in E$ tels que $f(e_1) = f(e_2) = e'$. Par définition, ceci veut dire que les deux ensembles $\{x \in E \mid x \prec e_1\}$ et $\{x \in E \mid x \prec e_2\}$ sont isomorphes à $\{x' \in E' \mid x' \prec' e'\}$, et donc $\{x \in E \mid x \prec e_1\}$ et $\{x \in E \mid x \prec e_2\}$ sont aussi isomorphes. En appliquant, le même raisonnement que précédemment, on démontre alors que e_1 et e_2 sont nécessairement égaux.

Il nous reste alors à montrer que f est strictement croissante ce qui entrainera qu'elle est un isomorphisme de son domaine de définition vers son image. Soient $e_1, e_2 \in E$ tels que $e_1 \prec e_2$, et tels que $f(e_1) = e'_1$ et $f(e_2) = e'_2$. Par définition, il existe un isomorphisme g de $\{x \in E \mid x \prec e_2\}$ dans $\{x' \in E' \mid x' \prec' e'_2\}$. Par hypothèse, $e_1 \in \{x \in E \mid x \prec e_2\}$, et donc on a $g(e_1) \prec' e'_2$. La restriction de g à $\{x \in E \mid x \preceq e_1\}$ dans $\{x' \in E' \mid x' \preceq' g(e_1)\}$ est encore un isomorphisme. Ainsi, on a par définition que $f(e_1) = g(e_1)$ d'où l'on peut conclure directement que $e'_1 \prec' e'_2$.

À partir de là, 4 cas sont à considérer :

1. $Dom(f) = E$ et $Im(f) = E'$, $(E, \preceq)$ et $(E', \preceq')$ sont isomorphes.
2. $Dom(f) = E$ et $Im(f)$ est un début de E' , $(E, \preceq)$ est isomorphe à un début de $(E', \preceq')$.
3. $Dom(f)$ est un début de E et $Im(f) = E'$, $(E', \preceq')$ est isomorphe à un début de $(E, \preceq)$.
4. $Dom(f)$ et $Im(f)$ sont des débuts de E et E' , respectivement. On montre que ce cas n'est pas possible. Par hypothèse, ceci veut dire qu'il existe $e \in E$ et $e' \in E'$ tels que $Dom(f) = \{x \in E \mid x \prec e\}$ et $Im(f) = \{x' \in E' \mid x' \prec' e'\}$. $Dom(f)$ et $Im(f)$ étant isomorphe, par définition on a $f(e) = e'$ et donc $e \in Dom(f)$ et $e' \in Im(f)$, ce qui est une contradiction. ■

La preuve ci-dessus peut laisser penser que la proposition 10 est aussi correcte quand $(E, \preceq)$ et $(E', \preceq')$ sont des ensembles totalement ordonnés mais dont les ordres ne sont pas forcément bons. Le problème est que dans ce cas-là, nous ne pourrions pas appliquer le lemme 9 que nous utilisons pour montrer que la correspondance f définie dans la preuve est une fonction injective.

En fait, dans le cas où $(E, \preceq)$ et $(E', \preceq')$ sont des ensembles totalement ordonnés mais dont les ordres ne sont pas forcément bons, on ne pourrait rien conclure. Par exemple, si on considère $(\mathbb{Z}, \leq)$ et $(\mathbb{Q}^+, \leq)$, le domaine et l'image de la correspondance f sont vides ; aucun début de $(\mathbb{Z}, \leq)$ n'est isomorphe à un début de $(\mathbb{Q}^+, \leq)$.

Induction mathématique

L'intérêt des ensembles bien-fondés est qu'ils sont intimement liés à la notion d'induction mathématique. En effet, l'induction mathématique sur un ensemble ordonné $(E, \preceq)$ est définie par l'équivalence : Soit P une propriété portant sur les éléments de E ,

$$\forall e \in E, P(e) \iff [\forall e' \in E, (e' \prec e \Rightarrow P(e')) \Rightarrow P(e)]$$

On peut être intrigué par le fait que ce principe d'induction mathématique ne comporte pas de cas de base. En fait, celui-ci est caché dans la partie droite de l'équivalence. En effet, soit e un élément minimal de E . Si on a vérifié la partie droite de l'équivalence ci-dessus, on a entre autres vérifié

$$(\forall e' \in E, e' \prec e \Rightarrow P(e')) \Rightarrow P(e)$$

Or, $P(e)$ est vraie si $\forall e' \in E, e' \prec e \Rightarrow P(e')$ est vraie. Mais il n'y a pas d'élément $e' \in E$ tel que $e' \prec e$ quand e est minimal dans E pour $\prec$, et donc $\forall e' \in E, e' \prec e \Rightarrow P(e')$ est toujours vraie, et donc $P(e)$ l'est aussi.

Bien entendu, l'induction mathématique n'est pas valide pour tout les ensembles ordonnés. Dans le cas contraire, ceci voudrait dire que nous pourrions faire de la preuve par récurrence sur l'ensemble $\mathbb{R}$ muni de la relation d'ordre classique $\leq$. En fait, seuls les ensembles bien-fondés admettent l'induction mathématique comme l'établit le théorème suivant :

Théorème 11.

$(E, \preceq)$ est bien fondé si, et seulement s'il admet l'induction mathématique comme valide.

Démonstration : Pour un ensemble bien fondé $(E, \preceq)$ l'équivalence suivante est satisfaite pour tout sous-ensemble S de E :

$$S \neq \emptyset \iff (\exists e \in E, (e \in S) \text{ et } (\forall e' \in E, e' \prec e \Rightarrow e' \notin S))$$

Soit P une propriété sur E . Par définition, P dénote un sous-ensemble de E . Notons alors S le sous-ensemble de E tel que $S = \{e \mid \text{non}P(e)\}$. Par définition, si P admet une preuve par induction, nous avons alors $S = \emptyset$. Par une simple application des équivalences logiques suivantes :

- $\exists x.P(x) \iff \text{non}(\forall x.\text{non}P(x))$
- $\text{non}(A \text{ et } B) \iff \text{non}A \text{ ou } \text{non}B$
- $(A \Rightarrow B) \iff (\text{non}A \text{ ou } B)$

les équivalences suivantes sont donc satisfaites :

$$\begin{aligned} S \neq \emptyset &\iff (\exists e \in E, (e \in S) \text{ et } (\forall e' \in E, e' \prec e \Rightarrow e' \notin S)) \\ &\iff S \neq \emptyset \iff (\forall e \in E, (e \notin S) \text{ ou } \text{non}(\forall e' \in E, e' \prec e \Rightarrow e' \notin S)) \\ &\iff \forall e \in E, P(e) \iff (\forall e \in E, P(e) \text{ ou } \text{non}(\forall e' \in E, e' \prec e \Rightarrow P(e'))) \\ &\iff \forall e \in E, P(e) \iff (\forall e \in E, (\forall e' \in E, e' \prec e \Rightarrow P(e')) \Rightarrow P(e)) \end{aligned}$$

Le théorème ci-dessus généralise la récurrence usuelle sur les entiers naturels à tout ensemble bien-fondé. En effet, rappelons le principe de preuve par induction sur les entiers naturels muni de l'ordre standard $\leq$. Ce principe peut s'énoncer de deux façons. La première connue sous le nom de principe de récurrence mathématique s'énonce de la façon suivante : Soit $P(n)$ une propriété dont l'énoncé dépend de $n \in \mathbb{N}$, si

(B) $P(0)$ est vraie.

(I) $\forall n \in \mathbb{N}, (P(n) \Rightarrow P(n+1))$

alors : $\forall n \in \mathbb{N}, P(n)$ est vraie.

Parfois, dans des cas plus complexes où pour établir $P(n+1)$ on ait besoin d'utiliser explicitement le fait que P soit vraie aux étapes $0, 1, \dots, n-1, n$, on utilise le second principe d'induction dont on reconnaitra aisément une application directe du principe d'induction mathématique généralisé énoncé ci-dessus. Il s'énonce de la façon suivante : Soit $P(n)$ une propriété dont l'énoncé dépend de $n \in \mathbb{N}$, si

(I') $\forall n \in \mathbb{N}, ((\forall k < n, P(k)) \Rightarrow P(n))$

alors, $\forall n \in \mathbb{N}, P(n)$ est vraie.

On verra dans la suite de ce chapitre que l'induction mathématique généralisée s'applique aussi à un autre principe de preuve très important en informatique, l'**induction structurale** (cf. chapitre 4).

Induction sans ordre bien fondé : lemme de Zorn et ses avatars

Comme nous venons de le voir, l'induction mathématique ne marche que pour les ensembles bien fondés. Mais quel type de preuve utilisons-nous pour démontrer une propriété sur tous les éléments d'un ensemble qui n'est pas bien fondé ? On dispose alors d'un lemme puissant, le **lemme de Zorn** dont on verra qu'il est équivalent à un autre lemme de la théorie des ensembles, le **théorème de Zermelo** qui justement énonce l'existence d'un bon ordre pour tout ensemble si l'on accepte l'axiome du choix. Ainsi, on peut *a priori* faire de la récurrence sur tous les ensembles. Le problème est que le théorème de Zermelo énonce l'existence de ce bon ordre mais ne dit pas comment le construire. C'est pourquoi, l'axiome du choix, le lemme de Zorn ou encore le théorème de Zermelo (tous les trois équivalents en fait) sont utilisés dans les preuves non constructives, i.e. on prouve l'existence d'un ensemble vérifiant certaines propriétés sans pour autant caractériser aucun ensemble vérifiant cette propriété, et ainsi contourner le manque d'induction explicite.

Énonçons ce résultat¹.

Théorème 12 (Lemme de Zorn).

Tout ensemble inductif $(E, \preceq)$ admet un élément maximal.

Démonstration : Commençons d'abord par introduire quelques notions que nous utiliserons par la suite pour démontrer ce résultat.

1. La preuve de ce lemme est très fortement inspirée de celle que l'on peut trouver dans le document de Rémi Peyre sur le lemme de Zorn.

Définition 11 (Chaîne ultime et continuable).

Soit $(E, \preceq)$ un ensemble ordonné. Nous dirons qu'une chaîne X de E est **ultime** si elle n'admet aucun majorant strict (i.e. s'il n'y a aucun $\alpha \in E \setminus X$ pour lequel $\forall x \in X, x \prec \alpha$), et **continuable** dans le cas contraire.

On observe alors qu'une chaîne ultime X d'un ensemble inductif E a nécessairement un plus grand élément, qui est maximal dans E . En effet, une chaîne ultime est une chaîne dont on a poussé la progression par la relation $\prec$ jusqu'à ne plus pouvoir continuer. Là, la chaîne X peut se terminer de deux façons *a priori*

1. il y a en fin de chaîne une infinité d'éléments tous plus grands les uns que les autres,
2. il y a un élément de la chaîne qui est le dernier.

Le premier cas est impossible. En effet, s'il se produisait, la chaîne ayant un majorant par définition des ensembles inductifs, nous pourrions toujours l'ajouter à la fin de la chaîne ce qui contredirait l'ultimité de celle-ci. Donc, X a un plus grand élément $m \in X$. Ce dernier est alors nécessairement maximal dans E . En effet, si ce n'était pas le cas, par définition des ensembles inductifs, il existerait $x \in E$ tel que $m \preceq x$ ce qui contredirait encore l'ultimité de X car l'on pourrait une nouvelle fois ajouter ce x à la fin de la chaîne X . m est donc l'élément maximal de E .

La démonstration du lemme de Zorn se ramène alors à démontrer que tout ensemble ordonné admet une chaîne ultime. En effet, si E est de plus inductif, on vient de démontrer ci-dessus que E a alors nécessairement un élément maximal. Ce dernier énoncé est connu sous le nom du **Principe de maximalité de Hausdorff**.

Pour commencer, on choisit pour toute chaîne continuable X de E un majorant strict noté $m(X)$.² Commençons d'abord par définir la notion de bonne chaîne qui nous sera utile dans la suite de la preuve.

Définition 12 (Bonne chaîne).

Soit $(E, \preceq)$ un ensemble ordonné. Soit X une chaîne de E . X est dite être une **bonne chaîne** quand, pour tout début I de X , $m(I)$ appartient à X et est le plus petit élément de $X \setminus I$.

Montrons alors pour toutes bonnes chaînes distinctes X_1 et X_2 de E que l'une est un début de l'autre³. Notons X^* l'ensemble de tous les $x \in X_1 \cap X_2$ tel que l'ensemble $\{y \in X_1 | y \prec x\}$ est un début de X_2 .⁴ Cet ensemble n'est pas vide. En effet, la chaîne vide $\emptyset$ est début de toute bonne chaîne et donc de X_1 et X_2 . Par définition, on a alors que $m(\emptyset) \in X_1 \cap X_2$.

X^* peut éventuellement coïncider avec X_1 ou X_2 mais pas les deux car X_1 et X_2 sont distinctes. Supposons alors que $X^* \subset X_1$ (inclusion stricte). Dans ce cas-là, X^* est un début de X_1 . En effet, soit $x \in X^*$, et soit $y \in X_1 \setminus X^*$. Si $y \prec x$ (par définition, $y \in X_2$ alors), alors $y \in X_1 \cap X_2$ et donc l'ensemble $\{y' \in X_1 | y' \prec y\}$ est aussi un début de X_2 , d'où nous pouvons déduire que $y \in X^*$ ce qui est une contradiction. Ainsi, on a nécessairement que $x \prec y$. Montrons alors que $X^* = X_2$ ce qui conclura cette première partie de preuve. Si X^* était strictement incluse dans X_2 , pour les

2. En fait, ce choix est possible par l'**axiome du choix** de la théorie des ensembles et dont l'énoncé dit que pour toute famille d'ensembles $(X_i)_{i \in I}$ non vides, il existe une **fonction de choix** $c = I \rightarrow \bigcup_{i \in I} X_i$ telle que pour

tout $i \in I$, $c(i) \in X_i$. Cet axiome s'utilise en mathématique quand on ne sait pas comment choisir un élément particulier dans des ensembles (e.g. le plus petit élément d'une classe d'équivalence) ou que l'on veuille encore effectuer une infinité de choix simultanément. Dans notre cas, I est l'ensemble de toutes les chaînes continuables et pour $i \in I$, X_i est l'ensemble des majorants stricts de la chaîne continuable i .

3. On retrouve la proposition 10 mais ici étendue aux bonnes chaînes.

4. On aurait pu de façon équivalente choisir l'ensemble $\{y \in X_2 | y \prec x\}$ comme début de X_1 .

mêmes raisons que précédemment elle en serait un début. Comme les chaînes X_1 et X_2 sont bonnes, $m(X^*)$ serait alors dans chacune des chaînes X_i pour $i = 1, 2$, et serait par définition le plus petit élément de $X_i \setminus X^*$. Ainsi, l'ensemble $\{y \in X_1 \mid y \prec m(X^*)\}$ est un début de X_2 , et donc, d'après la définition de X^* , on aurait $m(X^*) \in X^*$ ce qui est une contradiction.

Notons alors $\overline{X} \subseteq E$ l'ensemble défini comme la réunion de toutes les bonnes chaînes. Montrons alors que $\overline{X}$ est une bonne chaîne. Soient x et y deux éléments de $\overline{X}$. Par définition, il existe deux bonnes chaînes X_i et X_j telles que $x \in X_i$ et $y \in X_j$. L'on sait que soient X_i et X_j coïncident ou bien l'une est début de l'autre, mais dans les deux cas une est incluse dans l'autre. Supposons alors que $X_i \subseteq X_j$. x et y sont alors tous les deux éléments de X_j et donc sont comparables par $\preceq$. Ainsi, $\overline{X}$ est une chaîne. Montrons qu'elle est en plus une bonne chaîne. Soit I un début de $\overline{X}$. Par définition, tous les points x de I appartiennent à des bonnes chaînes X_x , et donc pour tous points $x \neq y \in I$, X_x ou X_y est début l'une de l'autre. Ainsi, x et y sont comparables et donc I est une chaîne. I est de plus une bonne chaîne. En effet, soit $X_{m(I)}$ la bonne chaîne contenant $m(I)$. Par hypothèse, pour tout $x \in I$, la bonne chaîne X_x est un début de $X_{m(I)}$, et donc I est aussi un début de $X_{m(I)}$. Soit I' un début de I . I' est aussi un début de $X_{m(I)}$, et donc $m(I')$ est le plus petit élément de $X_{m(I)} \setminus I'$. Mais $I \setminus I' \subset X_{m(I)} \setminus I'$ d'où nous pouvons déduire que $m(I')$ est aussi le plus petit élément de $I \setminus I'$.

I étant continuable, $I \cup \{m(I)\}$ est encore une bonne chaîne, d'où nous déduisons que $m(I) \in \overline{X}$. Montrons que c'est le plus petit élément de $\overline{X} \setminus I$. Considérons l'ensemble $J = \{x \in \overline{X} \mid x \prec m(I)\}$. Par le fait que tout point de $\overline{X}$ appartient à une bonne chaîne, et que les bonnes chaînes sont ordonnées par la relation "être un début de", J est un début d'une bonne chaîne et donc un début de $\overline{X}$. En suivant le même processus de preuve que pour I ci-dessus, on peut montrer que J est aussi une bonne chaîne dont I par définition est un début. Or, par définition, $m(I) \notin J$ ce qui est une contradiction avec le fait pour J d'être une bonne chaîne.

Pour finir, $\overline{X}$ est ultime, sinon (i.e. $m(\overline{X}) \notin \overline{X}$), $\overline{X} \cup \{m(\overline{X})\}$ serait une bonne chaîne, et donc $m(\overline{X}) \in \overline{X}$ ce qui serait absurde. ■

Pour ce qui nous intéresse dans cette première partie du cours, l'intérêt du lemme Zorn est qu'il permet de montrer le théorème suivant, plus connu sous le nom de **Théorème de Zermelo**⁵ :

Théorème 13.

Tout ensemble X peut être muni d'un bon ordre.

Démonstration : Notons S l'ensemble défini par :

$$S = \{(Y, \preceq) \mid Y \subseteq X, \preceq \text{ est un bon ordre sur } Y\}$$

Par définition, cet ensemble n'est pas vide car il contient au moins le couple $(\emptyset, \emptyset)$. Définissons sur S la relation R par :

$$(Y, \preceq) R (Y', \preceq') \text{ ssi } Y \subset Y' \text{ et } Y \text{ est début de } Y' \text{ avec } \preceq = \preceq'|_Y.$$

Il est facile de vérifier que R est une relation d'ordre strict sur S . Elle est de plus inductive car toute chaîne de (S, R) est majorée par sa réunion. Par le lemme de Zorn, notons $(\overline{Y}, \leq)$ un élément maximal de (S, R) . Si $\overline{Y} \subset X$, alors en prenant n'importe quel $x \in X \setminus \overline{Y}$, on peut définir le bon ordre $\leq'$ sur $\overline{Y} \cup \{x\}$ en gardant pour tous les éléments de $\overline{Y}$ l'ordre $\leq$ et en posant pour tout $y \in \overline{Y}$, $y \leq' x$. Mais alors, par définition, $\leq'$ prolonge $\leq$ ce qui est exclu par maximalité, et donc $\overline{Y} = X$, et ainsi le bon ordre sur $\overline{Y}$ est un bon ordre sur X . ■

La démonstration que nous avons effectuée dans la preuve du théorème de Zermelo est caractéristique de la façon dont on utilise généralement le lemme de Zorn.

5. En fait, l'énoncé du théorème de Zermelo est équivalent au lemme de Zorn et donc à l'axiome du choix.

4. Treillis, définitions inductives et preuve par induction structurelle

Les définitions inductives/récurrentes d'ensembles sont omniprésentes en informatique. Par exemple, toutes les structures de données que l'on manipule en programmation se définissent de cette façon : une liste d'éléments est soit la liste vide, soit l'ajout d'un élément dans une liste déjà construite, de même, un arbre binaire est soit l'arbre vide, soit l'ajout d'une racine à deux arbres binaires qui dénotent respectivement l'arbre gauche et l'arbre droit. L'on pourrait bien sûr définir de tels ensembles explicitement comme il est d'usage en mathématiques. Ainsi, les listes dont les éléments sont pris dans un ensemble E seraient définies par toutes les applications $f : I \rightarrow E$ où $I = \{0, 1, \dots, n\}$ est un segment de $\mathbb{N}$. De la même façon, les arbres binaires pourraient être définis comme l'ensemble des graphes connexes acycliques, tel que le degré de chaque noeud soit au plus 3. Bien sûr, de telles définitions explicites ne sont pas suffisantes quand l'on veut "programmer" sur ces structures de données. Seules leurs définitions inductives par la considération de cas de base et de cas plus généraux obtenus à partir de cas plus simples, permettent de "programmer" sur ces structures. Cette construction des objets manipulés permet de plus un type de raisonnement puissant qui fait écho à la section précédente puisqu'il est un cas particulier d'induction, le *raisonnement par induction structurelle*. Ici, nous allons nous intéresser à définir formellement cette notion de définition inductive à partir de théorèmes de point fixe obtenus dans le cadre de la théorie des treillis. Nous étudierons aussi une autre application des treillis : la sémantique dénotationnelle des langages de programmation.

4.1. Treillis et points fixes

Définition

Les treillis sont des ensembles ordonnés avec la contrainte supplémentaire que toute paire d'éléments possède à la fois une borne supérieure et une borne inférieure.

Définition 13 (Treillis).

Un ensemble ordonné $(E, \preceq)$ est un **treillis** si, et seulement si toute paire d'éléments $\{x, y\}$ de E possède une borne supérieure, noté $x \sqcup y$, et une borne inférieure, notée $x \sqcap y$.

Si E est un treillis, on peut considérer qu'il est muni de deux opérateurs binaires $\sqcup : E \times E \rightarrow E$ et $\sqcap : E \times E \rightarrow E$.

Exemple 14.

- $\mathbb{N}$ muni de l'ordre de divisibilité est un treillis, les opérations binaires $\sqcup$ et $\sqcap$ étant respectivement le ppcm et le pgcd.
- Soit E un ensemble quelconque. $\mathcal{P}(E)$ ordonné par inclusion est un treillis où $\sqcup$ et $\sqcap$ sont respectivement $\cup$ et $\cap$.

Treillis complets et fonctions continues**Définition 14** (Treillis complet).

Un treillis $(E, \preceq, \sqcup, \sqcap)$ est **complet** si tout sous-ensemble S de E admet une borne supérieure et une borne inférieure. On les note respectivement $\sqcup S$ et $\sqcap S$.

Exemple 15.

Soit E un ensemble quelconque. $\mathcal{P}(E)$ ordonné par inclusion est un treillis complet où $\sqcup\{E_i | i \in I\} = \bigcup_{i \in I} E_i$ et $\sqcap\{E_i | i \in I\} = \bigcap_{i \in I} E_i$.

Remarque 16.

Par définition, si E est un treillis complet alors la borne inférieure (resp. supérieure) de E est majorée (resp. minorée) par tous les éléments de E . Ainsi, E possède un élément minimum $\perp$ et maximum $\top$. Mais ça on pouvait s'y attendre car les treillis complets sont avant tout des ensembles inductifs, même beaucoup plus que des ensembles inductifs car tous les sous-ensembles (et pas seulement les chaînes) possèdent une borne supérieure (et pas simplement un majorant) qui est le plus petit des majorants. Donc, par application du lemme de Zorn, l'ensemble E possède un élément maximal, ici $\top$.

Comme on le verra dans la suite de ce chapitre, les définitions inductives d'un ensemble se définissent par une équation à point fixe, i.e. une équation de la forme $F(E) = E$, et parmi tous les ensembles E , points fixes de l'équation, l'ensemble recherché sera le plus petit. Comme il est d'usage en analyse réelle, pour résoudre/calculer les points fixes d'une fonction $f : \mathbb{R} \rightarrow \mathbb{R}$, on dispose de la méthode itérative qui consiste pour toute fonction continue f de considérer la suite :

$$\begin{aligned} x_0 &= \text{une certaine valeur initiale} \\ x_i &= f(x_{i-1}) \end{aligned}$$

L'on sait que si cette suite possède une limite x , on peut montrer aisément que x est un point fixe de la fonction f .¹ C'est de ce type de méthode dont on va s'inspirer pour résoudre les équations

1. En aucun cas, on assure qu'un point fixe existe, mais quand il existe on sait alors le calculer. Il existe d'autres résultats qui assurent l'existence et l'unicité d'un tel point fixe quand l'application f est dite contractante.

à points fixes sur les treillis.

Étant donné un treillis complet $(E, \preceq, \sqcup, \sqcap)$ et une application $f : E \rightarrow E$, l'ensemble des points fixes de f est un sous-ensemble de E . Si ce sous-ensemble admet un minimum, on l'appellera le *plus petit point fixe*, et s'il admet un maximum, on l'appellera le *plus grand point fixe*. Nous avons un premier résultat qui assure l'existence d'un plus petit et d'un plus grand point fixe quand l'application f est un homomorphisme sur $(E, \preceq)$, i.e. elle est monotone pour l'ordre $\preceq$.

Théorème 17.

Soit $(E, \preceq, \sqcup, \sqcap)$ un treillis complet. Soit $f : E \rightarrow E$ une application monotone. Alors, f possède un plus petit et un plus grand point fixe.

Démonstration : Soit X l'ensemble défini par :

$$X = \{x \in E \mid x \preceq f(x)\}$$

Triviallement, $\perp \in X$. Par définition, pour tout $x \in X$, on a $x \preceq \sqcup X$, et donc par monotonie, $f(x) \preceq f(\sqcup X)$. Comme $x \preceq f(x)$, on a $x \preceq f(\sqcup X)$, et donc $f(\sqcup X)$ est un majorant de X . On a alors $\sqcup X \preceq f(\sqcup X)$. Comme f est monotone, on a aussi $f(\sqcup X) \preceq f(f(\sqcup X))$, et donc $f(\sqcup X) \in X$, d'où l'on peut déduire $f(\sqcup X) \preceq \sqcup X$. Par antisymétrie de $\preceq$, on conclut alors que $f(\sqcup X) = \sqcup X$. Ainsi, $\sqcup X$ est un point fixe de f . Si f a un autre point fixe y . Par définition de X , $y \in X$, et donc $y \preceq \sqcup X$.

On montre de même que $\sqcap\{x \in E \mid x \succeq f(x)\}$ est le plus grand point fixe de f . ■

Exemple 18.

Soit $G = (V, E)$ un graphe orienté où V est l'ensemble des noeuds du graphe et $E \subseteq V \times V$ l'ensemble des arcs. Pour tout $X \subseteq V$, on note $Pred(X)$ l'ensemble des prédécesseurs des sommets de X :

$$Pred(X) = \{v \in V \mid \exists v' \in X, (v, v') \in E\}$$

Soit $X_0 \subseteq V$. L'application $f : \mathcal{P}(V) \rightarrow \mathcal{P}(V)$ définie par $X \mapsto X_0 \cup pred(X)$ est monotone pour l'inclusion. Elle admet donc un plus petit et un plus grand point fixe. Le plus petit point fixe de f est l'ensemble Y des sommets depuis lesquels on peut atteindre un sommet de X_0 (définition inductive de la co-accessibilité : Y est le plus petit ensemble contenant X_0 et tel que $pred(Y) \subseteq Y$ - cf. la section 4.2).

Le plus grand point fixe de f est l'ensemble $Z = Y \cup I$ où I est l'ensemble des sommets depuis lesquels il existe dans G un chemin infini. En effet, on a trivialement que $I \subseteq pred(I)$ sinon ceci signifierait qu'il existe un sommet v prédécesseur d'un sommet v' de I qui ne serait pas dans I . Mais, alors on peut mener un chemin infini à partir de v puisque l'on peut en mener un à partir de v' . Donc, $v \in I$. De plus, on a $Y = f(Y)$ (Y est son plus petit point fixe). Donc, on a aussi $Y \cup I \subseteq f(Y) \cup f(I)$, i.e. $Z \subseteq f(Z)$.

Montrons maintenant que Z est le plus grand ensemble vérifiant $Z \subseteq f(Z)$. Soit $X \subseteq V$ tel que $X \subseteq f(X)$, et soit v un sommet de X . Nous avons alors 2 possibilités :

1. si $v \in Y$, alors on a a fortiori $v \in Z$.
2. sinon (i.e. $v \notin Y$ et donc $v \notin X_0$ et $v \notin Pred(Y)$), alors $v \in pred(X)$, i.e. il existe un sommet $v_1 \in X$ tel que $(v, v_1) \in E$. Mais du fait que $pred(Y) \subseteq Y$, on en déduit que $v_1 \notin Y$. On peut alors par récurrence construire un chemin infini $(v, v_1, v_2, \dots)$. On a ainsi $v \in I$, et donc là encore $v \in Z$.

Pour permettre de calculer un des points fixes de f (ici ce sera le plus petit), il nous faut maintenant disposer d'une méthode itérative qui itération après itération, tende vers ce point fixe. Les éléments étant ordonnés, chaque itération doit avoir pour effet d'améliorer la solution selon l'ordre $\preceq$, i.e. si x est la solution obtenue à l'itération i , à $i + 1$ on doit avoir que $x \preceq f(x)$. Ceci nous donne alors la forme récursive de notre schéma itératif. Il nous manque le cas de base. Un candidat naturel se présente, l'élément minimal du treillis complet $\perp$. $\perp$ étant l'élément minimal dans E et f étant monotone, on a $\perp \preceq f(\perp) \preceq f(f(\perp)) \preceq \dots \preceq f^n(\perp) \preceq \dots$. Il nous reste alors à vérifier que $\sqcup\{f^n(\perp) \mid n \in \mathbb{N}\}$ avec $f^0(\perp) = \perp$ est un point fixe de f , i.e. $f(\sqcup\{f^n(\perp) \mid n \in \mathbb{N}\}) = \sqcup\{f^{n+1}(\perp) \mid n \in \mathbb{N}\}$. Par définition, tout sous-ensemble $E_k = \{f^i(\perp) \mid i \geq k\}$ avec $k \in \mathbb{N}$ a la même borne supérieure que $\{f^n(\perp) \mid n \in \mathbb{N}\}$. De plus, on a pour tout $k \in \mathbb{N}$, $E_k = f(E_{k-1})$. On a alors l'équivalence suivante :

$$f(\sqcup E_0) = \sqcup E_0 \iff f(\sqcup E_0) = \sqcup f(E_0)$$

Toute application $f : E \rightarrow E$ qui vérifierait la condition à droite de l'équivalence mais sur tout sous-ensemble E' de E (et non pas seulement la chaîne E_0) est dite *continue*. En fait, la continuité est plus générale, car elle n'impose pas que l'application f soit définie sur le même ensemble.

Définition 15.

Soit $(E, \preceq)$ et $(E', \preceq')$ deux ensembles ordonnés. Une application $f : E \rightarrow E'$ est **continue** (on dit aussi **sup-continue**) si elle préserve les bornes supérieures des sous-ensembles non vides de E , i.e. pour tout $S \subseteq E$ tel que $S \neq \emptyset$ si $\sqcup S$ existe alors $f(\sqcup S) = \sqcup f(S)$.

Théorème 19.

Si f est une application continue d'un treillis complet sur lui-même, alors le plus petit point fixe de f est égal à $\sqcup \{f^n(\perp) \mid n \in \mathbb{N}\}$.

Démonstration : On a vu ci-dessus qu'il était bien un point fixe. Il nous reste à montrer qu'il est le plus petit.

Si y est un autre point fixe de f , on montre par induction sur n que $f^n(\perp) \preceq y$. $\perp$ étant l'élément minimal, on a $f^0(\perp) \preceq y$. Supposons que $f^n(\perp) \preceq y$. f étant continue, elle est monotone. On a donc aussi $f^{n+1}(\perp) \preceq f(y)$. y étant un point fixe de f , on a alors $f^{n+1}(\perp) \preceq y$.

Ainsi, y est aussi un majorant de $\{f^n(\perp) \mid n \in \mathbb{N}\}$, et donc $\sqcup \{f^n(\perp) \mid n \in \mathbb{N}\} \preceq y$. ■

Dans la preuve ci-dessus, on n'utilise pas la structure de treillis. En fait, les seules conditions qui nous intéressent sont que l'ensemble ordonné $(E, \preceq)$, domaine et co-domaine de la fonction f , possède un élément minimal, et que chaque chaîne admette une borne supérieure. Enfin, la fonction a besoin d'être continue uniquement sur les chaînes. On parle alors d'**ordre partiel complet** (CPO en anglais pour Complete Partial Order) et de *continuité de Scott* du nom du logicien qui le premier caractérisa les CPOs. C'est à partir de ce type de structure algébrique que la sémantique dénotationnelle des langages de programmation a été définie, et non pas celle de treillis complet (cf. la section 4.3).

4.2. Définitions inductives et induction structurelle

Définitions inductives

La définition inductive d'un ensemble E consiste à construire chacun des éléments à partir d'éléments de l'ensemble E que l'on aurait déjà construits. Pour que ceci ait un sens, il faut se donner explicitement un ensemble B d'éléments que l'on suppose appartenir à E sans les construire, et des règles de construction. Comme il est d'usage en mathématiques, pour ne pas tomber dans les paradoxes de la théorie des ensembles, on définira toujours inductivement un sous-ensemble X d'un ensemble déjà connu E .

Définition 16 (Définition inductive).

Soit E un ensemble. Une **définition inductive** d'un sous-ensemble X de E est la donnée :

- d'un sous-ensemble B de E appelé **ensemble de base**, et
- d'un ensemble Γ de fonctions (qui peuvent être partielles) $F_i : E^{n_i} \rightarrow E$ où $n_i \in \mathbb{N}$ dénote l'arité de la fonction F_i . On appelle souvent les fonctions F_i les **règles de construction** de l'ensemble inductif.

telle que X est le plus petit ensemble (au sens de l'inclusion) vérifiant :

(B) $B \subseteq X$

(I) $\forall F_i : E^{n_i} \rightarrow E, \forall (x_1, \dots, x_{n_i}) \in E^{n_i}, F_i(x_1, \dots, x_{n_i}) \in X$

Exemple 20.

- Le sous-ensemble X de $\mathbb{N}$ défini inductivement par :
 - (B) $0 \in X$
 - (I) $x \in X \Rightarrow x + 1 \in X$
 n'est autre que $\mathbb{N}$. Ainsi, (B) et (I) constituent une définition inductive de $\mathbb{N}$.
- Soit A un ensemble. Notons X le sous-ensemble de A^* défini inductivement par :^a
 - (B) $\varepsilon \in X$ (ε est le mot vide équivalent à la suite $s : \emptyset \rightarrow A$ dans A^*).
 - (I) $\alpha \in X \Rightarrow \forall a \in A, a.\alpha \in X$ X n'est autre que A^* .
- Soit $A = \{ (,) \}$ l'ensemble (ou alphabet) constitué de deux parenthèses (ouvrante et fermante). Le sous-ensemble D de A^* des parenthésages bien formés, appelé aussi *langage de Dyck*, est inductivement défini par :
 - (B) $\varepsilon \in D$
 - (I) $x, y \in D \Rightarrow (x) \in D$ et $xy \in D$
- Soit A un ensemble. L'ensemble AB des arbres binaires étiquetés sur A est le sous-ensemble de $(A \cup \{ \emptyset, (,), ; \})^*$ défini inductivement par :
 - (B) $\emptyset \in AB$ ($\emptyset$ dénote alors l'arbre vide)
 - (I) $g, d \in AB \Rightarrow \forall a \in A, (a; g; d) \in AB$ (l'arbre binaire de racine a , d'arbre gauche g et d'arbre droit d)

^a A^* est l'ensemble de toutes les séquences finies (aussi appelées *mots*) que nous pouvons construire sur l'ensemble A pris comme un *alphabet*. Nous définirons plus précisément cet ensemble dans la troisième partie de ce cours.

Pour montrer qu'il existe toujours un plus petit sous-ensemble X de E contenant B et satisfaisant la condition (I), définissons la fonction $F : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$ par :

$$F(Y) = \{y \in E | (y \in B) \text{ ou } (\exists F_i, \exists (y_1, \dots, y_{n_i}) \in Y^{n_i}, y = F_i(y_1, \dots, y_{n_i}))\}$$

F est trivialement croissante pour l'inclusion. Elle est même continue sur les chaînes (le montrer en exercice). Par le théorème 17, on définit X comme le plus petit point fixe de la fonction F . Par définition, X est le plus petit ensemble vérifiant $F(X) = X$, et d'après la preuve du théorème 17, c'est le plus petit ensemble vérifiant la propriété $F(X) \subseteq X$. X est donc bien le plus

petit ensemble contenant B et fermé par les fonctions F_i . Ainsi, $X = \bigcap_{Y \in \mathcal{F}} Y$

où $\mathcal{F} = \{Y \mid F(Y) \subseteq Y\}$ qui est une autre façon de caractériser l'ensemble X .

Par le théorème 19, X peut aussi se définir par la suite suivante :

$$X_0 = \emptyset$$

$$X_i = X_{i-1} \cup \{y \in E \mid (y \in B) \text{ ou } \exists F_i, \exists y_1, \dots, y_{n_i} \in X_{i-1}, y = F_i(y_1, \dots, y_{n_i})\}$$

Et donc $X = \bigcup_{i \in \mathbb{N}} X_i$.²

Nous pouvons alors donner un premier principe d'induction.

Corollaire 1.

Soit E un ensemble. Soit $X \subseteq E$ un ensemble défini inductivement à partir d'un ensemble de base B et d'un ensemble Γ de fonctions $F_i : E^{n_i} \rightarrow E$. Si $Y \subseteq X$ contient B et est aussi clos par les fonctions de Γ , alors $Y = X$.

Démonstration : Par hypothèse, on a $F(Y) \subseteq Y$, et donc $X \subseteq Y$. Or par hypothèse $Y \subseteq X$ ce qui nous permet de conclure que $X = Y$. ■

Preuve par induction structurelle

Par définition, les ensembles X inductivement définis possèdent la propriété que chacun de leurs éléments est obtenu par composition licite de fonctions de Γ . Ainsi, associons à chaque fonction $F_i \in \Gamma$ un nom de fonction $\overline{F}_i$ et notons $\overline{\Gamma} = \{\overline{F}_i \mid F_i \in \Gamma\}$. Considérons alors l'ensemble $A = B \cup \overline{\Gamma} \cup \{(\cdot, \cdot), \cdot\}$. On peut considérer l'ensemble des expressions $T_\Gamma(B) \subseteq A^*$ comme défini inductivement par :

(B) $B \subseteq T_\Gamma(B)$

(I) $t_1, \dots, t_{n_i} \in T_\Gamma(B) \Rightarrow (\forall F_i : E^{n_i} \rightarrow E, \overline{F}_i(t_1, \dots, t_{n_i}) \in T_\Gamma(B))$

On peut définir une application $_^X : T_\Gamma(B) \rightarrow X$ par :

$$x \in B \mapsto x$$

$$\overline{F}_i(t_1, \dots, t_{n_i}) \mapsto F_i(t_1^X, \dots, t_{n_i}^X)$$

X étant inductivement défini à partir de B et des fonctions dans Γ , l'application $_^X$ est surjective. Maintenant, sur $T_\Gamma(B)$ on peut définir la relation binaire $\preceq_X$ comme la fermeture réflexive et transitive de :

$$t \preceq_X t' \Leftrightarrow t' = \overline{F}_i(\dots, t, \dots)$$

$\preceq_X$ est la relation "être une sous-expression de".

Théorème 21.

$\preceq_X$ est un ordre bien fondé sur $T_\Gamma(B)$.

2. On suppose ici que tout élément $x \in B$ se définit par une fonction constante $x : \rightarrow E$ (i.e. une fonction $x : E^0 \rightarrow E$) qui dénote l'élément x dans E .

Démonstration : Montrons que $\preceq_X$ est anti-symétrique. Par le théorème 19 appliquée à la fonction $F : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$, l'ensemble $T_\Gamma(B) = \bigcup_{j \in \mathbb{N}} T^j$ où pour tout $j \in \mathbb{N}$, T^j est l'ensemble défini par³ :

$$T^0 = \emptyset \\ T^j = T^{j-1} \cup \{\bar{F}_i(t_1 \dots, t_{n_i}) \mid \forall k, 1 \leq k \leq n_i, t_k \in T^{j-1}\}$$

Ainsi, pour tout $j > 0$ et pour toute expression $\bar{F}_i(t_1 \dots, t_{n_i}) \in T^j \setminus T_{j-1}$, toutes les sous-expressions t_k pour $k = 1, \dots, n_i$ appartiennent à des ensembles T^l avec $l < j$. On en déduit alors que deux expressions t et t' de $T_\Gamma(B)$ qui vérifieraient à la fois $t \preceq_X t'$ et $t' \preceq_X t$ ne peuvent être qu'égaux (une expression ne peut pas être à la fois sur et sous-expression d'une autre).

Montrons que $\preceq_X$ est de plus bien fondé. Par la définition inductive de $T_\Gamma(B)$, étant donnée une expression $\bar{F}_i(t_1, \dots, t_{n_i})$, l'ensemble de ses sous-expressions est forcément de cardinalité finie et a pour éléments minimaux tous les éléments appartenant à B . ■

On peut alors appliquer l'induction mathématique sur les ensembles bien fondés. Ainsi, étant donnée une propriété P portant sur les éléments de $T_\Gamma(B)$, ce principe de preuve s'exprime ici par :

$$\forall t \in T_\Gamma(B), P(t) \iff [\forall t' \in T_\Gamma(B), (\forall t' \in T_\Gamma(B) t' \prec_X t \Rightarrow P(t')) \Rightarrow P(t)]$$

Par le théorème 19 appliqué à la fonction $F : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$, on a vu que l'ensemble $T_\Gamma(B) = \bigcup_{j \in \mathbb{N}} T^j$ où pour tout $j \in \mathbb{N}$, T^j est l'ensemble défini par :

$$T^0 = \emptyset \\ T^j = T^{j-1} \cup \{\bar{F}_i(t_1 \dots, t_{n_i}) \mid F_i \in \Gamma, \forall k, 1 \leq k \leq n_i, t_k \in T^{j-1}\}$$

Le principe d'induction ci-dessus s'exprime aussi de la façon suivante : Si les deux conditions sont vérifiées

1. $P(x)$ est vraie pour tout $x \in B$
2. $\forall F_i \in \Gamma, (\forall k, 1 \leq k \leq n_i, P(t_k)) \Rightarrow P(\bar{F}_i(t_1, \dots, t_{n_i}))$

alors, $P(t)$ est vraie pour tout $t \in T_\Gamma(B)$.

Ce type de preuve s'appelle **induction structurelle**. Il s'étend naturellement à X . En effet, soit P une propriété portant sur les éléments de X . On peut définir la propriété P' sur $T_\Gamma(B)$ par : $\forall t \in T_\Gamma(B), P'(t) \Leftrightarrow P(t^X)$. L'application $_^X$ étant surjective, on a alors nécessairement :

$$(\forall t \in T_\Gamma(B), P'(t)) \Leftrightarrow (\forall x \in X, P(x))$$

Définition récursive d'une fonction

Sur les ensembles définis inductivement, nous pouvons profiter de la structure constructive des éléments de l'ensemble pour définir récursivement des fonctions sur cet ensemble. C'est l'approche suivie quand on écrit un programme implantant une fonction f . En effet, on utilise la structure inductive des structures de données en entrée de la fonction f pour décrire de façon récursive son comportement. Comme nous le verrons dans la seconde partie de ce document, c'est à partir de cette notion de récursivité que l'on définit rigoureusement la notion d'algorithme.

Bien sûr, étant donnés un ensemble X défini inductivement et une fonction $f : X \rightarrow A$, pour

3. Là encore, nous considérons que tous les éléments $x \in B$ sont des fonctions constantes $x : \rightarrow E$, et donc $T^1 = B$.

assurer la propriété pour f d'être fonctionnelle, il faut que l'on s'assure qu'à tout élément $x \in X$, on ne peut associer qu'une unique valeur par f . La fonction f étant définie récursivement à partir de la structure inductive des éléments de X (i.e. des éléments dans $T_\Gamma(B)$), il faut que l'on s'assure alors que toutes les expressions t dans $T_\Gamma(B)$ s'évaluent par $_^X$ de façon unique. Quand l'ensemble X vérifie une telle propriété, on dira qu'il est *non ambigu*.

Définition 17 (Définition inductive non ambigu).

Une définition inductive d'un ensemble X est dite **non ambigu** si l'application $_^X$ est bijective, i.e. pour tout $x \in X$, il existe une unique expression $t \in T_\Gamma(B)$ telle que $t^X = x$.

Exemple 22.

- La définition des arbres binaires donnée dans l'exemple 20 est non ambigu. Nous n'avons qu'un choix unique de construction d'un arbre binaire à chaque étape.
- À l'inverse, la définition inductive suivante de $\mathbb{N} \times \mathbb{N}$ est ambigu :

(B) $(0, 0) \in \mathbb{N} \times \mathbb{N}$

(I) $(n, m) \in \mathbb{N} \times \mathbb{N}$ et $n \geq m \Rightarrow (n + 1, m) \in \mathbb{N} \times \mathbb{N}$ et $(n, m + 1) \in \mathbb{N} \times \mathbb{N}$

Il suffit de considérer n'importe quel couple d'entiers positifs (n, m) avec $n \neq 0$ et $m \neq 0$ qui peut s'obtenir à partir du couple $(n - 1, m - 1)$ en ajoutant une unité au premier argument puis au second, ou l'inverse. Une définition inductive non ambigu de $\mathbb{N} \times \mathbb{N}$ peut être la suivante :

(B) $(0, 0) \in \mathbb{N} \times \mathbb{N}$ et $\forall n, m \in \mathbb{N}, n < m \Rightarrow (n, m) \in \mathbb{N} \times \mathbb{N}$

(I) $(n, m) \in \mathbb{N} \times \mathbb{N} \Rightarrow (n + m, m) \in \mathbb{N} \times \mathbb{N}$

Cette définition de $\mathbb{N} \times \mathbb{N}$ n'est pas la plus naturelle. En fait, comme on pouvait s'y attendre, elle est dictée par la définition récursive d'une fonction sur les couples d'entiers positifs, ici la fonction modulo

$$n \text{ modulo } m = \begin{cases} n & \text{si } n < m \\ (n - m) \text{ modulo } m & \text{sinon} \end{cases}$$

Intuitivement, pour définir récursivement une fonction sur les éléments d'un ensemble non ambigu, on commence par définir la fonction sur les éléments de l'ensemble de base puis inductivement sur les éléments nouveaux construits à partir d'éléments déjà connus.

Définition 18 (Définition récursive d'une fonction).

Soit $X \subseteq E$ un ensemble défini inductivement et non ambigu. Soit A un ensemble quelconque. La **définition récursive** d'une application $f : X \rightarrow A$ est la donnée d'une application $g : B \rightarrow A$ et pour tout $F_i : E^{n_i} \rightarrow E \in \Gamma$ d'une application $h_{F_i} : E^{n_i} \times A^{n_i} \rightarrow A$ telle que :

- $f(x) = g(x)$ pour tout $x \in B$
- $f(F_i(x_1, \dots, x_{n_i})) = h_{F_i}(x_1, \dots, x_{n_i}, f(x_1), \dots, f(x_{n_i}))$

Exemple 23.

Reprenons l'application modulo définie sur la définition non ambiguë de $\mathbb{N} \times \mathbb{N}$. Dans cette définition, l'ensemble $B = \{(0, 0)\} \cup \{(n, m) | n < m\}$ et $\Gamma = \{F : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}\}$ où F est la fonction définie par $(n, m) \mapsto (n + m, m)$ pour tout couple $(n, m) \in \mathbb{N} \times \mathbb{N}$ et $n, m \neq 0$, et les fonctions g et h_F se définissent par : $g : (n, m) \mapsto n$ et $h_F : ((n, m), p) \mapsto p$. La définition récursive de modulo (i.e. la fonction f) s'écrit alors :

- $f(n, m) = g(n, m) = n$ pour tout $(n, m) \in B$
- $f(F(n, m)) = f(n + m, m) = h_F((n, m), f(n, m)) = n \text{ modulo } m = f(n, m)$

Nous verrons dans la seconde partie de ce document que les fonctions qui se définissent selon la définition 18 caractérisent bien la notion intuitive d'algorithme.

4.3. Application : sémantique dénotationnelle des langages de programmation

Les programmes écrits dans n'importe quel langage de programmation sont des objets syntaxiques auxquels il faut pouvoir donner une signification pour permettre d'analyser leur comportement, cette signification pouvant aussi bien être *dénotationnelle* (caractérisation abstraite du comportement) qu'*opérationnelle* (caractérisation concrète du comportement). L'ensemble des programmes que l'on peut écrire dans tout langage de programmation a une particularité, il suit une définition inductive non ambiguë. On peut donc définir des fonctions récursives sur cet ensemble. Les sémantiques dénotationnelle et opérationnelle se définissent justement par de telles fonctions⁴. Ici, nous allons nous intéresser plus particulièrement à la sémantique dénotationnelle d'un langage de programmation jouet (donc très simple dans sa syntaxe) mais suffisamment puissant pour exprimer tous les problèmes calculables (cf. la seconde partie de ce document pour une définition formelle de la notion de calculabilité). Pourquoi présenter la sémantique dénotationnelle d'un tel langage ? parce qu'elle est une application directe des treillis et des résultats de points fixes associés. Enfin, l'intérêt d'une sémantique formelle (i.e. mathématiquement définie) des langages de programmation, est de permettre de raisonner sur les programmes générés pour démontrer rigoureusement leur correction partielle, i.e. montrer sans les exécuter que les programmes font bien ce que l'on attend d'eux.

Un langage de programmation simple

Ici, nous donnons la syntaxe de notre langage de programmation d'étude que nous appelons *WHILE*. *WHILE* est un langage dit **impératif**, i.e. que l'instruction de base est l'affectation. Les langages PYTHON, C, C++, COBOL, FORTRAN, JAVA, etc. sont des langages impératifs. Il existe aussi deux autres paradigmes de programmation, celle dite **fonctionnelle** où l'on évalue des fonctions (e.g. le langage CAML) et celle dite **logique** où l'on infère des faits (e.g. PROLOG - cf. le cours électif en S8 "Fondements logiques de l'informatique").

Comme tout langage de programmation, *WHILE* est composé de types de données prédéfinis, d'expressions sur ces types et d'instructions. À la fois, les expressions et les instructions vont

4. Il existe aussi une sémantique axiomatique des langages de programmation qui définit non pas une fonction mais un prédicat de façon récursive sur les programmes.

suivre une définition inductive. On supposera que l'on ne peut pas définir d'autres types à l'exception de ceux donnés explicitement. En ce sens, le langage proposé n'est pas réaliste.

Les types autorisés dans le langage sont les suivants : les entiers *int*, les booléens *bool*, les caractères ASCII *char*, les chaînes de caractères *string*, et les listes *list[t]* dont tous les éléments sont d'un même type donné *t*, ce dernier pouvant être un des types précédents. Entre autres, il peut être un type *list[t']*. Formellement, l'ensemble des types $\mathbb{T}$ est l'ensemble défini inductivement par :

- $\mathbb{T}_0 = \{int, bool, char, string\}$
- $\mathbb{T}_n = \mathbb{T}_{n-1} \cup \{list[t] | t \in \mathbb{T}_{n-1}\}$

Sur ces types on peut écrire des expressions. Comme il est d'usage dans les langages de programmation, on va utiliser la notation à la BNF (Backus-Naur Form) pour donner la définition inductive de ces expressions.

- les expressions arithmétiques : $a ::= n | v | a + a | a \times a | -a$ ($n \in \mathbb{Z}$ utilisé comme une constante, et v est une variable de type *int*)
- les expressions booléennes : $b ::= true | false | v | not\ b | b\ and\ b | e == e | e < e | e \leq e$ où v est une variable de type *bool* et e est une expression arithmétique, un caractère ou une expression sur les chaînes de caractère. Bien sûr les expressions prises dans une égalité ou une inégalité doivent être de même type.
- les expressions sur les chaînes de caractères : $c ::= [] | [char_1, \dots, char_n] | v | c[i] | c + c | size(c)$ où $char_j$ est un des 256 caractères ASCII, v une variable de type *string* et i est une expression arithmétique qui dénote une position dans la chaîne c . On supposera que les positions dans une chaîne de caractères commencent à 1.
- les expressions sur les listes : $l ::= [] | [e_1; \dots; e_n] | v | l[i] | l + l | size(l)$ où tous les e_j sont des expressions de même type, et v est une variable de type *list[t]*. Comme pour les chaînes de caractères, on supposera que les positions dans une liste commencent à 1.

Enfin, on définit les instructions de notre langage *WHILE*. Là encore, nous allons utiliser une notation à la BNF. On suppose s'être donnée une famille indexée par les types d'ensembles de variables $V = (V_t)_{t \in \mathbb{T}}$.

$$I ::= skip | x = e | c[i] = char | l[i] = e | I; I | if\ b\ then\ I\ else\ I | while\ b\ do\ I$$

où x , c et l sont des variables de type $t \in \mathbb{T}$, *string* et *list[t]*, respectivement, e est une expression de même type que la variable x , i est une variable ou expression de type *int* et dénote la position dans la chaîne de caractère c ou la liste l , et b est une expression booléenne.

Nous aurions aussi pu introduire la récursivité, mais ceci aurait demandé au préalable d'introduire des notations fonctionnelles qui auraient alourdi la présentation du langage sans apporter de connaissances supplémentaires à la sémantique dénotationnelle d'un langage de programmation.

Un programme P dans le langage *WHILE* se définit alors :

```

Prog P(x_1:t_1; ...; x_n:t_n) : y_1:t'_1; ...; y_m:t'_m
  z_1:t''_1; ...; z_p:t''_p
begin
  I
end

```

Les variables x_i définissent les arguments en entrée du programme P , tandis que les variables y_j sont les variables de sorties. Les variables z_l sont des variables locales utiles aux calculs.

Exemple 24.

Donnons le programme qui recherche un élément dans une liste.

```

Prog Recherche(e:t;l:list[t]):b:bool
  i:int
  begin
    i = 1;
    if (l == [])
      then b = false
      else b = (l[i] == e);
      while not b
        i = i + 1;
        b = (i > size(l)) or (l[i] == e);
        b = (i < size(l))
    end
  end

```

Sémantique dénotationnelle du langage *WHILE*

Un programme dans notre langage *WHILE* transforme les états du programme définis par les contenus des variables. Ainsi, la sémantique dénotationnelle d'un programme sera alors définie par une fonction partielle sur l'ensemble de ces états. La fonction est partielle car il se peut que le programme ne termine pas. Ainsi, la sémantique d'un programme P est une fonction $\llbracket P \rrbracket : S \rightarrow S$ où S est l'ensemble des états du programme, i.e. des fonctions de V dans la sémantique associée aux types. Donc, commençons par associer une sémantique $\llbracket t \rrbracket$ à tout type $t \in \mathbb{T}$. L'ensemble $\mathbb{T}$ étant défini de façon inductive, nous allons définir leur sémantique profitant de cette structure.

- $\llbracket \text{int} \rrbracket = \mathbb{Z}$, $\llbracket \text{bool} \rrbracket = \mathbb{B} = \{0, 1\}$, $\llbracket \text{char} \rrbracket = 256$ caractères ASCII, $\llbracket \text{string} \rrbracket = \llbracket \text{char} \rrbracket^*$
- $\llbracket \text{list}[t] \rrbracket = \llbracket t \rrbracket^*$

Soit un programme P dont les variables d'entrée, de sortie et locales sont choisies parmi un ensemble de variables $V = (V_t)_{t \in \mathbb{T}}$. Par définition, pour un type donné $t \in \mathbb{T}$, le contenu des variables dans V_t doit être choisi parmi les valeurs dans $\llbracket t \rrbracket$. Ainsi, chaque état du programme se définit par une famille indexée par $\mathbb{T}$ d'applications $V_t \rightarrow \llbracket t \rrbracket$. On notera ces états σ . Pour toute variable $x \in V_t$, $\sigma(x) \in \llbracket t \rrbracket$ est la valeur de x dans cet état. Enfin, $\sigma[x/val]$ où $x \in V_t$ et $val \in \llbracket t \rrbracket$ désigne le nouvel état σ' tel que pour tout $y \neq x \in V$, $\sigma'(y) = \sigma(y)$ et $\sigma'(x) = val$. L'ensemble S des états est alors l'ensemble des applications σ de V dans $\llbracket \mathbb{T} \rrbracket$, noté aussi $\llbracket \mathbb{T} \rrbracket^V$, qui respectent le typage (i.e. pour tout $x \in V_t$, $\sigma(x) \in \llbracket t \rrbracket$).

À partir d'un état σ , nous pouvons maintenant évaluer les expressions. Soit e une expression du langage et σ un état du programme. On note $\llbracket e \rrbracket_\sigma$ son évaluation, et on la définit de la façon suivante :

- Évaluation des expressions arithmétiques : $\llbracket n \rrbracket_\sigma = n$, $\llbracket v \rrbracket_\sigma = \sigma(v)$, $\llbracket e_1 @ e_2 \rrbracket_\sigma = \llbracket e_1 \rrbracket_\sigma @ \llbracket e_2 \rrbracket_\sigma$ où $@ \in \{+, \times\}$, et $\llbracket -e \rrbracket_\sigma = -\llbracket e \rrbracket_\sigma$
- Évaluation des expressions booléennes : $\llbracket \text{true} \rrbracket_\sigma = 1$, $\llbracket \text{false} \rrbracket_\sigma = 0$, $\llbracket v \rrbracket_\sigma = \sigma(v)$, $\llbracket \text{not } b \rrbracket_\sigma = 1 - \llbracket b \rrbracket_\sigma$, $\llbracket b_1 \text{ and } b_2 \rrbracket_\sigma = \llbracket b_1 \rrbracket_\sigma \times \llbracket b_2 \rrbracket_\sigma$, $\llbracket e_1 == e_2 \rrbracket_\sigma = (\llbracket e_1 \rrbracket_\sigma = \llbracket e_2 \rrbracket_\sigma)$, $\llbracket e_1 < e_2 \rrbracket_\sigma = (\llbracket e_1 \rrbracket_\sigma < \llbracket e_2 \rrbracket_\sigma)$, $\llbracket e_1 \leq e_2 \rrbracket_\sigma = (\llbracket e_1 \rrbracket_\sigma \leq \llbracket e_2 \rrbracket_\sigma)$. Dans le cas où e_1 et e_2 sont des chaînes de caractères, $<$ et $\leq$ désignent les ordres lexicographiques strict et total, respectivement.

- Évaluation des expressions sur les chaînes de caractères : $\llbracket [] \rrbracket_\sigma = \varepsilon$ (le mot vide), $\llbracket [char_1, \dots, char_n] \rrbracket_\sigma = char_1 \dots char_n$, $\llbracket v \rrbracket_\sigma = \sigma(v)$, $\llbracket c[i] \rrbracket_\sigma = c_i$ quand $\llbracket c \rrbracket_\sigma = c_1 \dots c_n$ et $1 \leq \llbracket i \rrbracket_\sigma \leq n$, sinon $\llbracket c[i] \rrbracket_\sigma = \varepsilon$, $\llbracket c_1 + c_2 \rrbracket_\sigma = \llbracket c_1 \rrbracket_\sigma \cdot \llbracket c_2 \rrbracket_\sigma$, et $\llbracket size(c) \rrbracket_\sigma = n$ si $\llbracket c \rrbracket_\sigma = c_1 \dots c_n$.
- Évaluation des expressions sur les listes : $\llbracket [] \rrbracket_\sigma = \varepsilon$, $\llbracket [e_1, \dots, e_n] \rrbracket_\sigma = \llbracket e_1 \rrbracket_\sigma \dots \llbracket e_n \rrbracket_\sigma$, $\llbracket v \rrbracket_\sigma = \sigma(v)$, $\llbracket l[i] \rrbracket_\sigma = a_i$ quand $\llbracket l \rrbracket_\sigma = a_1 \dots a_n$ et $1 \leq \llbracket i \rrbracket_\sigma \leq n$, sinon $\llbracket l[i] \rrbracket_\sigma = \varepsilon$, $\llbracket l_1 + l_2 \rrbracket_\sigma = \llbracket l_1 \rrbracket_\sigma \cdot \llbracket l_2 \rrbracket_\sigma$, et $\llbracket size(l) \rrbracket_\sigma = n$ si $\llbracket l \rrbracket_\sigma = l_1 \dots l_n$.

Évaluons maintenant les instructions toujours en profitant de la structure inductive de leur définition. Chaque instruction ι va donc définir une fonction partielle $\llbracket \iota \rrbracket : S \rightarrow S$. Ci-dessous, nous allons donner la sémantique des instructions à l'exception de la boucle "while", celle-ci demandant un traitement particulier.

- $\llbracket skip \rrbracket = Id_S$ (l'application identité sur S)
- $\llbracket x = e \rrbracket : \sigma \mapsto \sigma[x/\llbracket e \rrbracket_\sigma]$
- $\llbracket c[i] = e \rrbracket : \sigma \mapsto \sigma[c/\alpha]$ quand $\sigma(c) = c_1 \dots c_n$ et $1 \leq \llbracket i \rrbracket_\sigma \leq n$, et dans ce cas-là $\alpha = c'_1 \dots c'_n$ tel que pour tout j , $1 \leq j \leq n$, $j \neq \llbracket i \rrbracket_\sigma \Rightarrow c'_j = c_j$ et $j = \llbracket i \rrbracket_\sigma \Rightarrow c'_j = \llbracket e \rrbracket_\sigma$, sinon $\llbracket c[i] = e \rrbracket$ est indéfini pour σ .
- $\llbracket l[i] = e \rrbracket : \sigma \mapsto \sigma[l/\alpha]$ quand $\sigma(l) = l_1 \dots l_n$ et $1 \leq \llbracket i \rrbracket_\sigma \leq n$, et dans ce cas-là $\alpha = l'_1 \dots l'_n$ tel que pour tout j , $1 \leq j \leq n$, $j \neq \llbracket i \rrbracket_\sigma \Rightarrow l'_j = l_j$ et $j = \llbracket i \rrbracket_\sigma \Rightarrow l'_j = \llbracket e \rrbracket_\sigma$, sinon $\llbracket l[i] = e \rrbracket$ est indéfini pour σ .
- $\llbracket I; I' \rrbracket = \llbracket I' \rrbracket \circ \llbracket I \rrbracket$
- $\llbracket if\ b\ then\ I\ else\ I' \rrbracket : \sigma \mapsto \begin{cases} \llbracket I \rrbracket(\sigma) & \text{si } \llbracket b \rrbracket_\sigma = 1 \\ \llbracket I' \rrbracket(\sigma) & \text{sinon} \end{cases}$

Terminons la sémantique de notre langage *WHILE* en traitant le cas de la boucle "while". Par définition, la boucle while satisfait le comportement suivant :

$$while\ b\ do\ I \equiv if\ b\ then\ I; while\ b\ do\ I\ else\ skip$$

Sémantiquement, on a alors :

$$\llbracket while\ b\ do\ I \rrbracket : \sigma \mapsto \begin{cases} \llbracket while\ b\ do\ I \rrbracket(\llbracket I \rrbracket(\sigma)) & \text{si } \llbracket b \rrbracket_\sigma = 1 \\ \sigma & \text{sinon} \end{cases}$$

Considérons alors la fonction $f : S_p^S \rightarrow S_p^S$ où S_p^S est l'ensemble des fonctions partielles de S dans S :

$$w \mapsto (\sigma \mapsto \begin{cases} w(\llbracket I \rrbracket(\sigma)) & \text{si } \llbracket b \rrbracket_\sigma = 1 \\ \sigma & \text{sinon} \end{cases})$$

La sémantique de la boucle "while" se réécrit alors :

$$\llbracket while\ b\ do\ I \rrbracket = f(\llbracket while\ b\ do\ I \rrbracket)$$

Ainsi, la sémantique de la boucle "while" se définit par un point fixe de la fonction f . On attend d'une boucle "while" qu'elle ait un nombre fini d'étapes d'exécution. Le point fixe recherché est alors le plus petit. Par le théorème 19, l'on sait exprimer (voir calculer) ce point fixe. Mais ceci demande à ce que S_p^S et f soit respectivement, un ordre partiel complet⁵ et une application continue. Associons à S_p^S l'ordre $\preceq$ suivant :

$$g \preceq g' \Leftrightarrow dom(g) \subseteq dom(g') \text{ et } \forall \sigma \in dom(g), g(\sigma) = g'(\sigma)$$

5. En fait, il n'est pas possible d'associer une structure de treillis à $(S_p^S, \preceq)$. En effet, à deux fonctions g et g' il n'est pas possible d'associer une borne supérieure car elles ne sont pas nécessairement en accord sur leur intersection des domaines $dom(g)$ et $dom(g')$. Ceci est aussi vrai pour leur borne inférieure.

où $\text{dom}(g)$ désigne le sous-ensemble de S pour lequel g est définie. $\preceq$ est un ordre car défini à partir de l'inclusion. $(S_p^S, \preceq)$ possède un élément minimal, la fonction partielle $\perp : S \rightarrow S$ définie nul part. Enfin, toute chaîne de fonctions partielles (g_i) possède une borne supérieure qui n'est autre que $\bigcup_i g_i$. Ainsi, $(S_p^S, \preceq)$ est un ordre partiel complet.

Pour pouvoir appliquer le théorème 19, il suffit de montrer que f préserve la borne supérieure des chaînes de fonctions dans S_p^S . Soit (g_i) une chaîne dont la borne supérieure est g . On observe que f est une fonction monotone. Ainsi, l'ensemble $(f(g_i))$ est aussi une chaîne dont la borne supérieure est h . On doit montrer que $f(g) = h$. Par définition, on a

$$f(g)(\sigma) = \begin{cases} g(\llbracket I \rrbracket(\sigma)) & \text{si } \llbracket b \rrbracket_\sigma = 1 \\ \sigma & \text{sinon} \end{cases}$$

Si $\llbracket b \rrbracket_\sigma = 0$, alors $f(g)(\sigma) = \sigma$, et donc $\forall i, f(g_i)(\sigma) = \sigma$, d'où l'on déduit aussi que $h(\sigma) = \sigma$. Si $\llbracket b \rrbracket_\sigma = 1$, alors on a $f(g)(\sigma) = g(\llbracket I \rrbracket(\sigma))$. Ici, on a alors deux possibilités :

1. $g(\llbracket I \rrbracket(\sigma))$ est indéfini. Ceci veut dire que pour tout i , $g_i(\llbracket I \rrbracket(\sigma))$ est indéfini, et donc $f(g_i)(\sigma)$ aussi, d'où l'on déduit que $h(\sigma)$ est indéfini.
2. $g(\llbracket I \rrbracket(\sigma))$ est défini. Ceci veut dire qu'il existe i tel que $g_i(\llbracket I \rrbracket(\sigma))$ est défini, et donc $f(g_i)(\sigma)$ aussi. On a alors $h(\sigma) = f(g_i)(\sigma) = g_i(\llbracket I \rrbracket(\sigma)) = g(\llbracket I \rrbracket(\sigma)) = f(g)(\sigma)$.

On peut appliquer le théorème 19 et donner la dénotation suivante aux boucles "while" :

$$\llbracket \text{while } b \text{ do } I \rrbracket = \sqcup \{f^n(\perp) \mid n \in \mathbb{N}\}$$

Exemple 25.

En reprenant notre exemple 24, la méthode itérative calcule pour la boucle while la solution $g : S \rightarrow S$ de l'équation à point fixe $g = f(g)$ comme la limite de la suite de fonctions $g_0, g_1, \dots, g_n, \dots$ définie inductivement par :

$$\begin{aligned} g_0 &= \perp \\ g_n &= f(g_{n-1}) \end{aligned}$$

Ainsi, par définition, g_0 est la fonction partielle définie nulle part. g_1 est alors la fonction définie par :

$$\sigma \mapsto \begin{cases} \perp(\llbracket i = i + 1; b = (i > \text{size}(l)) \text{ or } (l[i] == e) \rrbracket(\sigma)) & \text{si } \sigma(b) = 0 \\ \sigma & \text{sinon} \end{cases}$$

$\perp$ étant la fonction définie nulle part, la fonction g_1 est seulement définie pour tout état $\langle e, l_1 \dots l_m, 1, i \rangle$ tel que $m \geq i, \forall j, 1 \leq j \leq i - 1, l_j \neq e$ et $l_i = e$. La fonction g_1 se comporte alors comme l'identité sur ces états.

Si nous examinons g_2 , nous avons

$$\begin{aligned} g_1(\llbracket i = i + 1; b = (i > \text{size}(l)) \text{ or } (l[i] == e) \rrbracket(\sigma)) & \quad \text{si } \sigma(b) = 0 \\ \sigma & \quad \text{sinon} \end{aligned}$$

Donc, g_2 est définie pour tout état $\sigma = \langle e, l_1 \dots l_m, b, i \rangle$ tel que :

- $\forall j, 1 \leq j \leq i, l_j \neq e, m \geq i$, et
- soit $m \geq i + 1$ et $l_{i+1} = e$, soit $m = i$

Pour tous ces états σ , g_2 est définie par :

$$g_2(\sigma) = \begin{cases} \langle e, l_1 \dots l_m, 1, i + 1 \rangle & \text{si } m = i \text{ ou } l_{i+1} = e \\ \langle e, l_1 \dots l_m, 0, i + 1 \rangle & \text{sinon} \end{cases}$$

En procédant de façon récursive, la fonction $g_n : S \rightarrow S$ est la fonction partielle définie pour tout état $\sigma = \langle e, l_1 \dots l_m, i, b \rangle$ tel que $i \leq n, \forall j, 1 \leq j \leq i, l_j \neq e, m \geq n$, de la façon suivante :

$$g_n(\sigma) = \begin{cases} \langle e, l_1 \dots l_m, 1, k \rangle & \text{si } i + 1 \leq k \leq i + n \leq m \text{ et} \\ & ((\forall j, i + 1 \leq j \leq k - 1, l_j \neq e \text{ et } l_k = e) \text{ ou } m = i + (k - 1)) \\ \langle e, l_1 \dots l_m, 0, m + 1 \rangle & \text{sinon} \end{cases}$$

Par le théorème 19, la fonction g associée à la boucle while dans le programme de recherche d'un élément dans une liste, n'est autre que la fonction définie pour tout état $\sigma = \langle e, l, i, b \rangle$ tel que $1 \leq i \leq |l|, \forall j, 1 \leq j \leq i, l_j \neq e$, de la façon suivante :
Pour tous ces états σ , g est défini par :

$$g(\sigma) = \begin{cases} \langle e, l, 0, 1 \rangle & \text{si } l = \varepsilon \\ \langle e, l, 1, k \rangle & \text{si } i + 1 \leq k \leq |l| \text{ et} \\ & ((\forall j, i + 1 \leq j \leq k - 1, l_j \neq e \text{ et } l_k = e) \text{ ou } m = i + (k - 1)) \\ \langle e, l, 0, |l| + 1 \rangle & \text{sinon} \end{cases}$$

SOUS PARTIE 2

Calculabilité

5. Introduction

La formalisation du raisonnement occupe les esprits depuis longtemps. En inventant/découvrant la logique, les Grecs furent les premiers à émettre l'idée que tout raisonnement – de nature scientifique ou philosophique – pouvait être réduit à un calcul, c'est-à-dire à une technique permettant de prouver un énoncé en respectant des règles autorisées. Leur but était de mettre fin aux désaccords lors des discussions. Au XVII^{ème} siècle, G. Leibniz affirma même qu'il était possible de réduire tout raisonnement et toute théorie à une combinaison d'éléments de base tels que les nombres, les lettres, les couleurs, etc. Il ébaucha alors une *algèbre de pensée* et définît la première table de calcul binaire. C'est cette *algèbre de pensée* qui inspira d'ailleurs G. Boole lorsqu'il réduisit le raisonnement logique au calcul algébrique à base binaire, que l'on appelle *algèbre de Boole*.

Depuis E. Kant, on savait que seules les questions scientifiques pouvaient être prouvées par un raisonnement rigoureux et non ambigu, et que les énoncés métaphysiques comme "Dieu existe-t-il ?" ou "L'âme est-elle immortelle ?" amenaient nécessairement à un désaccord persistant sur leur valeur de vérité. Sachant qu'un raisonnement est rigoureux et non ambigu s'il peut être traduit dans un langage mathématique, les mathématiques devenaient donc le langage des sciences. Il était alors nécessaire d'étudier les objets mathématiques, à plus forte raison parce que certains objets mathématiques incongrus venaient d'apparaître (par exemple une courbe sans tangente ou les géométries non euclidiennes). Le XIX^{ème} siècle et le premier tiers du XX^{ème} siècle furent l'époque de la réflexion sur les objets mathématiques. De cette réflexion sont nées, au cours du XX^{ème} siècle, la *théorie de la démonstration* et la notion de *calculabilité*. Le but de ces recherches était de mettre fin à la "crise des fondements", principalement après la découverte de paradoxes dans le raisonnement mathématique. Pour l'ensemble des participants, l'idée était de ramener les mathématiques à la seule arithmétique et à l'utilisation de procédés finitistes (L. Kronecker, L. Brower, D. Hilbert). À la même époque, A. Thue posait un problème linguistique qui eut un grand retentissement dans le monde mathématique mais aussi dans un monde qui n'existait pas encore : celui de l'informatique. Son problème s'énonçait ainsi : étant donné un alphabet et une grammaire (c'est-à-dire un ensemble de règles), un mot quelconque sur l'alphabet peut-il être dérivé de l'alphabet à l'aide de la grammaire ? Ce problème, connu sous le nom de *problème du mot pour les systèmes formels*, motiva de nombreux chercheurs. En effet, le résoudre équivalait à systématiser le raisonnement, c'est-à-dire à l'automatiser. L'équivalence est évidente lorsqu'on pose le problème de la façon suivante : étant donné un ensemble d'hypothèses et une démarche de raisonnement, une assertion quelconque est-elle une conséquence logique des hypothèses de départ ?

Dès 1931, la thèse finitiste fut mise en échec par les *théorèmes d'incomplétude* de K. Godel. Comme conséquence logique, il fut démontré que le problème du mot était *indécidable* pour de nombreux systèmes formels (par A. Church ou A. Turing pour l'arithmétique en 1936, par E. Post et A. Markov en parallèle pour les semi-groupes en 1947, par W. Boone pour les groupes en 1957, etc.). Cependant, de cet échec est née la notion d'indécidabilité. On savait maintenant montrer que certains problèmes mathématiques ne pouvaient être résolus par aucun algorithme ; il devenait alors intéressant de chercher ceux pour lesquels un algorithme existait. L'apparition des ordinateurs donna une motivation supplémentaire à cette recherche : même partielle, l'au-

tomatisation du raisonnement trouvait là toute sa justification. La partie automatique pourrait être confiée à l'ordinateur : il suffirait de lui fournir les hypothèses, les règles de déduction et le mot à prouver, et d'attendre sa réponse. Toutefois, ceci ne suffisait pas. En effet, rien implique que l'algorithme soit utilisable pratiquement : le temps et l'espace mémoire nécessaires à son exécution pourraient largement dépasser ce dont on peut espérer disposer. Ainsi, un algorithme utilisable se doit d'être relativement *efficace* (i.e il doit n'utiliser qu'une quantité "raisonnable" de ressources nécessaires au calcul en temps et en espace mémoire). Pour cela, nous devons être capable de distinguer un algorithme efficace à ceux qui ne le sont pas. Une théorie permettant de répondre à cette attente a été développée dans les années 70 et continue à occuper un nombre important de chercheurs en informatique théorique dans le monde : *la complexité*.

Dans cette partie, nous allons donner une définition rigoureuse à ces deux notions de décidabilité et complexité. Nous allons alors aborder la formalisation de la décidabilité de deux façons. Tout d'abord, du point de vue des mathématiques standards où l'on a l'habitude de manipuler des fonctions. On parlera alors de fonctions récursives. Enfin, on caractérisera la propriété pour un problème d'être décidable par une version mathématique plus opérationnelle des machines à calculer ou des ordinateurs, les *machines de Turing*. On montrera alors que ces deux notions qui au premier abord peuvent sembler différentes, sont en fait équivalentes. En fait, il existe une multitude de définitions mathématiques de la calculabilité traduisant pour certains les différents paradigmes de programmation (λ -calcul pour la programmation fonctionnelle, machine à registre pour la programmation impérative, et la résolution pour la programmation logique), mais tous ont été démontrés équivalents, ce qui peut nous laisser supposer très fortement que les formalisations de la calculabilité telle qu'elles ont été définies il y a maintenant presque un siècle et dont nous en présentons deux ici, ont cerné parfaitement ce qui peut être calculable par un ordinateur (*thèse de Church*). Nous finirons cette partie par la présentation de la théorie de la complexité. Cette théorie sera étudiée dans le cadre des machines de Turing plus aptes à prendre en compte la notion de calcul. Nous donnerons alors la définition des deux grandes classes de complexité, P et NP, qui contiennent respectivement les algorithmes dont le temps d'exécution est "humainement" abordable et ceux qui ne le sont pas. On finira alors par étudier la sous-classe dans NP des algorithmes dits canoniques (i.e. tels que tous les autres de la classe NP peuvent se ramener par une transformation raisonnable à n'importe lequel de ces problèmes) : les algorithmes *NP-complets*.

6. Fonctions primitives récursives et récursives

6.1. Codage

Le calcul est une activité qui manipule généralement des mots de longueur finie définis sur un alphabet donné. Ces mots peuvent aussi bien représenter des entiers, des arbres, ou tout autres structures de données mais aussi des programmes. On a vu dans la première partie de ce document, que l'ensemble de ces structures acceptait une définition inductive. Une propriété simple des définitions inductives est que chacun de leurs éléments peut être obtenu par une application finie des règles de production/construction. Ainsi, l'ensemble des mots finis que l'on manipule dans une activité calculatoire a la propriété d'être dénombrable. Or, on sait depuis Cantor que tous les ensembles dénombrables sont en bijection avec l'ensemble des entiers naturels. Ainsi, il existe une fonction bijective, appelée *codage*, qui permet de faire une correspondance un à un entre ces objets manipulés par tout programme et les entiers naturels. Un exemple de codage peut alors être le suivant : Soit A^* l'ensemble des mots finis construits sur un alphabet fini ou de cardinalité dénombrable A (ce que nous avons l'habitude de manipuler en programmation pour représenter les structures de données). L'alphabet A étant au plus de cardinalité dénombrable, on peut numéroter ses éléments et donc établir une bijection entre A et $\mathbb{N}$. On a alors le résultat suivant :

Proposition 26.

Pour chaque entier naturel p , il existe une fonction $\alpha_p : \prod_{i=1}^p \mathbb{N} \rightarrow \mathbb{N}$ qui est une bijection de $\prod_{i=1}^p \mathbb{N}$ dans $\mathbb{N}$.

Démonstration On commence par construire α_2 . Pour cela, on va suivre la méthode qui a été utilisée par Cantor pour démontrer que $\mathbb{Q}$ est en bijection avec $\mathbb{N}$. On va numéroter les couples d'entiers naturels (x, y) en suivant les diagonales $x + y = k$ pour k un entier naturel croissant (i.e. en suivant l'ordre des entiers k dans $\mathbb{N}$). On commence par la diagonale $x + y = 0$ (qui ne contient qu'un seul couple), puis on passe à la diagonale $x + y = 1$ en commençant par le bas (i.e. par le couple $(1, 0)$), etc. La diagonale $x + y = n$ a exactement $n + 1$ éléments. Donc avant le couple $(p + n, 0)$ il y a exactement $1 + 2 + \dots + (n + p) = \frac{1}{2}(n + p)(n + p + 1)$. Le couple (p, n) qui se trouve sur la même diagonale que $(p + n, 0)$, a exactement n places après lui. On peut donc poser $\alpha_2 : (p, n) \mapsto \frac{1}{2}(n + p)(n + p + 1) + n$. Par définition, cette fonction est bijective de

$\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$. On en déduit immédiatement pour chaque $p \geq 1$ une bijection $\alpha_p : \prod_{i=1}^p \mathbb{N} \rightarrow \mathbb{N}$

définie par récurrence sur p par :

$$\alpha_1 : n \mapsto n$$

$$\alpha_{p+1} : (n_1, \dots, n_p, n_{p+1}) \mapsto \alpha_2(\alpha_p((n_1, \dots, n_p)), n_{p+1})$$

□

On peut alors coder toute suite d'entiers à partir de ces fonctions α_p pour $p \in \mathbb{N}$. En effet, soit $\mathcal{S}^* = \bigcup_{p \in \mathbb{N}} \mathbb{N}^p$. On définit l'application $\alpha : \mathcal{S}^* \rightarrow \mathbb{N}$ pour toute suite d'entiers σ de longueur p par :

$$\alpha(\sigma) = \alpha_2(p, \alpha_p(\sigma))$$

Théorème 27.

α est une bijection.

Démonstration Composition de deux fonctions bijectives. □

La notion de calculabilité dont je rappelle qu'elle caractérise ce qui peut être calculé par un ordinateur, peut alors se définir par des fonctions $f : \mathbb{N}^k \rightarrow \mathbb{N}$ où $\mathbb{N}^k = \prod_{i=1}^k \mathbb{N}$ avec pour convention que $\mathbb{N}^0$ est l'ensemble ne contenant que la suite vide. Ces fonctions peuvent être partielles (i.e. indéfinies sur certaines valeurs) afin de prendre en compte le fait qu'un programme puissent ne jamais s'arrêter sur certaines valeurs. La question que l'on peut se poser alors, est "est-ce que toute fonction de ce type est effectivement calculable par un ordinateur?". Depuis les travaux de Cantor, nous pouvons répondre non à cette question. La raison est que l'ensemble des fonctions $f : \mathbb{N} \rightarrow \mathbb{N}$ a déjà une cardinalité strictement supérieure à $\mathbb{N}$. C'est une conséquence triviale du théorème fondamental de Cantor qui énonce pour tout ensemble E , l'inéquation suivante : $|E| < |\mathcal{P}(E)|$ où $\mathcal{P}(E)$ est l'ensemble des parties de E et $|E|$ dénote la cardinalité de E . En effet, pour l'ensemble des fonctions $f : \mathbb{N} \rightarrow \mathbb{N}$, il y a toutes les fonctions $f : \mathbb{N} \rightarrow \{0, 1\}$. Ce sous-ensemble de fonctions est isomorphe à l'ensemble des parties de $\mathbb{N}$ (connu aussi comme isomorphe à $\mathbb{R}$). Or on vient de voir que la cardinalité de ce sous-ensemble est plus grand que $\mathbb{N}$. Maintenant, par leur définition récursive, l'ensemble des programmes que l'on peut écrire dans un langage de programmation donné est forcément dénombrable. Ainsi, l'ensemble $\mathcal{F} = \bigcup_{k \in \mathbb{N}} \mathcal{F}_k$

où $\mathcal{F}_k$ est l'ensemble de toutes les fonctions $f : \mathbb{N}^k \rightarrow \mathbb{N}$ contient nécessairement des fonctions non calculables par un ordinateur. Existe-t-il alors un moyen de caractériser le sous-ensemble des fonctions qui sont effectivement calculables? Oui, et ceci a été fait au siècle dernier par d'éminents logiciens et mathématiciens tels que K. Gödel, A. Turing, A. Church, J. Herbrand ou encore W. Ackermann. Cette caractérisation s'est faite en deux étapes. Tout d'abord, un premier sous-ensemble a été défini les *fonctions primitives récursives*. Cependant, cet ensemble, bien que fort riche, ne traduit pas complètement l'ensemble des fonctions pour lesquelles on peut effectivement trouver un algorithme qui les calcule. En effet, W. Ackermann a défini une telle fonction, la *fonction d'Ackermann*, dont on démontrera qu'elle n'est pas primitive récursive. En fait, avec le recul que nous avons maintenant sur les langages de programmation, ce que l'on constate alors est que les fonctions primitives récursives traduisent les fonctions que l'on peut calculer avec une boucle **For**. Cependant, il existe aussi une multitude de fonctions qui ont besoin d'une boucle **While** et dont le comportement peut aussi ne pas s'arrêter. C'est justement, ce que

permettront de caractériser les *fonctions récursives* en introduisant un *schéma de minimisation non bornée* et donc la notion de fonctions entières (dites aussi numériques) partielles.

6.2. Les fonctions primitives récursives

Définition

Le mécanisme de récursion comme moyen de définition de fonctions numériques est connu depuis Dedekind et Peano. Par exemple, l'addition peut se définir récursivement de la façon suivante :

$$\begin{aligned}x + 0 &= x \\x + (y + 1) &= (x + y) + 1\end{aligned}$$

Dans l'exemple ci-dessus, la récursion porte alors sur le second argument de l'addition. La multiplication aussi peut se définir avec un schéma récursif identique :

$$\begin{aligned}x \times 0 &= 0 \\x \times (y + 1) &= (x \times y) + x\end{aligned}$$

Là encore, la récursion porte sur le second argument de la multiplication. Ce schéma récursif se généralise au travers de la notion de fonction primitive récursive dont la définition formelle est la suivante :

Définition 19 (Fonction primitive recursive).

L'ensemble des **fonctions primitives récursives** est le plus petit des sous-ensembles $\mathcal{E}$ de $\mathcal{F}$ (l'ensemble de toutes les fonctions dans $\mathbb{N}$ aussi appelées **fonctions numériques**) satisfaisant les conditions suivantes :

1. $\mathcal{E}$ contient toutes les fonctions constantes de $\mathbb{N}^k$ dans $\mathbb{N}$ pour tout entier $k \in \mathbb{N}$.
2. $\mathcal{E}$ contient toutes les projections $\pi_k^i : (x_1, \dots, x_k) \mapsto x_i$ pour tous entiers $i \in \mathbb{N}$ et $k \in \mathbb{N}$ tels que $1 \leq i \leq k$.
3. $\mathcal{E}$ contient la fonction successeur $s : x \mapsto x + 1$.
4. $\mathcal{E}$ est clos par composition, ce qui veut dire que si n et p sont des entiers naturels, que $f_1, \dots, f_n$ sont des fonctions de $\mathcal{E}$ définies de $\mathbb{N}^p$ dans $\mathbb{N}$ et $g : \mathbb{N}^n \rightarrow \mathbb{N}$ est aussi une fonction de $\mathcal{E}$, alors la fonction $g(f_1, \dots, f_n) : \mathbb{N}^p \rightarrow \mathbb{N}$ qui à tout $(x_1, \dots, x_p) \in \mathbb{N}^p$ associe $g(f_1(x_1, \dots, x_p), \dots, f_n(x_1, \dots, x_p))$, est une fonction de $\mathcal{E}$.
5. $\mathcal{E}$ est clos par récurrence, ce qui veut dire que, si p est un entier naturel, $g : \mathbb{N}^p \rightarrow \mathbb{N}$ et $h : \mathbb{N}^{p+2} \rightarrow \mathbb{N}$ sont deux fonctions de $\mathcal{E}$, alors la fonction $f : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$ définie pour tout $(x_1, \dots, x_p) \in \mathbb{N}^p$ et tout $y \in \mathbb{N}$ par :
 - a) $f(x_1, \dots, x_p, 0) = g(x_1, \dots, x_p)$
 - b) $f(x_1, \dots, x_p, y + 1) = h(x_1, \dots, x_p, y, f(x_1, \dots, x_p, y))$
est une fonction de $\mathcal{E}$.

Dans la fermeture par récurrence, la fonction g dénote la **condition initiale** et h l'**étape de récurrence** de la définition par récurrence de la fonction f .

Pour montrer qu'une fonction est primitive récursive, il suffit de montrer comment l'obtenir à l'aide des clauses 4. et 5., et à partir des fonctions décrites en 1., 2. et 3., ou plus généralement, par la définition 19, à partir de fonctions dont on sait déjà qu'elles sont primitives récursives. Ainsi, la définition de l'ensemble des fonctions primitives récursives suit bien une définition inductive.

Exemple 28.

Montrons que l'addition est primitive récursive. On a vu ci-dessus que nous pouvions la définir par récurrence sur son second argument. Il suffit alors de trouver deux fonctions primitives récursives g et h définissant respectivement la condition initiale et l'étape de récurrence de sa définition par récurrence. Selon la définition récursive de l'addition donnée ci-dessus, on doit avoir respectivement :

- $g(x) = x$, et
- $h(x, y, x + y) = (x + y) + 1$

Pour g , seule la fonction constante π_1^1 peut convenir. La fonction h s'obtient en composant la fonction successeur avec la projection de la façon suivante : $h(x, y, x + y) = s(\pi_3^3(x, y, x + y))$. On obtient ainsi la définition suivante :

$$\begin{aligned} x + 0 &= \pi_1^1(x) \\ x + (y + 1) &= s(\pi_3^3(x, y, x + y)) \end{aligned}$$

Ensemble primitif récursif

Tout ensemble A se définit par une fonction caractéristique $\chi_A : A \rightarrow \{0, 1\}$ dont la définition est la suivante :

$$\begin{aligned} \chi_A(a) &= 1 \text{ si } a \in A \\ \chi_A(a) &= 0 \text{ sinon} \end{aligned}$$

De la même façon que pour les fonctions primitives récursives, si A est un sous-ensemble de $\mathbb{N}^k$ pour un entier naturel k donné, la fonction caractéristique χ_A peut aussi se définir selon un schéma récursif. On peut alors étendre la propriété d'être primitif récursif aux fonctions caractéristiques et donc à tout sous-ensemble $A \subseteq \mathbb{N}^k$ pour un entier naturel k donné.

Définition 20 (Ensemble primitif récursif).

Un ensemble $A \subseteq \mathbb{N}^k$ pour un entier naturel k donné est **primitif récursif** si, et seulement si sa fonction caractéristique $\chi_A : \mathbb{N}^k \rightarrow \{0, 1\}$ est primitive récursive.

Exemple 29.

Définissons d'abord la fonction $sg : \mathbb{N} \rightarrow \{0, 1\}$ par :

$$sg(m) = \begin{cases} 0 & \text{si } m = 0 \\ 1 & \text{sinon} \end{cases}$$

Cette fonction est primitive récursive. En effet, on peut la définir par récurrence de la façon suivante :

$$\begin{aligned} sg(0) &= 0 \\ sg(m+1) &= 1 \end{aligned}$$

Ce qui peut se définir par :

$$\begin{aligned} sg(0) &= cste_0^0 \\ sg(m+1) &= cste_1^1(\pi_2^1(m, sg(m))) \end{aligned}$$

où $cste_i^j : \mathbb{N}^j \rightarrow \mathbb{N}$ pour $i, j \in \mathbb{N}$ est la fonction constante qui à tout tuple $(n_1, \dots, n_j)$ fait correspondre l'entier i . À partir de cette fonction, on peut alors montrer que le sous-ensemble $A \subseteq \mathbb{N} \times \mathbb{N}$ défini par :

$$(n, m) \in A \iff n < m$$

est primitif récursif. En effet, on peut définir sa fonction caractéristique par le schéma suivant :

$$\chi_A(n, m) = sg(m - n)$$

Or, nous avons monté ci-dessus qu'à la fois les fonctions sg et la soustraction étaient primitives récursives.

6.3. Les fonctions récursives

Récursion primitive et algorithmique

Comme nous l'avons déjà dit plus haut, d'un point de vue de la programmation séquentielle, les fonctions et les ensembles primitifs récursifs se caractérisent exactement par des programmes dont la seule structure d'itération est la boucle finie. Ceci se voit bien par les exemples de fonctions récursives primitives qui montrent que l'ensemble $\mathcal{E}$ défini dans la définition 19 est stable par certaines opérations faisant intervenir des ensembles bornés. Ainsi, si $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ est une fonction primitive récursive, alors la fonction $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ définie par :

$$f(x_1, \dots, x_k, n) = \sum_{i=0}^n g(x_1, \dots, x_k, i)$$

est primitive récursive. En effet, nous avons :

$$\begin{aligned} f(x_1, \dots, x_k, 0) &= g(x_1, \dots, x_k, 0) \\ f(x_1, \dots, x_k, m+1) &= f(x_1, \dots, x_k, m) + g(x_1, \dots, x_k, m+1) \end{aligned}$$

En utilisant la forme stricte de la définition de la primitive récursion, on écrirait :

$$f(x_1, \dots, x_k, m+1) = h(x_1, \dots, x_k, m, f(x_1, \dots, x_k, m))$$

avec $h(\vec{n}) = g(\pi_{k+2}^1(\vec{n}), \dots, \pi_{k+2}^k(\vec{n}), s(\pi_{k+2}^{k+1}(\vec{n}))) + \pi_{k+2}^{k+2}(\vec{n})$ où $\vec{n} = (n_1, \dots, n_k, n_{k+1}, n_{k+2})$

Il en est de même pour la quantification bornée. La quantification bornée universelle (resp. existentielle) permet de préciser qu'une propriété (définie par sa fonction caractéristique) est vraie pour toutes les (resp. pour certaines) valeurs d'un de ses arguments inférieures à une certaine borne. Précisément,

$$\forall i \leq m, \chi(\vec{x}, i)$$

est vrai si $\chi(\vec{x}, i) = 1$ pour tout $i \leq m$. Si la fonction $\chi : \mathbb{N}^{k+1} \rightarrow \{0, 1\}$ est primitive récursive, alors la fonction caractéristique de $\forall i \leq m, \chi : \mathbb{N}^{k+1} \rightarrow \{0, 1\}$ l'est aussi. En effet, la fonction caractéristique de $\forall i \leq m, \chi(\vec{x}, i)$ est définie par :

$$\prod_{i=1}^m \chi(\vec{x}, i)$$

qui vaut bien 1 quand tous les $\chi(\vec{x}, i) = 1$ pour $i \leq m$. Nous avons déjà montré que la multiplication est primitive récursive, et en suivant la même démarche que pour la somme bornée ci-dessus, il est très facile de montrer que le produit borné est aussi primitif récursif, ce qui nous permet de déduire que $\prod_{i=1}^m \chi(\vec{x}, i)$ l'est aussi.

La dernière opération que nous allons voir pour laquelle l'ensemble des fonctions primitives récurives est aussi stable, est l'opération dite de **minimisation bornée**. La minimisation bornée sera utilisée pour représenter les programmes effectuant une recherche sur une structure bornée mais dont on peut arrêter la recherche dès que l'on a trouvé le premier élément satisfaisant la condition d'arrêt, et donc sans parcourir la structure complète. Un exemple est la recherche d'un élément tel qu'un entier, dans un tableau trié. Ainsi, la minimisation bornée définira les programmes utilisant une boucle while dont la condition d'arrêt porte sur une structure de donnée dont on connaît la taille.

Cette opération est importante car c'est en retirant la borne dans le schéma de mnimisation que nous pourrions définir exactement ce qui peut être calculable par un ordinateur. La minimisation bornée est le moyen de rechercher le plus petit élément d'un ensemble $A \subseteq \mathbb{N}^{k+1}$. Cette fonction notée souvent $\mu_{\leq}^A : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ se définit par : Soit A un ensemble primitif récursif de $\mathbb{N}^{k+1}$,

$$\begin{aligned} \mu_{\leq}^A(x_1, \dots, x_k, n) &= 0 \text{ s'il n'existe pas de } p \leq n \text{ tel que } (x_1, \dots, x_k, p) \in A \\ \mu_{\leq}^A(x_1, \dots, x_k, n) &= p \text{ si } p \leq n \text{ est le plus petit entier tel que } (x_1, \dots, x_k, p) \in A \end{aligned}$$

Pour démontrer que cette fonction est primitive récursive, nous allons utiliser la **définition par cas** qui peut être vue comme une conditionnelle imbriquée classique dans les langages de programmation. La définition d'une fonction $f : \mathbb{N}^k \rightarrow \mathbb{N}$ par cas est la suivante : Soit $g_1, \dots, g_n, g_{n+1}$ des fonctions de $\mathbb{N}^k$ dans $\mathbb{N}$ et $\chi_1, \dots, \chi_n$ des fonctions caractéristiques de sous-ensembles de $\mathbb{N}^k$ (les conditions)

$$f(\vec{x}) = \begin{cases} g_1(\vec{x}) & \text{si } \chi_1(\vec{x}) = 1 \\ \vdots & \\ g_n(\vec{x}) & \text{si } \bigwedge_{i \leq n-1} \chi_i(\vec{x}) = 0 \text{ et } \chi_n(\vec{x}) = 1 \\ g_{n+1}(\vec{x}) & \text{si } \bigwedge_{i \leq n} \chi_i(\vec{x}) = 0 \end{cases}$$

Si les fonctions $g_1, \dots, g_n, g_{n+1}$ et $\chi_1, \dots, \chi_n$ sont primitives récursives alors f l'est aussi. En effet, on a

$$f(\vec{x}) = \begin{cases} g_1(\vec{x}) \times \chi_1(\vec{x}) + \\ \dots + \\ g_i(\vec{x}) \times (\chi_i(\vec{x}) - \sum_{j=1}^{i-1} \chi_j(\vec{x})) + \\ \dots + \\ g_n(\vec{x}) \times (\chi_n(\vec{x}) - \sum_{j=1}^{n-1} \chi_j(\vec{x})) + \\ g_{n+1}(\vec{x}) \times (1 - \sum_{j=1}^n \chi_j(\vec{x})) \end{cases}$$

À partir de là, on peut montrer que le schéma de minimisation bornée est primitive récursive. En effet, on a la définition par récurrence suivante de $\mu_{\leq}^A$:

$$\mu_{\leq}^A(x_1, \dots, x_k, 0) = 0$$

$$\mu_{\leq}^A(x_1, \dots, x_k, n+1) = \begin{cases} \mu_{\leq}^A(x_1, \dots, x_k, n) & \text{si } \sum_{i=1}^n \chi_A(x_1, \dots, x_k, i) \geq 1 \\ n+1 & \text{si } \sum_{i=1}^n \chi_A(x_1, \dots, x_k, i) = 0 \text{ et } \chi_A(x_1, \dots, x_k, n+1) = 1 \\ 0 & \text{pour tous les autres cas} \end{cases}$$

Comme nous l'avons déjà vu précédemment, l'ensemble des fonctions numériques (i.e. définies sur les entiers) n'est pas primitive récursive. La raison est que la cardinalité de l'ensemble des fonctions primitives récursives du fait de leur définition inductive à partir d'un ensemble fini d'opérations de base et de schémas de clôture, est nécessairement dénombrable, mais la cardinalité de l'ensemble des fonctions numériques n'est pas dénombrables (une simple application du théorème de Cantor). Maintenant, la question que l'on peut se poser est "*existe-t-il des fonctions que l'on sait calculer (par un programme informatique) mais qui ne seraient pas primitives récursives ?*". Malheureusement, la réponse à cette question est "oui" comme le prouve le théorème suivant :

Théorème 30.

Il existe des fonctions calculables qui ne sont pas primitives récursives.

Démonstration Pour démontrer ce théorème, nous allons utiliser la méthode de la diagonale introduite par Cantor pour montrer que les réels étaient en nombre plus grand que les entiers, et utilisée après par Turing et Godel pour démontrer l'existence de problèmes non calculables. Comme l'ensemble des fonctions primitives récursives est de cardinalité dénombrable, nous pouvons ordonner/énumérer ses éléments. On peut en effet représenter les fonctions primitives récursives sous formes de chaînes de caractères dans un alphabet bien choisi et utiliser le codage que nous avons défini précédemment et dont nous avons vu qu'il était lui même primitive récursive, en ayant au préalable numéroté les lettres de l'alphabet. Soit donc $f_0, f_1, \dots, f_n, \dots$ une telle énumération. Par définition, le nombre d'arguments de ces fonctions varie. Avec le codage des suites d'entiers que nous avons donné dans la section 26, nous pouvons sans perte de généralité, nous restreindre au cas des fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Considérons alors le tableau suivant :

A	0	1	2	...	j	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...	$f_0(j)$	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	...	$f_1(j)$	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...	$f_2(j)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	...	$f_i(j)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

Chaque case $A(i, j)$ de ce tableau contient un entier naturel qui est $f_i(j)$. Définissons alors la nouvelle fonction $g : \mathbb{N} \rightarrow \mathbb{N}$ de la façon suivante :

$$g(n) = f_n(n) + 1 = A(n, n) + 1$$

Cette fonction ne peut pas être primitive récursive sinon elle serait identique à une des fonctions f_k apparaissant dans le tableau A ci-dessus ce qui n'est pas possible. En effet, si ceci se pouvait, $g(k)$ devrait être égale à $f_k(k)$ mais de par sa définition, $g(k) = f_k(k) + 1$. Ainsi, g ne peut pas apparaître dans le tableau A et donc n'est pas primitive récursive. Maintenant, la fonction g est calculable. En effet, pour calculer $g(n)$ on peut procéder ainsi :

1. On énumère les fonctions primitives récursives jusqu'à f_n . Ceci peut se faire algorithmiquement en énumérant selon le codage utilisé pour représenter les fonctions primitives récursives jusqu'à la fonction recherchée.
2. Une fois la fonction f_n obtenue, on évalue f_n . Ce qui est calculable puisque f_n est primitive récursive.
3. Finalement, on évalue $f_n(n) + 1$. Ici aussi, ceci est calculable car $f_n(n) + 1 = s(f_n(n))$.

□

Ce théorème montre qu'il existe plus de fonctions que l'on sait calculer que de fonctions primitives récursives. Maintenant, on peut se demander s'il existe vraiment de telles fonctions ou bien le résultat ci-dessus n'est que purement théorique. En fait, il en existe que l'on sait construire. La première a été définie par W. Ackermann et fait l'objet de l'exercice suivant.

Minimisation non bornée

Avec le recul que nous avons des langages de programmation, nous savons bien que les boucles For ou plus généralement des boucles dont on connaît la borne d'arrêt, sont insuffisantes pour décrire l'ensemble des algorithmes, et beaucoup de programmes utilisent d'autres boucles non

bornées ou enore bornées par des conditions (utilisation de la boucle While) que nous n'arrivons pas toujours à satisfaire ce qui fait qu'ils ne terminent pas. Que faut-il alors pour prendre en compte l'ensemble des fonctions calculables ? Il nous faut le moyen de pouvoir parcourir l'ensemble des entiers jusqu'à trouver le plus petit élément satisfaisant une propriété P sans pour cela imposer à chaque fois un segment de $\mathbb{N}$ (i.e. une borne n). Le schéma de minimisation tel qu'il a été présenté dans la section précédente mais en retirant la borne n permettrait de caractériser un tel schéma. Maintenant, l'élément recherché peut ne pas exister. Comment prendre en compte un tel cas ? En considérant des fonctions partielles, c'est-à-dire pas nécessairement définies partout.

Définition 21 (Fonctions partielles).

Une **fonction partielle** de $\mathbb{N}^p$ dans $\mathbb{N}$ est un couple (A, f) tel que $A \subseteq \mathbb{N}^p$ et $f : A \rightarrow \mathbb{N}$ est une application. A est appelé le **domaine de définition** de f . On notera $\mathcal{F}_p^*$ l'ensemble des fonctions partielles de $\mathbb{N}^p$ dans $\mathbb{N}$.

Étendons cette notion de fonctions partielles aux schémas de clôture (i.e. la composition et la définition par récurrence).

Définition 22 (Composition partielle).

Soient $f_1, f_2, \dots, f_n \in \mathcal{F}_p^*$ et $g \in \mathcal{F}_n^*$. La **fonction composée** $h = g(f_1, \dots, f_n)$ est l'élément de $\mathcal{F}_p^*$ définie par :

- $h(x_1, \dots, x_p)$ n'est pas définie si l'une des $f_i(x_1, \dots, x_p)$ n'est pas définie ou bien si toutes le sont, $g(f_1(x_1, \dots, x_p), \dots, f_n(x_1, \dots, x_p))$ n'est pas définie.
- Dans le cas contraire, $h(x_1, \dots, x_p)$ est définie et est égale à $g(f_1(x_1, \dots, x_p), \dots, f_n(x_1, \dots, x_p))$.

Définition 23 (Récurrence partielle).

Soient $g : \mathcal{F}_p^*$ et $h \in \mathcal{F}_{p+2}^*$. Alors, la fonction $f \in \mathcal{F}_{p+1}^*$ est définie pour tout $(x_1, \dots, x_p) \in \mathbb{N}^p$ et tout $y \in \mathbb{N}$ par :

1. $f(x_1, \dots, x_p, 0) = g(x_1, \dots, x_p)$ si $g(x_1, \dots, x_p)$ est définie, sinon $f(x_1, \dots, x_p, 0)$ est indéfinie.
2. $f(x_1, \dots, x_p, y + 1) = h(x_1, \dots, x_p, y, f(x_1, \dots, x_p, y))$ si $h(x_1, \dots, x_p, y, f(x_1, \dots, x_p, y))$ est définie, sinon $f(x_1, \dots, x_p, y + 1)$ est indéfinie.

La fonction f est **définie par récurrence** à partir de g et h .

Définition 24 (Minimisation non bornée).

Soit $A \subseteq \mathbb{N}^{p+1}$. La **fonction de minimisation non bornée** $\mu^A \in \mathcal{F}_p^*$ est définie pour tout $(x_1, \dots, x_p)$ par :

- $\mu^A(x_1, \dots, x_p) = k$ si k est le plus petit entier tel que $(x_1, \dots, x_p, k) \in A$
- $\mu^A(x_1, \dots, x_p)$ est indéfinie sinon.

On peut maintenant définir les fonctions récursives.

Définition 25 (Fonctions récursives).

L'ensemble des **fonctions récursives** est le plus petit sous-ensemble de $\mathcal{F}^* = \bigcup_{p \in \mathbb{N}} \mathcal{F}_p^*$

qui :

- contient toutes les fonctions (totales) constantes, les projections et la fonction successeur,
- est clos pour la composition, les définitions par récurrence partielles et la minimisation non bornée.

Un ensemble $A \subseteq \mathbb{N}^p$ est **rékursif** si, et seulement si sa fonction caractéristique χ_A est récursive.

Exemple 31.

Reprenons l'exemple de la fonction d'Ackermann. Ce qui fait que cette fonction n'est pas primitive récursive est qu'elle se définit par une double récurrence sur chacun de ses deux arguments et dont on ne peut ramener la condition d'arrêt sur une simple équation sur les entiers en entrée de la fonction. En fait, on peut montrer que cette fonction termine mais avec assez de difficulté ce qui n'est jamais le cas avec les fonctions primitives récursives. Dans un précédent exercice, nous avons montré que la fonction d'Ackermann n'était pas primitive récursive. Cependant, est-elle récursive ? On peut répondre par l'affirmative à cette question. En effet, si nous considérons le graphe de chacune des fonctions A_n pour tout $n \in \mathbb{N}$ défini par :

$$A_0 = \{(x, y) | y = 2^x\}$$

$$A_{n+1} = \{(0, 1)\} \cup \{(x, y) | \exists y' \in \mathbb{N}, (x-1, y') \in A_{n+1} \text{ et } (y', y) \in A_n\}$$

nous voyons bien dans la définition de l'ensemble A_{n+1} l'utilisation de la minisation non bornée. En effet, nous avons :

$$\chi_{A_0}(x, y) = \begin{cases} 1 & \text{si } y = 2^x \\ 0 & \text{sinon} \end{cases}$$

$$\chi_{A_{n+1}}(0, y) = \begin{cases} 1 & \text{si } y = 1 \\ 0 & \text{sinon} \end{cases}$$

$$\chi_{A_{n+1}}(x+1, y) = \chi_{A_n}(\mu^{A_{n+1}}(x), y)$$

la fonction 2^x ainsi que l'égalité sont primitives récursives. La définition ci-dessus de la fonction d'Ackermann est donc bien récursive.

6.4. Un interpréteur de programmes

Une question intéressante est de se demander s'il existe une fonction récursive qui peut simuler/interpréter n'importe quelle fonction récursive. Précisément, on voudrait une fonction récursive Int à laquelle on fournirait la description d'une fonction quelconque f de même qu'un tuple de valeurs initiales pour f et qui simulerait l'exécution de f sur ce tuple de valeurs initiales. De telles fonctions existent et sont appelées des **fonctions récursives universelles** ou **interpréteur**. Une fonction récursive f par définition, est la composition récursive des opérations de composition, schéma de récurrence et minimisation non bornée à partir des fonctions de bases (constantes, projections, et successeur). Ceci définit un arbre caractérisant le **programme** définissant le comportement de la fonction f . Formellement, les programmes sont définis de la façon suivante :

Définition 26 (Programmes).

L'ensemble des **programme** d'arité $p \in \mathbb{N}$ est le plus petit ensemble inductivement défini par :

- $(0, p)$ est un programme d'arité p . Ce programme représente la fonction qui à tout tuple $(x_1, \dots, x_p)$ de $\mathbb{N}^p$ associe 0. C'est la seule fonction constante que nous considérons, les autres fonctions constantes pouvant toujours être obtenues avec la fonction successeur par composition.
- s est un programme d'arité 1.
- pour tout $i \leq p \in \mathbb{N}$, (π, i, p) est un programme d'arité p .
- si $f_1, \dots, f_n$ sont des programmes d'arité p et h est un programme d'arité n , alors $h(f_1, \dots, f_n)$ est un programme d'arité p .
- si g est un programme d'arité p et h est un programme d'arité $p+2$, alors $rec(g, h)$ est un programme d'arité $p+1$.
- si χ est un programme d'arité $p+1$ à valeurs d'arrivée dans $\{0, 1\}$, alors $\mu(\chi)$ est un programme d'arité $p+1$.

L'ensemble des **programmes** contient tous les programmes d'arité p pour tout $p \in \mathbb{N}$. Une **exécution** est tout terme de la forme $App(f, \vec{x})$ où f est un programme d'arité p et $\vec{x} \in \mathbb{N}^p$.

Pour pouvoir définir une fonction qui interpréterait tous ces programmes, nous devons tout d'abord associer un entier à chacun d'eux. Les programmes pouvant être représentés par des arbres (de part leur définition récursive), nous pouvons aussi les représenter sous la forme d'une chaîne de caractères qui représente la notation fonctionnelle du programme sous sa forme préfixe et bien parenthésée. Ainsi, chaque caractère est soit le nom d'une fonction de base (constante, projections, successeur), une opération de clôture (composition, récurrence, minimisation), une parenthèse ouvrante ou fermante, ou la virgule. En associant un entier unique à ces différents caractères, on peut alors, par le codage que nous avons défini dans la section 26, associer un entier unique à chacune des chaînes représentant nos programmes et nos exécutions. Pour les exécutions, nous aurons pris soin de représenter tout entier x par le terme $s(\dots(s((0, 0))\dots))$ avec x occurrences du symbole s . Dans la suite, nous représenterons le code associé à un programme f (resp. une exécution $App(f, \vec{x})$) par $\overline{f}$ (resp. $\overline{App(f, \vec{x})}$). On peut maintenant définir notre interpréteur. Ce dernier a pour fonction d'interpréter des exécutions de la forme $App(f, \vec{x})$. L'ensemble des programmes suit une définition inductive non ambiguë. Nous pouvons alors au codage près, définir notre interpréteur comme une fonction récursive sur la structures des programmes.

Définition 27 (Interpréteur).

On définit la fonction $Int : \mathbb{N} \rightarrow \mathbb{N}$ par cas de la façon suivante : supposons que $x = \overline{App(f, x_1, \dots, x_p)}$

- Si f est $(0, p)$, alors $Int(x) = 0$.
- Si f est s , alors $Int(x) = x_1 + 1$.
- Si f est (π, i, p) , alors $Int(x) = x_i$.
- Si f est $h(f_1, \dots, f_n)$, alors si pour chaque i , $1 \leq i \leq n$, $Int(\overline{App(f_1, x_1, \dots, x_p)})$ est défini et $Int(\overline{App(h, (Int(\overline{App(f_1, x_1, \dots, x_p)}), \dots, Int(\overline{App(f_n, x_1, \dots, x_p)})))})$ est aussi défini

$$Int(x) = \overline{Int(App(h, (Int(\overline{App(f_1, x_1, \dots, x_p)}), \dots, Int(\overline{App(f_n, x_1, \dots, x_p)})))})}$$

$Int(x)$ est indéfini sinon.

- Si f est $rec(g, h)$, alors

$$Int(x) = \begin{cases} Int(\overline{App(g, x_1, \dots, x_{p-1})}) & \text{si } x_p = 0 \text{ et } Int(\overline{App(g, x_1, \dots, x_{p-1})}) \\ & \text{est défini} \\ \overline{Int(App(h, (x_1, \dots, x_p, Int(\overline{App(f, x_1, \dots, x_{p-1})}))})} & \text{si } x_p \neq 0, \\ & \text{et } Int(\overline{App(h, x_1, \dots, x_p - 1, Int(\overline{App(f, x_1, \dots, x_{p-1})}))}) \\ & \text{est défini} \\ \text{Indéfini} & \text{sinon} \end{cases}$$

- si f est $\mu(\chi)$, alors

- $Int(\overline{App((\mu(\chi)), x_1, \dots, x_p)}) = k$ si k est le plus petit entier tel que $Int(\overline{App(\chi, x_1, \dots, x_p, k)}) = 1$
- $Int(\overline{App((\mu(\chi)), x_1, \dots, x_p)})$ est indéfinie sinon.

Théorème 32.

Int est une fonction récursive.

Démonstration Int est défini par cas à partir de la fonction récursive α^{-1} et par composition à partir d'elle-même. Enfin, elle est bien partielle car si $Int(x)$ n'est pas un entier, alors Int n'est pas définie : l'interpréteur ne termine pas pour un programme qui ne termine pas. $\square$

Indécidabilité du problème de l'arrêt

Dans les sections précédentes, la notion de procédure effective a été définie en termes de problèmes portant sur les entiers pour lesquels nous pouvions définir des fonctions récursives (une par problème) calculant une solution à ces problèmes. Avec ces éléments, il est possible de démontrer que certains problèmes ne sont pas solubles par une procédure effective. Parmi ces problèmes non solubles effectivement, un particulièrement intéressant pour la programmation en particulier et l'informatique en général, est celui du problème de l'arrêt, c'est-à-dire de savoir s'il existe une fonction récursive qui est capable de décider pour toute fonction récursive et tout tuple d'entiers si cette fonction est définie ou non sur ce tuple. Pour démontrer qu'il n'existe pas

une telle fonction, nous allons encore utiliser un raisonnement par diagonalisation à la Cantor à partir de notre interpréteur défini dans la section précédente.

Définition 28 (Programme terminant).

Un programme f d'arité p est dit **terminant** si, et seulement si pour tout tuple $\vec{x} \in \mathbb{N}^p$, $Int(\overline{App(f, \vec{x})}) \in \mathbb{N}$.

Proposition 33.

Soit A un sous-ensemble récursif de l'ensemble des programmes qui terminent toujours. Alors, il existe une fonction récursive totale qui n'est représentée par aucun programme de A .

Démonstration Soit $G : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la fonction récursive suivante : pour tout programme f d'arité p de A et tout entier $n \in \mathbb{N}$,

$$G(\overline{f}, n) = Int(\overline{App(f, \vec{x})})$$

où $\vec{x} = n$.

La fonction G est trivialement récursive et totale puisque définie directement à partir d'une fonction dans A . De plus, par définition, c'est un interpréteur des programmes de A . Définissons alors la fonction G' par : $G'(n) = G(n, n) + 1$. Par définition, $G' : \mathbb{N} \rightarrow \mathbb{N}$ est une fonction récursive totale car G est une fonction récursive totale. Supposons qu'il y ait dans A un programme codé par l'entier x qui représente la fonction G' . On a alors $G(x, k) = G'(k) = G(k, k) + 1$. En particulier, pour $k = x$, on a $G(x, x) = G'(x) = G(x, x) + 1$ ce qui n'est pas possible. $\square$

Ce que montre la preuve ci-dessus, est qu'un langage de programmation qui ne permettrait d'exprimer que des programmes qui terminent est toujours incomplet, car il ne permet pas d'exprimer son propre interpréteur. C'est l'exemple des langages impératifs formés de la déclaration de variables, de l'affectation, du test et de la boucle for.

De la proposition ci-dessus, on a comme corollaire l'indécidabilité du problème de l'arrêt.

Théorème 34 (Indécidabilité du problème de l'arrêt).

L'ensemble des programmes terminant n'est pas récursif.

Démonstration En appliquant la proposition précédente, si l'ensemble des programmes terminant était récursif, il existerait par la proposition ci-dessus, une fonction récursive totale (et donc qui termine) mais qui ne serait représentée par aucun des programmes de cet ensemble, ce qui est une contradiction. $\square$

6.5. Thèse de Church

Il est clair que toute fonction récursive est calculable (i.e. implémentable par un algorithme). Mais qu'en est-il de l'inverse ? La réponse affirmative à cette question est la **thèse de Church**. Cette affirmation ne se prête pas à une démonstration parce que nous n'avons pas de définition précise à la notion d'algorithme. Cependant, à ce jour, on ne connaît aucun contre-exemple à la thèse de Church. À chaque fois que l'on connaît une fonction dont on avait l'intuition qu'elle est calculable, on a réussi à le prouver. De plus, il existe d'autres formalisations de la calculabilité (λ -calcul, machine à registre, etc.) et toutes ont été montrées équivalentes. Ceci milite fortement en faveur de la thèse de Church. Néanmoins, l'échec de la première tentative de formalisation de la calculabilité, celles des fonctions primitives récursives, doit nous rendre prudents.

7. Les machines de Turing

7.1. Définitions

Les fonctions récursives ne constituent pas un modèle très convaincant de la calculabilité. Elles manquent d'un aspect calculatoire naturel, c'est-à-dire effectuer un calcul pas à pas jusqu'à obtenir le résultat recherché.¹ Pour répondre à ce manque, A. Turing proposa en 1936 un nouveau modèle de calcul appelé **les machines de Turing**. Les machines de Turing sont un exemple d'automates (que l'on étudiera plus précisément dans la troisième partie de ce document) qui précèdent d'une dizaine d'années les machines de von Neumann. La définition d'une machine de Turing est la suivante :

Définition 29 (Machines de Turing).

Une **machine de Turing** est la donnée de trois ensembles finis (Σ, Q, R) où : on se donne un symbole $\square$

- Σ est l'alphabet de la machine. On suppose que $\square \notin \Sigma$.
- Q est l'ensemble fini des états de la machine.
- R est l'ensemble fini des transitions de la machine. Chaque transition se définit par un 5-uplets (q, s, q', s', d) où $q, q' \in Q$, $s, s' \in \Sigma \cup \{\square\}$, et $d \in \{0, +, -\}$. On note la transition $q \xrightarrow{s/s', d} q'$.

Un programme manipule des chaînes de symboles pour les transformer selon des règles de transformation et ce afin de rendre un résultat lui-même représenté par une chaîne de symboles. C'est justement ce que permet de traduire l'ensemble des éléments d'une machine de Turing. Ainsi, une machine de Turing peut se voir comme un automate et donc fonctionne par transitions entre configurations, à partir d'une configuration initiale, vers éventuellement une configuration terminale. Les configurations d'une machine contiennent plus d'information qu'un mot de $(\Sigma \cup \{\square\})^*$. Elles contiennent en plus :

- une mémoire de taille infinie sous forme d'un ruban divisé en cases. Chaque case contiendra un symbole de $\Sigma \cup \{\square\}$.
- la position de la tête de lecture.
- le mot écrit sur le ruban, élément de $(\Sigma \cup \{\square\})^*$ en omettant les $\square$ préfixes et suffixes.

L'ensemble des règles R définit alors le programme, c'est-à-dire le traitement permettant à la fois de transformer les configurations de la machine en remplaçant le symbole s par s' et laissant le pointeur sur place ($d = 0$), ou alors en le déplaçant à droite ($d = +$) ou à gauche ($d = -$) à partir de la position courante. Ainsi, une règle s'applique que si dans l'état q , la configuration contient à la position courante de la tête de lecture, le symbole s .

1. Dans le chapitre précédent, on a obtenu ce calcul pas à pas mais de façon détournée au travers de la définition de notre interpréteur.

Les machines de Turing dérivent alors par réécriture de symbole selon les règles dans R , des configurations. Une configuration sera une **configuration terminale** quand aucune des règles de transformation de R n'est applicable pour cette configuration. Pour représenter plus formellement cette notion de dérivation, nous devons tout d'abord définir formellement la notion de configuration.

Définition 30 (Configuration).

Soit $\mathcal{M} = (\Sigma, Q, R)$ une machine de Turing. Une **configuration** pour $\mathcal{M}$ est un mot de la forme $\alpha q \beta$ où $q \in Q$, et $\alpha, \beta \in (\Sigma \cup \{\square\})^*$, avec α (resp. β) ne commençant (resp. ne finissant) pas par $\square$. On appellera **configuration initiale** tout mot de la forme $q\alpha$. Dans la suite, on supposera toujours une configuration initiale pour chaque machine de Turing considérée.

La configuration $\alpha q \beta$ exprime que le contenu de la bande est le mot $\alpha\beta$, que l'état courant est q et que la tête de lecture pointe sur le début de β .

Définition 31 (Étape de dérivation).

Une **étape de dérivation** pour $\mathcal{M}$ est le triplet (C, ρ, C') tel que

- $C = \alpha s'' q s \beta$ et $C' = \alpha' q' \beta'$ sont des configurations pour $\mathcal{M}$,
- $\rho = (q, s, q', s', d) \in R$, et
- les mots α' et β' sont définis comme suit :
 - $\alpha' = \alpha s''$ et $\beta' = s' \beta$ si $d = 0$
 - $\alpha' = \alpha$ et $\beta' = s'' s' \beta$ si $d = -$
 - $\alpha' = \alpha s'' s'$ et $\beta' = \beta$ si $d = +$

On la note $C \xrightarrow{\rho} C'$.

Définition 32 (Dérivation et calcul).

Une **dérivation** pour $\mathcal{M}$ est une suite d'étapes de dérivation soit infinie, soit finie et dans ce cas-là se terminant par une configuration d'arrêt. On la note $C_0 \xrightarrow{\rho_1} C_1 \xrightarrow{\rho_2} \dots \xrightarrow{\rho_p} C_p$. La suite $\rho_1 \rho_2 \dots \rho_p$ est la **trace** du calcul.

Exemple 35.

On se propose de donner la machine de Turing qui permet de calculer le successeur d'un entier. Tout d'abord, nous devons représenter les entiers. Pour cela, nous allons utiliser la représentation unaire des entiers^a, i.e. celle qui représente les entiers par des "bâtons", autant que l'entier désigné. Nous supposons alors l'alphabet singleton $\Sigma = \{|\}$. Ainsi, pour représenter le nombre x , on aura sur les cases $1, 2, \dots, x$ des bâtons, et enfin le symbole $\square$ sur les autres cases. La configuration initiale est donc notée $q \underbrace{|| \dots ||}_{n \text{ fois}} \cdot$.

La machine de Turing permettant de calculer le successeur de tout nombre est défini par l'automate :



a. qui est la représentation la plus abstraite car contenant leur définition inductive à partir de la fonction successeur.

7.2. Les fonctions T-calculables

Là encore, les seules fonctions qu'une machine de Turing peut calculer sont les fonctions numériques, c'est-à-dire celles dont les domaines sont des tuples d'entiers. Bien sûr, pour qu'une machine de Turing puisse calculer la valeur d'une fonction $f : \mathbb{N}^k \rightarrow \mathbb{N}$ il faut évidemment coder les valeurs des tuples $(x_1, \dots, x_k)$ afin qu'ils puissent être lus par la machine de Turing. Pour cela, nous allons étendre la représentation des entiers sous forme unaire que nous avons donnée dans l'exemple précédent. Il faut maintenant séparer ces différents entiers mis sous forme unaire afin de représenter les tuples. On se propose alors de les séparer par une virgule. Le tuple $(x_1, \dots, x_k)$ sera donc défini par la configuration

$$q \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}$$

Définition 33 (Fonction T-calculable).

Une fonction partielle $f : \mathbb{N}^k \rightarrow \mathbb{N}$ est dite **T-calculable** si, et seulement s'il existe une machine de Turing construite sur l'alphabet $\Sigma = \{ |, ", " \}$ qui à partir de toute configuration initiale $q_0 \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}$ a pour comportement :

- de ne jamais s'arrêter si $f(x_1, \dots, x_k)$ n'est pas défini,
- ou bien de rendre la configuration $q_f \underbrace{|\dots|}_{y \text{ fois}}$ en un temps fini telle que

$$f(x_1, \dots, x_k) = y.$$

La question naturelle que l'on se pose est de savoir si les fonctions récursives que nous avons étudiées dans le chapitre précédent définissent la même notion de calculabilité que les machines de Turing. La réponse à cette question a été apportée par A. Turing lui-même au travers des deux théorèmes suivants :

Théorème 36.

Toute fonction récursive est calculable par une machine de Turing ou T-calculable.

Démonstration : Pour démontrer ce théorème, nous devons alors tout d'abord montrer que les fonctions de base (i.e. les fonctions constantes, les projections et la fonction successeur) sont T-calculables. Puis montrer que les fonctions T-calculables sont stables par composition, récurrence et minimisation non bornées. Dans l'exemple 35 et l'exercice ?? ci-dessus, nous avons déjà montré que les fonctions de base sont T-calculables. Il nous reste alors à montrer que l'ensemble des fonctions T-calculables sont stables par la composition, le schéma de récurrence et la minimisation non bornée. Les machines de Turing associées à chacune des opérations de clôture sont lourdes de notations dans leur définition. Nous ne donnerons ici que les grandes étapes de leur description.

— Commençons par montrer la clôture des fonctions T-calculables pour la composition. Soient donc $f_1, f_2, \dots, f_n \in \mathcal{F}_k^*$ et $g \in \mathcal{F}_n^*$ telles que toutes ces fonctions soient T-calculables. On sait alors que pour tout i , $1 \leq i \leq n$, il existe une machine de Turing $\mathcal{M}_i$ qui calcule f_i . De même, on sait qu'il existe une machine de Turing $\mathcal{N}$ qui calcule g . On peut alors construire la machine de Turing $\mathcal{M}$ de la façon suivante : la configuration initiale de la machine $\mathcal{M}$ est

$$q_0 \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}$$

1. on commence par réécrire la configuration $q_0 \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}$ en la configuration

$$\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}} \square \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}} q_1$$

2. On revient en arrière jusqu'au 1er symbole $\square$ rencontré. On a alors la configuration

$$\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}} \square q'_1 \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}$$

3. On applique l'étape ci-dessus $n-1$ fois pour obtenir la configuration

$$\underbrace{\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}} \square \dots \square \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}}_{n \text{ fois}} q_{n-1}$$

4. on revient en arrière jusqu'au 1er symbole " $\square$ " que l'on remplace par $\#$, et on applique les règles de la machine $\mathcal{M}_n$. À cette étape, on a alors la configuration

$$\underbrace{\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}} \square \dots \square \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}}_{n-1 \text{ fois}} \# \underbrace{|\dots|}_{f_n(\vec{x}) \text{ fois}} q_{f_{\mathcal{M}_n}}$$

5. on revient en arrière jusqu'au 1er symbole $\square$ rencontré que l'on remplace par $\#$, pour obtenir la configuration

$$\underbrace{\underbrace{\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}}_{n-2 \text{ fois}} \square \dots \square \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}}_{n-2 \text{ fois}} \# q_{0_{\mathcal{M}_{n-1}}} \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}} \# \underbrace{|\dots|}_{f_n(\vec{x}) \text{ fois}}$$

6. on applique les règles de la machine $\mathcal{M}_{n-1}$. À cette étape, on a alors la configuration

$$\underbrace{\underbrace{\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}}_{n-3 \text{ fois}} \square \dots \square \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_k \text{ fois}}}_{n-3 \text{ fois}} \# \underbrace{|\dots|}_{f_{n-1}(\vec{x}) \text{ fois}} q_{f_{\mathcal{M}_{n-1}}} \# \underbrace{|\dots|}_{f_n(\vec{x}) \text{ fois}}$$

7. On applique ce processus n fois jusqu'à obtenir la configuration $\underbrace{|\dots|}_{f_1(\vec{x}) \text{ fois}} q_{f_{\mathcal{M}_1}} \# \dots \# \underbrace{|\dots|}_{f_n(\vec{x}) \text{ fois}}$

8. On remplace tous les symboles $\#$ par virgule et on revient en arrière jusqu'au 1er symbole $\square$ rencontré. On obtient alors la configuration $q_{0_{\mathcal{N}}} \underbrace{|\dots|}_{f_1(\vec{x}) \text{ fois}}, \dots, \underbrace{|\dots|}_{f_n(\vec{x}) \text{ fois}}$

9. On applique alors les règles de la machine $\mathcal{N}$ pour aboutir à la configuration $\underbrace{|\dots|}_{g(f_1(\vec{x}), \dots, f_n(\vec{x})) \text{ fois}} q_{f_{\mathcal{N}}}$

où $q_{f_{\mathcal{N}}}$ devient l'état final de la machine $\mathcal{M}$.

— Montrons maintenant la clôture des fonctions T-calculables pour la récurrence. Soient donc $g \in \mathcal{F}_p^*$ et $h \in \mathcal{F}_{p+2}^*$ telles que ces deux fonctions soient T-calculables. On sait alors qu'il existe une machine de Turing $\mathcal{M}_g$ qui calcule g . De même, on sait qu'il existe une machine de Turing $\mathcal{M}_h$ qui calcule h . On peut alors construire la machine de Turing $\mathcal{M}$ de la façon suivante : la configuration initiale de la machine $\mathcal{M}$ est $q_0 \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, \underbrace{|\dots|}_{y \text{ fois}}$

1. On commence par réécrire la configuration initiale en la configuration

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square q_1$$

2. On revient en arrière pour aboutir à la configuration :

$$q_1 \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square$$

3. On réécrit la configuration précédente en la configuration :

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-2} q_2$$

4. On revient en arrière jusqu'à rencontrer le premier $\square$. On aboutit à la configuration :

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square q_1 \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-2}$$

5. On applique les étapes 3 et 4 jusqu'à obtenir la configuration :

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \dots \square \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \square \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p} q_2$$

6. On revient en arrière jusqu'au premier symbole $\square$ pour obtenir la configuration

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \dots \square \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \square q_{0_{\mathcal{M}_g}} \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}$$

7. On applique les règles de $\mathcal{M}_g$. On obtient

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \dots \square \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \square \underbrace{|\dots|}_{g(\vec{x})} | q_{f_{\mathcal{M}_g}}$$

8. On revient en arrière jusqu'au premier $\square$ que l'on remplace par $,$. On aboutit à la configuration :

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \dots \square \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, q_3, \underbrace{|\dots|}_{g(\vec{x})}$$

9. On revient en arrière jusqu'au premier $\square$ rencontré. On a la configuration :

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \dots \square q_{0_{\mathcal{M}_h}} \underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{g(\vec{x})}$$

10. On applique les règles de $\mathcal{M}_h$. On obtient

$$\underbrace{|\dots|}_{x_1}, \dots, \underbrace{|\dots|}_{x_p}, \underbrace{|\dots|}_{y-1} \square \dots \square \underbrace{|\dots|}_{h(\vec{x}, 0, g(\vec{x}))} | q_{f_{\mathcal{M}_h}}$$

11. On réapplique l'étape 8, 9 et 10 jusqu'à obtenir la configuration :

$$\underbrace{|\dots|}_{f(\vec{x}, y)} | q_{f_{\mathcal{M}_h}}$$

— On finit en montrant la clôture des fonctions T-calculables pour la minimisation non bornée. Soit $A \subseteq \mathbb{N}^{p+1}$ un ensemble récursif de fonction caractéristique $\chi_A : \mathbb{N}^{p+1} \rightarrow \{0, 1\}$. Soit $\mathcal{M}$ la machine de Turing calculant χ_A . La machine de Turing $\mathcal{N}$ effectuant une minimisation non bornée sur A est défini de la façon suivante : la configuration initiale est $q_0 \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}$

1. On commence par réécrire à la fin de la séquence, l'entier 0 représenté par une virgule suivie de rien et le symbole $\#$. On obtient la configuration $\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, \# q_1$

2. On revient au début de la séquence que l'on réécrit après le symbole $\#$. On obtient la configuration $\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, \# \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, q_2$

3. On se place après le symbole $\#$, pour appliquer les transitions de $\mathcal{M}$. On a alors la configuration $\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, \# q_{0_{\mathcal{M}}} \underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}},$

4. On applique les transitions de la MT $\mathcal{M}$. On obtient alors la configuration $\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, \# \overset{0}{1} q_{f_{\mathcal{M}}}$

5. On revient en arrière jusqu'au symbole $\#$ pour tester si la cellule lue est 0 ou 1. Si elle contient 0, on retourne au symbole $\#$, que l'on remplace par $|$ et que l'on fait suivre par $\#$. On obtient alors configuration $\underbrace{|\dots|}_{x_1 \text{ fois}}, \dots, \underbrace{|\dots|}_{x_p \text{ fois}}, \# q_1$. On applique les étapes 2., 3., et 4.,

et on répète ce processus tant que nous n'avons pas 1 après le $\#$.

Si la cellule lue contient 1, on s'arrête et la solution est l'entier compris entre la dernière virgule et le symbole $\#$. ■

Théorème 37.

Toute fonction T-calculable est récursive.

Démonstration : Soit $\mathcal{M}$ une machine de Turing calculant une fonction $f : \mathbb{N}^p \rightarrow \mathbb{N}$. Montrons alors qu'il existe une fonction récursive $h : \mathbb{N}^p \rightarrow \mathbb{N}$ telle que $h(x_1, \dots, x_p) = f(x_1, \dots, x_p)$. Pour définir la fonction h , nous allons tout d'abord définir un certain nombre de fonctions primitives récursives qui opèrent sur les configurations de $\mathcal{M}$. Avant cela, nous devons d'abord associer un entier unique à chaque configuration de la machine $\mathcal{M}$. Nous allons alors définir la fonction qui à partir d'une configuration c donne l'entier unique qui la représente. Pour cela, nous allons utiliser le codage gc suivant : on suppose que $Q = \{q_1, q_2, \dots, q_k\}$ est l'ensemble fini des états de la machine $\mathcal{M}$ où q_1 (resp. q_k) désigne l'état initial (resp. final) de $\mathcal{M}$

- $gv(q_i) = i$ pour $i = 1, \dots, k$
- $gv(\square) = k + 1$
- $gv(,) = k + 2$
- $gv(|) = k + 3$

On pose alors

$$gc(\omega) = \sum_{i=0}^l (k+4)^{l-i} gc(\omega_i)$$

où $|\omega| = l$ et ω_i désigne l'élément se trouvant à la position i dans ω commençant à la position i . La fonction gc donne alors l'entier dont la représentation en base $k+4$ est la configuration ω . Ce codage est effective (i.e. injectif) dans le sens où il associe un unique entier à chaque configuration. Par contre, il existe certains entiers naturels qui ne correspondent à aucune configuration (tous ceux contenant des 0 en base $k+4$), mais ce n'est pas grave car nous cherchions une représentation des configurations par des entiers, et non l'inverse.

Les fonctions que nous allons utiliser pour montrer que toutes les fonctions T-calculables sont récursives sont énumérées ci-dessous.

1. $init(\vec{x})$ donne l'entier correspondant par la fonction gc à la configuration initiale de la machine $\mathcal{M}$ pour le tuple en entrée $\vec{x}$. Cette fonction se définit par : si $\vec{x} = (x_1, \dots, x_p)$ alors

$$init(\vec{x}) = \left\{ \begin{array}{l} \sum_{i=1}^p x_i + (p-1) \\ (k+4)^{\sum_{i=1}^p x_i} + \left(\sum_{j=0}^p (k+4)^j \right) + (p-1) \\ + \left(\sum_{i=1}^p (k+4)^{x_i} \right) + (p-1) \\ + \left(\sum_{i=1}^p (k+4)^{x_i} \right) + (p-1) \end{array} \right. \quad (k+3) \quad (k+2)$$

Le premier argument de $init$ calcule la valeur en base $k+4$ de l'état initial q_1 , le second argument calcule les différentes valeurs des "bâtons" associés à chacun des entiers x_i et le dernier calcule la valeur des "," apparaissant dans la configuration initiale associée au tuple $\vec{x}$. Bien que la fonction $init$ peut sembler compliquée dans sa définition, elle n'est définie qu'à partir de fonctions dont on a déjà vu qu'elles étaient primitives récursives (la somme et l'exponentiation principalement). Cette fonction peut donc être considérée comme primitive récursive elle-même.

2. $config_suivant(x)$ a pour valeur la configuration de $\mathcal{M}$, si elle existe, qui suit immédiatement la configuration x définie par un entier qui représente cette configuration par la fonction de codage gc . Si aucune configuration suit x alors $config_suivant(x) = 0$. Ici, la fonction $config_suivant(x)$ se définit par cas à partir de la fonction caractéristique $\chi(x_1, x_2)$ où x_1 (resp. x_2) est l'entier dénotant la configuration C_1 (resp. C_2) qui est vraie si il existe une transition ρ de $\mathcal{M}$ tel que $gc^{-1}(x_1) \xrightarrow{\rho} gc^{-1}(x_2)$ et 0 sinon. Bien entendu, ceci demande au préalable d'avoir représenté les transitions comme des relations incluses dans $\mathbb{N}^5$ selon le codage des états et des symboles donné par la fonction gc . On peut aisément montrer que la fonction χ est primitive réursive. À l'inverse, on ne donnera pas explicitement la définition $config_suivant(x)$ à cause de la lourdeur de sa définition comme peut nous le laisser supposer la définition de la fonction $init$ ci-dessus. L'élève intéressé est invité à se convaincre qu'elle est bien primitive réursive.
3. $config(x, n)$ est la fonction qui donne la configuration après l'exécution de n étapes à partir de la configuration x . Elle est définie par le schéma de récurrence suivant :

$$\begin{aligned} config(x, 0) &= x \\ config(x, n+1) &= config_suivant(config(x, n)) \end{aligned}$$

4. La fonction caractéristique $stop$ qui est vraie pour la configuration x finale et faux pour toutes les autres. Elle se définit par : $stop(x) = 1 - sg(config_suivant(x))$.
5. $sortie(x)$ donne pour une configuration finale si elle existe, la valeur associée.

La fonction h recherchée est alors définie par :

$$h(\vec{x}) = sortie(config(init(\vec{x}), nb_de_pas(init(\vec{x})))$$

où $nb_de_pas(x) = \mu^A(x)$ avec $A = \{(x, n) | stop(config(x, n))\}$. ■

La preuve de ces deux théorèmes montrent la robustesse de la formalisation de la calculabilité même si ceci ne peut pas se démontrer rigoureusement (voir la section 6.5). Maintenant, si les machines de Turing donnent une description plus convaincante du calcul que les fonctions récursives, leur influence sur la conception des langages de programmation évolués a été très réduite. Du point de vue théorique, de nombreuses constructions et démonstrations se font via les machines de Turing. En outre, elles permettent la définition précise de nombre de "pas" et de "cases" nécessaires à un calcul, donc de complexités temporelle et spatiale d'un algorithme. Les machines de Turing ont donc été et sont à la base de la plupart des travaux sur la théorie de la complexité et la définition des classes de complexité telles que P et NP et la définition des problèmes dits NP -complets. Enfin, le modèle des machines de Turing est souvent étendu (plusieurs bandes, têtes de lecture, etc.) pour faciliter le traitement de certains problèmes ; sans que cela augmente leur capacité de calcul.

7.3. Machines de Turing universelles

Là encore, une question intéressante est de se demander s'il existe une machine de Turing qui peut simuler n'importe quelle machine de Turing. La réponse à cette question est un simple corollaire des théorèmes 32 et 36. En effet, nous avons vu que toutes les fonctions récursives étaient T-calculables et que justement l'interpréteur de tout programme était récursif et donc T-calculable. La traduction par une machine de Turing de cet interpréteur comme exprimée dans la preuve du théorème 36 est la définition d'une machine de Turing universelle. Bien entendu, par les deux théorèmes qui montrent l'équivalence entre l'ensemble des fonctions T-calculables et des fonctions récursives, le problème de l'arrêt demeure aussi un problème indécidable pour les machines de Turing.

7.4. Exemples de problèmes indécidables

À la suite du premier résultat d'indécidabilité sur la terminaison des programmes (qu'ils soient représentés par des fonctions récursives ou des machines de Turing ou tout autre modèle de calcul), assez rapidement d'autres problèmes ont aussi été montrés indécidables. Entre autres, par la possibilité des machines de Turing de manipuler des mots sur un alphabet sans faire de codage au préalable, des résultats d'indécidabilité dans le monde des mots ont été établis. Ces résultats ont été obtenus en **réduisant** un problème connu comme indécidable au problème traité sur les mots. Définissons d'abord formellement cette notion de réduction que nous utiliserons sous une autre forme dans le chapitre suivant traitant de la complexité théorique des machines de Turing.

Définition 34 (Réduction).

Soient A et B deux ensembles de mots définis sur les alphabets finis Σ_1 et Σ_2 , respectivement. Nous dirons que A est **réductible** à B si, et seulement si il existe une fonction totale $f : \Sigma_1^* \rightarrow \Sigma_2^*$ telle que :

- f est calculable par un algorithme.
- pour tout mot $\alpha \in \Sigma_1^*$, $\alpha \in A \Leftrightarrow f(\alpha) \in B$.

La fonction f s'appelle la réduction de A à B .

Ainsi, s'il existe un algorithme qui décide de l'appartenance ou non d'un mot de Σ_2 dans B , alors il existe aussi un algorithme qui décide de l'appartenance ou non d'un mot de Σ_1 dans A . On réduit donc la décidabilité de A à celle de B . Inversement, si l'appartenance dans A est indécidable, elle l'est aussi pour B . C'est cette contraposée que l'on utilise pour démontrer l'indécidabilité d'un problème.

Dans la suite, on va s'intéresser à deux problèmes sur les mots connus comme indécidables :

1. Problème du mot dans les systèmes semi-Thuien, et
2. Problème de la correspondance de Post.

Ces deux problèmes sont souvent utilisés pour démontrer que d'autres problèmes sont indécidables (e.g. indécidabilité de la validation dans la logique des prédicats du premier ordre à partir de la correspondance de Post - cf. le cours en S8 "fondements logiques de l'informatique").

Problème du mot dans les systèmes semi-Thuien

Les systèmes que nous allons voir ici sont des instances d'un problème plus général appelé le **problème du mot dans les systèmes de réécriture**. Les systèmes de réécriture sont l'objet de la dernière partie du cours de "fondements logiques de l'informatique" proposé en S8. Définissons ici simplement les systèmes semi-Thuien.

Définition 35 (Système semi-Thuien).

Un **système semi-Thuien** $\mathcal{S}$ est la donnée d'un alphabet fini Σ et d'un ensemble fini de couples (α, β) de mots sur Σ tels que α n'est pas le mot vide. Un tel système introduit naturellement les règles de réécriture suivantes (une par couple) : si un mot α' peut se factoriser en $\alpha_1.\alpha.\alpha_2$, alors en appliquant la règle on produit un nouveau mot $\beta' = \alpha_1.\beta.\alpha_2$. On note une telle production par $\alpha' \rightarrow_{\mathcal{S}} \beta'$. Ainsi, $\rightarrow_{\mathcal{S}}$ est une relation binaire sur Σ^* (i.e. $\rightarrow_{\mathcal{S}} \subseteq \Sigma^* \times \Sigma^*$), et donc on note $\xrightarrow{*}_{\mathcal{S}}$ sa fermeture réflexive et transitive.

Définition 36 (Problème du mot).

Étant donné un système semi-Thuien $\mathcal{S}$, on appelle **problème du mot**, le problème de décision qui consiste à savoir pour tout couple de mots (α, β) si $\alpha \xrightarrow{*}_{\mathcal{S}} \beta$.

Théorème 38.

Le problème du mot pour les systèmes semi-Thuien est indécidable.

Démonstration : À toute machine de Turing $\mathcal{M}$, on peut associer un système semi-Thuien $\mathcal{S}$. En effet, les configurations de la machine sont les mots sur l'alphabet $\Sigma \cup Q \cup \{\square\}$, et toute transition $q \xrightarrow{s/s'd} q'$ peut être transformée en la règle de réécriture suivante :

- $qs \rightarrow qs'$ si $d = 0$
- $qs \rightarrow s'q'$ si $d = +$
- $xqs \rightarrow q'xs'$ si $d = -$

La transformation est trivialement calculable. De plus, ce système semi-Thuien simule complètement la machine de Turing $\mathcal{M}$. On part d'une configuration initiale $q_0\alpha$ de la machine et l'on réécrit cette configuration en appliquant les règles ci-dessus. Très aisément, la machine $\mathcal{M}$ s'arrête si, et seulement si pour toute configuration initiale $q_0\alpha \xrightarrow{*}_{\mathcal{S}} \alpha'q'\beta'$ où $\alpha'q'\beta'$ est une configuration d'arrêt. Ainsi, le problème de la terminaison pour les MTs est équivalent au problème du mot pour les systèmes semi-Thuien obtenus à partir de la réduction ci-dessus des MTs. Le problème de la terminaison étant indécidable, le problème du mot pour les systèmes semi-Thuien l'est aussi. ■

Correspondance de Post

Avant de présenter ce problème, on va introduire quelques notations utiles à son énoncé. Soit $x = (\alpha_1, \dots, \alpha_n)$ une suite de mots tous définis sur un même alphabet V . Posons $I = \{1, \dots, n\}$. Soit $\sigma = i_1 i_2 \dots i_p$ un mot dans I^+ . On note σx le mot sur V :

$$\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_p}$$

Par exemple, soit $x = (bb, ab, b)$ et $\sigma = 1223$. σx est alors égale à $bbababb$.

Définition 37 (Correspondance de Post).

On appelle **problème de la correspondance de Post** le problème de décision qui consiste à partir de deux suites finies non vides x et y toutes les deux indexées par le même ensemble I et dont tous les mots sont définis sur un même alphabet Σ , à chercher un mot $\sigma \in I^+$ tel que $\sigma x = \sigma y$.

Théorème 39.

Le problème de la correspondance de Post est indécidable.

Démonstration : Pour cela, on va réduire le problème du mot pour les systèmes semi-Thuien au problème de la correspondance de Post.

Tout d'abord, tout système semi-Thuien $\mathcal{S}$ défini sur un alphabet Σ fini peut se ramener à un système semi-Thuien équivalent dont l'alphabet ne comporte que deux lettres, par exemple $\{a, b\}$. En effet, Σ étant de taille finie, on peut numérotiser ses éléments, prendre leur notation binaire et répercuter cette notation sur chacun des mots de Σ^* et donc sur les règles du système $\mathcal{S}$. En renommant 0 par a et 1 par b , on obtient bien un système semi-Thuien équivalent à $\mathcal{S}$ mais sur $\{a, b\}$.

Donc, soit le système semi-Thuien $\mathcal{S}$ sur l'alphabet $\Sigma = \{a, b\}$ comportant n règles $\alpha_i \rightarrow \beta_i$, et soit (α, β) une instance du problème du mot pour $\mathcal{S}$. On construit alors le problème de la correspondance de Post sur l'alphabet $\Sigma' = \{a, b, a', b', [,], >, <\}$. Pour tout mot $\alpha \in \Sigma^*$, on écrit α' le mot sur Σ' obtenu à partir de α en remplaçant les lettres a et b par a' et b' , respectivement.

On prend $I = \{1, \dots, 8 + 2n\}$ indexant les deux suites x et y des mots sur Σ' définies par :

$$\begin{aligned} x_1 &= [\alpha < & x_2 &=] & x_3 &= < & x_4 &= > & x_5 &= a & x_6 &= a' & x_7 &= b & x_8 &= b' \\ y_1 &= [& y_2 &= > \beta] & x_3 &= > & x_4 &= < & x_5 &= a' & x_6 &= a & x_7 &= b' & x_8 &= b \\ \forall i &= 1, 3, 5, \dots, 2n-1 & x_{8+i} &= \beta_i & y_{8+i} &= \alpha'_i & x_{8+i+1} &= \beta'_i & y_{8+i+1} &= \alpha_i \end{aligned}$$

Soit $\alpha = u_1 \rightarrow_{\mathcal{S}} u_2 \rightarrow_{\mathcal{S}} \dots \rightarrow_{\mathcal{S}} u_p = \beta$ une solution au problème du mot (α, β) . Alors, selon que p soit pair ou impair, le mot ω

1. p est pair, $\omega = [u_1 < u'_2 > u_3 < \dots < u'_p > u_p]$
2. p est impair, $\omega = [u_1 < u'_2 > u_3 < \dots < u'_{p-1} > u_p]$

est une solution au problème de la correspondance de Post ci-dessus. En effet, nous avons les deux décompositions suivantes

$$\omega = \begin{aligned} &[u_1 < |u'_2 > |u_3 < | \dots |] \\ &[|u_1 < |u'_2 > | \dots | > u_p] \end{aligned}$$

Ainsi, la solution est $\sigma = i_1 \dots i_p$ si p est impair, $\sigma = i_1 \dots i_{p+1}$ si p est pair. Dans les deux cas, pour chaque j , $2 \leq j \leq p-1$ (resp. $2 \leq j \leq p$) pour voir dans le cas où j est pair que $u'_j >$ correspond à $u_{j-1} <$, notons que nous pouvons écrire $u_{j-1} = \delta_1 \alpha_k \delta_2$ et $u'_j = \delta'_1 \beta'_k \delta'_2$. On a alors la correspondance triviale qui consiste à utiliser les indices 5 et 7 autant de fois que la taille des mots δ_1 et δ_2 le requièrent pour renommer les a en a' et b en b' et l'indice $8+k+i$. Dans le cas où j est impair, la démarche est identique.

Inversement, si σ est une solution au problème de la correspondance de Post, nécessairement le mot $\sigma x = \sigma y = \omega$ est de la forme $[\alpha < \dots > \beta]$. Il n'est pas possible de commencer et de finir par un indice i autre que 1 et 2 car les mots x_i et y_i sont différents sur toutes leurs lettres. Là encore, par la forme des mots dans la suite x et y , on a

$$\omega = \begin{array}{l} [\alpha < | \dots > \beta] \\ [|\alpha < | \dots | > \beta] \end{array}$$

Nous avons alors nécessairement une correspondance du mot $\alpha <$ avec un certain $\delta' >$ et du mot $> \beta$ avec un certain $< \tau'$ tels que $\alpha \rightarrow_S^* \delta$ et $\tau \rightarrow_S^* \beta$. Ainsi, le mot a la forme plus précise

$$\omega = \begin{array}{l} [\alpha < |\delta' > | \dots | < \tau' | > \beta] \\ [|\alpha < |\delta' > | \dots | < \tau' | > \beta] \end{array}$$

On peut réitérer le procédé sur $\delta' >$ et $< \tau'$. Le mot ω étant de taille finie, ce procédé terminera et donc on aboutira à $\alpha \rightarrow_S^* \beta$. ■

Théorème de Rice

Le théorème de Rice est un résultat important de la théorie de la calculabilité qui dit (en gros) que toute propriété non triviale (i.e. qui n'est pas toujours vraie ou toujours fausse) sur les programmes est indécidable. Avant d'énoncer ce théorème, nous devons d'abord introduire les notions de langages acceptés et décidés par une machine de Turing (MT). On parlera aussi de langage *récursivement énumérable* et *rékursif*.

Définition 38 (Mot accepté).

Soit $\mathcal{M}$ une MT sur un alphabet Σ . Soit ω un mot de Σ (i.e. $\omega \in \Sigma^*$). ω est **accepté** par $\mathcal{M}$ si l'exécution de la machine $\mathcal{M}$ pour reconnaître ω mène à une configuration dont l'état est final (considéré alors comme état acceptant).

Un mot ω n'est pas accepté par une machine $\mathcal{M}$ si soit la machine s'arrête sur un état non final, soit elle ne termine pas.

Définition 39 (Langage accepté et décidé).

Soit $\mathcal{M}$ une MT sur un alphabet Σ . Soit ω un mot de Σ . Notons $\mathcal{L}(\mathcal{M})$ le langage des mots $\omega \in \Sigma^*$ acceptés par $\mathcal{M}$.

$\mathcal{L}(\mathcal{M})$ est appelé le **langage accepté** par $\mathcal{M}$.

$\mathcal{L}(\mathcal{M})$ est dit **langage décidé** si de plus $\mathcal{M}$ n'a pas d'exécution infinie.

Ainsi, dans un langage $\mathcal{L} \subseteq \Sigma^*$ décidé par une MT $\mathcal{M}$, $\mathcal{M}$ définit une procédure effective qui pour tout mot $\omega \in \Sigma^*$ répond oui si $\omega \in \mathcal{L}(\mathcal{M})$, et non si $\omega \notin \mathcal{L}(\mathcal{M})$. A l'inverse, dans un langage accepté, on ne sait pas répondre pour certains mots dont l'exécution est infinie.

Définition 40 (Langage rékursif et récursivement énumérable).

Un langage $\mathcal{L} \subseteq \Sigma^*$ est dit **rékursif (R)** (resp. **récursivement énumérable (RE)**) s'il est décidé (resp. accepté) par une MT.

Proposition 40.

Le complément d'un langage R est aussi R .

Démonstration : Soit $\mathcal{L}$ un langage décidé par une MT $\mathcal{M}$. Il est facile de définir une MT $\mathcal{M}'$ qui répond oui sur un mot quand $\mathcal{M}$ répond non, et inversement. $\mathcal{M}'$ étant définie à partir de $\mathcal{M}$ en inversant ses réponses, elle termine pour tout mot ω , et donc le langage $\mathcal{L}(\mathcal{M}')$ est récursif. ■

Proposition 41.

Si un langage $\mathcal{L}$ et son complément sont RE, alors ils sont tous les deux R.

Démonstration : Il suffit de composer les deux MTs pour avoir une MT capable de reconnaître l'appartenance ou non d'un mot à $\mathcal{L}$ et donc aussi à son complémentaire en inversant les réponses. ■

Une conséquence des deux propositions ci-dessus est qu'étant donné un langage $\mathcal{L}$, on a l'un des trois cas suivants : On note $\overline{\mathcal{L}}$ le complémentaire de $\mathcal{L}$

1. $\mathcal{L}$ et $\overline{\mathcal{L}}$ sont tous les deux R.
2. $\mathcal{L}$ et $\overline{\mathcal{L}}$ ne sont pas RE.
3. $\mathcal{L}$ n'est pas RE mais $\overline{\mathcal{L}}$ est RE mais non R.

Ainsi, un langage est **indécidable** quand lui ou son complémentaire ne sont pas RE.

Un langage important pour notre propos est le langage $\mathcal{LU}$ sous-jacent à toute MT universelle. Les mots acceptés par cette machine sont les codifications dans un alphabet approprié que l'on notera $\langle \mathcal{M}, \omega \rangle$, d'une MT $\mathcal{M}$ sur un alphabet Σ et d'un mot $\omega \in \Sigma^*$. $\mathcal{LU}$ se définit alors :

$$\langle \mathcal{M}, \omega \rangle \in \mathcal{LU} \Leftrightarrow \omega \in \mathcal{L}(\mathcal{M})$$

Par définition, le langage $\mathcal{LU}$ est RE car on a montré l'existence de ces MTs universelles. Par contre, il n'est pas récursif car on sait qu'il existe des MTs $\mathcal{M}$ qui ne terminent pas sur certains mots ω et donc la MT universelle ne peut pas non plus répondre sur l'appartenance à $\mathcal{LU}$ de tels $\langle \mathcal{M}, \omega \rangle$. Ainsi, par les propositions ci-dessus, on a aussi que le complémentaire $\overline{\mathcal{LU}}$ est aussi non R. En fait, comme $\mathcal{LU}$ est RE, $\overline{\mathcal{LU}}$ n'est pas non plus RE et donc est indécidable. C'est ce langage que nous allons utiliser dans la preuve du théorème de Rice pour obtenir notre résultat d'indécidabilité recherché.

Théorème 42 (Théorème de Rice).

Toute propriété P non triviale sur les langages RE est indécidable ^a.

- ^a. Par propriété P non triviale, on entend que P satisfait aux deux conditions suivantes :
1. Pour toutes MTs $\mathcal{M}, \mathcal{M}'$, si $\mathcal{M} \in P$ et $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$, alors $\mathcal{M}' \in P$.
 2. Il existe deux MTs $\mathcal{M}, \mathcal{M}'$ telles que $\mathcal{M} \in P$ et $\mathcal{M}' \notin P$.

Démonstration : P est un sous-ensemble de MTs. On peut supposer que P ne contient pas le langage vide. En effet, si P le contient, on peut alors considérer le complément de P et montrer son indécidabilité par la construction ci-dessous. En effet, un problème est indécidable si, et seulement si son complémentaire est indécidable.

Réduisons le langage $\mathcal{LU}$ à P . À toute instance $\langle \mathcal{M}, \omega \rangle$ du problème de l'arrêt, on associe la MT $\mathcal{M}'$ qui a le comportement suivant : On se donne une des MTs $\mathcal{M}_P$ d'un des langages de P (ceci est possible car P est non triviale et donc n'est pas vide).

- quelque soit le mot δ qu'elle a entrée, elle simule la MT $\mathcal{M}$ sur ω ;
- si $\mathcal{M}$ accepte ω , alors on exécute $\mathcal{M}_P$ sur δ ;
- si $\mathcal{M}$ n'accepte pas ω (i.e. le rejette ou a une exécution infinie), alors $\mathcal{M}'$ n'accepte pas δ (et donc au final le langage $\mathcal{L}(\mathcal{M}') = \emptyset$)

Cette réduction est correcte. En effet, on a :

- $\mathcal{L}(\mathcal{M}') = \emptyset$ et donc ne satisfait pas la propriété P si, et seulement si $\mathcal{M}$ n'accepte pas ω et donc $\langle \mathcal{M}, \omega \rangle \notin \mathcal{LU}$;
- $\mathcal{L}(\mathcal{M}') = \mathcal{L}(\mathcal{M}_P)$ et donc satisfait la propriété P si, et seulement si $\mathcal{M}$ accepte ω et donc $\langle \mathcal{M}, \omega \rangle \in \mathcal{LU}$;

■

Un corollaire du théorème de Rice est qu'il n'est pas possible de décider algorithmiquement pour un programme donné s'il est correct, i.e. s'il vérifie sa spécification. En effet, cette propriété de correction est non triviale (des programmes sont correctes et d'autres non pour une spécification donnée).

8. Complexité

Dans ce chapitre nous étudierons essentiellement la complexité en temps des algorithmes et n'arborerons pas la complexité en espace, cette dernière étant moins contrainte par l'explosion de la taille des mémoires des ordinateurs.

8.1. Complexité en temps

Toute machine de Turing (MT) $\mathcal{M}$ peut se voir comme un *problème de décision*, i.e. un langage $\mathcal{L}(\mathcal{M}) \subseteq \Sigma^*$ sur son alphabet Σ tel que :

$$\mathcal{L}(\mathcal{M}) = \{\alpha \in \Sigma^* | \mathcal{M} \text{ s'arrête sur } \alpha\}$$

Quand $\mathcal{L}(\mathcal{M})$ est récursif, $\mathcal{M}$ est appelé un **décideur**, i.e. toutes les branches de calcul de $\mathcal{M}$ s'arrêtent. On va alors *mesurer* ces branches qui permettra d'avoir une idée sur le comportement d'un algorithme. Cette mesure doit donc être *abstraite*, i.e. indépendante de la représentation des données en entrée et de la machine (physique) sur lequel il est exécuté.

On est donc intéressé par :

- la taille des données plus que leur valeur
- les ordres de grandeur et la croissance de cette mesure pour des données de grande taille.

Ainsi, soit $\mathcal{M}$ une Machine de Turing qui est un décideur. La complexité en temps est une fonction $\tau_{\mathcal{M}} : \mathbb{N} \rightarrow \mathbb{N}$ telle que $\tau_{\mathcal{M}}(n)$ soit le nombre maximal de pas qu'effectue $\mathcal{M}$ lorsqu'on lui soumet un mot sur son alphabet de taille n . Ainsi, si $f : \mathbb{N} \rightarrow \mathbb{N}$ est une fonction, on dit que $\mathcal{M}$ est en complexité en temps $f(n)$ si $\tau_{\mathcal{M}}(n) \leq f(n)$ pour tout n . En fait, on s'intéressera à une complexité asymptotique (i.e. pour n grand) en se contentant des ordres de grandeur et de la croissance de la fonction f .

Définition 41 (Complexité asymptotique).

Soit $\mathcal{M}$ une MT et $f : \mathbb{N} \rightarrow \mathbb{N}$ une fonction. La **complexité asymptotique** de $\mathcal{M}$ est dite en $O(f(n))$ si, et seulement si, il existe deux constantes $c \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ telles que $\tau_{\mathcal{M}}(n) \leq c.f(n)$ pour tout $n \geq n_0$.

Deux classes de complexité ont émergé naturellement, la classe contenant tous les problèmes que l'on peut résoudre de façon *déterministe* (i.e. sans choix dans le parcours d'arbres des exécutions) en temps dit polynomial (i.e. que la fonction f définit un polynôme en n), et ceux que l'on peut résoudre toujours en temps polynomial mais de façon *non déterministe* (i.e. à chaque pas de calcul un choix dans la branche à suivre peut se présenter).

Définition 42.

Une machine de Turing $\mathcal{M} = (\Sigma, Q, R)$ est **déterministe** si R peut être définie par une fonction $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{0, +, -\}$, et **non déterministe** sinon (i.e. $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma \times \{0, +, -\})$ est une relation).

Définition 43 (Complexité d'une MT).

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$. La classe de complexité $TIME(f(n))$ contient l'ensemble des langages qui sont décidés par des MTs (déterministes) en temps $O(f(n))$.

La classe $NTIME(f(n))$ contient l'ensemble des langages qui sont décidés par des MTs non déterministes en temps $O(f(n))$.

8.2. Les classes P et NP

Succinctement, la classe P contient tous les problèmes résolus par une MT déterministe en temps polynomial tandis que la classe NP contient tous ceux résolus par une MT non déterministe aussi en temps polynomial. Ce qui fait que les problèmes de la classe NP sont considérés comme être résolus de façon non efficace est leur coût exponentiel quand ces derniers sont résolus par des machines déterministes (machines usuelles), et ce provenant des choix dans l'arbre d'exécution comme l'établit le résultat suivant :

Théorème 43.

Soit $f : \mathbb{N} \rightarrow \mathbb{N}$ telle que pour tout $n \in \mathbb{N}$, $f(n) \geq n$. Soit $\mathcal{M}$ une MT non déterministe de complexité $O(f(n))$. Il existe une MT déterministe $\mathcal{M}'$ qui décide le même langage en $2^{O(f(n))}$.

Démonstration La MT $\mathcal{M}'$ doit simuler toutes les exécutions de $\mathcal{M}$. Toutes ces exécutions sont de longueur bornée par $O(f(n))$. Soit m un majorant du nombre de choix à chaque transition. L'exploration de toutes ces exécutions, jusqu'à leur longueur maximale $f(n)$, demande au plus $m^{f(n)}$. Maintenant, la simulation de chacun de ces calculs est bornée par $O(f(n))$. Donc le temps de la calcul de $\mathcal{M}'$ est bornée par $O(f(n)) \times m^{f(n)}$, et donc par $2^{\log_2(m)(f(n)+O(f(n)))}$, i.e. $2^{O(f(n))}$. $\square$

La définition des classes P et NP est alors la suivante :

Définition 44 (Classes P et NP).

Les **classes P et NP** sont les réunions des classes $TIME(n^k)$ et $NTIME(n^k)$ pour $k \geq 1$, respectivement.

Exemple 45 (Un exemple de problème P : PATH).

Étant donné un graphe $G = (X, A)$ et 2 sommets $s, s' \in X$, existe-t-il un chemin de s à s' ? L'on représente classiquement les graphes par leur matrice d'adjacence, i.e. une matrice $X \times X$ telle que chaque entrée $a_{s,s'}$ est le nombre d'arêtes reliant s à s' .

Théorème 45.

PATH $\in$ P.

Démonstration On marque une et une seule fois chaque sommet rencontré en commençant par s , i.e. que l'on exige que chaque sommet marqué ne sera jamais revisité. Si l'on rencontre s' on s'arrête et on répond "Oui", sinon (i.e. tous les sommets rencontrés sont marqués et on n'a pas rencontré s') on répond "Non". Ainsi, à la 1ère étape on va visiter au plus dans la matrice $|X| - 1$ sommets, à la seconde étape $|X| - 2$ sommets, etc.

Cet algorithme est borné par $O(n^2)$ où n est le nombre de sommets (pire cas, celui du graphe en peigne). $\square$

Exemple 46.

Autre exemple- Coprimalité. Tester pour 2 nombres entiers a et b si ils sont premiers entre eux. On applique l'algorithme d'Euclide qui calcule de façon déterministe le pgcd de a et b , et dont on sait que la complexité est $O(\log_2 \sup(a, b))$.

Ainsi, pour un problème de NP si l'on dispose d'une solution, on doit être capable de la vérifier en temps polynomiale. On parle alors de *certificat*.

Définition 45 (Certificat).

Soit $\mathcal{L} \subseteq \Sigma^*$ un langage sur l'alphabet Σ . $\mathcal{L} \in NP$ si, et seulement s'il existe $\mathcal{L}_R \subseteq \Sigma^* \times \Sigma^*$ t.q. :

- $\mathcal{L}_R \in P$
- $\forall x \in \Sigma^*, x \in \mathcal{L} \Leftrightarrow \exists y \in \Sigma^*, (x, y) \in \mathcal{L}_R$

La MT en temps polynomiale qui décide $\mathcal{L}_R$ est le **certificat**.

Exemple 47 (Problème COMPOSITE).

Ce problème consiste à tester si un nombre est composable (i.e. n'est pas premier). Ce problème de décision se définit par :

$$COMPOSITE = \{n | \exists a, b > 1, n = ab\}$$

De là, on peut définir le langage $\mathcal{L}_R$ de la façon suivante :

$$\mathcal{L}_R = \{(x, y) | 1 < y < \sqrt{x}, y \text{ divise } x\}$$

Maintenant, vérifier si y divise x se fait en temps polynomial.

En fait, il est facile de montrer que COMPOSITE est dans P. En effet, la recherche des diviseurs de n est bornée par $\sqrt{n}$.

Le problème dual défini par le langage PRIME contenant tous les nombres premiers est aussi dans P mais ceci n'a été démontré que récemment (cf. Théorème d'Agrawal, Kayal, et Saxena 2002).

Exemple 49 (Exemple de problème NP : HAMIL).

Étant donné un graphe $G = (X, A)$, existe-t-il un chemin qui passe par tous les sommets du graphe ? Un chemin vérifiant cette propriété est appelé **chemin Hamiltonien** du nom de l'astronome et mathématicien anglais William Rowan Hamilton.

Théorème 49.

HAMIL $\in$ NP.

Démonstration Le problème HAMIL peut se traduire par le problème de décision représenté par le langage :

$$HAMIL = \{G | \exists \sigma X, \sigma X \text{ est un chemin}\}$$

σX étant une substitution des sommets de X .

De là, on peut définir simplement le certificat :

$$\mathcal{L}_R = \{(G, \sigma X) | \sigma X \text{ est un chemin de } G\}$$

Maintenant, contrôler qu'une substitution des sommets est un chemin dans G se fait en temps polynomial. $\square$

Ce problème est naturellement exponentiel car il demande de vérifier dans le pire cas toutes les substitutions de sommets du graphe dont le nombre est $|X|!$.

8.3. Structure de la classe NP

On peut démontrer simplement que $P \subseteq NP$. En effet, l'algorithme de vérification est l'algorithme de décision, ce dernier étant déjà de complexité polynomiale pour une MT déterministe. On n'a donc pas besoin de certificat ou plus exactement le certificat que l'on considère est pour tout langage $\mathcal{L} \subseteq \Sigma^*$ calculable par une MT déterministe en temps polynomiale, le langage $\mathcal{L}_R \subseteq \Sigma^* \times \Sigma^*$ suivant :

$$\mathcal{L}_R = \{(x, \varepsilon) | x \in \mathcal{L}\}$$

où ε désigne le mot vide sur Σ .

Démontrer l'inclusion inverse consisterait à démontrer que $P = NP$. Or, ceci est une conjecture, plus exactement sa négation, qui fait partie des problèmes à 1M\$ de l'institut Clay, et qui n'a toujours pas reçu de réponse. Par contre, on peut démontrer que la classe NP est close par union, intersection, et concaténation (cf. l'exercice énoncé ci-dessous). À l'inverse, on ne sait pas si la classe NP est close par complémentaire, i.e. pour un langage $\mathcal{L}$ dans NP, on ne sait pas répondre pour $\Sigma^* \setminus \mathcal{L}$.

Problèmes NP-complets et NP-durs

Face à toutes ces questions, les chercheurs ont tenté de connaître de façon plus précise la structure de la classe NP, et ce dans l'espoir de répondre à ces questions (et il y en a bien d'autres). Parmi ces chercheurs, un particulièrement a oeuvré dans la description de la structure de la classe NP : S.-A. Cook. Ce dernier a en effet défini une sous-classe de NP, de problèmes dont les définitions contiennent toute la difficulté de la classe NP. Cette sous-classe connue sous le nom de la classe des problèmes NP-complets, contient alors tous les problèmes dont la résolution engendre nécessairement, à une transformation polynomiale près, la résolution de tous les problèmes de la classe NP. Ainsi, si pour un de ces problème NP-complet, nous étions capables de trouver une MT déterministe en temps polynomial qui résolvait ce problème, alors nous aurions toujours à une transformation polynomiale près, un algorithme raisonnable qui résolverait tous les problèmes NP et ainsi on aurait démontré que $P = NP$. Inversement, si nous montrons qu'il ne peut exister une MT déterministe en temps polynomial pour seulement un problème de la classe NP-complet, alors nous aurons montré que $P \neq NP$. Bien sûr, pour l'instant ni l'un, ni l'autre de ces cas n'ont encore été obtenus. Néanmoins, la définition de cette sous-classe a permis de grandes avancées dans le domaine de la théorie de la complexité. Nous allons donc dans la section suivante définir cette sous-classe des problèmes dits NP-complets. Comme nous l'avons déjà dit ci-dessus, la classe NP-complet contient tous les problèmes dont toute propriété sur leur complexité devient, à une transformation polynomiale près, une propriété de tous les problèmes NP. Que veut bien dire "à une transformation polynomiale près" ? On comprend aisément l'adjectif "polynomial" car on ne veut pas que cette transformation engendre un surcoût de complexité. La question est plutôt de savoir ce que l'on entend par transformation ? Ce que l'on veut obtenir est donc un moyen permettant de montrer qu'un problème n'est pas plus dur qu'un autre, i.e. que toutes instances d'un problème peut à transformation près se traduire en une instance d'un autre problème. Ainsi, si un premier problème se réduit en un second, le premier problème est (au moins) aussi facile que le second - si la réduction est facile, i.e. en temps polynomial. On pourra donc utiliser la réduction de deux façon :

1. pour montrer qu'un problème est NP-complet : si le premier est réputé NP-complet (i.e. dur), le second l'est aussi ;

2. pour montrer qu'un problème est facile : si le deuxième est facile, le premier l'est aussi.

C'est en fait surtout le premier raisonnement que l'on utilise généralement. Définissons alors cette notion de réduction ou transformation polynomiale.

Définition 46 (Transformation polynomiale).

Soient $\mathcal{L}_1 \subseteq \Sigma_1^*$ et $\mathcal{L}_2 \subseteq \Sigma_2^*$ deux langages. Une **transformation polynomiale** de $\mathcal{L}_1$ vers $\mathcal{L}_2$ est une fonction $f : \Sigma_1^* \rightarrow \Sigma_2^*$ qui satisfait les conditions suivantes :

1. f est calculable en temps polynomial, i.e. par une MT déterministe en temps polynomial.
2. $f(x) \in \mathcal{L}_2 \iff x \in \mathcal{L}_1$

On note $\mathcal{L}_1 \rightsquigarrow \mathcal{L}_2$.

On remarque simplement que la relation $\rightsquigarrow$ est transitive car définie à partir de la loi $\circ$, et la composition de deux fonctions polynomiales est trivialement une fonction polynomiale.

Exemple 50.

Soient les deux problèmes suivants :

1. Problème 1 : Un jeu composé de N participants et d'une liste L de paires de participants : les paires d'ennemis. Le but est de créer p équipes de telle sorte qu'aucune équipe ne contienne une paire d'ennemis.
2. Problème 2 : Soit $G = (X, A)$ un graphe et k un nombre de couleurs. G est-il k -coloriable ? ^a

Le langage associé au problème 1 est alors :

$$\mathcal{L}_1 = \{((N, L), p) \mid \text{possibilité de faire } p \text{ équipes sans paire d'ennemis}\}$$

et le langage associé au problème 2 est le suivant :

$$\mathcal{L}_2 = \{(G, k) \mid G \text{ est } k\text{-coloriable}\}$$

Deux transformations polynomiales $f_1 : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ et $f_2 : \mathcal{L}_2 \rightarrow \mathcal{L}_1$ peuvent être simplement définies :

1. Pour f_1 , étant donnée une instance $((N, L), p)$ de $\mathcal{L}_1$, on peut définir le graphe G dont les sommets sont numérotés de 1 à N et dénotent les participants au jeu et les arcs sont toutes les paires de la liste L . On pose alors $k = p$. On a bien G est k -coloriable si, et seulement si il est possible de faire p équipes. La transformation f_1 n'étant qu'un renommage, elle est trivialement polynomiale en $N + |L|$.
2. De la même manière, pour f_2 , étant donné un graphe $G = (X, A)$ et un entier positif k , on peut définir le jeu $((N, L), p)$ en posant $N = |X|$, $L = A$ et $p = k$. Là encore, on peut faire p équipes si, et seulement si G est k -coloriable. Enfin, la transformation n'étant qu'un calcul de cardinalité et un renommage, elle est aussi polynomiale en temps de calcul.

a. G est dit **k -coloriable** si, et seulement s'il existe une application g de X dans $\{1, 2, \dots, k\}$ telle que pour tout $(x, y) \in A$, $g(x) \neq g(y)$.

Proposition 51.

Si $\mathcal{L}' \in P$ et $\mathcal{L} \rightsquigarrow \mathcal{L}'$, alors $\mathcal{L} \in P$.

Démonstration Soit u une instance du problème posé par $\mathcal{L}$. Pour décider si $u \in \mathcal{L}$, il suffit d'appliquer la transformation f à u et tester si $f(u) \in \mathcal{L}'$. Maintenant, tester cette appartenance est par hypothèse effectuée en temps polynomial et la transformation est elle-même calculable en temps polynomial. Ainsi, tester l'appartenance de u à $\mathcal{L}$ consiste à composer deux MTs déterministe en temps de calcul polynomial, qui donne donc une MT déterministe qui reste encore en temps de calcul polynomial. $\square$

On peut maintenant définir la classe des problèmes NP-complets.

Définition 47 (NP-complétude).

Un langage $\mathcal{L}$ est **NP-complet** si :

- $\mathcal{L} \in NP$;
- $\forall \mathcal{L}' \in NP, \mathcal{L}' \rightsquigarrow \mathcal{L}$.

La classe des problèmes NP-complets impose comme contrainte sur les langages qu'ils soient reconnus par une MT non déterministe en temps polynomial. Or, il existe des problèmes décidables qui ne sont pas reconnus en temps polynomial même sur une machine non déterministe. Pour de tels problèmes possédant aussi la particularité que tous les problèmes NP peuvent se réduire en eux par une transformation polynomiale, on dira qu'ils sont **NP-durs**. En résumé, démontrer qu'un problème est NP-dur établit qu'il n'a vraisemblablement pas de solution calculable en temps polynomial, démontrer qu'il est NP-complet assure au moins l'existence d'une solution polynomiale sur une MT non déterministe.

Proposition 52.

Si $\mathcal{L}$ est NP-dur et $\mathcal{L} \rightsquigarrow \mathcal{L}'$, alors $\mathcal{L}'$ est NP-dur.

Démonstration Par hypothèse, $\mathcal{L}'$ contient à la transformation polynomiale près donnée par $\mathcal{L} \rightsquigarrow \mathcal{L}'$, un sous-langage permettant de réduire tous les problèmes NP. $\square$

Un premier problème NP-complet

La question naturelle que l'on peut se poser est de savoir si de tels problèmes NP-complets (NP-durs) existent ? La réponse est positive et a été apportée par S.-A. Cook en 1970 dans un papier à la base de la théorie de la complexité des algorithmes et a valu à l'auteur le Turing award équivalent du prix Nobel en informatique et informatique théorique. Le premier problème que S.-A. Cook a montré être NP-complet vient du monde de la logique mathématique, et plus précisément de la logique dite propositionnelle. La logique propositionnelle est un formalisme qui étudie les valeurs de vérité de propositions complexes obtenues par composition de propositions élémentaires à partir des connecteurs logiques standards que sont la conjonction (notée $\wedge$), la disjonction (notée $\vee$), l'implication ($\Rightarrow$), et la négation (notée $\neg$). Formellement, la logique propositionnelle se définit de la façon suivante :

- On se donne un ensemble P dont les éléments sont appelés des **variables propositionnelles**. Une formule propositionnelle sur P est alors une suite de symboles pris dans $P \cup \{\Rightarrow, \neg, \wedge, \vee, (,)\}$ et construite selon les règles suivantes :
 1. toute variable propositionnelle est une formule sur P .
 2. si φ et ψ sont des formules sur P alors $\varphi @ \psi$ est une formule sur P avec $@ \in \{\wedge, \vee, \Rightarrow\}$.
 3. si φ est une formule sur P alors $\neg\varphi$ est une formule sur P .
 4. toute formule sur P est obtenue par application répétée, un nombre fini de fois des étapes 1. à 3. ci-dessus¹.

1. C'est une définition inductive.

Pour le problème qui nous intéresse ici, on se restreindra à un type de formule particulier, les formules en **forme normale conjonctive**. Ce sont toutes les formules de la forme :

$$E_1 \wedge \dots \wedge E_k$$

et chaque E_i est une **clause**, i.e. une disjonction de variables propositionnelles ou de négations de variables propositionnelles, à savoir une formule de la forme :

$$x_{i1} \vee \dots \vee x_{il}$$

où x_{ij} est soit p , soit $\neg p$ avec $p \in P$. En fait, on peut montrer que toute formule peut être transformée algorithmiquement en une formule équivalente (i.e. avec les mêmes valeurs de vérité - voir juste ci-dessous) en forme normale conjonctive.

- Pour calculer la valeur de vérité d'une formule φ , on commence par donner une valeur de vérité aux variables propositionnelles dans P . Puis, on applique les tables de vérité de chacun des connecteurs apparaissant dans φ , les valeurs de vérité données aux variables propositionnelles, appelées **modèles**, caractérisant une ligne dans le tableau. Donc, étant donné un modèle défini par une application $\nu : P \rightarrow \{0, 1\}$, la valeur de vérité d'une formule φ , notée $\nu^*(\varphi)$, se définit inductivement sur la structure de φ de la façon suivante :
 - si $\varphi \in P$ alors $\nu^*(\varphi) = \nu(\varphi)$,
 - si $\varphi = \varphi_1 \wedge \varphi_2$ (resp. $\varphi_1 \vee \varphi_2$, resp. $\varphi_1 \Rightarrow \varphi_2$) alors $\nu^*(\varphi) = 1$ si, et seulement si $\nu^*(\varphi_1) \times \nu^*(\varphi_2) = 1$ (resp. $\nu^*(\varphi_1) + \nu^*(\varphi_2) \geq 1$, resp. $\nu^*(\varphi_1) \leq \nu^*(\varphi_2)$),
 - si $\varphi = \neg \varphi_1$ alors $\nu^*(\varphi) = 1$ si, et seulement si $\nu^*(\varphi_1) = 0$.

On dit alors que φ est **satisfiable** s'il existe un modèle ν tel que $\nu^*(\varphi) = 1$.

Le problème qui a été démontré par Cook comme premier problème NP-complet et connu sous l'acronyme SAT, est celui de la satisfiabilité pour les formules en forme normale conjonctive.

Théorème 53.

SAT est NP-complet.

Démonstration Montrons tout d'abord que SAT est NP. Soit $\mathcal{L}_R = \{(\varphi, \nu) | \nu^*(\varphi) = 1\}$ où $\nu : P \rightarrow \{0, 1\}$ est une valuation des variables propositionnelles (ν peut se représenter par une séquence de variables propositionnelles pour dénoter celles mises à 1). Étant donnée une valuation, il est facile en temps polynomial en la taille de φ (parcours d'arbre) de vérifier sa valeur de vérité.

On va maintenant montrer comment transformer toute MT déterministe polynomiale au problème SAT. Pour cela, on va définir une formule propositionnelle φ dont les variables représentent l'historique du calcul qu'effectuerait la MT, un choix de valeurs des variables fournissant une branche du calcul.

On se donne une MT $\mathcal{M}$ non déterministe d'alphabet Σ . Soit $p(n)$ la complexité de $\mathcal{M}$ pour décider de l'appartenance d'un mot de longueur n au langage calculé. Pour construire les clauses, nous aurons besoin des variables suivantes :

- pour $q \in Q$ et $0 \leq k \leq p(n)$, $Q(q, k) = 1$ ssi l'état de la MT à l'étape k est q .
- pour $1 \leq i \leq p(n)$ et $0 \leq k \leq p(n)$, $T(i, k) = 1$ ssi la position de la MT à l'étape k est i .
- pour $1 \leq i \leq p(n)$, $0 \leq k \leq p(n)$ et $j \in \Sigma$, $R(i, j, k) = 1$ ssi à l'étape k , la case i contient le symbole j .

Le nombre de ces variables est $O(p(n)^2)$.

La formule φ que nous allons construire va être la conjonction d'une longue liste de clauses qui expriment que chaque étape du calcul obéit aux règles de la machine $\mathcal{M}$.

- Pour tout $i \leq p(n)$, le symbole $j \in \Sigma$ est l'état initial de la machine à la position i . Ceci s'exprime par la clause $R(i, j, 0)$.
- Pour tout état $q \in Q$, on a la clause $Q(q, 0)$ si l'état initial de la machine est q , la clause $\neg Q(q, 0)$ sinon.
- À l'état initial, la tête est au début du ruban. Ceci s'exprime par la clause $T(1, 0)$.
- À chaque instant, une case du ruban ne peut contenir qu'un unique symbole. Ceci s'exprime pour tout $i, k \leq p(n)$ et pour tout $j \neq j' \in \Sigma$ par la clause $\neg R(i, j, k) \vee \neg R(i, j', k)$.
- À chaque étape $k \leq p(n)$, la tête de lecture ne peut pas être simultanément à deux positions $i \neq i' \leq p(n)$ différentes. Ceci s'exprime par la clause $\neg T(i, k) \vee \neg T(i', k)$.
- À chaque étape $k \leq p(n)$, $\mathcal{M}$ ne peut pas être simultanément dans deux états $q \neq q' \in Q$ différents. Ceci s'exprime par la clause $\neg Q(q, k) \vee \neg Q(q', k)$.
- À chaque étape $k < p(n)$, la machine $\mathcal{M}$ ne modifie que la case $i \leq p(n)$ où la tête de lecture est positionnée. Ceci s'exprime par la clause $R(i, j, k) = R(i, j, k+1) \vee T(i, k)$ où $x = y$ est l'abréviation pour $(\neg x \wedge \neg y) \vee (x \wedge y)$.
- À chaque étape $k \leq p(n)$ et selon la position $i \leq p(n)$ de la machine $\mathcal{M}$, toute transition (q, j, q', j', d) de $\mathcal{M}$ peut être applicable si $\mathcal{M}$ est dans l'état q et le symbole lu à la position i est bien j . Ceci s'exprime par la clause

$$T(i, k) \wedge Q(q, k) \wedge R(i, j, k) \Rightarrow T(i + d, k + 1) \wedge Q(q', k + 1) \wedge R(i, j', k + 1)$$

- La clause $Q(q_f, p(n))$ où q_f est l'état final de $\mathcal{M}$.

Le temps de calcul de cette transformation correspond à la taille de la formule φ générée. Or, on constate que le nombre de clauses est en $O(p(n)^3)$. La transformation est donc polynomiale, comme requis. En outre, sa satisfiabilité est, par construction même, équivalente à l'existence d'une branche du calcul qui accepte le mot passé en argument (celui se trouvant sur le ruban initial). $\square$

La NP-complétude de SAT ayant été établie, il est maintenant possible de montrer par transformation polynomiale que d'autres problèmes sont NP-complets.

Exemple 54.

Nous allons montrer la NP-complétude pour une restriction de SAT. En effet, nous allons nous restreindre à des formules en forme normale conjonctive comportant exactement 3 littéraux par clause. Ce problème connu sous l'acronyme 3-SAT va donc manipuler des formules de la forme $E_1 \wedge \dots \wedge E_k$ où chaque E_i est la disjonction d'exactly 3 littéraux (variables propositionnelles ou négation de variables propositionnelles). Nous allons alors montrer que $\text{SAT} \rightsquigarrow \text{3-SAT}$. Pour cela, on remplace les clauses ne contenant pas 3 littéraux par :

- Une clause x_1 comportant un seul littéral est remplacée par :

$$(x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \neg y_2) \wedge (x_1 \vee \neg y_1 \vee y_2) \wedge (x_1 \vee \neg y_1 \vee \neg y_2)$$

- Une clause $(x_1 \vee x_2)$ contenant 2 littéraux est remplacée par :

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

- Une clause $(x_1 \vee \dots \vee x_l)$ avec $l \geq 4$ est remplacée par :

$$(x_1 \vee x_2 \vee y_1) \wedge \left(\bigwedge_{3 \leq i \leq l} (\neg y_{i-2} \vee x_i \vee y_{i-1}) \right) \wedge (\neg y_{l-3} \vee x_{l-1} \vee x_l)$$

où $y, y_1, \dots, y_{l-3}$ sont des variables nouvelles.

La construction que nous venons de décrire est aisément implantée par un algorithme polynomiale. Par la proposition 52 ci-dessus, nous avons donc démontré que 3-SAT est NP-dur. Maintenant, en suivant le même raisonnement que pour SAT, 3-SAT appartient à NP. Ainsi, 3-SAT est aussi NP-complet.

SOUS PARTIE 3

Théorie des langages

9. Introduction

Dans la partie sur la calculabilité, nous avons vu qu'un moyen simple de représenter un algorithme était par un graphe dont les noeuds définissent les états du système et les transitions, les actions de lecture ou d'écriture de caractères et de déplacement de la tête de lecture. C'est le modèle de calcul des *Machines de Turing*. Ici, nous allons étudier un formalisme plus restrictif permettant de modéliser des procédures effectives de reconnaissance des mots dans un langage, et donc très utilisé dans la modélisation des systèmes discrets comme les systèmes informatiques, les mots étant les exécutions possibles du système.

Au départ, cette forme de modélisation est née d'une tentative de représentation formelle des langues naturelles. Elle connut une expansion considérable lorsque l'on s'aperçut de son adéquation à la description à la fois des langages de programmation (description des analyseurs syntaxiques et compilateurs) et des programmes eux-mêmes. Dans le premier cas, le formalisme des automates est utilisé pour la définition de procédures effectives (i.e. calculables) de reconnaissance de langages (i.e. montrer qu'un mot construit sur un alphabet Σ donné appartient ou non à une partie $L \subseteq \Sigma^*$, le langage défini). Cette approche, appelée souvent *théorie des langages*, a reçu beaucoup d'attention de la part des chercheurs en informatique et a eu de nombreuses applications dans la définition des compilateurs, des traitements de texte (correcteur orthographique et grammatical), la compréhension automatique des langues naturelles, etc. Plus récemment, la théorie des langages est aussi très utilisée pour les recherches de caractères dans un texte et a dans ce cadre de nombreuses applications en bio-informatique, entre autres dans l'analyse de séquences qui consiste à reconnaître des motifs particuliers (i.e. des mots construits sur l'alphabet $\{A, T, G, C\}$) dans une protéine à partir de la séquence de celle-ci.

Dans le cadre de la modélisation des programmes et des systèmes informatiques, les automates sont utilisés pour modéliser et raisonner sur le comportement de ces derniers. Ainsi, l'automate représente une modélisation formelle du comportement du programme dont le langage reconnu correspond à l'ensemble des exécutions possibles, appelées aussi *traces*. L'analyse du système consiste alors à vérifier certaines propriétés sur ces traces telles qu'il n'y ait pas de blocage, d'état non atteignable, ou encore l'existence d'exécutions finies et infinies. Ces propriétés s'expriment dans des logiques dites temporelles, et leur vérification s'automatise parfois. On parle alors de "Model-Checking" (le cours "Méthode de Conception Formelle" dans l'option "Ingénierie des Systèmes Informatiques et Avancés" traite de ce sujet).

Ici, nous ne traiterons pas de ces applications et extensions de la théorie des automates et des langages. Nous présenterons plutôt les fondements mathématiques et outils formels sous-jacents à l'ensemble de ces différents cas d'application.

10. Langages rationnels

10.1. Monoïde

Comme nous l'avons déjà vu dans la partie précédente, les langages, que ces derniers soient des expressions ou des instructions d'un langage de programmation, les exécutions d'un programme ou encore un problème de décision (i.e. les mots sur lesquels une machine de Turing donnée s'arrête), sont des ensembles de suites de symboles construits sur un alphabet. Ici, nous allons voir une famille de langages très utilisée en informatique du fait de leur structuration, les *langages rationnels ou réguliers*.

Les langages étant des ensembles, il existe plusieurs façons de les définir, et nous en avons vu une dans la première partie de ce document au travers des définitions inductives. Ici, nous allons étudier une autre façon plus effective (mais en même temps très proche) que les définitions inductives pour définir un langage. L'intérêt du procédé que nous allons présenter est de donner une façon simple de savoir si un mot appartient ou non au langage défini. Mais avant, comme il est d'usage en mathématique, les langages que nous allons considérer seront toujours des parties d'une structure algébrique particulière, les *monoïdes*.

Définition 48 (Monoïde).

Un **monoïde** est un ensemble M muni d'une loi interne $\cdot : M \times M \rightarrow M$ associative, et d'un élément neutre e (i.e. $\forall m \in M, e.m = m.e = m$).

Comme toute structure algébrique, les monoïdes peuvent être comparés au travers de la notion d'*homomorphisme*, c'est-à-dire des applications entre les ensembles qui préservent l'élément neutre et la loi interne.

Définition 49 (Homomorphisme).

Un **homomorphisme** entre deux monoïdes $(M_1, \cdot, e_1)$ et $(M_2, \times, e_2)$ est une application $f : M_1 \rightarrow M_2$ telle que :

- $f(e_1) = e_2$
- $\forall \alpha, \beta \in M_1, f(\alpha.\beta) = f(\alpha) \times f(\beta)$

Parmi les monoïdes, un particulièrement nous intéresse, le monoïde Σ^* des mots finis défini inductivement sur l'alphabet Σ par :

- $\varepsilon \in \Sigma^*$
- si $\alpha, \beta \in \Sigma^*$, alors $\alpha.\beta \in \Sigma^*$

Ainsi, $(\Sigma^*, \cdot, \varepsilon)$ est bien un monoïde. La loi interne s'appelle la **concaténation**, et l'élément neutre ε le **mot vide**. Ce monoïde particulier est aussi appelé le **monoïde libre** engendré par Σ . La propriété d'être libre lui vient de la propriété suivante :

Théorème 55.

Soient Σ un ensemble, $(M, \cdot, e)$ un monoïde et $f : \Sigma \rightarrow M$ une application. Il existe un unique homomorphisme $h : \Sigma^* \rightarrow M$ tel que pour tout $a \in \Sigma$, $h(a) = f(a)$.

Démonstration : Il suffit de poser $h(e) = \varepsilon$ et $h(a_1 \dots a_n) = f(a_1) \dots f(a_n)$. h est bien un homomorphisme. L'unicité est une simple conséquence du fait que $h(a) = f(a)$ pour tout $a \in \Sigma$. ■

10.2. Langages réguliers et automates

Définitions

Un *langage* va donc être une partie de Σ^* pour un alphabet Σ donné, i.e. un élément de $\mathcal{P}(\Sigma^*)$. Il est facile de voir que l'ensemble des langages $\mathcal{P}(\Sigma^*)$ sur une même signature Σ est stable pour un certains nombre d'opérations telles que l'union, l'intersection et le passage au complémentaire. Cet ensemble est aussi stable pour le produit et l'itération où :

- **Produit de langages.** Soit $L, L' \in \mathcal{P}(\Sigma^*)$, $L.L' = \{\alpha.\beta \mid \alpha \in L, \beta \in L'\}$.
- **Itération d'un langage.** Soit $L \in \mathcal{P}(\Sigma^*)$. Notons L^* le langage obtenu à partir de L de la façon suivante :

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^1 &= L \\ L^{i+1} &= L.L^i \\ L^* &= \bigcup_{i \geq 0} L^i. \text{ On note aussi } L^+ = \bigcup_{i > 0} L^i \end{aligned}$$

On a donc :

$$L^+ = L.L^* = L^*.L \text{ et } L^* = \{\varepsilon\} \cup L^+$$

Comme on l'a vu dans la partie précédente, une machine de Turing est un moyen effectif pour engendrer un langage, le problème de décision qu'elle définit. Ici, nous allons suivre le même procédé en définissant aussi un moyen effectif pour engendrer un langage au travers de la notion **d'automate fini** qui s'abstrait des actions d'écriture et de lecture, et des têtes de lecture, pour ne garder que les symboles lus à chaque transition. Formellement, les automates finis se définissent ainsi :

Définition 50 (Automate fini).

Soit Σ un alphabet. Un **automate fini** $\mathcal{A}$ sur Σ est un tuple (Q, F, q_0, τ) où :

- Q est un ensemble fini dont les éléments sont appelés les **états** de l'automate ;
- $F \subseteq Q$ est l'ensemble des états dits **finaux** ;
- $q_0 \in Q$ est l'**état initial** ;
- $\tau \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$ est une relation dont les éléments s'appellent des **transitions**.

Dans la suite, une transition $(q, a, q') \in \tau$ sera notée $q \xrightarrow{a} q'$.

L'automate $\mathcal{A}$ est dit **déterministe** quand $\tau : Q \times \Sigma \cup \{\varepsilon\} \rightarrow Q$ est une fonction.

Quand τ est une application alors l'automate est dit **complet**.

Définition 51 (Langage reconnu par un automate).

Soit $\mathcal{A} = (Q, F, q_0, \tau)$ un automate fini sur un alphabet Σ . Le **langage reconnu** par $\mathcal{A}$ est le langage $L(\mathcal{A}) \subseteq \Sigma^*$ défini par : $\forall a_1 \dots a_n \in \Sigma^*$,

$$a_1 \dots a_n \in L(\mathcal{A}) \Leftrightarrow (\exists q_1, \dots, q_n \in Q, (\forall i, 0 \leq i \leq n-1, q_i \xrightarrow{a_i} q_{i+1}) \text{ et } q_n \in F)$$

Il est facile de compléter un automate sans changer le langage reconnu. Soit $\mathcal{A} = (Q, F, q_0, \tau)$. On définit l'automate complété $\mathcal{A}' = (Q', F', q'_0, \tau')$ comme suit :

- $Q' = Q \cup \{p\}$, $F' = F$, et $q'_0 = q_0$;
- $\tau' = \{(q, a, p) \mid \nexists q' \in Q, q \xrightarrow{a} q'\} \cup \{(p, a, p) \mid a \in \Sigma\}$

Par construction $\mathcal{A}'$ est complet. De plus, p n'étant pas un état terminal, on a $L(\mathcal{A}) = L(\mathcal{A}')$.

Dans la suite, on ne considérera que des automates complets.

Étant donné un automate $\mathcal{A}$ sur un alphabet Σ , il est facile d'écrire un algorithme qui pour un mot de Σ^* répond par l'affirmative si le mot appartient à $L(\mathcal{A})$. Ainsi, par l'exercice ci-dessus, les langages reconnus par un automate sont calculables. On appellera de tels langages des *langages réguliers*.

Définition 52 (Langages réguliers).

Soit Σ un alphabet. Un langage $L \subseteq \Sigma^*$ est dit **régulier** si, et seulement s'il existe un automate $\mathcal{A}$ sur Σ tel que $L = L(\mathcal{A})$.

Pour prouver qu'un langage est régulier, une méthode évidente consiste à construire explicitement un automate. Bien sûr, on peut imposer d'autres contraintes à l'automate comme avoir le plus petit nombre d'états ou encore être déterministe. On verra dans la suite que pour l'essentiel, ces contraintes sont faciles à obtenir. Néanmoins, pour simplifier la construction des automates, il est souvent utile de savoir que les langages réguliers sont aussi stables pour les opérations ensemblistes que nous avons définies ci-dessus.

Théorème 56.

L'ensemble des langages réguliers est stable par passage au complémentaire, intersection, union, produit et itération.

Démonstration : Nous allons donner ci-dessous, les preuves pour chacune des opérations mentionnées : nous rappelons que tous les automates considérés sont complets

- *Passage au complémentaire.* Soit $\mathcal{A} = (Q, F, q_0, \tau)$ un automate (complet) et déterministe (on verra dans la suite que tout automate peut se déterminer - ceci n'est donc pas une restriction). L'automate $\overline{\mathcal{A}}$ qui reconnaît le langage complémentaire est défini par $(Q, Q \setminus F, q_0, \tau)$. Nous avons bien que tous les mots qui ne sont pas reconnus par $\mathcal{A}$ sont maintenant reconnus par $\overline{\mathcal{A}}$.
- *Intersection.* L'idée est d'exécuter en parallèle les deux automates sur un même mot et ne reconnaître que ceux pour lesquels les deux automates aboutissent dans un état final. Ceci s'obtient très bien par le produit Cartésien d'automates. Soient $\mathcal{A}_1 = (Q_1, F_1, q_0^1, \tau_1)$ et $\mathcal{A}_2 = (Q_2, F_2, q_0^2, \tau_2)$ deux automates définis sur une même signature Σ . Le **produit Cartésien** de $\mathcal{A}_1$ et $\mathcal{A}_2$, noté $\mathcal{A}_1 \times \mathcal{A}_2$, est l'automate fini $\mathcal{A} = (Q, F, q_0, \tau)$ défini par :

- $Q = Q_1 \times Q_2$ et $F = F_1 \times F_2$;
- $q_0 = (q_0^1, q_0^2)$;
- $\forall a \in \Sigma, (q_1, q_2) \xrightarrow{a} (q'_1, q'_2) \in \tau \Leftrightarrow q_1 \xrightarrow{a} q'_1 \in \tau_1 \text{ et } q_2 \xrightarrow{a} q'_2 \in \tau_2$.

L'automate $\mathcal{A}$ reconnaît le langage $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

- *Union.* L'union de deux automates $\mathcal{A}_1 = (Q_1, F_1, q_0^1, \tau_1)$ et $\mathcal{A}_2 = (Q_2, F_2, q_0^2, \tau_2)$ est définie par l'automate $\mathcal{A}' = (Q', F', q'_0, \tau')$ avec :
 - $Q' = Q_1 \cup Q_2 \cup \{q'_0\}$ et $F' = F_1 \cup F_2$
 - $\tau' = \tau_1 \cup \tau_2 \cup \{q'_0 \xrightarrow{\varepsilon} q_0^1, q'_0 \xrightarrow{\varepsilon} q_0^2\}$

Une autre construction de l'union existe à partir du produit cartésien de $\mathcal{A}_1$ et $\mathcal{A}_2$ en posant que l'état initial est (q_0^1, q_0^2) et les états finaux sont $F_1 \times Q_2 \cup Q_1 \times F_2$. Cette construction marche parce que l'on a supposé $\mathcal{A}_1$ et $\mathcal{A}_2$ complets.

- *Produit.* Soient $\mathcal{A}_1 = (Q_1, F_1, q_0^1, \tau_1)$ et $\mathcal{A}_2 = (Q_2, F_2, q_0^2, \tau_2)$ deux automates définis sur une même signature Σ . On ajoute à Σ le mot vide ε dont on rappelle qu'il est neutre pour la concaténation. On définit alors l'automate fini $\mathcal{A} = (Q, F, q_0, \tau)$ par :

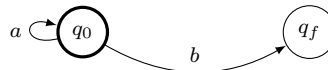
- $Q = Q_1 \cup Q_2, F = F_2$, et $q_0 = q_0^1$;
- $\tau = \tau_1 \cup \tau_2 \cup \{q_1 \xrightarrow{\varepsilon} q_0^2 \mid q_1 \in F_1\}$

L'automate $\mathcal{A}$ ainsi défini reconnaît bien exactement le langage $L(\mathcal{A}_1).L(\mathcal{A}_2)$.

- *Itération.* Soit $\mathcal{A} = (Q, F, q_0, \tau)$ un automate sur Σ . On définit l'automate $\mathcal{A}^* = (Q', F', q'_0, \tau')$ sur $\Sigma' = \Sigma \cup \{\varepsilon\}$ de la façon suivante :

- $Q' = Q \cup \{q'_0\}$ (q'_0 est un nouvel état qui n'apparaissait pas dans Q)
- $F' = F \cup \{q'_0\}$
- $\tau' = \tau \cup \{q \xrightarrow{\varepsilon} q_0 \mid q \in F'\}$

Remarquons que nous avons ajouté un nouvel état initial et pas considéré q_0 comme un nouvel état final. En effet, pourquoi ne pas imposer que q_0 soit un nouvel état final ? Si nous faisons un tel choix, nous changeons alors le langage de départ et donc on ne calculerait pas $L(\mathcal{A})^*$. En effet, supposons l'automate $\mathcal{A} = (\{q_0, q_f\}, \{q_f\}, q_0, \tau)$ défini sur l'alphabet $\{a, b\}$ et dont les transitions dans τ sont définies par :



Il est aisé de voir que le langage calculé $L(\mathcal{A}) = \{a^n.b \mid n \in \mathbb{N}\}$ où $a^0 = \varepsilon$ et $\forall n \geq 1, a^n = \underbrace{a \dots a}_{n \text{ fois}}$.

Si dans notre construction de l'automate $\mathcal{A}^*$, nous imposons que q_0 deviennent un état terminal,

nous reconnaitrions alors tous les mots a^n pour $n \in \mathbb{N}$, qui bien sûr n'appartiennent pas à $L(\mathcal{A})$ et donc ne devraient pas être reconnus par $\mathcal{A}^*$. ■

Lemme de pompage et langage réguliers

Nous allons présenter une propriété importante des langages réguliers. Elle est souvent utilisée pour démontrer qu'un langage n'est pas régulier.

Théorème 57 (Lemme de pompage).

Soit L un langage régulier dans un alphabet Σ . Il existe un entier naturel p (la longueur du pompage) tel que pour tout mot α de L de longueur au moins p , il existe trois mots $x, y, z \in \Sigma^*$ tels que :

- $y \neq \varepsilon$
- le mot $x.y$ a une longueur (son nombre de lettres) inférieure ou égale à p
- $\alpha = x.y.z$
- $\forall k \in \mathbb{N}, x.y^k.z \in L$.

Démonstration : Soit un automate $\mathcal{A} = (Q, F, q_0, \tau)$ qui reconnaît L . Posons $p = |Q|$. Soit $\alpha = a_1 \cdots a_n$ un mot de longueur au moins égale à p . Ceci signifie alors que $n \geq p$. Soit $q_1, \dots, q_n \in Q$ la séquence d'états telle que pour tout i , $0 \leq i \leq n-1$, $q_i \xrightarrow{a_i} q_{i+1}$. D'après le principe des tiroirs, il y a parmi les $p+1$ premiers états, deux endroits i, j , $0 \leq i < j \leq p$, tels que $q_i = q_j$. Posons alors $x = a_1 \dots a_i$, $y = a_{i+1} \dots a_j$ et $z = a_{j+1} \dots a_n$. Les deux premières conditions sont vérifiées ($x.y = a_1 \dots a_j$ et $j \leq p$). Vérifions la troisième. Soit $k \in \mathbb{N}$. Montrons alors que le mot $x.y^k.z$ est reconnu par l'automate $\mathcal{A}$. Il commence par lire x en suivant le chemin $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_i} q_i$. Puis, il lit y en suivant le chemin $q_i \xrightarrow{a_{i+1}} q_{i+1} \xrightarrow{a_{i+2}} \dots \xrightarrow{a_j} q_j$. Comme $q_j = q_i$, il peut recommencer le même chemin, et donc relire y , et répéter ainsi ce processus k fois. À partir de là, il peut lire z en suivant le chemin $q_j \xrightarrow{a_{j+1}} q_{j+1} \xrightarrow{a_{j+2}} \dots \xrightarrow{a_n} q_n$. L'état $q_n \in F$ puisque le mot $\alpha \in L$. ■

On peut se demander pourquoi on appelle le théorème, le lemme de pompage ? C'est simplement parce qu'il indique que chaque mot d'un langage régulier peut être *gonflé* par la répétition d'une de ses parties.

Exemple : Montrons que le langage $L = \{0^n.1^n \mid n \in \mathbb{N}\}$ n'est pas un langage régulier. Supposons le contraire. Par le théorème ci-dessus, il existe $p \in \mathbb{N}$ tel que pour tout mot $0^n.1^n \in L$, il existe x, y, z satisfaisant les conditions du théorème. Supposons alors $\alpha = 0^p.1^p$. Par la seconde condition du théorème, x et y sont forcément composés que de 0. Supposons alors que $x = 0^i$, $y = 0^j$ avec $i+j \leq p$ et $z = 0^l.1^p$ avec $l \in \mathbb{N}$ tel que $i+j+l = p$. Il est clair qu'à partir d'un certain $k \in \mathbb{N}$, $i+kj > p$ et donc le mot $x.y^k.z = 0^{i+jk}.0^l.1^p \notin L$ car $i+jk+l > p$.

Les automates finis sont donc un modèle de calcul plus faible que les machines de Turing car ils ne permettent pas de reconnaître les langages de l'exercice précédent ou de l'exemple 10.2 alors que ces derniers sont calculables. La raison est que les MTs disposent d'une mémoire, le ruban de taille infinie, qui permet de conserver un historique des calculs effectués. Une autre forme d'automates a alors été définie comme une extension des automates finis, les *automates à pile*, qui se rapprochent des MTs par le fait qu'ils manipulent une mémoire non bornée mais à laquelle on accède en suivant une politique LIFO (Last In First Out) (i.e. une pile d'où leur nom). Ce type d'automate permet de reconnaître les langages de l'exercice précédent. Néanmoins, là

encore, nous disposons d'un résultat équivalent au lemme du pompage qui montre que certains langages pourtant calculables, ne sont pas modélisables par un automate à pile. La raison est que la pile associée par sa politique LIFO, contraint la façon dont on accède aux éléments dans la pile. Nous ne présenterons pas ce type d'automates qui n'ont pas connus la même notoriété que les automates finis du fait de leur faible application (à part bien sûr la reconnaissance de certains langages) entre autres dans la modélisation des systèmes.

Expressions régulières

Les expressions régulières sont un autre moyen très simple de définir les langages réguliers.

Définition 53 (Expressions régulières).

Soit Σ un alphabet. L'ensemble des **expressions régulières**, noté $\mathcal{ER}(\Sigma)$, est défini inductivement par :

- $\emptyset, \varepsilon \in \mathcal{ER}(\Sigma)$;
- $\forall a \in \Sigma, a \in \mathcal{ER}(\Sigma)$;
- $E_1 + E_2 \in \mathcal{ER}(\Sigma)$ si $E_1, E_2 \in \mathcal{ER}(\Sigma)$;
- $E_1.E_2 \in \mathcal{ER}(\Sigma)$ si $E_1, E_2 \in \mathcal{ER}(\Sigma)$;
- $E^* \in \mathcal{ER}(\Sigma)$ si $E \in \mathcal{ER}(\Sigma)$

À partir d'une expression régulière, on peut définir le langage qu'elle caractérise.

Définition 54.

Soit Σ un alphabet et $E \in \mathcal{ER}(\Sigma)$. Le langage $L(E)$ est défini par induction sur la structure de l'expression E par :

- $L(\emptyset) = \emptyset, L(\varepsilon) = \{\varepsilon\}$, et $L(a) = \{a\}$
- $L(E_1 + E_2) = L(E_1) \cup L(E_2)$
- $L(E_1.E_2) = L(E_1).L(E_2)$
- $L(E^*) = L(E)^*$

On a alors le résultat important suivant :

Théorème 58.

Un langage est régulier si, et seulement s'il existe une expression régulière qui le décrit.

Démonstration : Soit E une expression régulière. Montrons par récurrence structurelle sur E que le langage $L(E)$ est régulier.

Il est très simple de construire un automate pour les langages $\emptyset, \{\varepsilon\}$ et $\{a\}$. Le pas de récurrence est une conséquence directe du théorème 56 qui nous assure que l'ensemble des langages réguliers est stable par union, produit et itération.

Soit $\mathcal{A} = (Q, F, q_0, \tau)$ un automate sur un alphabet Σ . Pour démontrer que le langage $L(\mathcal{A})$ peut se décrire par une expression régulière, nous allons appliquer la procédure suivante :

- On considère tous les chemins de $\mathcal{A}$ allant de l'état initial vers les états finaux de F .
- Pour chacun des chemins, on peut construire l'expression régulière en concaténant les lettres apparaissant sur les transitions. Les boucles sont traitées par l'introduction de l'opérateur d'itération $*$.
- L'expression régulière recherchée est alors la somme des expressions régulières définissant les chemins.

De façon plus rigoureuse, nous allons définir inductivement les expressions que l'on peut construire sur les chemins. Supposons que l'ensemble des états $Q = \{q_0, \dots, q_n\}$. Étant donnés deux états q_i et q_j de Q , on va construire l'ensemble des mots que l'on peut obtenir en partant de l'état q_i pour finir dans l'état q_j . On commence alors par construire l'ensemble des mots que l'on obtient directement sans passer par des états intermédiaires de q_i à q_j , puis les mots que l'on obtient en passant par q_0 , puis par q_0 et q_1 , etc. Pour définir l'ensemble de ces mots, on introduit la notation $R(i, j, k)$ comme l'**ensemble des mots** permettant de passer de l'état q_i à l'état q_j en passant par les états $\{q_0, q_1, \dots, q_{k-1}\}$, pour tout i, j, k , $0 \leq i, j \leq n$ et $0 \leq k \leq n + 1$. Définissons alors par récurrence sur k l'ensemble $R(i, j, k)$:

- $R(i, j, 0) = \begin{cases} \{a|q_i \xrightarrow{a} q_j\} & \text{si } i \neq j \\ \{\varepsilon\} \cup \{a|q_i \xrightarrow{a} q_j\} & \text{si } i = j \end{cases}$
 - Les mots dans $R(i, j, k + 1)$ qui permettent de passer de q_i à q_j sont alors :
 - tous les mots qui permettent de passer de q_i à q_j en ne passant que par des états dans $\{q_0, \dots, q_{k-1}\}$, i.e. les mots de $R(i, j, k)$;
 - tous les mots qui permettent de passer q_i à q_k , puis de q_k à q_k un certains nombre de fois, puis de q_k à q_j , i.e. $R(i, k, k).R(k, k, k)^*.R(k, j, k)$.
- Nous avons donc

$$R(i, j, k + 1) = R(i, j, k) + R(i, k, k).R(k, k, k)^*.R(k, j, k)$$

Le langage $L(\mathcal{A})$ s'exprime alors par l'expression régulière

$$E = \sum_{q_j \in F} R(0, j, n + 1)$$

■

10.3. Opérations sur les automates

Un certain nombre d'opérations peuvent être appliquées aux automates. Nous en avons déjà présenté deux en section 10.2 : la complétion et le produit Cartésien d'automates. Ici, nous allons présenter deux nouvelles opérations classiquement utilisées dans la reconnaissance des langages réguliers :

1. La détermination d'un automate.
2. La minimisation d'un automate.

Détermination d'un automate

L'intérêt de la détermination d'un automate est qu'il est plus facile de décider de l'appartenance d'un mot dans un langage L quand l'automate qui le définit est déterministe. La construction est assez simple. Étant donné un automate $\mathcal{A} = (Q, F, q_0, \tau)$ sur un alphabet Σ , nous pouvons représenter la relation de transition τ comme une application $\tau : Q \times \Sigma \rightarrow \mathcal{P}(Q)$. Il suffit alors de définir l'automate $\mathcal{A}_d = (\mathcal{P}(Q), F', \{q_0\}, \tau')$ où :

- $\tau' = \{(Q_1, a, Q_2) \mid Q_2 = \{q_2 \mid \exists q_1 \in Q_1, q_1 \xrightarrow{a} q_2 \in \tau\}\}$
- $F' = \{Q' \mid Q' \cap F \neq \emptyset\}$

Clairement, $\mathcal{A}_d$ est un automate déterministe.

Théorème 59.

$$L(\mathcal{A}) = L(\mathcal{A}_d).$$

Démonstration : La preuve est laissée en exercice. ■

Minimisation d'un automate

Soit $L \subseteq \Sigma^*$ un langage régulier. Notons $\sim_L \subseteq \Sigma^* \times \Sigma^*$ la relation binaire définie par :

$$\alpha \sim_L \beta \iff (\forall \delta \in \Sigma^*, \alpha.\delta \in L \iff \beta.\delta \in L)$$

$\sim_L$ est clairement une relation d'équivalence. De plus, pour tous mots $\alpha, \beta \in L$, si $\alpha \sim_L \beta$ alors pour tout $a \in \Sigma$, nous avons $\alpha.a \sim_L \beta.a$. En effet, si $\alpha \sim_L \beta$ alors pour tout $\delta \in \Sigma^*$, $\alpha.a.\delta \in L \iff \beta.a.\delta \in L$, et donc $\alpha.a \sim_L \beta.a$. Définissons alors l'automate $\mathcal{T} = (T, F, [\varepsilon]_{\sim_L}, \tau)$ de la façon suivante :

- $T = \Sigma^*_{/\sim_L}$
- $F = L_{/\sim_L}$
- $\tau([\alpha]_{\sim_L}, a) = [\alpha.a]_{\sim_L}$

τ est définie sous sa forme fonctionnelle car $\mathcal{T}$ est par définition déterministe (nous avons vu que si $\alpha \sim_L \beta$ alors $\alpha.a \sim_L \beta.a$). De plus, elle est une application (i.e. $\mathcal{T}$ est complet) car elle est définie pour tous les mots de T (pour les mêmes raisons que ci-dessus). Il nous faut montrer que $\mathcal{T}$ est un automate fini. Pour cela, considérons un automate $\mathcal{A}$ déterministe et complet reconnaissant L . $\mathcal{A}$ étant déterministe et complet, pour tout mot $\alpha = a_1 \dots a_n \in \Sigma^*$, il existe un unique chemin dans $\mathcal{A}$ dont les transitions sont étiquetées par $a_1 \dots a_n$, et ce chemin a la forme $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ (si $\alpha \in L$, alors $q_n \in F$). Notons alors $\kappa_\alpha = q_n$, et définissons la relation $\sim_{\mathcal{A}} \subseteq \Sigma^* \times \Sigma^*$ par :

$$\alpha \sim_{\mathcal{A}} \beta \iff \kappa_\alpha = \kappa_\beta$$

$\sim_{\mathcal{A}}$ étant fondée sur l'égalité d'états, elle est clairement une relation d'équivalence. De plus, on a :

1. Le nombre de classes d'équivalence dans $\sim_{\mathcal{A}}$ est fini. En effet, à toute classe d'équivalence $[\alpha]_{\sim_{\mathcal{A}}}$ on peut associer l'unique état κ_α , et le nombre d'états dans $\mathcal{A}$ est fini.
2. $\alpha \sim_{\mathcal{A}} \beta \implies (\forall a \in \Sigma, \alpha.a \sim_{\mathcal{A}} \beta.a)$ (une conséquence du fait que $\mathcal{A}$ est déterministe et complet).
3. $\alpha \sim_{\mathcal{A}} \beta$ et $\alpha \in L(\mathcal{A}) \iff \beta \in L(\mathcal{A})$

On a alors le résultat suivant :

Proposition 60.

Pour tout $\mathcal{A}$ tel que $L(\mathcal{A}) = L$, $\sim_{\mathcal{A}} \subseteq \sim_L$.

Démonstration : Supposons $\alpha \sim_{\mathcal{A}} \beta$, et montrons que

$$\forall \delta \in \Sigma^*, \alpha.\delta \in L \Leftrightarrow \beta.\delta \in L$$

Par le second point ci-dessus, pour tout $\delta \in \Sigma^*$, on a $\alpha.\delta \sim_{\mathcal{A}} \beta.\delta$, et donc par le troisième point, on peut conclure l'équivalence. ■

Corollaire 2.

$\mathcal{T}$ est un automate fini.

Démonstration : Par la proposition 60, $\sim_L$ est une relation d'équivalence sur Σ^* moins fine que toutes les relations $\sim_{\mathcal{A}}$. Elle comporte donc moins de classe d'équivalence. Donc, par le premier point ci-dessus, le nombre d'états dans $\mathcal{T}$ est fini. ■

Montrons alors que $\mathcal{T}$ reconnaît le langage L et de plus qu'il est minimal en nombre d'états, i.e. pour tout autre automate $\mathcal{A} = (Q, F, q_0, \tau')$ tel que $L(\mathcal{A}) = L$, on $|T| \leq |Q|$.

Théorème 61.

$$L(\mathcal{T}) = L.$$

Démonstration : Montrons que $L(\mathcal{T}) \subseteq L$. Soit $\alpha \in L(\mathcal{T})$ avec $\alpha = a_1 \dots a_n$. $\mathcal{T}$ étant déterministe et complet, il existe un unique chemin dont les transitions sont étiquetées par α , et ce chemin a la forme $[\varepsilon]_{\sim_L} \xrightarrow{a_1} [a_1]_{\sim_L} \xrightarrow{a_2} \dots \xrightarrow{a_n} [\alpha]_{\sim_L}$. Par définition, on a $[\alpha]_{\sim_L} \in L/\sim_L$. On a alors directement que $\alpha \in L$.

Montrons maintenant que $L \subseteq L(\mathcal{T})$. Soit $\alpha = a_1 \dots a_n \in L$. Par construction de $\mathcal{T}$, $[\alpha]_{\sim_L} \in F$. On peut alors construire l'unique chemin dans $\mathcal{T}$ suivant :

$$[\varepsilon]_{\sim_L} \xrightarrow{a_1} [a_1]_{\sim_L} \xrightarrow{a_2} \dots \xrightarrow{a_n} [\alpha]_{\sim_L}$$

et donc, $\alpha \in L(\mathcal{T})$. ■

Théorème 62.

$\mathcal{T}$ est minimal en nombre d'états parmi tous les automates déterministes complets qui reconnaissent L .

Démonstration : Pour tout automate $\mathcal{A} = (Q, F, q_0, \tau)$ reconnaissant le langage L , on peut construire l'automate $\mathcal{T}(\mathcal{A}) = (Q', F', [\varepsilon]_{\sim_{\mathcal{A}}}, \tau')$ de la façon suivante :

- $Q' = \Sigma^*_{/\sim_{\mathcal{A}}}$
- $F' = L_{/\sim_{\mathcal{A}}}$
- $\tau'([\alpha]_{\sim_{\mathcal{A}}}, a) = [\alpha.a]_{\sim_{\mathcal{A}}}$

En suivant les mêmes étapes que dans la preuve du théorème 61, on montre aisément que $L(\mathcal{T}(\mathcal{A})) = L(\mathcal{A}) = L$. Mais, comme à chaque classe d'équivalence $[\alpha]_{\sim_{\mathcal{A}}}$ on peut associer de façon unique l'état κ_α , on a $|Q'| \leq |Q|$. Par la proposition 60, on a aussi $|T| \leq |Q'|$, et donc $|T| \leq |Q|$. ■

11. Langages algébriques

Dans le chapitre précédent sur les langages rationnels, nous avons vu qu'il existait des langages qui n'étaient pas rationnels bien que rékursifs. Nous pouvons citer par exemple, le langage $\mathcal{L} = \{a^n.b^n \mid n \in \mathbb{N}\}$ défini sur l'alphabet $\{a, b\}$. Existe-t-il une famille de langage intermédiaire aux langages rékursifs permettant de définir de tels ensembles ? La réponse est oui, ce sont les *langages algébriques* appelés aussi *langages hors-contexte*. Au départ, ces langages ont été introduits pour étudier les langues naturelles. Après, ils ont reçu une attention plus particulièrement du monde de la recherche informatique parce que ce modèle s'est avéré adapté pour décrire la syntaxe des langages de programmation. Le terme "algébrique" vient du fait que les langages algébriques sont les solutions de systèmes polynomiaux.

11.1. Grammaire algébrique

Définitions

Une grammaire algébrique est le moyen d'engendrer un langage algébrique.

Définition 55 (Grammaire algébrique).

Une **grammaire algébrique** $\mathcal{G}$ est un tuple (T, V, v_0, R) où :

- T est un ensemble fini dont les éléments sont appelés **terminaux** ;
 - V est un ensemble fini dont les éléments sont appelés **variables** ;
 - $v_0 \in V$ est une variable appelée **axiome** ;
 - $R \subseteq V \times (T \cup V)^*$ est une relation finie dont les éléments sont appelés des **règles**.
- et tel que $T \cap V = \emptyset$ (T et V sont disjoints).

Dans la suite, tout élément $(v, \alpha) \in R$ sera noté $v \rightarrow \alpha$.

Notations 63.

Quand nous aurons plusieurs règles de la forme $v \rightarrow \alpha_1, \dots, v \rightarrow \alpha_n$, nous les rassemblerons en une seule règle $v \rightarrow \alpha_1 + \dots + \alpha_n$.

Ainsi, les règles d'une grammaire algébrique doivent être comprises comme des réécritures locales au sein de mots¹, c'est-à-dire que dans un mot donné, on recherchera le membre gauche d'une règle que l'on remplacera par le membre droit tout en conservant les préfixe et suffixe du mot de départ. On parle de *dérivation*.

1. Nous verrons dans le chapitre 23, que les grammaires algébriques sont un cas particulier de systèmes plus généraux appelés *système de réécriture*.

Définition 56 (Dérivation).

Soit $\mathcal{G} = (T, V, v_0, R)$ une grammaire algébrique. Soient $\alpha, \beta \in (T \cup V)^*$ deux mots sur l'alphabet $T \cup V$. β **dérive** de α , noté $\alpha \rightarrow \beta$, s'il existe deux mots $\alpha_1, \alpha_2 \in (T \cup V)^*$ et une règle $v \rightarrow \omega \in R$ tels que $\alpha = \alpha_1.v.\alpha_2$ et $\beta = \alpha_1.\omega.\alpha_2$.

Définition 57 (Langage algébrique).

Soit G une grammaire. Le langage engendré par G , noté $\mathcal{L}(\mathcal{G})$, est l'ensemble de mots sur T défini par :

$$\mathcal{L}(\mathcal{G}) = \{\alpha \in T^* \mid v_0 \xrightarrow{*} \alpha\}$$

Tout langage $\mathcal{L}$ défini sur un alphabet Σ est dit **algébrique** s'il existe une grammaire $\mathcal{G}$ dont Σ est l'ensemble des éléments terminaux telle que $\mathcal{L} = \mathcal{L}(\mathcal{G})$.

De la même façon, à partir de n'importe quelle variable $v \in V$, on peut aussi calculer le langage $\mathcal{L}(v) = \{\alpha \in T^* \mid v \xrightarrow{*} \alpha\}$. Dans cas-là, on a bien sûr $\mathcal{L}(\mathcal{G}) = \mathcal{L}(v_0)$.

Exemple 64.

Considérons la grammaire $\mathcal{G} = (T, V, v_0, R)$ avec :

- $T = \{a, b\}$;
- $V = \{v_0\}$;
- $R = \{v_0 \rightarrow a.v_0.b + \varepsilon\}$

Il est facile de montrer que le langage engendré par cette grammaire est $\{a^n b^n \mid n \in \mathbb{N}\}$.

Théorème 65.

Les langages réguliers sont algébriques.

Démonstration : Soit un automate fini $\mathcal{A} = (Q, F, q_0, \tau)$ sur une signature Σ . On construit alors la grammaire $\mathcal{G} = (\Sigma, Q, q_0, R)$ où $R = \{q \rightarrow aq' \mid q \xrightarrow{a} q' \in \tau\} \cup \{f \rightarrow \varepsilon \mid f \in F\}$. Il est facile de montrer que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G})$. ■

Réciproquement, on peut montrer que toute grammaire dont les règles sont de la forme $S \rightarrow aT$ ou $S \rightarrow a$ pour $a \in T$ et $S, T \in V$ engendre un langage rationnel.

Exemple 66 (Exemples remarquables).

- *Langage de Dyck*. Ce langage est important dans le cadre des langages algébriques car il existe un résultat de représentation de tels langage à partir du langage de Dyck. Intuitivement, le langage de Dyck (que l'on a déjà défini dans la section 4.2) définit les expressions bien parenthésées. Ici, nous donnons une définition plus générale de ce langage. Soit la grammaire $\mathcal{G} = (T, V, v_0, R)$ où :
 - $T = \{a_1, \dots, a_n, \bar{a}_1, \dots, \bar{a}_n\}$;
 - $V = \{v_0, v_1\}$;
 - $R = \{v_0 \rightarrow v_0 v_1 + \varepsilon, v_1 \rightarrow a_1 v_0 \bar{a}_1 + \dots + a_n v_0 \bar{a}_n\}$.

On retrouve notre langage précédent, en considérant que chaque a_i est une parenthèse ouvrante et $\bar{a}_i$ une paratenthèse fermante.

- *Langage de Lukasiewicz*. Soit la grammaire $\mathcal{G} = (T, V, v_0, R)$ où :
 - $T = \{a, \bar{a}\}$;
 - $V = \{v_0\}$;
 - $R = \{v_0 \rightarrow a v_0 v_0 + \bar{a}\}$.

Le langage de Lukasiewicz contient alors l'ensemble des mots ω tel que $|\omega|_{\bar{a}} = |\omega|_a + 1$ où $|\omega|_{\bar{a}}$ (resp. $|\omega|_a$) dénote le nombre de $\bar{a}$ (resp. a) apparaissant dans le mot ω , et pour tout préfixe propre α de ω , on a $|\alpha|_{\bar{a}} \leq |\alpha|_a$.

11.2. Formes normales de grammaires

Il existe plusieurs formes normales de grammaire (i.e. des grammaires où les règles ont des formes particulières). Elles ont l'avantage d'être plus facile à manipuler. Ici, nous allons énoncer les plus courantes, et nous montrerons que toute grammaire algébrique peut se transformer en une grammaire possédant la forme normale recherchée.

Définition 58 (Grammaire réduite).

- Une grammaire $\mathcal{G} = (T, V, v_0, R)$ est dite **réduite** pour v_0 si :
- Pour tout $v \in V$, $\mathcal{L}(v) \neq \emptyset$;
 - Pour tout $v \in V$, il existe $\alpha, \beta \in (T \cup V)^*$ tel que $v_0 \xrightarrow{*} \alpha v \beta$.

La première condition signifie que toute variable peut produire un mot, et la seconde que toute variable est atteignable à partir de l'axiome v_0 .

Théorème 67.

Pour toute grammaire $\mathcal{G} = (T, V, v_0, R)$, il existe une grammaire réduite $\mathcal{G}' = (T', V', v'_0, R')$ pour v'_0 telle que $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.

Démonstration : La preuve se fait en deux étapes : Il est à noter que l'ordre des étapes est important

1. On supprime toutes les variables $v \in V$ telles que $\mathcal{L}(v) = \emptyset$. Pour cela, soit la fonction $F : 2^{T \cup V} \rightarrow 2^{T \cup V}$ définie par $X \mapsto T + \{v \in V \mid \exists \alpha \in X^*, v \rightarrow \alpha \in R\}$. La fonction F est trivialement croissante. Elle est de plus continue pour le treillis complet $(2^{T \cup V}, \subseteq)$. Par le second théorème du calcul des points fixes, il existe un plus petit point fixe et le fait que les ensembles manipulés sont tous finis, il existe $n \in \mathbb{N}$ tel que $U = F^n(\emptyset)$. Il est alors facile de montrer que pour tout $v \in U \cap V$, on a $\mathcal{L}(v) \neq \emptyset$.
2. On supprime toutes les variables non atteignables à partir de v_0 . Pour cela, soit la fonction $F : 2^V \rightarrow 2^V$ définie par $X \mapsto \{v_0\} \cup \{v \in V \mid \exists v' \in V, v \rightarrow \alpha v' \beta \in R\}$. Là encore, il est facile de montrer que F est continue pour le treillis complet $(2^V, \subseteq)$. Par le second théorème du calcul des points fixes, il existe un plus petit point fixe et le fait que les ensembles manipulés sont tous finis, il existe $n \in \mathbb{N}$ tel que $W = F^n(\emptyset)$. Il est alors facile de montrer que pour tout $v \in W$, il existe $\alpha, \beta \in T^*$ tels que $s_0 \xrightarrow{*} \alpha v \beta$.

On pose alors $T = T$, $V' = W$, $s'_0 = s_0$, et $R' = \{v \rightarrow \alpha \in R \mid v \in V' \wedge \alpha \in (T \cup V')^*\}$. ■

Définition 59 (Grammaire propre).

Une grammaire $\mathcal{G} = (T, V, v_0, R)$ est **propre** si elle vérifie :

- pour toute règle $v \rightarrow \alpha \in R$, $\alpha \neq \varepsilon$ ou $v = v_0$;
- Il n'existe pas de règle de la forme $v \rightarrow v' \in R$ avec $v, v' \in V$.

Théorème 68.

Pour toute grammaire $\mathcal{G} = (T, V, v_0, R)$, il existe une grammaire propre $\mathcal{G}' = (T', V', v'_0, R')$ telle que $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.

Démonstration : La preuve se fait en deux étapes : là encore, l'ordre des étapes est important

1. On pose $T' = T$, $V' = V \cup \{v'_0\}$, et $R' = R \cup \{v'_0 \rightarrow v_0\}$.
2. On supprime les règles de la forme $v \rightarrow \varepsilon$ de R' si $v \neq v_0$. Soit la fonction $F : 2^V \rightarrow 2^V$ définie par $X \mapsto \{v \in V \mid v \rightarrow \varepsilon\} \cup \{v \in V \mid \exists \alpha \in X^*, v \rightarrow \alpha \in R\}$. La fonction F est trivialement croissante. Elle est de plus continue pour le treillis complet $(2^V, \subseteq)$. Par le second théorème du calcul des points fixes, il existe un plus petit point fixe et le fait que les ensembles manipulés sont tous finis, il existe $n \in \mathbb{N}$ tel que $U = F^n(\emptyset)$. Il est alors facile de montrer que pour tout $v \in U$, on a $v \xrightarrow{*} \varepsilon$. À partir de là, on supprime toutes les règles de la forme $v \rightarrow \varepsilon$ de R' , et pour tout $v \in U$, toute règle $u \rightarrow \alpha v \beta \in R$ est remplacée par la règle $u \rightarrow \alpha \beta$.
3. On supprime les règles de la forme $v \rightarrow v'$ avec $v, v' \in V$. On part de l'ensemble R' et on calcule l'ensemble $W = \{(v, v') \in V \times V \mid v \xrightarrow{*} v'\}$. Comme on a supprimé toutes les règles de la forme $v \rightarrow \varepsilon$ dans R' , on a $W = \xrightarrow{*}$ où $\rightarrow$ est restreinte aux variables. Puis, on supprime toutes les règles $v \rightarrow v'$, puis pour tout $(v, v') \in W$ et pour tout $v' \rightarrow \alpha$ avec $\alpha \notin V$, on ajoute à R' la règle $v \rightarrow \alpha$.

On vérifie sans difficulté que la grammaire est propre, et qu'elle engendre le même langage. ■

Corollaire 3.

Le problème du mot pour les langages algébriques est décidable, c'est-à-dire étant donnée une grammaire $\mathcal{G} = (T, V, v_0, R)$ et un mot $\alpha \in T^*$, on peut décider si α appartient ou non au langage $\mathcal{L}(\mathcal{G})$.

Démonstration : Soit n la taille du mot α (i.e. $n = |\alpha|$). Deux cas sont à traiter :

1. $n = 0$. Dans ce cas là, il suffit de vérifier si $v_0 \xrightarrow{*} \varepsilon$. Pour cela, on pourra utiliser l'algorithme que nous avons défini dans la preuve du théorème 68 pour calculer l'ensemble U des variables v telles que $v \xrightarrow{*} \varepsilon$, et vérifier si $v_0 \in U$.
2. si $n \geq 0$. Calculer à partir de $\mathcal{G}$ la grammaire réduite et propre $\mathcal{G}'$. Soit L_n l'ensemble des mots dans $(T \cup V)^*$ dérivables en au plus n étapes en prenant $L_0 = \{v_0\}$ et $L_{k+1} = L_k \cup \{\beta \mid \exists \alpha, \alpha \rightarrow \beta\}$. La grammaire $\mathcal{G}'$ étant propre, aucun membre droit des règles n'est réduit à une variable ou bien ε , et donc à chaque réduction, la taille du mot augmente. Le mot recherché sera rencontré (s'il appartient au langage) au plus tard dans l'ensemble L_n . ■

Définition 60 (Forme de Chomsky).

Une grammaire $\mathcal{G} = (T, V, v_0, R)$ est en **forme normale de Chomsky** si toutes les règles ont l'une des formes suivantes :

- $v \rightarrow v_1 v_2$ avec $v_1, v_2 \in V$;
- $v \rightarrow t$ avec $t \in T$.

Théorème 69.

Pour toute grammaire $\mathcal{G} = (T, V, v_0, R)$, il existe une grammaire en forme normale de Chomsky $\mathcal{G}' = (T', V', v'_0, R')$ telle que $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.

Démonstration : On suppose que $\mathcal{G}$ est une grammaire propre. On applique les étapes suivantes :

1. On pose $T' = T$, $V' = V \cup \{v_t \mid t \in T\}$, $v'_0 = v_0$ et $R' = R \cup \{v_t \rightarrow t \mid t \in T\}$;
2. On remplace dans R' toutes les règles de la forme $v \rightarrow \alpha$ par la règle $v \rightarrow \beta$ où β est obtenu à partir de α en remplaçant tous les terminaux t par la variable v_t ;
3. On remplace dans R' toutes les règles de la forme $v \rightarrow v_1 \dots v_n$ par les n règles suivantes :
 - $v \rightarrow v_1 v'_2$;
 - $v'_i \rightarrow v_i v'_{i+1}$ pour tout i , $2 \leq i \leq n-2$;
 - $v'_{n-1} \rightarrow v_{n-1} v_n$.
 On aura ajouté au préalable les $n-2$ variables $v'_2, \dots, v'_{n-1}$ à V' . ■

Il existe une dernière forme normale, la *forme normale de Greibach*.

Définition 61 (Forme normal de Greibach).

Une grammaire $\mathcal{G} = (T, V, v_0, R)$ est en **forme normale de Greibach** si toutes les règles ont la forme $v \rightarrow \alpha$ avec $\alpha \in TV^*$. Si de plus chaque α appartient $T + TV + TV^2$, la grammaire est en **forme normale de Greibach quadratique**.

Il existe aussi un théorème qui montre que toute grammaire peut être transformée en une grammaire en forme normale de Greibach et en forme normale de Greibach quadratique. La preuve n'est pas triviale et donc nous ne la donnerons pas dans ce document.

Algorithme CYK

L'algorithme donné dans la preuve du corollaire 3 pour décider du problème du mot n'est pas efficace. Sa complexité est en $O(n^r)$ où r est le nombre de règles. Il existe un meilleur algorithme, l'algorithme de Cocke-Younger-Kasami (CYK) qui repose sur de la programmation dynamique. Avant de donner le comportement de cet algorithme, introduisons la notation suivante :

Notations 70.

Soit α le mot $a_1 \dots a_n$. On note $\alpha[i, j]$ le facteur $a_i \dots a_j$ de α .

L'algorithme se décrit par les étapes suivantes :

Entrée. Soit une grammaire $\mathcal{G}$ en forme normale de Chomsky, et un mot α de taille n

Sortie. Vrai si $\alpha \in \mathcal{L}(\mathcal{G})$

- pour tout $0 \leq i \leq j \leq n$, calculer l'ensemble $E_{i,j}$ des variables v telles que $\alpha[i, j] \in \mathcal{L}(v)$. Ces ensembles se calculent par récurrence de la façon suivante :
 - $\forall i \leq n, E_{i,i} = \{v \in V \mid v \rightarrow a_i\}$
 - $\forall i < j \leq n, E_{i,j} = \{v \in V \mid \exists i \leq k \leq n, \exists v \rightarrow v_1 v_2 \in R, \alpha[i, k] \in \mathcal{L}(v_1) \text{ et } \alpha[k+1, j] \in \mathcal{L}(v_2)\}$
- Il suffit de vérifier que $v_0 \in E_{1,n}$.

Chaque ensemble $E_{i,j}$ se calcule à partir des ensembles $E_{i,k}$ et $E_{k+1,j}$, et donc leur calcul est proportionnel à la différence $j - i$, ce qui amène à un temps global cubique de l'algorithme.

11.3. Arbre de dérivation et itération**Définition 62** (Arbre de dérivation).

Soit $\mathcal{G} = (T, V, v_0, R)$ une grammaire. Un **arbre de dérivation** pour $\mathcal{G}$ est arbre fini dont les noeuds et les feuilles sont étiquetés par des éléments de $T \cup V \cup \{\varepsilon\}$ et tel que si v est l'étiquette d'un noeud interne et $a_1, \dots, a_n$ sont les étiquettes de ses fils, alors il existe une règle $v \rightarrow a_1 \dots a_n \in R$. Un Arbre de dérivation est dit **complet** (respectivement **partiel**) si le mot obtenu par les étiquettes des feuilles de l'arbre de la gauche vers la droite ne contient pas de variables (respectivement contient au moins une variable).

Les arbres de dérivation permettent de définir la notion d'ambiguïté d'une grammaire.

Définition 63 (Ambiguïté).

Une grammaire $\mathcal{G} = (T, V, v_0, R)$ est **ambigüe** s'il existe un mot ayant au moins de deux arbres de dérivation distincts étiqueté à la racine par l'axiome v_0 .

Exemple 71.

La grammaire $v_0 \rightarrow v_0 v_0 + a$ est ambigüe car le mot aaa a deux arbres de dérivations. À l'inverse la grammaire $v_0 \rightarrow av_0 + a$ ne l'est pas.

Les arbres de dérivation permettent de formaliser un lemme important analogue au lemme de pompage pour les langages rationnels, le lemme d'itération. Ce dernier comme pour le lemme de pompage permettra de montrer les limites de formalisation des langages algébriques.

Théorème 72 (Lemme d'itération).

Soit $\mathcal{G} = (T, V, v_0, R)$ une grammaire. Il existe un entier $n \in \mathbb{N}$ tel que pour tout mot $\alpha \in \mathcal{L}(\mathcal{G})$ de longueur au moins n (i.e. $|\alpha| \geq n$), il existe $\beta, \gamma, \delta, x, y \in T^*$ tels que :

1. $\alpha = \beta x \gamma y \delta$;
2. $|xy| > 0$;
3. $|x \gamma y| < n$;
4. $\forall i \geq 0, \beta x^i \gamma y^i \delta \in \mathcal{L}(\mathcal{G})$.

Pour avoir un énoncé purement grammatical, la condition 4. peut être remplacée par :
 $\exists v \in V, v_0 \xrightarrow{*} \beta v \delta, v \xrightarrow{*} x v y, v \xrightarrow{*} \gamma$.

Démonstration : On suppose que $\mathcal{G}$ est une grammaire propre. Soit $k = |V|$ et $m \in \mathbb{N}$ la longueur maximale des membres droits des règles de $\mathcal{G}$. Posons alors $n = m^{k+1}$. Soit $\alpha \in \mathcal{L}(\mathcal{G})$. Soit $\mathcal{A}$ un arbre de dérivation de α . Par hypothèse, chaque noeud interne a au plus m fils. Il est facile de montrer par récurrence le résultat suivant :

Lemme 73.

Dans un arbre de hauteur k avec au plus m fils par noeud interne, le nombre de feuilles est au plus m^k .

Ainsi, si la taille de α est au moins n , alors la hauteur de $\mathcal{A}$ est au moins $k + 1$. Il existe donc un chemin de la racine v_0 vers une feuille $t \in T$ qui contient au moins deux noeuds internes étiquetés par la même variable. Considérons un tel chemin de longueur maximale. Notons les sommets qui le composent de la feuille vers la racine $(f, t_0, t_1, \dots, t_k, \dots, r)$ où $f \in T$ est la feuille, $t_0 \in V$ son noeud parent, etc, et $r = v_0$. Comme il n'y a que k variables, nécessairement il existe $i, j \in \mathbb{N}$ avec $0 \leq i < j \leq k$ tels que $t_i = t_j$. Notons cette variable $v = t_i = t_j$.

Soient $\mathcal{A}_i$ et $\mathcal{A}_j$ les sous-arbres de dérivation de racine v . Soit γ le mot de l'arbre $\mathcal{A}_i$ et γ' celui de

$\mathcal{A}_j$. Comme le chemin que nous avons choisi est de longueur maximale, la hauteur de $\mathcal{A}_j$ est au plus k , et donc par le lemme ci-dessus la taille de γ' est au plus n . De plus, par l'hypothèse que $i < j$, $\mathcal{A}_i$ est un sous-arbre strict de $\mathcal{A}_j$, et donc γ' se factorise en $x\gamma y$ ou de façon équivalente $v \xrightarrow{*} x\gamma y$. De la même manière, en supprimant $\mathcal{A}_j$ de $\mathcal{A}$, α se factorise $\beta\gamma'\delta$, et donc en $\beta x\gamma y\delta$, et on obtient les dérivations $v_0 \xrightarrow{*} \beta v\delta$, $v \xrightarrow{*} xvy$, et $v \xrightarrow{*} \gamma$. ■

Comme pour les langages rationnels, ce résultat d'itération permet de montrer qu'un langage n'est pas agébrique.

Proposition 74.

Le langage $\mathcal{L} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ sur l'alphabet $\{a, b, c\}$ n'est pas agébrique.

Démonstration : Supposons qu'il le soit. Il existe alors un entier $n \in \mathbb{N}$ tel que tout mot $a^m b^m c^m \in \mathcal{L}$ avec $3m \geq n$, se factorise en $\beta x\gamma y\delta$ et tel que $\beta x^i \gamma y^i \delta \in \mathcal{L}$ pour tout $i \geq 0$. Chacun des mots x et y ne contient qu'une seule lettre a , b ou c sinon le mot $\beta x^i \gamma y^i \delta \notin a^* b^* c^*$. Il en découle que $\beta x^i \gamma y^i \delta \notin \mathcal{L}$ car le nombre de a , b et c ne sont plus égaux. ■

Comme pour les langages rationnels, ce lemme d'itération n'est qu'une condition nécessaire mais pas suffisante pour les langages algébriques. Il existe des langages qui satisfont les conditions du lemme d'itération mais qui pourtant ne sont pas des langages algébriques. Par exemple, le langage $\{a^n b^m \mid n \neq m\} \cup \{a^n b^n \mid n \text{ non premier}\}$ n'est pas algébrique et pourtant il vérifie les conditions du lemme d'itération.

Le lemme d'itération permet aussi de montrer que des langages sont inhéremment ambigus, c'est-à-dire que toutes les grammaires qui les engendrent sont ambiguës.

Proposition 75.

Le langage $\mathcal{L} = \{a^m b^m c^n \mid n, m \geq 0\} \cup \{a^m b^n c^n \mid n, m \geq 0\}$ sur l'alphabet $\{a, b, c\}$ est inhéremment ambigu.

Démonstration : Les deux langages $\{a^m b^m c^n \mid n, m \geq 0\}$ et $\{a^m b^n c^n \mid n, m \geq 0\}$ sont algébriques. En effet, on peut leur associer les grammaires suivantes :

- Pour le langage $\{a^m b^m c^n \mid n, m \geq 0\}$: $v_0 \rightarrow v_0 c + v_1$ et $v_1 \rightarrow a v_2 b + \varepsilon$;
- Pour le langage $\{a^m b^n c^n \mid n, m \geq 0\}$: $v_0 \rightarrow a v_0 + v_1$ et $v_1 \rightarrow b v_1 c + \varepsilon$;

Dans la section suivante, on montrera que les langages algébriques sont fermés par union de langages. Soit le mot $a^n b^n c^n$ un mot de $\mathcal{L}$. Soit n l'entier associé au langage $\mathcal{L}$, et appliquons le lemme d'itération au mot $a^n b^n c^{n+n!}$. D'après le lemme d'itération, il existe une dérivation $v_0 \xrightarrow{*} \beta v\delta \xrightarrow{*} \beta xvy\delta \xrightarrow{*} \beta x\gamma y\delta$ tel que $\beta x\gamma y\delta = a^n b^n c^{n+n!}$. Les conditions du lemme d'itération impliquent que $x = a^i$ et $y = b^i$ pour $i \leq k$. En appliquant $n!/i$ fois la dérivation $v \xrightarrow{*} xvy$, on obtient le mot $a^{n+n!} b^{n+n!} c^{n+n!}$.

On peut appliquer le même procédé à partir du mot $a^{n+n!} b^n c^n$ pour obtenir le mot $a^{n+n!} b^{n+n!} c^{n+n!}$ mais à partir de dérivations différentes. En effet, dans le 1er cas on a commencé par engendrer le mot $a^n b^n c^{n+n!}$ puis appliquer les règles ne contenant que des a et des b en partie droite pour engendrer les parties manquantes pour engendrer $a^{n+n!} b^{n+n!} c^{n+n!}$. À l'inverse, dans le second cas, on a d'abord engendré le mot $a^{n+n!} b^n c^n$ puis appliqué les règles ne contenant que des b et des c en partie droite pour engendrer les parties manquantes pour obtenir $a^{n+n!} b^{n+n!} c^{n+n!}$. Les arbres de dérivation correspondants sont alors différents. ■

11.4. Propriétés de clôture

On a déjà vu que les langages rationnels sont algébriques. Qu'en est-il des opérations usuelles rationnelles (union, produit, itération, intersection, complémentaire) ? Sont-elles préservées par les langages algébriques ? Certaines oui (produit, itération, union) et d'autres non (intersection, complémentaire). C'est ce que nous allons montrer dans cette section.

Proposition 76.

Les langages algébriques sont clos par union, produit et itération.

Démonstration : Soit $\mathcal{G} = (T, V, v_0, R)$ et $\mathcal{G}' = (T, V', v'_0, R')$ deux grammaires. On suppose sans perte de généralité que $V \cap V' = \emptyset$.

- *Union.* Il suffit de considérer la grammaire $\mathcal{U} = (T, V_u, u_0, R_u)$ où $V_u = V \cup V' \cup \{u_0\}$, et $R_u = R \cup R' \cup \{u_0 \rightarrow v_0 + v'_0\}$.
- *Produit.* Il suffit de considérer la grammaire $\mathcal{P} = (T, V_p, p_0, R_p)$ où $V_p = V \cup V' \cup \{p_0\}$, et $R_p = R \cup R' \cup \{p_0 \rightarrow v_0.v'_0\}$.
- *Itération.* Il suffit de considérer la grammaire $\mathcal{I} = (T, V_i, i_0, R_i)$ où $V_i = V \cup \{i_0\}$, et $R_i = R \cup \{i_0 \rightarrow v_0.i_0 + \varepsilon\}$ ■

Proposition 77.

Les langages algébriques ne sont pas clos par intersection et complémentaire.

Démonstration : Pour l'intersection, nous avons le contre-exemple suivant : le langage $\{a^n b^n c^n \mid n \geq 0\}$ non algébrique (voir la proposition 74) est l'intersection des langages $\{a^m b^m c^n \mid n, m \geq 0\}$ et $\{a^m b^n c^n \mid n, m \geq 0\}$ dont on a vu qu'ils étaient algébriques (voir proposition 75).

Maintenant, puisque la classe des langages algébriques est close par union, elle ne peut pas être close par complémentarité sinon elle serait close par intersection. ■

À l'inverse, pour l'intersection, nous avons le résultat suivant :

Proposition 78.

L'intersection d'un langage algébrique avec un langage rationnel est algébrique.

Démonstration : Soit $\mathcal{G} = (T, V, v_0, R)$ une grammaire, et soit $\mathcal{A} = (Q, F, q_0, \tau)$ un automate fini. Sans perte de généralité, on suppose que la grammaire $\mathcal{G}$ est en forme normale de Chomsky. Le langage $\mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$ est engendré par la grammaire $\mathcal{G}' = (T, V', v'_0, R')$ où :

- $V' = \{v_{pq} \mid v \in V \text{ et } p, q \in Q\} \cup \{v'_0\}$;
- $R' = \begin{cases} \{v'_0 \rightarrow v_{p_0 f} \mid v_{q_0 f} \in V' \text{ et } f \in F\} \\ \cup \\ \{v_{pq} \rightarrow a \mid v \rightarrow a \in R \text{ et } p \xrightarrow{a} q\} \\ \cup \\ \{v_{pq} \rightarrow x_{pr} y_{rq} \mid r \in Q \text{ et } v \rightarrow xy \in R\} \end{cases}$

Il nous reste alors à montrer que $\mathcal{L}(\mathcal{G}') = \mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$.

- $\subseteq$. À partir de tout arbre de dérivation $\mathcal{B}$ obtenu avec $\mathcal{G}'$, on peut construire l'arbre de dérivation $\mathcal{A}'$ pour $\mathcal{G}$ en transformant toute sous-partie de l'arbre $\mathcal{B}$ de la forme
 - $x_{pr} \leftarrow v_{pq} \rightarrow y_{rq}$ en $x \leftarrow v \rightarrow y$,
 - $v_{pq} \rightarrow a$ en $v \rightarrow a$,
 et en remplaçant la branche de départ $v'_0 \rightarrow v_{p_0f}$ par le noeud v_0 .
 De la même manière, on peut à partir de $\mathcal{B}$ générer un chemin dans l'automate $\mathcal{A}$ de la façon suivante : On lit les dernières branches de l'arbre $\mathcal{B}$ qui sont de la forme $v_{pq} \rightarrow a$ en allant de la gauche vers la droite, et on pour chacune on écrit $p \xrightarrow{a} q$. Par construction de la grammaire $\mathcal{G}'$, la branche à coté de $v_{pq} \rightarrow a$ est forcément de la forme $v_{qr} \rightarrow b$. On peut donc construire une séquence $p_0 \xrightarrow{a_0} p_1 \dots p_n \xrightarrow{a_n} f$ où $a_0 \dots a_n$ est un mot de $\mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$.
- $\supseteq$. Soit un mot $a_0 \dots a_n \in \mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$. Il existe donc à la fois un arbre de dérivation $\mathcal{B}$ dont les feuilles sont étiquetées de la gauche vers la droite par les lettres $a_0, \dots, a_n$, et une séquence $p_0 \xrightarrow{a_0} p_1 \dots p_n \xrightarrow{a_n} f$ où f est un état final. À partir de ces deux éléments, on peut construire un arbre de dérivation $\mathcal{B}'$ pour la grammaire $\mathcal{G}'$ de la façon suivante : On part de la branche la plus profonde et la plus à gauche dans l'arbre $\mathcal{B}$. Cette dernière est de la forme $v^0 \rightarrow a_0$. On la transforme en $v_{p_0p_1}^0 \rightarrow a_0$. On procède de la même façon sur la branche la plus profonde qui suit, et ce jusqu'à la plus à droite dans l'arbre $\mathcal{B}$. Ensuite, on va au niveau juste au-dessus de ces branches dans l'arbre $\mathcal{B}$. On a alors pour le noeud le plus à gauche une partie de l'arbre de la forme $v^0 \leftarrow v \leftarrow v^1$ (on rappelle que l'on a supposé la grammaire $\mathcal{G}$ en forme de Chomsky). On remplace cette partie pour construire $\mathcal{B}'$ par la partie $v_{p_0p_1}^0 \leftarrow v_{p_0p_2} \rightarrow v_{p_1p_2}^1$. On effectue cette transformation jusqu'au noeud le plus à droite. Puis, on effectue le même traitement sur le niveau juste supérieur, et ce jusqu'à la racine de l'arbre $\mathcal{B}$. Par construction, l'arbre $\mathcal{B}'$ a bien toutes ses feuilles prises de la gauche vers la droite étiquetées par respectivement $a_0, \dots, a_n$. ■

11.5. Automates à pile

Les automates à pile sont des extensions des automates finis dans le sens où ils se définissent par des ensembles finis d'états et de transitions étiquetées, à ceci près qu'ils possèdent en plus une mémoire auxiliaire gérée comme une pile.

Définition 64 (Automate à pile).

Soit Σ et Δ deux ensembles représentant respectivement l'alphabet d'entrée de l'automate et l'alphabet de la pile. Un **automate à pile** $\mathcal{A}$ sur Σ et Δ est un tuple (Q, F, q_0, p_0, τ) où :

- Q est un ensemble fini d'états ;
- $F \subseteq Q$ est l'ensemble des états terminaux ;
- $q_0 \in Q$ est l'état initial ;
- $p_0 \in \Delta$ est un symbole initial, appelé aussi symbole de fond de pile ;
- $\tau \subseteq Q \times \Delta \times \Sigma \cup \{\varepsilon\} \times Q \times \Delta^*$ est l'ensemble des transitions.

Dans la suite, une transition (q, p, a, q', α) sera noté $q, p \xrightarrow{a} q', \alpha$.
 L'automate $\mathcal{A}$ est dit **déterministe** quand $\tau : Q \times \Delta \times \Sigma \cup \{\varepsilon\} \rightarrow Q \times \Delta^*$ est une fonction. Il est dit **complet** si τ est une application.

Graphiquement, les états seront représentés comme précédemment pour les automates finis par des sommets, et les transitions par une flèche d'un état vers un autre état. Ce qui va changer avec les automates finis est l'étiquetage des transitions. Ainsi, une transition (q, p, a, q', α) sera graphiquement représentée par

$$q \xrightarrow{a/(p,\alpha)} q'$$

L'étiquette contient deux parties : la première est le symbole lu, et la seconde est formé du couple p, α .

À la différence des automates finis où les mots acceptés sont tous les mots qui à partir de l'état initial aboutissent à un état final en suivant les transitions de l'automate, ici, plusieurs mode d'acceptation sont possibles. Avant de donner ces différents modes d'acceptation, définissons la notion de configuration.

Définition 65 (Configuration et calcul).

Soit $\mathcal{A} = (Q, F, q_0, p_0, \tau)$ un automate à pile. Une **configuration** C est un élément de $Q \times \Delta^*$.

Une **étape de calcul** est une paire de configurations (C, C') , notée $C \xrightarrow{a} C'$ telle que $C = (q, p\beta)$, $C' = (q', \alpha\beta)$ et $q, p \xrightarrow{a} q', \alpha \in \tau$.

Un **calcul** de $\mathcal{A}$ est une suite d'étapes de calcul $C_0 \xrightarrow{a_0} C_1 \dots C_n \xrightarrow{a_n} C_{n+1}$, et le mot calculé $a_0 \dots a_n$ est l'étiquette du calcul.

Les différents modes d'acceptation sont alors les suivants :

- **par pile vide.** une configuration acceptante est une configuration de la forme (q, ε) où la pile est vide ;
- **par état final.** une configuration acceptante est une configuration de la forme (q_f, α) où $q_f \in F$;
- **par sommet de la pile.** une configuration acceptante est une configuration de la forme $(q, p_f\alpha)$ où $p_f \in \Delta_f$ avec $\Delta_f \subseteq \Delta$;
- **par combinaison.** toute combinaison des trois premiers.

Notons qu'il est toujours possible de supposer que le symbole de fond de pile se trouve en permanence en fond de pile (sauf éventuellement tout à la fin en cas de reconnaissance par pile vide) et jamais ailleurs. En effet, étant donné un automate à pile $\mathcal{A} = (Q, F, q_0, p_0, \tau)$, on peut construire l'automate $\mathcal{A}' = (Q \cup \{q'_0\}, F, q'_0, p'_0, \tau')$ où τ' contient les transitions :

- $q'_0, p'_0 \xrightarrow{\varepsilon} q_0, p_0 p'_0 \in \tau'$;
- pour toute transition $q, p \xrightarrow{a} q', \alpha \in \tau$, $q, p \xrightarrow{a} q', \alpha \in \tau'$;
- si l'acceptation est par pile vide, alors pour tout $q \in Q$, $q, p'_0 \xrightarrow{\varepsilon} q, \varepsilon$.

Proposition 79.

Tous ces modes d'acceptation sont équivalents.

Démonstration : Nous allons uniquement montrer l'équivalence entre le mode d'acceptation par pile vide et par état final. Les autres preuves sont similaires. Par la construction que nous avons donnée précédemment, on peut supposer sans perte de généralité que le symbole p_0 est toujours en fond de pile et donc que toute transition qui vide la pile est de la forme $q, p_0 \xrightarrow{a} q', \varepsilon$.

Soit un automate à pile $\mathcal{A}$ qui accepte par pile vide. On ajoute alors à l'ensemble des états, l'état q_f qui devient le seul état final du nouvel automate, et l'on remplace toute transition de la forme

$q, p \xrightarrow{a} q', \varepsilon$ par la transition $q, p \xrightarrow{a} q_f, \varepsilon$.

Soit un automate à pile $\mathcal{A}$ qui accepte par état final. On ajoute deux états q_+ et q_- , puis on effectue les ajouts et remplacements de règles suivants :

- Pour vider la pile on ajoute pour tout $p \in \Delta$, la règle $q_+, p \xrightarrow{\varepsilon} q_+, \varepsilon$;
- pour chaque transition $q, p \xrightarrow{a} q_f, \alpha$ où $q_f \in F$, on ajoute la règle $q, p \xrightarrow{a} q_+, \varepsilon$;
- chaque transition $q, p_0 \xrightarrow{a} q, \varepsilon$ où $q' \notin F$ (i.e. on vide la pile sans atteindre un état final), est remplacée par $q, p_0 \xrightarrow{a} q_-, p_0$.

Définition 66 (Langage reconnu).

Soit $\mathcal{A} = (Q, F, q_0, p_0, \tau)$ un automate à pile. Le **langage reconnu** par $\mathcal{A}$ est le langage $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ où pour tout mot $a_0 \dots a_n$ de $\mathcal{L}(\mathcal{A})$, il existe un calcul $C_0 \xrightarrow{a_0} C_1 \dots C_n \xrightarrow{a_n} C_{n+1}$ tel que $C_0 = (q_0, p_0)$ et C_{n+1} est une configuration acceptante.

Théorème 80.

Un langage $\mathcal{L}$ est algébrique si, et seulement s'il existe un automate à pile qui reconnaît $\mathcal{L}$.

Démonstration : Soit $\mathcal{L}$ un langage algébrique. Ceci signifie qu'il existe une grammaire $\mathcal{G} = (T, V, v_0, R)$ telle que $\mathcal{L}(\mathcal{G}) = \mathcal{L}$. On suppose ici que la grammaire est en forme normale de Greibach ce qui permet de ne pas utiliser de ε -transition. Définissons à partir de $\mathcal{G}$ l'automate à pile $\mathcal{A} = (\{q\}, \emptyset, q, v_0, \tau)$ sur les alphabets T et V , de la façon suivante :

- à toute règle de la forme $v \rightarrow \varepsilon$, on fait correspondre la transition $q, v \xrightarrow{\varepsilon} q, \varepsilon$;
- à toute règle de la forme $v \rightarrow a\alpha$, on fait correspondre la transition $q, v \xrightarrow{a} q, \alpha$;

On suppose alors que le mode d'acceptation est par pile vide. Dans ce cas-là, par une récurrence sur la taille des dérivations que les dérivations dans $\mathcal{G}$ correspondent exactement aux calculs dans $\mathcal{A}$.

On peut aussi donner un algorithme de construction de l'automate à pile, en supposant que les règles sont de la forme $v \rightarrow \alpha$ avec $\alpha \in V^*$ ou bien de la forme $v \rightarrow a$ avec $a \in T$. Il est facile de transformer toute grammaire $\mathcal{G}$ en une grammaire satisfaisant une telle condition. Il suffit pour cela d'introduire une nouvelle variable v_a pour tout $a \in T$. Cette transformation est identique au point 2. du théorème 69. On construit l'automate à pile de la façon suivante :

- à toute règle de la forme $v \rightarrow a$, on fait correspondre la transition $q, v \xrightarrow{a} q, \varepsilon$;
- à toute règle de la forme $v \rightarrow \alpha$, on fait correspondre la transition $q, v \xrightarrow{\varepsilon} q, \alpha$;

Montrons la réciproque. Soit $\mathcal{A}$ un automate à pile dont le mode d'acceptation est à pile vide. On peut supposer de plus que le symbole de fond de pile se trouve en permanence en fond de pile, et donc que toutes les transitions qui vident la pile aboutissent toujours sur le même état q_f (i.e. toutes les transitions qui vident la pile sont de la forme $q, p_0 \xrightarrow{a} q_f, \varepsilon$). Notons $L_{q, q', p}$ le langage dont les éléments sont tous les mots α tel qu'il existe un calcul d'étiquette α de la configuration (q, p) à la configuration (q', ε) . Si un tel calcul est possible, il y a aussi un calcul de même étiquette α à partir de toute configuration $(q, v\beta)$ à la configuration (q', β) , et ce pour tout mot $\beta \in \Delta^*$. On a alors les égalités suivantes :

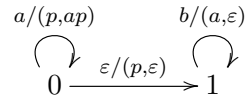
$$\mathcal{L}(\mathcal{A}) = \bigcup_{q' \in Q} L_{q_0, q', p_0}$$

$$L_{q,q',v} = \begin{cases} \{a \mid q, p \xrightarrow{a} q', \varepsilon\} \\ \cup \\ q, p \xrightarrow{a} q_1, p_1 \dots v_n \quad aL_{q_1, q_2, p_1} \dots L_{q_n, q', p_n} \\ q_2, \dots, q_{n+1} \in Q \end{cases}$$

Si l'on pose que les variables de la grammaire sont tous les triplets (q, q', p) avec (q_0, q_f, p_0) comme axiome, et que l'ensemble des terminaux est Σ , alors ceci donne naissance aux règles suivantes : $(q, q', v) \rightarrow a$ et $(q, q', v) \rightarrow a.(q_1, q_2, v_1) \dots (q_n, q', v_n)$. ■

Exemple 81.

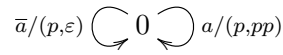
L'automate à pile ci-dessous reconnaît le langage $\{a^n b^n \mid n \in \mathbb{N}\}$.



La configuration initiale est $(0, p)$ et l'état final est 1. On reconnaît par pile vide et état final. Les calculs réussis vont de 0 à 1. Ils empilent autant de fois la lettre a dans l'état 0 qu'ils déplient des lettres a dans l'état 1. Pour passer de 0 à 1, l'automate dépile la lettre p qui est le fond de pile au début.

Exemple 82.

L'automate à pile ci-dessous reconnaît le langage de de Lucasiewicz.



La configuration initiale est $(0, p)$ et l'état final est 0. Ici, on reconnaît à la fois par pile vide et par état final, l'automate n'ayant qu'un état. La pile sert de compteur. À chaque fois que l'on lit un a , on incrémente la pile que l'on décrémente quand on lit $\bar{a}$.

Deuxième partie .

Logiques mathématiques

12. Introduction

La notion de calculabilité que nous avons présentée dans la partie ?? est la conséquence d’une réponse à D. Hilbert à certains problèmes reconnus par ce dernier comme parmi les grands problèmes des mathématiques du 20ème siècle . D. Hilbert a énoncé 23 problèmes à la conférence des mathématiciens qui a eu lieu à Paris en 1900. Parmi ces problèmes, il y en a trois qui sont à la base de la définition de la notion de calculabilité présentée dans la partie ?? :

1. Donner la preuve de l’hypothèse du continu de Cantor.
2. Montrer la consistance des axiomes de l’arithmétique de Peano.
3. Définir un algorithme déterminant si une équation diophantienne a des solutions.

Bien que les réponses apportées à ces trois problèmes furent toutes négatives,¹ leur étude a permis de définir les techniques permettant de montrer qu’un problème est décidable/mécanisable ou non. Les mathématiciens qui ont apporté les réponses à ces trois problèmes étant des logiciens, les premiers problèmes dont ils ont cherché à montrer la décidabilité sont des problèmes de logiques. En fait, ceci s’est fait en plusieurs étapes. En effet, la logique est la formalisation du raisonnement mathématique. Depuis les travaux de Frege et Peano et leur formalisation de l’arithmétique, une logique particulière semblait bien adaptée pour formaliser le raisonnement mathématique : la **logique des prédicats du premier ordre**. Naturellement, les logiciens ont alors cherché à montrer la décidabilité de cette dernière, c’est-à-dire à savoir si pour un ensemble d’axiome donné (i.e. une théorie) l’ensemble des théorèmes de cette théorie pouvaient être déduits mécaniquement. Comme ce problème a été montré indécidable en général (voir le théorème de Church dans le chapitre suivant), ils ont cherché tout d’abord les fragments de la logique des prédicats du premier ordre puis les théories exprimables dans cette logique qui sont décidables. Ceci a alors donné naissance à la **théorie de la démonstration**.

12.1. Logique mathématique et informatique

Vis-à-vis de l’informatique, la logique agit à deux niveaux :

1. **Externe et théorique**. On reproduit ici la même démarche que la logique a avec les mathématiques, c’est-à-dire l’utiliser comme un fondement de l’informatique. Dans ce cas-là, la logique est utilisée comme une méta-informatique (calculabilité, indécidabilité, classification des problèmes selon leur complexité)
2. **Interne et pratique** : l’informatique est un “terrain” où les méthodes issues de la logique sont appliquées et expérimentées. En effet, la programmation est en soi une activité logique :

1. La réponse négative apportée au premier problème a été donnée par A. Cohen en 1963 (ce qui lui a valu la médaille Fields) où il a montré que l’hypothèse du continu était un axiome et donc ne pouvait pas être prouvée à partir des autres axiomes de la théorie des ensembles. La réponse négative au second problème a été donnée par K. Gödel. C’est le fameux théorème d’incomplétude de Gödel. La consistance de l’arithmétique a été démontrée par la suite par H. Gentzen mais en ajoutant à l’arithmétique de Peano la théorie des ensembles de Zermelo-Fraenkel avec l’axiome du choix. Enfin, la réponse négative au dernier problème a été donnée par Y. Matiassevitch en 1970 à la suite de travaux de J. Robinson.

- Concevoir un programme, c'est prouver une proposition de façon constructive.
- Exécuter un programme fonctionnel, c'est normaliser une preuve (i.e. la mettre sous une forme canonique).
- Exécuter un programme logique, c'est construire une preuve.
- Optimiser ou interpréter un programme, c'est construire un modèle non standard du programme (i.e. une abstraction de ce dernier), comme s'il s'agissait d'évaluer la vérité ou la consistance d'une formule logique.
- vérifier la correction d'un programme, c'est montrer que ce dernier est un modèle d'un ensemble de propriétés définissant le cahier des charges formel du programme.

Alors que la partie ?? s'est intéressée à la logique comme fondement de l'informatique, ici, nous allons nous intéresser à la logique selon le second point ci-dessus. Mais avant d'aborder les méthodes issues de la logique, nous allons présenter le monde formel dans lequel "vivent" ces méthodes. Ce monde formel est très riche et caractérise une multitude de logiques. Ici, nous allons nous intéresser aux deux logiques à la base de toutes les autres et partagées à la fois par les mathématiques et l'informatique :

1. La logique propositionnelle, et
2. la logique des prédicats du premier ordre.

En fait, seule la logique des prédicats du premier ordre est vraiment intéressante car elle possède un pouvoir d'expression suffisant pour décrire la plupart des théories mathématiques et les systèmes informatiques², et exprimer des propriétés sur ces derniers. Cependant, l'intérêt de la logique propositionnelle est la simplicité de sa description. Elle permet donc d'aborder plus simplement la définition d'une logique. Enfin, la logique des prédicats du premier ordre est une extension de la logique propositionnelle.

12.2. Structure d'une logique

Il existe une multitude de logique (logique propositionnelle, logique des prédicats du premier ordre, logique modale, etc.) mais toutes adoptent une même structure dans leur présentation établissant une distinction claire entre la **syntaxe** (langage, formules) et la **sémantique** (l'interprétation du langage, les valeurs de vérité des formules) :

- La syntaxe a pour rôle :
 - d'explicitier les langages (**alphabet** ou **signatures**) sur lesquels seront exprimées les théories et les théorèmes de ces théories.
 - d'explicitier les règles de construction des théories et des théorèmes que l'on veut établir par ces logiques.
- La sémantique donne une dénotation mathématique à tous les éléments syntaxiques. Elle se caractérise par la définition :
 - des **modèles** qui sont une abstraction mathématique des objets du monde réel visés par notre sémantique (tels que des programmes si nous nous intéressons à la spécification formelle des systèmes informatiques).
 - de la **satisfaction** d'une propriété (formule) par un modèle, et par extension d'un théorème par une théorie. En logique, cette relation de satisfaction est toujours notée $\models$.

2. du moins les systèmes informatiques dits standards, c'est-à-dire ceux définis par des structures de données complexes mais des structures de contrôles simples.

Les logiques, surtout quand nous sommes intéressés par leur effectivité (i.e. étudier les parties décidables des logiques), sont munies d'un troisième volet, un **Calcul**. Le calcul est un ensemble de règles syntaxiques définissant des raisonnements élémentaires, permettant en combinant ces règles de montrer qu'une formule du langage est un théorème d'une théorie simplement par des manipulations syntaxiques. L'intérêt des ces manipulations syntaxiques est permettre entre autres d'étudier la décidabilité des logiques et des théories qu'elles permettent d'exprimer. Cette étude est rendue possible car seules des manipulations syntaxiques sont mises en jeu, et c'est justement ce que sait faire un ordinateur, manipuler des symboles. Le calcul caractérise alors une relation d'inférence, notée $\vdash$, à partir d'un ensemble de règles d'inférence permettant d'obtenir symboliquement des formules à partir d'autres formules supposées correctes (les axiomes de la théorie). On parle aussi de **système formel**. Les deux logiques que nous allons présenter dans la suite de ce chapitre suivront ce schéma de description, i.e. syntaxe, sémantique et calcul.

13. Systèmes formels

Post [1897-1954] a introduit la notion de systèmes formels comme support mécanique du raisonnement dans l'objectif d'être capable de déduire tous les faits dérivés d'un ensemble de faits et de règles. Il s'agit d'une classe de définitions inductives portant sur des langages définis sur un alphabet. Ainsi, les systèmes formels sont une abstraction des calculs associés à chaque logique. Nous les présentons ici afin de formaliser des notions telles que celles de déduction et de relation d'inférence $\vdash$ qui seront utilisées dans la présentation des différents calculs pour les logiques propositionnelles et du premier order qui seront présentés dans les sections qui suivent.

13.1. Définition

Définition 67.

Un *système formel*, également appelé *calcul*, est un triplet $C = (A, \mathcal{F}, \mathcal{R})$ où :

- A est un ensemble, appelé *alphabet*, dont les éléments sont appelés *symboles*,
- $\mathcal{F}$ est un sous-ensemble de A^* , dont les éléments sont appelés les *formules* bien formées,
- $\mathcal{R}$ est un ensemble fini de relations d'arité au moins égale à 1 sur les formules ; chaque relation r de $\mathcal{R}$ est appelé une *règle de déduction*, ou encore une *règle d'inférence*, et porte donc sur $\mathcal{F}^n$ où n est l'arité de r .

Pour r dans $\mathcal{R}$, on note $a(r)$ son arité.

Intuitivement, $\mathcal{A}lF$ dénote l'ensemble de tous les énoncés possibles. En pratique, il existe un moyen (algorithme) pour décider si un mot sur A est ou non une formule. Plus précisément, la plupart des ensembles de formules qui seront considérées par la suite seront définis comme des ensembles définis inductivement.

Une règle d'inférence r sert en fait à définir une étape élémentaire de preuve. Si $\varphi_1 \dots \varphi_n$ vérifient que $(\varphi_1, \dots, \varphi_n) \in r$, cela signifie intuitivement que φ_n est déduit de φ_1 et ... et φ_{n-1} . Là aussi, en pratique, une règle d'inférence n'est pas quelconque, elle est définie par un moyen algorithmique très simple pour savoir si le prédicat est vrai ou faux pour un n -uplet de formules donné.

Terminologie :

- On dit alors que $(\varphi_1, \dots, \varphi_n)$ est une *instance* de la règle r , que $\varphi_1 \dots \varphi_{n-1}$ sont les *prémisses* de cette instance, et que φ_n en est la conclusion.
- Pour des raisons beaucoup plus historiques que logiques, on distingue le cas particulier des règles d'arité 1 et on les appelle des *schémas d'axiomes*. Une instance d'un schéma d'axiomes est donc réduit à une formule qui en est sa conclusion ; on l'appelle alors un *axiome*. Un axiome n'est déduit de rien (puisque l'arité n de r est égale à 1) et est donc d'une certaine façon "admis".

Ainsi, certaines présentations des systèmes formels distinguent les axiomes des règles d'inférence et sont données sous la forme d'un quadruplet $(A, \mathcal{A}IF, Ax, \mathcal{A}IR)$ incluant la donnée Ax des schémas d'axiomes séparément des règles d'inférence.

- Si le contexte est clair, les ensembles A et $\mathcal{A}IF$ sont laissés implicites, et un système formel peut être réduit à $\mathcal{A}IR$ ou $(Ax, \mathcal{A}IR)$.

Notation : étant donnée une instance de règle $r(\varphi_1, \varphi_2, \dots, \varphi_n)$, on la note de la façon suivante :

$$\varphi_1 \quad \varphi_2 \quad \dots \quad \varphi_{n-1} \vdash_r \varphi_n$$

ou

$$\frac{\varphi_1 \quad \varphi_2 \quad \dots \quad \varphi_{n-1}}{\varphi_n} r$$

On peut même supprimer le nom de la règle r lorsque c'est évident par le contexte.

13.2. Dédutions

À partir d'un calcul, on peut construire des preuves en utilisant des instances de règles les unes après les autres de telle sorte que les prémisses utilisées à chaque étape soient toutes déjà prouvées dans les étapes précédentes.

Définition 68.

Étant donné un calcul $C = (A, \mathcal{F}, \mathcal{R})$ et un ensemble de formules $\Gamma \subseteq \mathcal{F}$, une *dédution* dans C à partir des hypothèses Γ est une séquence $\psi_1 \dots \psi_k$ de formules de $\mathcal{F}$ telle que pour tout i de 1 à k :

- ou ψ_i appartient à Γ
- (ou ψ_i est un axiome de C)
- ou encore il existe une instance $\frac{\varphi_1 \dots \varphi_{n-1}}{\varphi_n} r$ d'une règle de $\mathcal{R}$ dont la conclusion φ_n soit égale à la formule ψ_i et chacune des prémisses φ_m ($1 \leq m \leq n-1$) soit égale à l'une des formules de la preuve qui précèdent ψ_i (i.e. ψ_j avec $j < i$).

On remarque que la définition précédente ne contient en fait que deux cas. En effet, le deuxième cas n'est qu'un cas particulier du troisième, où l'arité de la règle r est 1 ; c'est pourquoi on l'a écrit entre parenthèses.

Définition 69.

Étant donné un calcul $C = (A, \mathcal{F}, \mathcal{R})$, un ensemble de formules $\Gamma \subseteq \mathcal{F}$ et une formule $\varphi \in \mathcal{F}$, on dit que Γ infère φ s'il existe une déduction dans C à partir de Γ dont la dernière formule soit égale à φ (i.e. $\psi_1 \dots \psi_k$ où ψ_k est égale à φ).

On peut alors définir une relation $\vdash_C$ sur $\mathcal{P}(\mathcal{F}) \times \mathcal{F}$, appelée *relation d'inférence* caractérisée par :

$$\Gamma \vdash_C \varphi \text{ ssi } \Gamma \text{ infère } \varphi$$

(l'indice C de $\vdash_C$ est souvent omis lorsqu'il n'y a pas d'ambiguïté sur le système formel considéré)

Proposition 83.

Soit $C = (A, \mathcal{F}, \mathcal{R})$ un calcul et $\vdash$ la relation d'inférence associée. La relation $\vdash$ possède les propriétés suivantes :

- **Réflexivité** : $\forall \varphi \in \mathcal{F}, \{\varphi\} \vdash \varphi$
À partir d'une formule, on peut toujours en déduire elle-même.
- **Transitivité** : $\forall \Gamma, \Gamma' \subseteq \mathcal{F}, \forall \varphi \in \mathcal{F}$, si $\Gamma \vdash \Gamma'$ et si $\Gamma \cup \Gamma' \vdash \varphi$ alors $\Gamma \vdash \varphi$
(avec la notation $\Gamma \vdash \Gamma'$ pour $\forall \phi \in \Gamma', \Gamma \vdash \phi$)
Cette propriété traduit l'idée de pouvoir mener des preuves en passant par des lemmes intermédiaires : pour prouver φ à partir de Γ , on commence par prouver un ensemble de lemmes Γ' , et on l'utilise pour prouver φ .
- **Monotonie** : $\forall \Gamma, \Gamma' \subseteq \mathcal{F}, \forall \varphi \in \mathcal{F}$, si $\Gamma \vdash \varphi$ et si $\Gamma \subseteq \Gamma'$ alors $\Gamma' \vdash \varphi$
Cette propriété signifie que si l'on peut mener une preuve de φ à partir d'un ensemble Γ d'hypothèses, alors on peut également (*a fortiori*) mener une preuve de φ à partir de plus d'hypothèses.
- **Compacité** : si $\Gamma \vdash \varphi$ alors il existe un sous-ensemble fini Γ_0 de Γ tel que $\Gamma_0 \vdash \varphi$

Si φ a été prouvée à partir d'un ensemble infini d'hypothèses Γ , on peut toujours en faire une preuve à partir d'un nombre fini d'hypothèses.

Démonstration : — Réflexivité : pour prouver $\{\varphi\} \vdash_C \varphi$, il suffit de considérer la séquence de formules réduite à φ (licite car φ est une hypothèse).

- Transitivité : montrons que si $\Gamma \vdash_C \Gamma'$ et si $\Gamma \cup \Gamma' \vdash \varphi$, alors $\Gamma \vdash \varphi$. De $\Gamma \cup \Gamma' \vdash \varphi$ on déduit qu'il existe une déduction $\psi_1 \dots \psi_{k-1} \varphi$ dans C à partir de $\Gamma \cup \Gamma'$ qui termine sur φ . Pour chacune des formules ψ_i , elle peut être obtenue par une règle de $\mathcal{R}$ (schémas d'axiomes compris), ou parce qu'elle appartient à $\Gamma \cup \Gamma'$. Si $\psi_i \in \Gamma'$ avec $\psi_i \notin \Gamma$, puisque $\Gamma \vdash \Gamma'$, il existe une preuve $\eta_1 \dots \eta_{m-1} \psi_i$ de ψ_i . Il suffit donc de remplacer ces ψ_i dans la séquence $\psi_1 \dots \psi_{k-1} \varphi$ par leur preuve entière $\eta_1 \dots \eta_{m-1} \psi_i$. On obtient alors une preuve φ à partir de Γ .
- Monotonie : montrons que si $\Gamma \vdash \varphi$ et si $\Gamma \subseteq \Gamma'$, alors $\Gamma' \vdash \varphi$. Soit $\psi_1, \dots, \psi_k$ une déduction de φ à partir de Γ . Alors, *a fortiori*, $\psi_1, \dots, \psi_k$ est aussi une déduction de φ à partir de Γ' .
- Compacité : on veut montrer que si $\Gamma \vdash \varphi$, alors il existe un sous-ensemble fini Γ_0 de Γ tel que $\Gamma_0 \vdash \varphi$. Soit $\psi_1, \dots, \psi_k$ une déduction de φ à partir de Γ . Considérons le sous-ensemble

$$\Gamma_0 = \{\psi_i \mid \psi_i \text{ apparait dans la déduction de } \varphi, \psi_i \in \Gamma\}$$

Γ_0 est fini par construction et $\psi_1, \dots, \psi_k$ est une déduction de φ à partir de Γ_0 . ■

13.3. Théorèmes et arbres de preuve

Définition 70.

Etant donné un calcul $C = (A, \mathcal{F}, \mathcal{R})$, l'ensemble des théorèmes de C est le sous-ensemble $Th(C)$ de $\mathcal{F}$ défini inductivement par :

- (tout axiome de C appartient à $Th(C)$)
- si toutes les prémisses d'une instance d'une règle d'inférence r de $\mathcal{R}$ appartiennent à $Th(C)$, alors sa conclusion appartient aussi à $Th(C)$.

(Ici également, le premier cas n'est qu'un cas particulier du second.)

Proposition 84.

$$\forall \varphi \in \mathcal{F}, \quad \varphi \in Th(C) \iff \emptyset \vdash_C \varphi$$

Démonstration : Montrons $\varphi \in Th(C) \implies \emptyset \vdash_C \varphi$ (autrement dit, tout théorème de C possède une déduction à partir de $\emptyset$).

Définissons la propriété P sur les formules de $\mathcal{ALF}$ par :

$$\forall \varphi \in \mathcal{ALF}, P(\varphi) \text{ ssi } \exists \psi_1, \dots, \psi_k \text{ déduction de } \varphi \text{ à partir de } \emptyset$$

Cas de base : inutile (cas particulier de l'étape d'induction)

Etape d'induction : supposons que les formules φ_j pour $j \in [1, n-1]$ vérifient la propriété P et que nous avons $r(\varphi_1, \dots, \varphi_{n-1}, \varphi_n)$ ($n \geq 1$). Posons, pour φ_j avec $j \in [1, n-1]$, $\psi_1^j, \dots, \psi_{k_j}^j$ la déduction de φ_j à partir de $\emptyset$ (on a donc $\psi_{k_j}^j = \varphi_j$). Alors

$$\psi_1^1, \dots, \psi_{k_1}^1, \psi_1^2, \dots, \psi_{k_2}^2, \dots, \psi_1^{n-1}, \dots, \psi_{k_{n-1}}^{n-1}, \varphi_n$$

est une déduction de φ_n à partir de $\emptyset$. En effet, $\psi_1^1, \dots, \psi_{k_1}^1, \dots, \psi_1^{n-1}, \dots, \psi_{k_{n-1}}^{n-1}$ est une juxtaposition de $n-1$ déductions à partir de $\emptyset$, il s'agit donc encore d'une déduction à partir de $\emptyset$. Lorsqu'on y ajoute la formule φ_n , cela constitue encore une déduction à partir de $\emptyset$, car φ_n est inféré à l'aide de la règle r appliquée aux arguments $\varphi_1, \varphi_{n-1}, \varphi_n$. ■

Définition 71.

Etant donné un calcul $C = (A, \mathcal{F}, \mathcal{R})$, l'ensemble des termes de preuves associé à C est l'ensemble des termes $T(\mathcal{ALR} \times \mathcal{ALF})$ avec pour toute règle r de $\mathcal{R}$, pour toute formule φ de $\mathcal{ALF}$, $a((r, \varphi)) = a(r) - 1$.

Pour t de la forme $(r, \varphi)(t_1, \dots, t_{n-1})$ dans $T(\mathcal{R} \times \mathcal{ALF})$, notons $Rule(t) = r$ et $Root(t) = \varphi$.

Définition 72.

Définissons $Proof(C)$ le sous-ensemble de $T(\mathcal{F}_C \times \mathcal{ALF})$ défini inductivement par :

- pour $\varphi \in \mathcal{F}$ avec r d'arité 1, $(r, \varphi) \in Proofs(C)$
- pour r d'arité $n > 1$, pour $t_1, \dots, t_{n-1}$ dans $Proofs(C)$ avec $(Root(t_1), \dots, Root(t_{n-1}), \varphi_n) \in r$, alors $(r, \varphi_n)(t_1, \dots, t_{n-1}) \in Proofs(C)$.

Les arbres de preuve sont en général dessinés avec le symbole de tête (racine de l'arbre) en bas.

$$\frac{\frac{\overline{(r_0, \varphi_0)} \quad \overline{(r_0, \varphi_1)}}{(r_1, \varphi_2)} \quad \overline{(r_0, \varphi_3)}}{(r_2, \varphi_4)}$$

La présentation ci-dessous est en général préférée :

$$\frac{\frac{\overline{\varphi_0} \ r_0 \quad \overline{\varphi_1} \ r_1}{\varphi_2} \quad \overline{\varphi_3} \ r_0}{\varphi_4} \ r_2$$

13.4. Correction et complétude des systèmes formels au regard d'une interprétation

Lorsque des systèmes formels sont définis, il est attendu un certain nombre de bonnes propriétés à leur sujet.

Propriétés syntaxiques, algorithmiques

Un système formel est décidable lorsqu'il existe une procédure de décision qui permet en un temps fini de décider si un mot quelconque du système est un théorème ou n'est pas un théorème. Un système formel qui n'est pas décidable peut être

- semi-décidable s'il existe une procédure qui permet de décider en un temps fini si une formule est un théorème, et s'il n'existe pas de procédure qui permet de décider en un temps fini si une formule n'est pas un théorème.
- indécidable s'il n'existe pas de procédures décidant si une formule est un théorème ou n'est pas un théorème.

Propriétés sémantiques

En général, les systèmes formels sont définis pour capturer tout ou partie d'une réalité. Cette réalité est en général définie en des termes mathématiques, et est appelée de façon générique en informatique *sémantique* ou *interprétation*.

Définition 73 (Correction et complétude des systèmes formels).

Etant donné un *système formel* $C = (A, \mathcal{F}, \mathcal{R})$

Soit I un sous-ensemble de $\mathcal{F}$.

C est **correct** par rapport à I ssi $Th(C) \subseteq I$.

C est **complet** par rapport à I ssi $I \subseteq Th(C)$.

I est souvent appelée *sémantique* ou *interprétation*. I est l'ensemble de référence qui capture l'ensemble des propriétés considérées comme vraies. Dans la suite, cet ensemble I sera défini via la notion de modèles et la relation de satisfaction $\models$.

En général, la correction est une propriété relativement facile à établir, car elle se prouve par induction sur l'ensemble des théorèmes : l'application d'une règle de déduction préserve par construction la sémantique.

En revanche, la complétude est une propriété plus difficile à établir, car il s'agit de montrer que toute formule de I peut être dérivée du système formel.

SOUS PARTIE 4

Logique propositionnelle

14. Introduction

La logique propositionnelle est la logique la plus simple. Toutes les autres logiques (logique des prédicats du premier ordre, logique d'ordre supérieur, logique modale, etc.) sont toutes des extensions de cette logique élémentaire. Intuitivement, les expressions manipulées par cette logique ont pour rôle de formaliser des énoncés du langage courant pouvant être vrais ou faux. À titre d'exemple, supposons la phrase suivante :

“Si le prévenu a commis le vol, c'est que ce vol a été minutieusement préparé, ou alors le prévenu avait un complice. Si le vol a été minutieusement préparé, alors, si le prévenu avait un complice, un butin plus important aurait été emporté. Or, le butin n'a pas été important. Donc, le prévenu n'a pas commis de vol.”

À la lecture de cette phrase, on peut se poser la question : Cette argumentation est-elle convaincante ? Avant de répondre à cette question, il faut tout d'abord "décortiquer" la phrase ci-dessus. On constate aisément qu'elle comporte 4 phrases :

1. Si le prévenu a commis le vol, c'est que ce vol a été minutieusement préparé, ou alors le prévenu avait un complice.
2. Si le vol a été minutieusement préparé, alors, si le prévenu avait un complice, un butin plus important aurait été emporté.
3. Or, le butin n'a pas été important.
4. Donc, le prévenu n'a pas commis de vol.

Ce que la phrase ci-dessus veut donc exprimer si l'on utilise la notation $\vdash$ pour signifier la déduction est :

$$Prop\ 1, Prop\ 2, Prop\ 3 \vdash Prop\ 4$$

Le problème est qu'à partir de l'énoncé $Prop\ 1, Prop\ 2, Prop\ 3 \vdash Prop\ 4$, nous ne pouvons rien dire de plus. Cependant, si l'on décortique chacune des 4 phrases ci-dessus, on constate qu'elles sont composées de propositions plus élémentaires combinées ensemble par des connexions de diverses formes "**ne ... pas**" ($\neg$), "**si ... alors**" ($\Rightarrow$), "**ou**" ($\vee$). Les propositions élémentaires obtenues sont :

- a. “Le prévenu a commis le vol.”
- b. “Le vol a été minutieusement préparé.”
- c. “Le prévenu avait un complice.”
- d. “Le butin est important.”

De là, on peut traduire les 4 phrases précédentes par les formules propositionnelles suivantes :

- $Prop\ 1 = a \Rightarrow ((b \wedge \neg c) \vee (c \wedge \neg b))$
- $Prop\ 2 = b \Rightarrow (c \Rightarrow d)$
- $Prop\ 3 = \neg d$
- $Prop\ 4 = \neg a$

Dans la suite de ce chapitre, nous allons formaliser les différents éléments qui définissent, c'est-à-dire la syntaxe (signature et formules), la sémantique (ensemble de modèles et relation de satisfaction $\models$) et la démonstration logique (relation d'inférence $\vdash$).

15. Logique propositionnelle : syntaxe

15.1. Formules propositionnelles

Définition 74 (Signature).

Une **signature** P est un ensemble dont les éléments sont appelés des **variables propositionnelles**.

Définition 75 (Formules propositionnelles).

Soit P une signature.

L'ensemble des **formules propositionnelles** sur P , noté $For(P)$ est le plus petit sous-ensemble de $(P \cup \{\Rightarrow, \neg, \wedge, \vee, (,)\})^*$ construit selon les règles suivantes :

1. toute variable propositionnelle est dans $For(P)$.
2. si φ et ψ sont dans $For(P)$ alors $(\varphi @ \psi)$ est dans $For(P)$ avec $@ \in \{\wedge, \vee, \Rightarrow\}$.
3. si φ est dans $For(P)$ alors $\neg\varphi$ est dans $For(P)$.

Les formules de $For(P)$ sont aussi appelées P -formules. Les symboles $\neg, \wedge, \vee$ et $\Rightarrow$ sont appelés **connecteurs logiques**.

Remarque 85.

- En fait, l'ensemble des formules propositionnelles est un ensemble de termes avec comme fonctions d'arité 0, les variables de P , comme fonction d'arité 1, l'opérateur $\neg$, aussi appelé négation, et comme fonctions binaires, les opérateurs $\wedge, \vee, \Rightarrow$, respectivement appelés conjonction, disjonction et implication. Dans la définition ci-dessus, la notation infixe a été privilégiée pour les fonctions d'arité 2, et les parenthèses ont été omises pour l'opérateur $\neg$.
- L'ensemble des formules $For(P)$ est donc par construction un ensemble défini inductivement. Autrement dit, dans la grande majorité des cas, les propriétés sur $For(P)$ seront donc établies par induction, et les fonctions avec comme ensemble de départ $For(P)$ seront définies par induction.
- Les formules propositionnelles sont parfois présentées avec le seul opérateur binaire $\Rightarrow$, ou bien au contraire, en ajoutant l'opérateur binaire $\Leftrightarrow$ (équivalence), ou encore en ajoutant les constantes $\top$ et $\perp$.
- Par abus, les parenthèses externes seront parfois omises (e.g. $p \vee q$ au lieu de $(p \vee q)$). Plus généralement, lorsque les parenthèses sont omises, les formules seront reconstituées selon les règles de précedence/priorité suivantes : $\neg$ plus prioritaire que $\wedge$, $\wedge$ plus prioritaire que $\vee$, $\vee$ plus prioritaire que $\Rightarrow$. De plus $\Rightarrow$ s'associe à droite. Ainsi, $p \Rightarrow q \Rightarrow p$ se lit $p \Rightarrow (q \Rightarrow p)$ et $\neg p \wedge q \Rightarrow r$ se lit $((\neg p) \wedge q) \Rightarrow r$

Exemple 86.

$(\neg p \wedge ((q \vee r) \Rightarrow s))$ est une formule sur $\{p, q, r, s\} \subset \mathcal{P}$

15.2. Quelques fonctions définies sur les formules propositionnelles

Comme $For(P)$ est un ensemble défini inductivement, les fonctions définies sur $For(P)$ sont en général définies en suivant un schéma d'induction structurelle :

Définition 76 (Sous-formules).

Soit φ une formule de $For(P)$.

L'ensemble $SF(\varphi)$ des sous-formules de φ est :

1. $\{\varphi\}$ si $\varphi \in P$.
2. $\{\varphi\} \cup SF(\varphi')$ si φ s'écrit $\neg\varphi'$.
3. $\{\varphi\} \cup SF(\varphi_1) \cup SF(\varphi_2)$ si φ s'écrit $(\varphi_1 @ \varphi_2)$ avec $@ \in \{\wedge, \vee, \Rightarrow\}$.

Définition 77 (Variables d'une formule).

Soit φ une formule de $For(P)$.

L'ensemble $var(\varphi)$ des variables de φ est :

1. $\{\varphi\}$ si $\varphi \in P$.
2. $var(\varphi')$ si φ s'écrit $\neg\varphi'$.
3. $var(\varphi_1) \cup var(\varphi_2)$ si φ s'écrit $(\varphi_1 @ \varphi_2)$ avec $@ \in \{\wedge, \vee, \Rightarrow\}$.

Définition 78 (Substitution p par ψ dans φ).

Soit φ et ψ deux formules de $For(P)$, et p une variable propositionnelle de P .

La substitution de p par ψ dans φ , notée $\varphi[\psi/p]$, est la formule définie inductivement comme suit :

- Si φ est p , $\varphi[\psi/p]$ est ψ
- Si φ est une variable propositionnelle q avec $p \neq q$, $\varphi[\psi/p]$ est q
- Si φ s'écrit $\neg\varphi'$, $\varphi[\psi/p]$ est $\neg\varphi'[\psi/p]$
- Si φ s'écrit $\varphi_1 @ \varphi_2$ avec $@ \in \{\vee, \wedge, \Rightarrow\}$, $\varphi[\psi/p]$ est $\varphi_1[\psi/p] @ \varphi_2[\psi/p]$

La définition de substitution peut être facilement étendue au cas de n substitutions (n variables propositionnelles p_i , chacune étant substituée par une formule ψ_i). On écrit alors $\varphi[\psi_1/p_1, \dots, \psi_n/p_n]$. Les substitutions sont appliquées en parallèle. Ce qui n'est pas le cas si l'on considère $\varphi[\psi_1/p_1][\psi_2/p_2] \dots [\psi_n/p_n]$ où p_1 est substituée par ψ_1 , puis p_2 par ψ_2 .

Une substitution σ est alors une fonction de l'ensemble des variables propositionnelles P vers $For(P)$ (en pratique $Dom(\sigma)$ est fini).

16. Sémantique

16.1. Validation des formules

Définition 79 (P -modèle).

Soit P une signature.

Un P -**modèle** (aussi appelé **valuation** ou **interprétation**) v est une application de P dans $\mathbb{B} = \{0, 1\}$.

Pour v valuation, p variable propositionnelle et b élément de $\mathbb{B}$, alors $v + [p \mapsto b]$ est la valuation v' vérifiant :

- $v'(p) = b$
- $\forall q \neq p, v'(q) = v(q)$

Pour v valuation, p variable propositionnelle et b élément de $\mathbb{B}$, alors $v + [p \mapsto b]$ (ou simplement $v[p \mapsto b]$) dénote la valuation v' vérifiant :

- $v'(p) = b$
- $\forall q \neq p, v'(q) = v(q)$

Si $P = \{p_1, \dots, p_n\}$ est un ensemble de variables fini, alors une valuation v peut s'écrire simplement $[p_1 \mapsto b_1, \dots, p_n \mapsto b_n]$ avec pour tout i , $v(p_i) = b_i$.

$\mathbb{B}$ peut être muni des applications suivantes (pour constituer ce qu'on appelle l'algèbre de Boole) :

- $\neg : \mathbb{B} \rightarrow \mathbb{B}$ définie par $\bar{1} = 0$ et $\bar{0} = 1$
- $+$: $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ définie par $x + y = 0$ ssi $x = 0$ et $y = 0$
- $\times$: $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ définie par $x \times y = 1$ ssi $x = 1$ et $y = 1$

Définition 80 (Validation de formules).

Une P -formule φ est dite **valide** (ou **vraie**) pour un P -modèle v (ou **satisfaite** par v), noté $v \models \varphi$, si $v^*(\varphi) = 1$ où l'application $v^* : For(P) \rightarrow \mathbb{B}$ est inductivement définie de la façon suivante :

- si φ est dans P , alors $v^*(\varphi) = v(\varphi)$,
- si φ est $\neg\varphi_1$ alors $v^*(\varphi) = 1$ ssi $v^*(\varphi_1) = 0$.
- si φ est $\varphi_1 \wedge \varphi_2$ alors $v^*(\varphi) = 1$ ssi $v^*(\varphi_1) \times v^*(\varphi_2) = 1$
- si φ est $\varphi_1 \vee \varphi_2$, alors $v^*(\varphi) = 1$ ssi $v^*(\varphi_1) + v^*(\varphi_2) \geq 1$,
- si φ est $\varphi_1 \Rightarrow \varphi_2$, alors $v^*(\varphi) = 1$ ssi $v^*(\varphi_1) \leq v^*(\varphi_2)$

Si $v^*(\varphi) = 0$, on note aussi $v \not\models \varphi$.

$v \models \varphi$ est aussi parfois écrit $\llbracket \varphi \rrbracket v$.

Si les seules variables propositionnelles qui apparaissent dans φ sont $p_1, p_2, \dots, p_n$ (autrement dit $\text{var}(\varphi) = \{p_1, p_2, \dots, p_n\}$, alors on note aussi simplement $\varphi(p_1, p_2, \dots, p_n)$.

Proposition 87.

Soit $\varphi(p_1, p_2, \dots, p_n)$ une formule propositionnelle.

Soient v et v' deux valuations vérifiant $\forall p \in \{p_1, p_2, \dots, p_n\}, v(p) = v'(p)$.

Alors $v^*(\varphi) = v'^*(\varphi)$.

Pour $P = \{p_1, \dots, p_n\}$, on note $[p_1 \mapsto b_1, p_2 \mapsto b_2, \dots, p_n \mapsto b_n]$ la valuation associant aux variables p_i les valeurs b_i .

Définition 81.

Soient φ et ψ deux formules propositionnelles.

- φ est dite **satisfiable** (resp. **falsifiable**) s'il existe un P -modèle v valide (resp. non valide) pour φ
(i.e. $\exists v : P \rightarrow \mathbb{B}, v \models \varphi$ (resp. $v \not\models \varphi$)).
- φ est une **tautologie** ou **valide** (resp. une **antilogie** ou encore **insatisfiable** ou **contradictoire**) si elle est (resp. n'est pas) valide pour tous les P -modèles v
(i.e. $\forall v : P \rightarrow \mathbb{B}, v \models \varphi$ (resp. $v \not\models \varphi$)).
- φ et ψ sont **équivalentes**, noté $\varphi \equiv \psi$, ssi pour tout P -modèle v , $v^*(\varphi) = v^*(\psi)$.

16.2. Quelques propriétés remarquables

Proposition 88.

Soient φ , ψ et ξ trois formules propositionnelles.

- φ est une tautologie ssi $\neg\varphi$ est une antilogie
- φ est satisfiable ssi $\neg\varphi$ est falsifiable
- Double négation

$$\neg\neg\varphi \equiv \varphi$$

- Idempotence

$$(\varphi \wedge \varphi) \equiv \varphi \quad (\varphi \vee \varphi) \equiv \varphi$$

- Lois de Morgan

$$\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi) \quad \neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$$

- Absorption

$$(\varphi \wedge (\varphi \vee \psi)) \equiv \varphi \quad (\varphi \vee (\varphi \wedge \psi)) \equiv \varphi$$

- Commutativité

$$(\varphi \vee \psi) \equiv (\psi \vee \varphi) \quad (\varphi \wedge \psi) \equiv (\psi \wedge \varphi)$$

- Associativité

$$(\varphi \vee (\psi \vee \xi)) \equiv ((\varphi \vee \psi) \vee \xi) \quad (\varphi \wedge (\psi \wedge \xi)) \equiv ((\varphi \wedge \psi) \wedge \xi)$$

- Distributivité

$$(\varphi \vee (\psi \wedge \xi)) \equiv ((\varphi \vee \psi) \wedge (\varphi \vee \xi)) \quad (\varphi \wedge (\psi \vee \xi)) \equiv ((\varphi \wedge \psi) \vee (\varphi \wedge \xi))$$

—

$$\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$$

$$\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$$

Par conventions, $\top$ (ou *true*) et $\perp$ (of *false*) sont respectivement des notations usuelles pour les formules $p \vee \neg p$ et $p \wedge \neg p$. Remarquons que pour toute valuation v , $v(\top) = 1$ et $v(\perp) = 0$. Ainsi, $\top$ est une tautologie et $\perp$ est une antologie.

De nouveaux connecteurs ou bien des connecteurs déjà existants peuvent être définis en fonction de connecteurs :

Définition 82 (Définition de connecteurs).

$$\varphi \Leftrightarrow \psi \equiv (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$$

$$\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$$

Dans la définition ci-dessus, le connecteur $\varphi \Rightarrow \psi$ est défini, via équivalence ($\equiv$) à la formule $\neg\varphi \vee \psi$

Proposition 89.

Toute formule propositionnelle (écrite avec les connecteurs $\neg, \wedge, \vee, \Rightarrow$) est équivalente à une formule construite uniquement avec les connecteurs $\neg$ et $\wedge$.
 $\{\neg, \wedge\}$ est un **système complet** de connecteurs.

Démonstration : Par induction structurelle sur la forme des formules.

Soit φ une formule. On suppose vraie la propriété sur les sous-formules de φ .

- si φ est une variables propositionnelle p , alors φ convient.
- si φ est $\neg\psi$ avec ψ' formule équivalente à ψ construite avec $\neg$ et $\wedge$, alors $\neg\psi'$ convient.
- si φ est $(\psi_1 \wedge \psi_2)$ avec ψ'_1 et ψ'_2 formules équivalentes resp. à ψ_1 et ψ_2 et construites avec $\neg$ et $\wedge$, alors $(\psi'_1 \wedge \psi'_2)$ convient.
- si φ est $(\psi_1 \vee \psi_2)$ avec ψ'_1 et ψ'_2 formules équivalentes resp. à ψ_1 et ψ_2 et construites avec $\neg$ et $\wedge$, alors $\neg(\neg\psi'_1 \wedge \neg\psi'_2)$ convient.
- si φ est $(\psi_1 \Rightarrow \psi_2)$ avec ψ'_1 et ψ'_2 formules équivalentes resp. à ψ_1 et ψ_2 et construites avec $\neg$ et $\wedge$, alors φ est équivalente à $(\neg\psi'_1 \vee \psi'_2)$, donc à $\neg(\psi'_1 \wedge \neg\psi'_2)$ (cf. point précédent). ■

Soit $P = \{p_1, p_2, \dots, p_n\}$ un ensemble fini de variables propositionnelles. Soit Val_P l'ensemble des valuations $v : P \rightarrow \{0, 1\}$ définies sur P (on a $|Val_P| = 2^n$). Toute formule propositionnelle φ définit de façon canonique une fonction $int(\varphi)$ de Val_P dans $\mathbb{B}$ par

$$\begin{aligned} int(\varphi) : Val_P &\rightarrow \{0, 1\} \\ v &\mapsto v^*(\varphi) \end{aligned}$$

Théorème 90 (Complétude fonctionnelle).

Soit $P = \{p_1, p_2, \dots, p_n\}$ un ensemble fini de variables propositionnelles. Toute fonction $f : Val_P \rightarrow \{0, 1\}$ coïncide avec la fonction interprétation $int(\varphi)$ d'une formule propositionnelle φ définie sur P à l'aide des connecteurs $\neg, \vee$ et $\wedge$.

Démonstration : Par récurrence sur le nombre de variables propositionnelles.

- Pour $n = 1$ ($P = \{p\}$), il y a quatre fonctions de $\{0, 1\}$ dans $\{0, 1\}$, qui correspondent aux fonctions interprétations des formules $p, \neg p, p \vee \neg p$ et $p \wedge \neg p$.
 Notons v_0 et v_1 les valuations qui associent respectivement les valeurs 0 et 1 à la variable p .
 Il y a quatre fonctions f_1, f_2, f_3 et f_4 de $\{v_0, v_1\}$ dans $\{0, 1\}$. Par exemple, f_1 définie par $f_1(v_0) = 1$ et $f_1(v_1) = 0$ coïncide avec $int(\neg p)$.

- Pour $n > 1$ ($P = \{p_1, \dots, p_n\}$), supposons la propriété vraie pour les $n - 1$ premières variables propositionnelles $P' = \{p_1, \dots, p_{n-1}\}$. Soit f une fonction de $\{0, 1\}^n$ dans $\{0, 1\}$. Soit v une valuation définie sur $P = \{p_1, p_2, \dots, p_n\}$. Considérons f_0 (resp. f_1) la restriction de f au $n - 1$ premières variables en posant $v(p_n) = 0$ (resp. $v(p_n) = 1$). Les fonctions f_0 et f_1 sont des fonctions définies pour les valuations définies sur P' et par hypothèse de récurrence correspondent resp. aux interprétation $int(\psi_0)$ et $int(\psi_1)$ de formules définies sur les variables $\{p_1, \dots, p_{n-1}\}$. La fonction f correspond alors à l'interprétation $int(\varphi)$ avec φ donnée par :

$$(\neg p_n \wedge \psi_0) \vee (p_n \wedge \psi_1)$$

On a bien $v^*(\varphi) = f(v)$. ■

Remarque : tout système complet de connecteurs permet de capturer toutes les fonctions de $Val_P \rightarrow \{0, 1\}$.

16.3. Conséquence sémantique

Définition 83.

Soit Γ un ensemble de formules propositionnelles et soit φ une formule propositionnelle. Un P -modèle v est un **modèle** de Γ , noté $v \models \Gamma$ si, et seulement si

$$\forall \varphi \in \Gamma, v \models \varphi$$

On dit que φ est **conséquence sémantique** de Γ , noté $\Gamma \models \varphi$ si et seulement si

$$\forall v : P \rightarrow \mathbb{B}, v \models \Gamma \implies v \models \varphi$$

16.4. Théorème de la compacité

Théorème 91.

Soit Γ un ensemble dénombrable (i.e. isomorphe à $\mathbb{N}$) de P -formules. Γ est satisfiable si, et seulement si toutes ses parties finies sont satisfiables.

Démonstration : La condition **nécessaire** est triviale.

Montrons la condition **suffisante**. Supposons alors que tous les sous-ensembles finis de Γ ont un P -modèle.

Γ étant dénombrable, l'ensemble P des variables propositionnelles apparaissant dans les formules de Γ l'est aussi. On peut donc numéroté les variables propositionnelles en $(p_i)_{i \in \mathbb{N}}$. On note $P = \{p_i \mid i \in \mathbb{N}\}$.

Supposons que tout sous-ensemble fini de Γ a un modèle. Nous définissons une application $v : P \rightarrow \{0, 1\}$ par induction sur les indices des variables de P .

Soit $(v_i)_{i \in \mathbb{N}}$ la suite de fonctions suivantes :

- v_0 est la fonction vide, définie nulle part ;

- le domaine de v_i est $\{p_j \mid 0 \leq j < i\}$.
 Soit w_i et x_i les fonctions prolongeant v_i et définies sur le domaine $\{p_j \mid 0 \leq j < i + 1\}$, telles que $w_i(p_i) = 0$ et $x_i(p_i) = 1$.
 (autrement dit $w_i = v_i + [p_i \mapsto 0]$ et $x_i = v_i + [p_i \mapsto 1]$)
 Si tout sous-ensemble fini de Γ a un modèle qui est une extension de w_i alors $v_{i+1} = w_i$ sinon $v_{i+1} = x_i$.
 Soit $v : P \rightarrow \{0, 1\}$ la fonction de domaine P (tout entier) et définie :
 pour tout $i \in \mathbb{N}$, $v(p_i) = v_{i+1}(p_i)$.

 Soit $\mathcal{Q}(i)$ la propriété telle que tout sous-ensemble fini de Γ a un modèle qui est une extension de v_i . Montrons par récurrence que cette propriété est vraie pour tout entier.
 - la propriété $\mathcal{Q}(0)$ est vraie. En effet, v_0 étant la fonction nulle part définie, elle peut être prolongée par chacun des modèles de tous les sous-ensembles vides de Γ .
 - Supposons $\mathcal{Q}(i)$ et montrons $\mathcal{Q}(i + 1)$.
 Analyse par cas selon la définition de v_{i+1}
 - $v_{i+1} = w_i$. Par définition de v_{i+1} , tout sous-ensemble fini de Γ a un modèle qui est une extension de v_{i+1} , donc $\mathcal{Q}(i + 1)$ est vérifiée.
 - $v_{i+1} = x_i$. Supposons que $\mathcal{Q}(i + 1)$ ne soit pas vérifiée. Alors,
 - par définition de v_{i+1} , il existe un sous-ensemble fini Δ de Γ , qui n'a pas de modèle en tant qu'extension de w_i .
 - puisque par hypothèse $\mathcal{Q}(i + 1)$ n'est pas vraie, il existe un sous-ensemble fini ∇ de Γ , qui n'a pas de modèle qui soit une extension de x_i . $\Delta \cup \nabla$ est un sous-ensemble fini de Γ , qui n'a pas de modèle qui soit une extension de v_i , ce qui contredit l'hypothèse $\mathcal{Q}(i)$.
 Par conséquent $\mathcal{Q}(i + 1)$ est vérifiée.
 et on en déduit : $\forall i \in \mathbb{N}$, $\mathcal{Q}(i)$.
 v est-il modèle de Γ ?
 Soit φ une formule de Γ . Soit k le plus grand indice d'une variable propositionnelle apparaissant dans φ . D'après la propriété $\mathcal{Q}$, le sous-ensemble fini $\{\varphi\}$ a un modèle w qui est une extension de v_{k+1} . Puisque v et w donnent la même valeur aux variables de φ , v est aussi modèle de φ . Puisque φ est une formule quelconque de Γ , v est modèle de Γ . ■

16.5. Décidabilité de la logique propositionnelle

Soit $\mathcal{U} \subseteq \mathcal{F}$ un sous-ensemble des formules propositionnelles. Un algorithme D est une **procédure de décision pour $\mathcal{U}$** si étant donné une formule arbitraire φ de $\mathcal{F}$, $D(\varphi)$ termine et renvoie *Ok* si $\varphi \in \mathcal{U}$ et *Nok* si $\varphi \notin \mathcal{U}$.

Proposition 92.

Il existe une procédure de décision pour la satisfiabilité des formules propositionnelles
 $Sat(\varphi)$ dénote de façon générique une procédure de décision de la satisfiabilité de la formule φ .

Il suffit de considérer tous les P -modèles v (avec P ensemble des variables de la formule considérée φ) et de calculer $v^*(\varphi)$. S'il existe v tel que $v^*(\varphi) = 1$, alors φ est satisfiable

Il s'agit de la méthode classique de la table de vérité des formules propositionnelles. Si φ contient n variables, alors il y a par défaut 2^n modèles à considérer, que l'on peut représenter sous forme de table de vérité.

Exemple 93.

Table de vérité pour la formule :

$$r \wedge (p \Rightarrow \neg q)$$

$r \wedge (p \Rightarrow \neg q)$ est satisfiable car il existe une ligne de la table de vérité à 1.

p	q	r	$\neg q$	$p \Rightarrow \neg q$	$r \wedge (p \Rightarrow \neg q)$
1	1	1	0	0	0
1	1	0	0	0	0
1	0	1	1	1	1
1	0	0	1	1	0
0	1	1	0	1	1
0	1	0	0	1	0
0	0	1	1	1	1
0	0	0	1	1	0

Définition 84 (Méthode par réfutation).

Soit Sat une procédure de décision de la satisfiabilité des formules propositionnelles
Considérons $Sat(\neg\varphi)$.

- Si $Sat(\neg\varphi)$ renvoie *Ok*, alors φ n'est pas valide ;
- Si $Sat(\neg\varphi)$ renvoie *Nok*, alors φ est valide.

Il s'agit d'une méthode par réfutation (la validité est établie par réfutation de la satisfiabilité de la négation de la formule)

Remarque 94.

Au lieu de chercher qu'une formule est vraie pour tous les modèles, un seul contre-exemple est recherché. C'est une approche efficace en pratique.

Le problème de la satisfiabilité des formules propositionnelles (problème SAT) est dans la classe **NP** des problèmes décidables en temps polynomial sur une machine de Turing non déterministe.

Théorème 95 (Cook-Levin).

SAT est **NP**-complet (i.e. à la fois dans **NP** et **NP**-difficile)

Autrement dit, à des réductions (transformations) polynomiales près, SAT est parmi les problèmes les plus difficiles de **NP**.

16.6. Méthode des tableaux

La méthode des tableaux profite de la structure inductive : on cherche en effet à réfuter une formule en tirant parti de la sémantique des connecteurs logiques,.

Formules α et formules β

Etant donnée une formule propositionnelle φ définie sur l'ensemble de variables propositionnelles P , la **méthode des tableaux** est une méthode de recherche des P -modèles de φ dirigée par la structure de φ , en particulier, dirigée par le **connecteur principal** de la formule (racine de l'arbre associé).

Définition 85 (Sous-formules).

Soit φ une formule de $For(P)$.

Le connecteur principal de φ , noté $CP(\varphi)$ est :

1. indéfini si $\varphi \in P$.
2. $\neg$ si φ s'écrit $\neg\varphi'$.
3. $@$ si φ s'écrit $(\varphi_1 @ \varphi_2)$ avec $@ \in \{\wedge, \vee, \Rightarrow\}$.

Idée de la méthode des tableaux : analyser - selon le connecteur principal - quelles sont les sous-formules qui contribuent à la valeur de vérité de la formule. Si φ est de la forme

- $\varphi_1 \wedge \varphi_2$, on considère l'ensemble $\{\varphi_1, \varphi_2\}$
- $\varphi_1 \vee \varphi_2$, on considère les deux possibilités φ_1 et φ_2 séparément (branchement)
- p ou $\neg p$, arrêt
- $\varphi \Rightarrow \psi$, on se ramène à $\neg\varphi \vee \psi$
- $\neg\varphi'$ avec φ' formule non réduite à une variable propositionnelle, on transforme la formule afin de descendre la négation au plus près des variables propositionnelles, par application des lois de Morgan, ...

Définition 86 (Formules de type α).

On définit les formules dites de type α , comme le sous-ensemble des formules $For(P)$ qui sont de l'une des formes suivantes, chacune des formules de type α définissant deux formules α_1 et α_2 associées :

- $\varphi \wedge \psi$
 $\alpha_1 = \varphi$ et $\alpha_2 = \psi$
- $\neg(\varphi \vee \psi)$
 $\alpha_1 = \neg\varphi$ et $\alpha_2 = \neg\psi$
- $\neg(\varphi \Rightarrow \psi)$
 $\alpha_1 = \varphi$ et $\alpha_2 = \neg\psi$
- $\neg\neg\varphi$
 $\alpha_1 = \alpha_2 = \varphi$

Pour une formule φ de type α , on note de façon générique $\alpha_1(\varphi)$ et $\alpha_2(\varphi)$ les formules correspondantes au schéma de construction des formules α .

Proposition 96.

Pour toute formule φ de type α , pour toute P -valuation v ,

$$v(\varphi) = 1 \text{ ssi } v(\alpha_1(\varphi)) = 1 \text{ et } v(\alpha_2(\varphi)) = 1$$

Définition 87 (Formules de type β).

On définit les formules dites de type β , comme le sous-ensemble des formules $For(P)$ qui sont de l'une des formes suivantes, chacune des formules de type α définissant deux formules β_1 et β_2 associées :

- $\varphi \vee \psi$
 $\beta_1 = \varphi$ et $\beta_2 = \psi$
- $\neg(\varphi \wedge \psi)$
 $\beta_1 = \neg\varphi$ et $\beta_2 = \neg\psi$
- $(\varphi \Rightarrow \psi)$
 $\beta_1 = \neg\varphi$ et $\beta_2 = \psi$

Pour une formule φ de type β , on note de façon générique $\beta_1(\varphi)$ et $\beta_2(\varphi)$ les formules correspondantes au schéma de construction des formules β .

Proposition 97.

Pour toute formule φ de type β , pour toute P -valuation v ,

$$v(\varphi) = 1 \text{ ssi } v(\beta_1(\varphi)) = 1 \text{ ou } v(\beta_2(\varphi)) = 1$$

Tableaux

Définition 88.

Un tableau est un arbre binaire - donc chaque sommet interne à un ou deux fils - étiqueté vérifiant :

- les sommets sont des ensembles de formules propositionnelles ;
- une branche contenant dans l'un de ses sommets une formule φ de type α peut être prolongée par un sommet étiqueté par $\{\alpha_1(\varphi), \alpha_2(\varphi)\}$ (**règle α**) ;
- une branche contenant dans l'un de ses sommets une formule φ de type β peut être prolongée par deux sommets étiquetés respectivement par $\{\beta_1(\varphi)\}$ et par $\{\beta_2(\varphi)\}$ (**règle β**).

en appelant **branche** tout chemin de la racine de l'arbre à l'une des feuilles de l'arbre.

Ainsi, les règles α crée un seul fils contenant les formules α_1 et α_2 (simple prolongement d'une branche) alors que les règles β crée deux fils contenant respectivement les formules β_1 et β_2 (bifurcation d'une branche).

Construction d'un tableau

- **Sommets** Une ou plusieurs formules
- **Racine** : formule propositionnelle (ou sa négation)
Cela peut éventuellement être un ensemble de formules
- Construction des sommets fils à partir d'un sommet s par **application de l'une des règles α ou β** sur l'une des formules qui apparaissent dans la branche allant de la racine à la feuille, qui n'a pas déjà donné lieu à une application de règle.

Pour une branche B , on note $For(B)$ l'ensemble des formules qui apparaissent dans l'un des sommets de la branche B de l'arbre.

Définition 89.

Soit T un tableau et soit B une branche de T .

- Une branche B est dite **close** s'il existe une formule φ (en pratique une variable propositionnelle) telle que φ et $\neg\varphi$ appartiennent à $For(B)$;
- Une branche qui n'est pas close est dite **ouverte**.
- Un tableau est dit **clos** (resp. **ouvert**) si toutes les branches sont closes (resp. s'il contient une branche ouverte).

La branche B est dite **développée** si

- pour toute formule φ de type α de $For(B)$, $\alpha_1(\varphi)$ et $\alpha_2(\varphi)$ appartiennent aussi à $For(B)$
- pour toute formule φ de type β de $For(B)$, $\beta_1(\varphi)$ ou $\beta_2(\varphi)$ appartiennent aussi à $For(B)$

Un tableau est dit **développé** si toutes ses branches sont soit closes, soit développées.

Proposition 98.

Soit Γ un ensemble de formules propositionnelles.

Il existe un tableau développé et fini pour Γ .

Démonstration : 1er cas : $|\Gamma| = 1$

La taille en nombre de connecteurs des formules α_i et β_j issues des applications des règles α et β est strictement inférieure à celle des formules dont elles sont issues.

Le processus termine donc nécessairement car dans une branche, l'application d'une règle ne peut avoir lieu qu'une fois sur une formule donnée.

2ème cas : $|\Gamma| > 1$ par récurrence sur le nombre de formules de Γ . ■

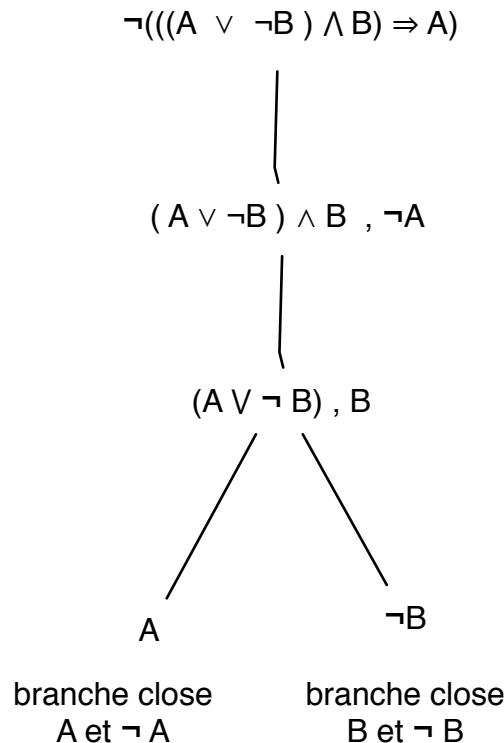
Remarque : pour une même formule φ , selon l'ordre et le choix des règles appliquées, plusieurs tableaux peuvent lui être associés.

Propriété 99.

Soit B une branche développée et ouverte d'un tableau.

Les propriétés suivantes sont vraies :

- il n'existe pas de variable propositionnelle p telle que $p \in \text{For}(B)$ et $\neg p \in \text{For}(B)$;
- pour toute formule φ de type α dans $\text{For}(B)$, alors $\alpha_1(\varphi)$ et $\alpha_2(\varphi)$ sont dans $\text{For}(B)$
- pour toute formule φ de type β dans $\text{For}(B)$, alors $\beta_1(\varphi)$ ou $\beta_2(\varphi)$ est dans $\text{For}(B)$

Exemple de tableau

Principales étapes de construction de l'arbre :

- La formule $\neg(((A \vee \neg B) \wedge B) \Rightarrow A)$ est de la forme $\neg(\varphi_1 \Rightarrow \varphi_2)$ donc de type α avec $\alpha_1 = ((A \vee \neg B) \wedge B)$ et $\alpha_2 = \neg A$.
- Création d'un fils avec les deux formules α_1 et α_2
- $((A \vee \neg B) \wedge B)$ est une formule de type α avec $\alpha_1 = (A \vee \neg B)$ et $\alpha_2 = B$
- Création d'un fils avec les deux (dernières formules α_1 et α_2
- $(A \vee \neg B)$ est de type β avec $\beta_1 = A$ et $\beta_2 = \neg B$
- Création de deux fils avec respectivement A et $\neg B$ comme étiquette

La branche terminant par la feuille étiquetée par A contient aussi $\neg A$, elle est donc close. De même, la branche terminant par la feuille étiquetée par B contient aussi $\neg B$, elle est donc close.

Correction de la méthode des tableaux

Définition 90.

Une formule φ est **prouvable par tableau** s'il existe un tableau clos avec la racine $\neg\varphi$.

On note alors $\vdash_{tab} \varphi$ (ou simplement $\vdash \varphi$)

Autrement dit, φ est prouvable, s'il existe un tableau à partir de $\neg\varphi$ dont toutes les branches sont closes (i.e. contiennent une variable propositionnelle et sa négation).

Définition 91.

Une branche B d'un tableau est **réalisable** s'il existe une valuation v telle que $\forall \psi \in For(B), v \models \psi$ (en particulier $v(p) = 1$ si $p \in For(B)$ et $v(p) = 0$ si $\neg p \in For(B)$).

Un tableau est dit **réalisable** s'il contient une branche réalisable.

Théorème 100 (Correction).

Toute formule prouvable (par la méthode des tableaux) est une tautologie.

(i.e. $\forall \varphi, \vdash_{tab} \varphi \implies \models \varphi$)

Démonstration :

Lemme 101.

L'application d'une règle α ou β préserve la réalisabilité des branches. Autrement dit, l'application d'une règle sur un tableau réalisable construit un tableau réalisable.

Démonstration : (lemme)

Soit B une branche réalisable et soit v une valuation telle que $\forall \psi \in For(B), v(\psi) = 1$.

- Soit B' la branche issue de B par application d'une règle α à partir d'une formule φ de $For(B)$.

$For(B') = For(B) \cup \{\alpha_1(\varphi), \alpha_2(\varphi)\}$ avec $\varphi \in For(B)$.

Par propriété des formules α , $v(\alpha_1(\varphi)) = 1$ et $v(\alpha_2(\varphi)) = 1$. B' est donc réalisable.

- Soit B_1 et B_2 les deux branches issues de B par application d'une règle β . $For(B_1) = For(B) \cup \{\beta_1(\varphi)\}$ et $For(B_2) = For(B) \cup \{\beta_2(\varphi)\}$ avec $\varphi \in For(B)$.

On a $v(\beta_1(\varphi)) = 1$ ou $v(\beta_2(\varphi)) = 1$. Et l'une des deux branches B_1 ou B_2 sont réalisables.

L'application des règles α et β préserve la réalisabilité du tableau. ■

Soit φ prouvable par la méthode des tableaux, alors il existe un tableau clos à partir de $\neg\varphi$. Chaque branche contient une variable p et sa négation $\neg p$, et donc n'est pas réalisable. Donc la racine de l'arbre n'est ($\neg\varphi$) n'est pas réalisable (sinon, la propriété de réalisabilité aurait été préservée lors de la construction du tableau à partir de $\neg\varphi$).

Donc il n'existe pas de valuation v vérifiant $v^*(\neg\varphi) = 0$. Soit $\forall v, v^*(\varphi) = 1$. Donc φ est une tautologie. ■

Complétude de la méthode des tableaux

Théorème 102 (Complétude).

Toute tautologie est prouvable

Corollaire 4.

φ est une tautologie ssi elle est prouvable

Lemme 103.

(1) Toute branche développée et ouverte d'un tableau est réalisable.

Démonstration : Soit B une branche développée et ouverte. Considérons la valuation définie par :

1. si $p \in \text{For}(B)$, $v(p) = 1$
2. si $\neg p \in \text{For}(B)$, $v(p) = 0$
3. si $p \notin \text{For}(B)$ et si $\neg p \notin \text{For}(B)$, posons $v(p) = 1$ (arbitraire)

La définition est bien fondée car p et $\neg p$ ne peuvent être conjointement dans $\text{For}(B)$ (B est ouverte).

Par induction structurelle (pseudo-structurelle en considérant que $\alpha_i(\varphi)$ et $\beta_i(\varphi)$ sont des sous-formules de φ). On suppose que toutes les sous-formules ψ de φ au sens α ou β , vérifie $v^*(\psi) = 1$.

C'est vrai pour les formules de la forme p ou $\neg p$: $v(p) = 1$ si $p \in \text{For}(B)$ et $v(p) = 0$ si $\neg p \in \text{For}(B)$.

Si φ n'est pas un littéral et de type α , alors $\alpha_1(\varphi)$ et $\alpha_2(\varphi)$ appartiennent à $\text{For}(B)$ (car B est développée) et sont évalués à 1 avec v par hypothèse d'induction. Et $v(\varphi) = 1$.

Si φ n'est pas un littéral et de type β , alors l'un des des formules $\beta_1(\varphi)$ ou $\beta_2(\varphi)$ appartient à $\text{For}(B)$ (car B est développée), et évaluée à 1 avec v par hypothèse d'induction. Et $v(\varphi) = 1$.

Donc $\forall \varphi \in \text{For}(B), v^*(\varphi) = 1$ ■

Lemme 104.

(2)

S'il existe un tableau clos avec $\neg\varphi$ comme racine, alors tout tableau développé de racine $\neg\varphi$ est clos.

Démonstration : Soit φ tautologie. En raisonnant par l'absurde, supposons que φ n'est pas prouvable par la méthode des tableaux. Il n'existe donc pas de tableau clos avec la racine $\neg\varphi$. Soit un tableau développé à partir de $\neg\varphi$. Le tableau contient une branche B ouverte et développée en partant de la racine $\neg\varphi$.

B est donc réalisable (cf lemme 1) : $\neg\varphi$ (élément de $\text{For}(B)$) est satisfiable. Contradiction avec φ tautologie. Il n'existe donc pas de tableau clos de racine $\neg\varphi$. ■

Démonstration : (Démonstration du théorème de complétude par l'absurde)

Soit φ une tautologie et supposons que $\neg\varphi$ n'est pas prouvable par tableau. Il n'existe donc pas de tableau clos de racine $\neg\varphi$.

Soit un tableau développé de racine $\neg\varphi$. Il n'est pas clos (lemme 2), et il existe donc une branche réalisable, donc $\neg\varphi$ est satisfiable (lemme 1). φ n'est pas une tautologie. Contradiction. ■

17. Calculs pour la logique propositionnelle

Il existe plusieurs calculs pour la logique propositionnelle. Ici, nous allons en évoquer trois :

1. Calcul à la Hilbert ;
2. Dédution naturelle ;
3. Calcul des séquents.

Le calcul à la Hilbert est le premier qui a été défini pour la logique propositionnelle. Comme son nom l'indique il a été défini par Hilbert s'appuyant sur des travaux entrepris par Frege. Son utilisation n'est pas aisée, c'est pourquoi d'autres calculs ont été proposés par la suite, entre autres la déduction naturelle et le calcul des séquents. Ces derniers ont montré leur importance en informatique théorique, et plus particulièrement dans la preuve automatique et le fondement des langages de programmation. Leur simplicité d'utilisation vient du fait qu'ils sont définis en profitant de la structure inductive des formules sur les connecteurs logiques.

Afin d'alléger la présentation des règles d'inférence, nous allons restreindre la logique propositionnelle à l'ensemble des connecteurs $\neg$ et $\Rightarrow$, (ou même pour le système à la Hilbert au seul connecteur $\Rightarrow$).

Rappelons que $\neg$ et $\Rightarrow$ forme un système complet de connecteurs. En effet

- $\varphi \wedge \psi$ et $\neg(\varphi \Rightarrow \neg\psi)$ sont deux formules équivalentes.
- $\varphi \vee \psi$ et $\neg\varphi \Rightarrow \psi$ sont deux formules équivalentes.

17.1. Système de preuves à la Hilbert

Nous esquissons ci-dessous le système de preuves réduit au seul connecteur $\Rightarrow$.

Considérons les règles suivantes :

$$\frac{}{A \Rightarrow (B \Rightarrow A)} [Axiome1]$$

$$\frac{}{(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))} [Axiome2]$$

$$\frac{A \Rightarrow B \quad A}{B} [Modus Ponens]$$

Considérons la preuve :

$$\frac{\frac{\frac{}{a} \text{ Axiome2} \quad \frac{\frac{}{b} \text{ Axiome1}}{\text{MP}}}{c} \quad \frac{}{d} \text{ Axiome1}}{e} \text{ MP}$$

avec

- $a : (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))$ Axiome 2
- $b : (p \Rightarrow ((p \Rightarrow p) \Rightarrow p))$ Axiome 1
- $c : ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))$ Modus Ponens entre a et b

- $d : (p \Rightarrow (p \Rightarrow p))$ Axiome 1
 - $e : (p \Rightarrow p)$ Modus Ponens entre c et d
- Soit

$$\frac{\frac{\frac{(p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))}{((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p))} \text{Axiome2} \quad \frac{(p \Rightarrow ((p \Rightarrow p) \Rightarrow p))}{(p \Rightarrow (p \Rightarrow p))} \text{Axiome1}}{(p \Rightarrow p)} \text{MP}$$

Cet exemple de preuve illustre de la difficulté d'exhiber une preuve dans le système (historique) proposé par Hilbert.

Ceci dit, une fois que $(p \Rightarrow p)$ a été démontré une fois, il est possible de réutiliser cette propriété, comme lemme (ou hypothèse) car la déduction au sein des systèmes formels est par nature transitive.

17.2. Système de la déduction naturelle

Le système de déduction naturelle permet d'introduire et retirer des hypothèses.

Les règles de déduction naturelle s'écrivent :

$$\frac{\boxed{\begin{array}{c} \varphi \\ \vdots \\ \psi \end{array}}}{\varphi \Rightarrow \psi} [\Rightarrow -i] \quad \frac{\varphi \quad \varphi \Rightarrow \psi}{\psi} [\Rightarrow -e]$$

$$\frac{\boxed{\begin{array}{c} \varphi \\ \vdots \\ \perp \end{array}}}{\neg \varphi} [\neg -i] \quad \frac{\varphi \quad \neg \varphi}{\perp} [\neg -e]$$

avec :

- $[\Delta - e]$ et $[\Delta - i]$ désignent respectivement les règles d'élimination (e) et d'introduction (i) du connecteur Δ .

Le calcul de la déduction naturelle contient une règle d'introduction et une règle d'élimination par connecteur (autant que possible).

- $\boxed{\begin{array}{c} \varphi \\ \vdots \\ \psi \end{array}}$ représente une preuve de ψ à partir de φ . Une règle contenant en prémisse une telle boîte se lit : "si on a réussi à prouver ψ à partir de φ ", alors on peut déduire la formule en conclusion de la règle. On dit alors que l'hypothèse φ est **déchargée**.

La déduction naturelle permet ainsi d'introduire des hypothèses à des fins de démonstration.

A la différence du système à la Hilbert qui contient beaucoup d'axiomes et une règle, la déduction naturelle ne contient pas d'axiome, et contient beaucoup de règles. Le mécanisme de décharge d'hypothèse fait que la déduction naturelle ne se présente pas comme un système formel.

Gerhard Gentzen a inventé le calcul de déduction naturelle ainsi que le calcul des séquents décrit ci-dessous, qui se présente sous la forme d'un système formel.

17.3. Calcul des séquents

L'un des problèmes majeurs du calcul à la Hilbert réside dans la possibilité de considérer lors d'une preuve d'une formule donnée, des formules indépendantes (i.e., qui ne sont ni des transformées, ni des parties de la formule à prouver - voir les règles du modus-ponens et du raisonnement par l'absurde). Ceci demande alors une connaissance approfondie de la formule à prouver pour savoir quelles formules il faut ajouter pour sa démonstration. Ici, nous proposons un nouveau système d'inférences appelé **calcul des séquents**. À l'inverse du système de preuves à la Hilbert, le calcul des séquents permettra la preuve d'une formule en décomposant ou en transformant la formule à prouver sans ajouter aucun élément extérieur.

Un séquent (réduit) s'écrit de façon générique

$$\Gamma \vdash \varphi$$

avec Γ ensemble de formules, et φ formule, écrites à l'aide des (seuls) connecteurs $\Rightarrow$ et $\neg$

Le séquent $\Gamma \vdash \varphi$ est valide si et seulement si pour toute valuation qui rend vraie toutes les formules φ_i de Γ , φ est évaluée à vrai pour cette valuation. On note alors $\Gamma \models \varphi$.

Les formules $\Gamma \vdash \varphi$ sont appelés *séquents* ou *jugements*. Le système manipule conjointement les hypothèses et les formules inférées. Le symbole $\vdash$ n'est pas un connecteur logique, juste un séparateur entre les hypothèses (ou antécédent) Γ et les conclusions (ou conséquent) φ .

Notation : $\Gamma \cup \Gamma'$ sera noté Γ, Γ' et $\Gamma \cup \{\psi\}$ sera noté Γ, ψ .

Il existe plusieurs calculs manipulant des séquents.

Définition 92 (Calcul réduit des séquents).

Une P -formule φ est un **théorème** d'un ensemble de P -formules Γ , noté $\Gamma \vdash \varphi$, si et seulement si on peut obtenir l'énoncé $\Gamma \vdash \varphi$ en utilisant les règles de déduction suivantes :

- **Utilisation d'une hypothèse** : si $\varphi \in \Gamma$ alors $\Gamma \vdash \varphi$

On note

$$\overline{\Gamma, \varphi \vdash \varphi}$$

- **Augmentation d'hypothèses** : si $\psi \notin \Gamma$ et $\Gamma \vdash \varphi$ alors $\Gamma, \psi \vdash \varphi$

$$\frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}$$

- **Règle de détachement (modus ponens)** : si $\Gamma \vdash \psi \Rightarrow \varphi$ et $\Gamma \vdash \psi$ alors $\Gamma \vdash \varphi$

$$\frac{\Gamma \vdash \psi \Rightarrow \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi}$$

- **Règle de synthèse (retrait d'hypothèse)** : si $\Gamma, \varphi \vdash \psi$ alors $\Gamma \vdash \varphi \Rightarrow \psi$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}$$

- **Règle de double négation** : $\Gamma \vdash \varphi$ si et seulement si $\Gamma \vdash \neg\neg\varphi$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \neg\neg\varphi} \text{ et } \frac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}$$

- **Règle du raisonnement par l'absurde** : si $\Gamma, \varphi \vdash \psi$ et $\Gamma, \varphi \vdash \neg\psi$ alors $\Gamma \vdash \neg\varphi$

$$\frac{\Gamma, \varphi \vdash \psi \quad \Gamma, \varphi \vdash \neg\psi}{\Gamma \vdash \neg\varphi}$$

La combinaison de ces règles engendrent des démonstrations de théorèmes.

Une **déduction** de $\Gamma \vdash \varphi$ est une suite finie d'énoncés $\Gamma_i \vdash \varphi_i$ dont le dernier est $\Gamma \vdash \varphi$, telle que tous les énoncés de cette suite sont obtenus en appliquant les règles à des énoncés qui les précèdent dans la suite.

Remarquons que le premier énoncé d'une preuve est nécessairement obtenu par la règle d'utilisation d'une hypothèse.

Exemple 105.

Prouvons l'énoncé $\vdash \neg\varphi \Rightarrow \neg\varphi$

1. $\neg\varphi \vdash \neg\varphi$ (utilisation d'hypothèse)
2. $\vdash \neg\varphi \Rightarrow \neg\varphi$ (retrait d'hypothèse)

$$\frac{\overline{\neg\varphi \vdash \neg\varphi} \text{ Utilisation d'hypothèse}}{\vdash \neg\varphi \Rightarrow \neg\varphi} \text{ Retrait d'hypothèse}$$

Quelques résultats utiles.

Rappelons qu'une **substitution** σ est une application de P (l'ensemble des variables propositionnelles) dans l'ensemble des P -formules, telle que $\text{dom}(\sigma) = \{p \mid \sigma(p) \neq p\}$ est fini.

On note σ^* l'application de l'ensemble des P -formules dans lui-même définie comme l'extension canonique de σ aux P -formules, i.e.

- $\sigma^*(p) = \sigma(p)$ pour tout $p \in P$
- $\sigma^*(\neg\varphi) = \neg\sigma^*(\varphi)$
- $\sigma^*(\varphi \Rightarrow \psi) = \sigma^*(\varphi) \Rightarrow \sigma^*(\psi)$

Proposition 106.

Soit σ une substitution. Si $\Gamma \vdash \varphi$ alors $\sigma^*(\Gamma) \vdash \sigma^*(\varphi)$ où $\sigma^*(\Gamma) = \{\sigma^*(\varphi) \mid \varphi \in \Gamma\}$.

Démonstration : Montrons alors par récurrence sur la taille des démonstrations, que Si $(\Gamma_i \vdash \varphi_i)_{i=1\dots n}$ est une démonstration de $\Gamma \vdash \varphi$, alors, $(\sigma^*(\Gamma_i) \vdash \sigma^*(\varphi_i))_{i=1\dots n}$ est une démonstration de $\sigma^*(\Gamma) \vdash \sigma^*(\varphi)$.

Cas de base. $\sigma^*(\Gamma_1) \vdash \sigma^*(\varphi_1)$ est prouvable car, puisque $\Gamma_1 \vdash \varphi_1$ est le premier énoncé prouvable, $\varphi_1 \in \Gamma_1$, et donc $\sigma^*(\varphi_1) \in \sigma^*(\Gamma_1)$.

Cas général. Supposons que $(\sigma^*(\Gamma_i) \vdash \sigma^*(\varphi_i))_{i=1\dots k}$ est une démonstration. Montrons sur la forme de la dernière règle appliquée pour obtenir la démonstration $(\Gamma_i \vdash \varphi_i)_{i=1\dots k+1}$ que $\sigma^*(\Gamma_{k+1}) \vdash \sigma^*(\varphi_{k+1})$.

- (modus ponens (MP)) l'énoncé $\Gamma_{k+1} \vdash \varphi_{k+1}$ a été obtenu à partir de $\Gamma_i \vdash \psi_i \Rightarrow \varphi_i$ et $\Gamma_j \vdash \psi_j$ tels que $\Gamma_i = \Gamma_j$, $\psi_i = \psi_j$ et $\varphi_i = \varphi_{k+1}$. Par hypothèse de récurrence, on $\sigma^*(\Gamma_i) \vdash \sigma^*(\psi_i) \Rightarrow \sigma^*(\varphi_i)$ et $\sigma^*(\Gamma_j) \vdash \sigma^*(\psi_j)$ et donc par MP, $\sigma^*(\Gamma_i) \vdash \sigma^*(\varphi_i)$.
- On procède de la même façon pour les autres règles. ■

Proposition 107 (Théorème de la déduction).

$\varphi_1, \dots, \varphi_n \vdash \varphi$ si, et seulement si $\vdash \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi$

Démonstration : Ceci revient à montrer que $\Gamma, \psi \vdash \varphi$ si, et seulement si $\Gamma \vdash \psi \Rightarrow \varphi$.

Le sens " $\Rightarrow$ " est la règle de synthèse (introduction du $\Rightarrow$).

Pour le sens " $\Leftarrow$ ", si $\Gamma \vdash \psi \Rightarrow \varphi$ alors $\Gamma, \psi \vdash \psi \Rightarrow \varphi$ par augmentation des hypothèses.

On a aussi $\Gamma, \psi \vdash \psi$ par utilisation des hypothèses, et donc par modus ponens, $\Gamma, \psi \vdash \varphi$. ■

Théorème 108 (Correction).

Si $\Gamma \vdash \varphi$ alors $\Gamma \models \varphi$

Démonstration : Ceci se prouve par *induction* sur la longueur des démonstrations. Étant donnée une démonstration $(\Gamma_i \vdash \varphi_i)_{i=1\dots n}$, on a $\Gamma_i \models \varphi_i$.

Cas de base. $\Gamma_1 \vdash \varphi_1$ est valide. En effet, s'il existe v telle que $v^*(\Gamma) = 1$ alors $v^*(\varphi) = 1$ car, puisque $\Gamma_1 \vdash \varphi_1$ est le premier énoncé prouvable, $\varphi_1 \in \Gamma_1$.

Cas général. Supposons que $(\Gamma_i \models \varphi_i)_{i=1\dots k+1}$. Par hypothèse de récurrence, on a $\Gamma_i \models \varphi_i$ pour chaque $i = 1, \dots, k$. Montrons alors sur la forme de la dernière règle appliquée pour obtenir la démonstration $(\Gamma_i \vdash \varphi_i)_{i=1\dots k+1}$ que $\Gamma_{k+1} \models \varphi_{k+1}$.

- (modus ponens (MP)) L'énoncé $\Gamma_{k+1} \vdash \varphi_{k+1}$ a été obtenu à partir de $\Gamma_i \vdash \psi_i \Rightarrow \varphi_i$ et $\Gamma_j \vdash \psi_j$ tels que $\Gamma_i = \Gamma_j$, $\psi_i = \psi_j$, et $\varphi_i = \varphi_{k+1}$. Soit v tel que $v \models \Gamma_{k+1}$. Par définition, $\Gamma_{k+1} = \Gamma_i = \Gamma_j$. Par hypothèse, $v^*(\psi_j) = v^*(\psi_i) = 1$, et donc $v^*(\varphi_{k+1}) = 1$.
- On procède de la même façon pour les autres règles. ■

Théorème 109 (Complétude).

$$\Gamma \models \varphi \implies \Gamma \vdash \varphi.$$

On va d'abord montrer le résultat de complétude pour les tautologies, c'est-à-dire on démontre d'abord :

$$\text{Si } \models \varphi \text{ alors } \vdash \varphi$$

Lemme 110.

Soit φ une formule construite sur les seules variables propositionnelles $p_1, p_2, \dots, p_n$. l_i dénote la variable p_i ou sa négation $\neg p_i$.

$v_{(l_i)}$ désigne alors la valuation associant la valeur 1 (resp. 0) à la variable p_i si $l_i = p_i$ (resp. $l_i = \neg p_i$).

Alors on a :

- $l_1, l_2, \dots, l_n \vdash \varphi$ ssi $v_{(l_i)}^*(\varphi) = 1$
- $l_1, l_2, \dots, l_n \vdash \neg \varphi$ ssi $v_{(l_i)}^*(\varphi) = 0$

Démonstration : La démonstration se fait par induction structurelle sur la forme de φ

- si φ est réduite à une variable propositionnelle p , on doit alors montrer $p \vdash p$ et $\neg p \vdash \neg p$ (car $v^*(\varphi) = 1$ ssi $l_1 = p$ et $v^*(\varphi) = 0$ ssi $l_1 = \neg p$).

Ces propriétés - $p \vdash p$ et $\neg p \vdash \neg p$ - sont triviales (par la règle d'utilisation d'hypothèses).

- si φ est de la forme $\neg \psi$

φ et ψ ont les mêmes variables. On peut appliquer l'hypothèse d'induction sur ψ .

— $l_1, l_2, \dots, l_n \vdash \psi$ ssi $v_{(l_i)}^*(\psi) = 1$

— $l_1, l_2, \dots, l_n \vdash \neg \psi$ ssi $v_{(l_i)}^*(\psi) = 0$

Or $v_{(l_i)}^*(\psi) = 1$ (resp. $= 0$) ssi $v_{(l_i)}^*(\varphi) = 0$ (resp. $= 1$).

Soit

— $l_1, l_2, \dots, l_n \vdash \psi$ ssi $v_{(l_i)}^*(\varphi) = 0$

— $l_1, l_2, \dots, l_n \vdash \neg \psi$ ssi $v_{(l_i)}^*(\varphi) = 1$

Soit

— $l_1, l_2, \dots, l_n \vdash \neg \neg \psi$ ssi $v_{(l_i)}^*(\varphi) = 0$ par utilisation de la règle de double négation

— $l_1, l_2, \dots, l_n \vdash \varphi$ ssi $v_{(l_i)}^*(\varphi) = 1$ car $\neg \psi$ est précisément φ .

Soit

— $l_1, l_2, \dots, l_n \vdash \neg \varphi$ ssi $v_{(l_i)}^*(\varphi) = 0$ car $\neg \psi$ est φ

- $l_1, l_2, \dots, l_n \vdash \varphi$ ssi $v_{(l_i)}^*(\varphi) = 1$
- si φ est de la forme $\psi \Rightarrow \rho$
 Considérons le cas où $v_{(l_i)}^*(\varphi) = 0$. On a alors $v_{(l_i)}^*(\psi) = 1$ et $v_{(l_i)}^*(\rho) = 0$.
 Par hypothèse d'induction, on a : $l'_1, l'_2, \dots, l'_k \vdash \psi$ et $l''_1, l''_2, \dots, l''_j \vdash \neg\rho$. avec $\{l'_1, l'_2, \dots, l'_k\} \cup \{l''_1, l''_2, \dots, l''_j\} = \{l_1, l_2, \dots, l_n\}$. On a par augmentation d'hypothèse $l_1, l_2, \dots, l_n \vdash \psi$ et $l_1, l_2, \dots, l_n \vdash \neg\rho$. Notons Γ les hypothèses $l_1, l_2, \dots, l_n$.
 On a donc $\Gamma \vdash \psi$ (a) et $\Gamma \vdash \neg\rho$ (b).
 1. $\Gamma, \psi \Rightarrow \rho \vdash \psi$ (augmentation d'hypothèse à partir de (a)).
 2. $\Gamma, \psi \Rightarrow \rho \vdash \neg\rho$ (augmentation d'hypothèse à partir de (b)).
 3. $\Gamma, \psi \Rightarrow \rho \vdash \psi \Rightarrow \rho$ (Hypothèse)
 4. $\Gamma, \psi \Rightarrow \rho \vdash \rho$ (MP entre (i) et (iii)).
 5. $\Gamma \vdash \neg(\psi \Rightarrow \rho)$ (Absurde entre (ii) et (iv))
 On obtient bien $l_1, l_2, \dots, l_n \vdash \varphi$ pour $v_{(l_i)}^*(\varphi) = 0$
 Considérons le cas où $v_{(l_i)}^*(\varphi) = 1$.
 3 sous-cas sont à envisager. Dans chacun des sous-cas, Γ est l'ensemble des l_i défini par les valeurs de $v_{(l_i)}^*$ pour les formules ψ et ρ . Il s'agit alors de montrer que les trois cas conduisent à la conclusion $\Gamma \vdash \varphi$.
 - $v_{(l_i)}^*(\psi) = 0$ et $v_{(l_i)}^*(\rho) = 1$.
 On a $\Gamma \vdash \neg\psi$ (a) et $\Gamma \vdash \rho$ (b)
 1. $\Gamma, \psi \vdash \rho$ (augmentation d'hypothèse par rapport à (b))
 2. $\Gamma \vdash \psi \Rightarrow \rho$ (retrait d'hypothèse à partir de (i))
 3. $\Gamma \vdash \varphi$
 - $v_{(l_i)}^*(\psi) = 0$ et $v_{(l_i)}^*(\rho) = 0$.
 On a $\Gamma \vdash \neg\psi$ (a) et $\Gamma \vdash \neg\rho$ (b)
 1. $\Gamma, \neg\rho \vdash \neg\psi$ (augmentation d'hypothèse par rapport à (a))
 2. $\Gamma \vdash \neg\rho \Rightarrow \neg\psi$ (retrait d'hypothèse à partir de (i))
 3. $\Gamma \vdash (\neg\rho \Rightarrow \neg\psi) \Rightarrow (\psi \Rightarrow \rho)$
 ($\vdash (\neg\rho \Rightarrow \neg\psi) \Rightarrow (\psi \Rightarrow \rho)$ lemme prouvé en PC)
 4. $\Gamma \vdash \psi \Rightarrow \rho$ (MP entre (ii) et (iii))
 5. $\Gamma \vdash \varphi$
 - $v_{(l_i)}^*(\psi) = 1$ et $v_{(l_i)}^*(\rho) = 1$.
 On a $\Gamma \vdash \psi$ et $\Gamma \vdash \rho$
 1. $\Gamma, \psi \vdash \rho$ (augmentation d'hypothèse)
 2. $\Gamma \vdash \psi \Rightarrow \rho$ (retrait d'hypothèse)
 3. $\Gamma \vdash \varphi$
 (raisonnement similaire au premier des sous-cas)
 On vient de montrer par induction le résultat du lemme. ■

Lemme 111.

Pour Γ ensemble de formules, et φ et ψ deux formules, si on a :

- $\Gamma, \varphi \vdash \psi$
- $\Gamma, \neg\varphi \vdash \psi$

alors $\Gamma \vdash \psi$.

Démonstration : Voir feuille de PC ■

Lemme 112.

Pour toute tautologie φ , on a $\vdash \varphi$. Autrement dit :

$$\forall \varphi, \text{ si } \models \varphi \text{ alors } \vdash \varphi$$

Démonstration : Soit φ une tautologie (définie sur les seules variables $p_1, p_2, \dots, p_n$).

D'après le premier lemme, pour toutes les configurations $l_1, l_2, \dots, l_n$ (l_i valant p_i ou $\neg p_i$), on a alors :

$$l_1, l_2, \dots, l_n \vdash \varphi$$

en effet, comme φ est une tautologie, alors $v_{(l_i)}^*(\varphi) = 1$ pour toutes les configurations $l_1, l_2, \dots, l_n$.

Pour une configuration donnée sur les variables $p_1, \dots, p_{n-1}$, notée Γ' , on a donc

$$\Gamma', p_n \vdash \varphi$$

$$\Gamma', \neg p_n \vdash \varphi$$

D'après le second lemme, on obtient :

$$\Gamma' \vdash \varphi$$

avec Γ' n'importe quelle configuration ne contenant pas la variable p_n .

En itérant le processus, toutes les variables peuvent être retirées des prémisses Γ' .

Et on obtient $\vdash \varphi$ pour φ tautologie. ■

Généralisons ce premier résultat aux conséquences sémantiques d'un ensemble de P -formules Γ . On suppose alors que $\Gamma \models \varphi$.

Lemme 113 (Lemme de finitude).

Si $\Gamma \models \varphi$ alors il existe un sous-ensemble fini $\Gamma' \subseteq \Gamma$, $\Gamma' \models \varphi$.

Démonstration (lemme) : Si $\Gamma \models \varphi$ alors $\Gamma \cup \{\neg \varphi\}$ est contradictoire. Par le théorème de la compacité, il existe alors un sous-ensemble fini $\Gamma' \subseteq \Gamma \cup \{\neg \varphi\}$ contradictoire, et donc $\Gamma' \cup \{\neg \varphi\}$ est aussi contradictoire, i.e. $\Gamma' \models \varphi$. ■

(Reprise de la preuve du théorème 109) Soit $\Gamma \models \varphi$. Par le lemme ci-dessus, il existe un sous-ensemble fini $\Gamma' \subseteq \Gamma$ tel que $\Gamma' \models \varphi$. Supposons alors que $\Gamma' = \varphi_1, \dots, \varphi_n$. Par le théorème de la déduction, on a $\models \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi$. Or, on sait que pour les tautologies, le résultat de complétude marche. On a donc, $\vdash \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi$. Par le théorème de la déduction, on a donc $\{\varphi_1, \dots, \varphi_n\} \vdash \varphi$, d'où nous déduisons $\Gamma \vdash \varphi$.

17.4. Calcul des séquents (complet)

On considère l'ensemble des connecteurs $\neg$, $\wedge$, $\vee$ et $\Rightarrow$, ainsi que les constantes $\top$ et $\perp$.
Un séquent se présente sous la forme

$$\varphi_1, \dots, \varphi_n \vdash \psi_1, \dots, \psi_k$$

et se lit comme $(\varphi_1 \wedge \dots \wedge \varphi_n) \Rightarrow (\psi_1 \vee \dots \vee \psi_k)$.

(si $n = 0$, alors $(\varphi_1 \wedge \dots \wedge \varphi_n)$ est $\top$, si $k = 0$ alors $(\psi_1 \vee \dots \vee \psi_k)$ est $\perp$)

Un séquent s'écrit de façon générique

$$\Gamma \vdash \Delta$$

Le séquent $\Gamma \vdash \Delta$ est valide si et seulement si pour toute valuation qui rend vraie toutes les formules φ_i de Γ , il existe une formule ψ_j de Δ qui est aussi vraie dans cette valuation. On note alors $\Gamma \models \Delta$.

Cette notation étend la notation adoptée précédemment $\Gamma \vdash \varphi$, puisque $\Gamma \vdash \varphi$ correspond trivialement au cas où Δ contient exactement une formule ($k = 1$).

$$\begin{array}{c} \frac{}{\varphi, \Gamma \vdash \Delta, \varphi} [Hyp] \\[10pt] \frac{}{\perp, \Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \perp} \\[10pt] \frac{\Gamma \vdash \Delta}{\top, \Gamma \vdash \Delta} \qquad \frac{}{\Gamma \vdash \Delta, \top} \\[10pt] \frac{\Gamma \vdash \Delta, \varphi}{\neg \varphi, \Gamma \vdash \Delta} \qquad \frac{\varphi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg \varphi} \\[10pt] \frac{\varphi, \psi, \Gamma \vdash \Delta}{\varphi \wedge \psi, \Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, \varphi \quad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \varphi \wedge \psi} \\[10pt] \frac{\varphi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, \varphi, \psi}{\Gamma \vdash \Delta, \varphi \vee \psi} \\[10pt] \frac{\Gamma \vdash \Delta, \varphi \quad \psi, \Gamma \vdash \Delta}{\varphi \Rightarrow \psi, \Gamma \vdash \Delta} \qquad \frac{\varphi, \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \varphi \Rightarrow \psi} \end{array}$$

Les règles sont ordonnées de la façon suivante : un connecteur par ligne, et la règle à gauche (resp. à droite) élimine un connecteur en prémisses (resp. en conclusion) du séquent.

On ajoute à l'ensemble de ces règles définissant le comportement logique des connecteurs propositionnels, l'ensemble de règles suivant, appelés *règles structurelles*

Règles structurelles

$$\frac{\Gamma \vdash \Delta}{\Gamma, \varphi \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \varphi}$$

Coupure :

$$\frac{\Gamma \vdash \Delta, \varphi \quad \Gamma, \varphi \vdash \Delta}{\Gamma \vdash \Delta}$$

On verra dans le chapitre consacré à logique des prédicats que la règle de coupure est une règle redondante que l'on peut éliminer. C'est le fameux résultat de l'élimination des coupures démontré par Gentzen pour montré la consistance de l'arithmétique.

Ce système se prête à l'automatisation du raisonnement car les séquents en prémisses d'une règle contient moins de connecteurs logiques que celui qui est en conclusion. On peut appliquer une stratégie qui consiste à éliminer un à un les connecteurs à partir d'un séquent à prouver. Par ailleurs, toutes les règles sont réversibles c'est-à-dire qu'une interprétation rend vrai un séquent en conclusion d'une règle si et seulement si elle rend vraie les séquents des prémisses. Et bien sûr, le système est correct et complet.

Nous concluons ce chapitre avec la donnée d'un arbre de preuves du calcul des séquents (réduit) :

Exemple 114.

Prouvons dans le calcul des séquents pour la logique propositionnelle, la tautologie $(\alpha \Rightarrow \beta) \Rightarrow (\neg\alpha \Rightarrow \neg\beta)$.

$$\begin{array}{c}
 \frac{\frac{\overline{\neg\beta, \alpha \vdash \alpha} \text{ Ax}}{\neg\beta \vdash \neg\alpha, \alpha} \vdash \neg \quad \frac{\frac{\overline{\beta \vdash \alpha, \beta} \text{ Ax}}{\neg\beta, \beta \vdash \neg\alpha} \neg \vdash}{\alpha \Rightarrow \beta, \neg\alpha \vdash \neg\beta} \Rightarrow \vdash \\
 \frac{\alpha \Rightarrow \beta \vdash \neg\alpha \Rightarrow \neg\beta}{\vdash (\alpha \Rightarrow \beta) \Rightarrow (\neg\alpha \Rightarrow \neg\beta)} \vdash \Rightarrow
 \end{array}$$

18. Méthodes algorithmiques

Bien que la logique propositionnelle soit décidable, des méthodes de preuve ont été définies dont la traduction algorithmique est plus naturelle. On présentera ici deux de ces méthodes :

1. L'algorithme DPPL ;
2. La méthode de la résolution.

L'algorithme DPPL est une méthode algorithmique permettant de résoudre la satisfiabilité d'une formule propositionnelle (problème SAT dont on sait qu'il est NP-complet). Bien sûr, cet algorithme n'est pas adaptable aux autres logiques entre autres la logique des prédicats du premier ordre dont on verra dans le chapitre suivant qu'elle est indécidable à la différence de la logique propositionnelle.

La méthode de la résolution est une méthode de preuve qui généralise le modus Ponens. Cette méthode est principalement utilisée dans les systèmes de preuve automatique et est surtout à la base du langage de programmation logique Prolog.

18.1. Mise sous forme normale

Rappel : Formules équivalentes

Selon les situations, les algorithmes de logique propositionnelle nécessitent un pré-traitement des formules (élimination de connecteurs, mise sous forme normale). Selon l'objectif, ce pré-traitement sera fait en préservant l'équivalence des formules ou l'équisatisfiabilité des formules. Rappelons que deux formules φ et ϕ sont dites équivalentes ($\varphi \equiv \phi$) si et seulement si pour tout P -modèle v , $v^*(\varphi) = v^*(\phi)$.

Deux formules φ et ψ sont dites **équisatisfiables** si et seulement si elles sont satisfiables ensemble, autrement dit, φ est satisfiable si et seulement si ψ est satisfiable. L'équisatisfiabilité de formules est une propriété plus faible que l'équivalence de formules. Ainsi p et $\top$ sont équisatisfiables, mais ne sont pas équivalentes.

Les mises en forme normale tirent parti des propriétés d'associativité et de commutativité des connecteurs, notamment $\wedge$ et $\vee$. Ainsi, sachant que les (12) formules suivantes sont équivalentes :

- $((p \wedge q) \wedge r)$
- $(p \wedge (q \wedge r))$
- $((p \wedge r) \wedge q)$
- $(p \wedge (r \wedge q))$
- $((q \wedge p) \wedge r)$
- $(q \wedge (p \wedge r))$
- $((q \wedge r) \wedge p)$
- $(q \wedge (r \wedge p))$
- $((r \wedge p) \wedge q)$
- $(r \wedge (p \wedge q))$
- $((r \wedge q) \wedge p)$
- $(r \wedge (q \wedge p))$

Ces formules pourraient être encodées par une formule qui s'écrirait comme $\wedge\{p, qr\}$ pour indiquer que les arguments du connecteur $\wedge$ sont à considérer à associativité et commutativité près. En pratique, elles seront notées

$$p \wedge q \wedge r$$

Une telle formule sera lue à associativité et commutativité près. La prise en charge du "à associativité et commutativité" est déléguée aux implémentations, tandis que les calculs et algorithmes s'affranchiront de ces détails et auront comme arguments de telles formules *abstraites*. Remarquons que ces formules à associativité et commutativité près ne sont plus définies comme un ensemble de termes. Une astuce pourrait être de munir les formules d'une relation d'ordre, de sorte que les arguments pourraient donnés en accord avec cette relation d'ordre.

Définition 93.

- Une formule est dite en **forme normale de négation (f.n.n)** si toute occurrence du connecteur $\neg$ s'applique directement à des variables propositionnelles (formules atomiques).
- Un **littéral** est une variable propositionnelle ou sa négation.
 p (resp. $\neg p$) est dit littéral positif (resp. négatif).
 Pour un littéral l de la forme p (resp. $\neg p$), $\neg l$ (aussi parfois noté $\bar{l}$) est le littéral $\neg p$ (resp. p).
- Une **clause** (aussi appelée disjonction élémentaire) est de la forme $l_1 \vee l_2 \vee \dots \vee l_m$ avec l_i littéral (par associativité et commutativité de $\vee$, l'ordre des littéraux importe peu).
 La **clause vide** ($m = 0$) est la formule fausse pour toutes les P -modèles. Elle est notée $\perp$ ou $\square$.
- Une **forme normale conjonctive (f.n.c)** est de la forme $D_1 \wedge D_2 \wedge \dots \wedge D_k$ avec D_i clause (par associativité et commutativité de $\wedge$, l'ordre des clauses importe peu).
 La **conjonction vide** ($k = 0$) est la formule toujours vraie ($\top$).
- Une **conjonction élémentaire** est de la forme $l_1 \wedge l_2 \wedge \dots \wedge l_m$ avec l_i littéral (par associativité et commutativité de $\wedge$, l'ordre des littéraux importe peu).
 Le cas $m = 0$ est la formule vraie ($\top$) pour toutes les P -modèles.
- Une **forme normale disjonctive (f.n.d)** est de la forme $C_1 \vee C_2 \vee \dots \vee C_k$ avec C_i conjonction élémentaire (par associativité et commutativité de $\vee$, l'ordre des conjonctions élémentaires importe peu).
 Si $k = 0$, la disjonction vide est assimilée à une formule toujours fausse ($\perp$ ou $\square$).

Exemple :

- $(\neg p \vee r) \wedge (p \Rightarrow (\neg q \wedge p))$ est en forme normale de négation (f.n.n)
 (les négations sont accolées aux variables propositionnelles)
- $((p \vee q) \vee r) \wedge (q \vee \neg r)$ et $(p \vee (q \vee r)) \wedge (q \vee \neg r)$ sont en f.n.c.

Elles sont simplement notées $(p \vee q \vee r) \wedge (q \vee \neg r)$

(Abus de notation "commutativité" et "associativité" de $\vee$, ou plus exactement "peu importe l'ordre de parenthésages pour les différents opérateurs $\vee$ ")

$(p \vee q \vee r)$ et $(q \vee \neg r)$ sont deux clauses

- $((p \wedge q) \wedge r) \vee ((q \wedge \neg r) \vee (r \wedge \neg q))$ est en f.n.d.

Elle est simplement notée $(p \wedge q \wedge r) \vee (q \wedge \neg r) \vee (r \wedge \neg q)$

Remarque 115.

La syntaxe des formules n'est plus la même : les connecteurs $\wedge$ et $\vee$ sont désormais n -aires (et plus simplement binaires). Par ailleurs, l'ordre des arguments importe peu (ils sont considérés à commutativité et associativité près). Lorsque nous souhaiterons parler de l'ensemble de ces formules, on notera $For_n(P)$. Prendre les arguments à commutativité et associativité près semble naturel en suivant une approche mathématique, cependant toute implémentation de ces formules devra gérer le "à commutativité et associativité près".

En particulier, pour exploiter les propriétés mathématiques, les clauses ou/et conjonctions élémentaires seront aussi manipulées sous forme ensembliste lorsque le contexte sera clair. Ainsi, une clause $l_1 \vee l_2 \vee \dots \vee l_n$ sera notée $\{l_1, l_2, \dots, l_n\}$.

En forme canonique, les clauses ne contiennent pas de doublons (pas deux occurrences d'un même littéral), et pas un littéral et sa négation.

Proposition 116.

Soit φ une formule propositionnelle.

- Il existe une formule φ_n en f.n.n équivalente à φ .
- Il existe une formule φ_c en f.n.c équivalente à φ .
- Il existe une formule φ_d en f.n.d équivalente à φ .

Démonstration : Il est possible de mettre les formules sous forme f.n.n, f.n.c ou f.n.d en utilisant :

- les définitions équivalentes des connecteurs $\Leftrightarrow$ et $\Rightarrow$,
e.g. $p \Rightarrow q$ peut se réécrire en $\neg p \vee q$
- les lois de Morgan concernant la négation d'une conjonction ou d'une disjonction,
e.g. $\neg(p \wedge q)$ peut se réécrire en $(\neg p) \vee (\neg q)$
(permet de descendre les négations au plus près des variables propositionnelles)
- la loi de double négation,
e.g. $\neg(\neg p)$ peut se réécrire en p
- la distributivité des connecteurs $\vee$ et $\wedge$ l'un par rapport à l'autre

Plus exactement, pour la mise sous forme conjonctive, considérons la fonction fnc définie inductivement sur l'ensemble des formules propositionnelles φ par :

- si φ est de la forme p ou $\neg p$ avec $p \in P$, alors $fnc(\varphi)$ est φ ;
- si φ est de la forme $\neg \neg \psi$, alors $fnc(\varphi)$ est $fnc(\psi)$;
- si φ est de la forme $\psi \Rightarrow \psi'$, alors $fnc(\varphi)$ est $fnc(\neg \psi \vee \psi')$;
- si φ est de la forme $\neg(\psi \wedge \psi')$, alors $fnc(\varphi)$ est $fnc(\neg \psi \vee \neg \psi')$;
- si φ est de la forme $\neg(\psi \vee \psi')$, alors $fnc(\varphi)$ est $fnc(\neg \psi \wedge \neg \psi')$;
- si φ est de la forme $(\psi \vee (\psi' \wedge \psi''))$, alors $fnc(\varphi)$ est $fnc((\psi \vee \psi') \wedge (\psi \vee \psi''))$;
- si φ est de la forme $((\psi \wedge \psi') \vee \psi'')$, alors $fnc(\varphi)$ est $fnc((\psi \vee \psi'') \wedge (\psi' \vee \psi''))$;
- sinon, si φ est de la forme $(\psi \vee \psi')$, alors $fnc(\varphi)$ est $fnc(\psi) \vee fnc(\psi')$;
- si φ est de la forme $(\psi \wedge \psi')$, alors $fnc(\varphi)$ est $fnc(\psi) \wedge fnc(\psi')$

La fonction fnc termine car les arguments de fnc décroissent selon l'évaluation $|\cdot|$ définie par :

- $|p| = 2$ pour $p \in P$
- $|\varphi \wedge \psi| = |\varphi| + |\psi| + 1$
- $|\varphi \vee \psi| = |\varphi| \times |\psi|$
- $|\neg \varphi| = 2^{|\varphi|}$
- $|\varphi \Rightarrow \psi| = 2^{|\varphi|+|\psi|+1}$

La fonction fnc est correcte, car toutes les configurations préservent l'équivalence des formules, et par ailleurs, le symbole $\neg$ descend sous $\wedge$ et $\vee$, et le symbole $\vee$ descend sous $\wedge$. ■

Formules équisatisfiables

Considérons φ la formule de taille $\Theta(n)$ définie comme suit :

$$(p_1 \wedge q_1) \vee \dots \vee (p_n \wedge q_n)$$

On remarque : $v \models \varphi$ ssi il existe i tel que $v(p_i) = 1$ et $v(q_i) = 1$.

Soit Φ une formule en forme normale conjonctive équivalente à φ . Elle s'écrit $\bigwedge_{j=1}^m C_j$ avec C_j clause (non réduite à une tautologie).

Chacune des clauses C_j contient pour chaque i dans $1..n$ au moins l'une des deux variables p_i ou q_i (sinon on peut trouver v modèle de φ qui invalide C_j , et donc Φ).

Pour $J \subseteq \{1, \dots, n\}$, définissons v_J par

- pour i dans J $v_J(p_i) = 1$
- pour i dans $\{1, \dots, n\} \setminus J$, $v_J(q_i) = 0$.

$v_J \not\models \varphi$. Par conséquent, il existe j vérifiant $v_J \not\models C_j$.

Soit

$$\begin{aligned} ind : \mathcal{P}(\{1, \dots, n\}) &\rightarrow \{1, \dots, n\} \\ J &\mapsto \min(\{j \mid J \not\models C_j\}) \end{aligned}$$

ind est injective. Soit $J \neq K \subseteq \mathcal{P}(\{1, \dots, n\})$ avec $ind(J) = ind(K)$.

Soit i dans $J \setminus K$ (à défaut, $K \setminus J$). On a alors $v_J(p_i) = 1$ et $v_K(q_i) = 1$. Par hypothèse, on a : $v_J \not\models C_j$ (donc $p_i \notin C_j$) et $v_K \not\models C_j$ (donc $q_i \notin C_j$). Or toutes les clauses contiennent pour tout i dans $1..n$, soit p_i soit q_i . Contradiction. La fonction ind est donc injective. On en déduit que $m \geq 2^n$ (cardinal de $\mathcal{P}(\{1, \dots, n\})$).

Finalement $|\Phi| \geq 2^n$.

La transformation de Tseitin transforme une formule φ en une formule en forme normale conjonctive équisatisfiable.

- Etape 1 : pour toute sous-formule ψ de φ , introduire une nouvelle variable propositionnelle p_ψ
- Etape 2 : pour toute sous-formule ψ de la forme $\psi_1 \Delta \psi_2$, transformer la formule $p_\psi \Leftrightarrow (p_{\psi_1} \Delta p_{\psi_2})$ en une formule en fnc (de taille bornée par une constante, puisqu'elle contient 3 variables).
- Etape 3 : considérer

$$p_\varphi \wedge \bigwedge_{\psi_1 \Delta \psi_2 \in SF(\varphi)} fnc(p_{\psi_1 \Delta \psi_2} \Leftrightarrow p_{\psi_1} \Delta p_{\psi_2})$$

18.2. Algorithme DPLL

L'algorithme DPLL a été proposé par Martin Davis, Hilary Putman, George Logemann et Donald Loveland en 1962 et prend en entrée, une formule en forme normale conjonctive (FNC)

telle que chaque clause ne contienne pas de littéral en double, ni une variable et sa négation (forme réduite de la clause).

Le principe général consiste à engendrer des solutions partielles $[p \setminus b]$, complétées pas à pas jusqu'à déduire au plus tôt la satisfiabilité de la formule initiale (ou une contradiction). L'algorithme met en place une stratégie pour choisir l'ordre des valuations partielles à considérer.

Valuations partielles

Définition 94 (Application d'une valuation partielle).

Soit C une clause de la forme $\{l_1, \dots, l_n\}$ avec l_i littéral, soit p une variable propositionnelle et $b \in \{0, 1\}$ une valeur booléenne.

$C[p \setminus b]$ est la formule propositionnelle

- $\top$ si p apparaît sous la forme p dans C ($p \in C$) et si $b = 1$,
- $\top$ si p apparaît sous la forme $\neg p$ dans C ($\neg p \in C$) et si $b = 0$,
- $C \setminus \{\neg p\}$ si p apparaît sous la forme $\neg p$ dans C ($\neg p \in C$) et si $b = 1$,
- $C \setminus \{p\}$ si p apparaît sous la forme p dans C ($p \in C$) et si $b = 0$
- C si p n'apparaît pas dans C (ni sous la forme p , ni sous la forme $\neg p$).

Pour une formule φ en forme normale conjonctive $C_1 \wedge C_2 \dots \wedge C_k$, la valuation partielle de φ par $[p \setminus b]$ est $C_1[p \setminus b] \wedge C_2[p \setminus b] \dots \wedge C_k[p \setminus b]$.

Exemple 117.

Soit φ la formule $(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z)$

En considérant la valuation partielle $[x \setminus 1]$, on obtient :

$$\varphi[x \setminus 1] : (x \vee y \vee z)[x \setminus 1] \wedge (x \vee \neg y \vee \neg z)[x \setminus 1]$$

$$\varphi[x \setminus 1] : \top \wedge \top$$

$$\varphi[x \setminus 1] : \top$$

et

$$\varphi[x \setminus 0] : (y \vee z) \wedge (\neg y \vee \neg z)$$

En considérant $[y \setminus 1]$, on obtient :

$$\varphi[y \setminus 1] : (x \vee y \vee z)[y \setminus 1] \wedge (x \vee \neg y \vee \neg z)[y \setminus 1]$$

$$\varphi[y \setminus 1] : \top \wedge (x \vee \neg z)$$

$$\varphi[y \setminus 1] : (x \vee \neg z)$$

et

$$\varphi[y \setminus 0] : (x \vee z)$$

Proposition 118.

Soit C une clause, soit p une variable propositionnelle, soit b une valeur booléenne.

Alors on a $v^*(C[p \setminus b]) = (v + [p \mapsto b])^*(C)$

Stratégie de choix des variables et des valeurs booléennes

En remarquant que $\varphi \equiv \varphi[p \setminus 0] \vee \varphi[p \setminus 1]$, choisir une variable p et une valeur b consiste à explorer l'une des deux branches. La variable p choisie est appelée **variable pivot**.

Quelle stratégie pour choisir l'ordre des variables pivot et leurs valeurs booléennes associées pour définir une valuation partielle susceptible de contribuer au mieux à la décision de la satisfiabilité de la formule en jeu ?

Principe : éviter autant que possible les retours en arrière (limiter le backtracking).

Définition 95 (Clauses unitaires).

Une **clause unitaire** est une clause ne contenant qu'un seul littéral (i.e. de la forme p ou $\neg p$)

Si φ contient une clause unitaire C_u , alors choix de la valuation partielle :

- $[p \setminus 1]$ si C_u de la forme p (car $C_u[p \setminus 1]$ vaut $\top$)
- $[p \setminus 0]$ si C_u de la forme $\neg p$

(car $C_u[p \setminus 0]$ vaut $\top$)

Itérer l'application de ce critère tant qu'il reste une clause unitaire. Remarquons que l'application d'une valuation partielle $[p \setminus b]$ sur φ est susceptible de faire apparaître de nouvelles clauses unitaires dans $\varphi[p \setminus b]$.

Exemple 119.

(Propagation des clauses unitaires)

$\varphi : (x \vee y) \wedge \neg y \wedge (\neg x \vee y \vee \neg z)$

- Clause unitaire $\neg y$

$\varphi[y \setminus 0] = x \wedge (\neg x \vee \neg z)$

- Clause unitaire x

$\varphi[y \setminus 0][x \setminus 1] = \neg z$

- Clause unitaire $\neg z$

$\varphi[y \setminus 0][x \setminus 1][z \setminus 0] = \top$

L'application des valuations partielles induites par le critère de clauses unitaires préserve strictement la satisfiabilité (i.e. φ et $\varphi[p \setminus b]$ sont satisfiables en même temps)

Définition 96.

Variable avec une seule polarité

Une variable p apparaît avec **une seule polarité** dans $\varphi : C_1 \wedge \dots \wedge C_n$ si les occurrences de p sont

- soit toutes positives (i.e. au sein de littéraux positifs), on parle alors de **polarité positive**
- soit toutes négatives (i.e. au sein de littéraux négatifs), on parle alors de **polarité négative**.

En présence d'une variable p avec une seule polarité dans φ

- appliquer $[p \setminus 1]$ si la polarité est positive
- appliquer $[p \setminus 0]$ si la polarité est négative

Exemple 120.

$\varphi : (x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg z \vee \neg w)$

- x est de polarité positive

$\varphi[x \setminus 1] : (y \vee \neg z) \wedge (\neg z \vee \neg w)$

- y est de polarité positive

$\varphi[x \setminus 1][y \setminus 1] : (\neg z \vee \neg w)$

- z est de polarité négative

$\varphi[x \setminus 1][y \setminus 1][z \setminus 0] : \top$

La satisfiabilité est préservée par choix de variable avec une unique polarité.

Algorithme DPLL

Choix d'une valuation partielle en suivant dans l'ordre les critères

- Critère de clause unitaire
- Critère de variable de même polarité
- A défaut, choisir une variable et une valeur booléenne au hasard (risque de backtrack si le choix échoue à exhiber une valuation)

Algorithme DPLL

entrée : une formule FNC φ sortie : valeur booléenne (0 ou 1)

1. Si $\varphi = \top$, retourner 1
2. Si $\varphi = \perp$ (ou simplement contient la clause $\perp$), retourner 0
3. Si φ contient une clause unitaire de la forme p , retourner $\text{DPLL}(\varphi[p \setminus 1])$
4. Si φ contient une clause unitaire de la forme $\neg p$, retourner $\text{DPLL}(\varphi[p \setminus 0])$
5. s'il existe une variable p de polarité positive, retourner $\text{DPLL}(\varphi[p \setminus 1])$
6. s'il existe une variable p de polarité négative, retourner $\text{DPLL}(\varphi[p \setminus 0])$
7. sinon, choisir une variable p de φ et retourner $\text{DPLL}(\varphi[p \setminus 0])$ ou $\text{DPLL}(\varphi[p \setminus 1])$

Vers les SAT solveurs

Les SAT solveurs implémentent l'algorithme DPLL . . . mais en pratique, avec beaucoup d'autres optimisations, structures de données, heuristiques.

18.3. Résolution

Quelques rappels et notations

Notations 121.

- Pour un littéral l , $\bar{l}$ désigne le littéral équivalent à $\neg l$.
Si l est de la forme p (resp. $\neg p$), $\bar{l}$ est $\neg p$ (resp. p).
- Une clause $l_1 \vee l_2 \dots \vee l_n$ est parfois notée sous forme ensembliste $\{l_1, l_2, \dots, l_n\}$
- Si les variables sont ordonnées $x_1, x_2 \dots x_n$, alors une clause peut être dénotée par la séquence des indices, surlignés ou pas, présents dans la clause.
Ainsi, la clause $x_2 \vee \neg x_4 \vee x_5 \vee \neg x_1$ est notée de la façon suivante : $\bar{1} \ 2 \ \bar{4} \ 5$.
Une autre convention est de précéder les littéraux négatifs du signe moins ($-$) :
ainsi la clause $x_2 \vee \neg x_4 \vee x_5 \vee \neg x_1$ est aussi dénotée $-1 \ 2 \ -4 \ 5$
- $l \vee C$ désigne une clause C' contenant le littéral l , C désignant alors la clause C' privée d'une occurrence d'un littéral l (éventuellement réduite à $\perp$).
- une forme normale conjonctive $D_1 \wedge D_2 \wedge \dots \wedge D_k$ est aussi notée sous forme ensembliste $\{D_1, D_2, \dots, D_k\}$.

Simplifications

- Si une clause contient plusieurs occurrences d'un même littéral, ne garder qu'une occurrence de ce littéral.

Ainsi, $x_1 \vee x_2 \vee \neg x_3 \vee x_1$ peut se réécrire en $x_1 \vee x_2 \vee \neg x_3$

- Si une clause contient un littéral et sa négation, la clause peut se ramener à la tautologie $\top$.

Ainsi, $x_1 \vee x_2 \vee \neg x_3 \vee \neg x_1$ peut se réécrire en $\top$.

(les formules réécrites sont équivalentes aux formules initiales : il y a donc préservation des valeurs de vérité)

Définition 97 (Valuation partielle).

Une valuation partielle, dénotée $[p \setminus v]$ avec p variable propositionnelle et v valeur de $\mathbb{B} = \{0, 1\}$ associe une valeur de vérité à une (seule) variable propositionnelle.

L'application d'une valuation partielle $[p \setminus v]$ à une formule φ , dénotée $\varphi[p \setminus v]$, est une formule propositionnelle ne contenant pas la variable propositionnelle p .

Soit φ une formule, en forme normale conjonctive (FNC) ou sous forme de clause.

- si φ est une clause $p \vee C$, alors $\varphi[p \setminus 0]$ est $C[p \setminus 0]$;
- si φ est une clause $p \vee C$, alors $\varphi[p \setminus 1]$ est $\top$;
- si φ est une clause $\neg p \vee C$, alors $\varphi[p \setminus 0]$ est $\top$;
- si φ est une clause $\neg p \vee C$, alors $\varphi[p \setminus 1]$ est $C[p \setminus 1]$;
- si φ est la clause vide $\perp$, alors $C[p \setminus v]$ est $\perp$;
- si φ est une clause $q \vee C$ avec $p \neq q$, alors $\varphi[p \setminus v]$ est $C[p \setminus v]$;
- si φ est une conjonction de clauses $C_1 \wedge \dots \wedge C_n$, alors $\varphi[p \setminus v]$ est $C_1[p \setminus v] \wedge \dots \wedge C_n[p \setminus v]$ avec la convention que $\varphi[p \setminus v]$ vaut $\top$ (resp. $\perp$) si pour tout i , $C_i[p \setminus v] = \top$ (resp. $\exists i, C_i[p \setminus v] = \perp$).

Proposition 122.

Soit φ une formule et p une variable propositionnelle.

φ est satisfiable si et seulement si $\varphi[p \setminus 1]$ est satisfiable ou $\varphi[p \setminus 0]$ est satisfiable
(p est appelée *variable pivot*)

Définition 98 (Clauses et formules de Horn).

Une clause $l_1 \vee l_2 \vee \dots \vee l_m$ est dite **clause de Horn** ssi il existe au plus un unique littéral l_i (avec $i \in \{1, \dots, m\}$) tel que l_i s'écrit p (l_i est un littéral positif).

Une **formule de Horn** est une conjonction de clauses de Horn.

Exemple 123.

- $\neg p \vee q \vee \neg r$, $\neg p \vee \neg r$, q sont des clauses de Horn
- $(\neg p \vee \neg q \vee \neg r \vee s) \wedge (p \vee \neg r)$ est une formule de Horn

Notations 124.

$\neg p \vee q \vee \neg r$ peut être écrite :

- $p \wedge r \rightarrow q$
- $p, r \rightarrow q$
- $q : \neg p, r$
- $q \leftarrow p, r$

(on peut alors avoir des formes réduites comme $p, q \rightarrow$ ou $\rightarrow q$)

Satisfiabilité des formules de Horn**Algorithme**

Entrée $F = \{H_1 \dots H_m\}$ formule de Horn avec H_i clause de Horn

Sortie F satisfiable ou F insatisfiable

1. Marquer toutes les variables propositionnelles à $\perp$ ($m(p) = \perp$)
2. Pour toutes les clauses de Horn H_i de la forme $\rightarrow p$, faire $m(p) = \top$
3. Tant que :
 - il existe une clause de Horn de la forme $p_1, \dots, p_m \rightarrow p$ avec pour tout i , $m(p_i) = \top$, avec $m(p) = \perp$, faire $m(p) = \top$.
 - ou il existe une clause de Horn de la forme $p_1, \dots, p_m \rightarrow$ avec pour tout i , $m(p_i) = \top$, retourner F insatisfiable
(Ce cas inclut le cas de la clause vide $\rightarrow$)
4. Retourner F satisfiable

Proposition 125.

Les propriétés suivantes sont vérifiées :

- L'algorithme termine
- Tout modèle v de F (i.e. pour tout H_i dans F , $v \models H_i$) vérifie :

$$\text{pour tout } p \in P, \text{ si } m(p) = \top \text{ alors } v(p) = 1$$

(avec m fonction de marquage des variables propositionnelles utilisée dans l'algorithme)

- L'algorithme est correct (l'algorithme retourne " F est satisfiable" ssi f est satisfiable).

Démonstration : — Terminaison de l'algorithme

Il n'y pas plus de n passages dans la boucle "tant que" avec n le nombre de variables propositionnelles. En effet, à chaque passage dans la boucle, une variable est marquée, ou bien il y a sortie de boucle.

Remarquons aussi qu'une clause est traitée au plus une fois dans la condition de boucle.

- Tout modèle v de F est tel que pour toute variable propositionnelle p vérifiant $m(p) = \top$, on a $v(p) = 1$

Démonstration par induction sur les étapes de l'algorithme. Soit v un modèle de F , on a alors pour tout H_i , $v \models H_i$.

- Etape (ii)
Les clauses H_i de la forme $\rightarrow p$, sont telles que $m(p) = \top$. Or $v \models H_i$, donc $v(p) = 1$
- Etape (iii)
Hypothèse d'induction : propriété vraie jusqu'au k ème passage dans la boucle.
Au $k + 1$ ème passage :
 - il existe une clause de Horn $H : p_1, \dots, p_m \rightarrow p$ avec pour tout i , $m(p_i) = \top$ avec $m(p) = \perp$ (permettant de marquer p à $\top$). Par hypothèse, pour tout i , $v(p_i) = 1$. Or $v \models H$, il en découle $v(p) = 1$
 - ou il existe une clause de Horn $H : p_1, \dots, p_m \rightarrow$ avec pour tout i , $m(p_i) = \top$. Par hypothèse pour tout i , $v(p_i) = 1$. Ce cas n'est pas possible, car $v \models H$ par hypothèse.
 Toutes les variables p marquées vérifient donc $v(p) = 1$.
- Correction de l'algorithme
 - Si l'algorithme retourne "F insatisfiable", cela veut dire qu'il y a une clause $H : p_1, \dots, p_m \rightarrow$ avec pour tout p_i , $m(p_i) = \top$.
S'il existait un modèle v de F , alors pour tout i , $v(p_i) = 1$ (cf propriété précédente), ce qui contredirait le fait que $v \models H$. Donc F est bien insatisfiable.
 - Si l'algorithme retourne "F satisfiable", considérons le modèle v vérifiant pour tout p , $(m(p) = \top \Rightarrow v(p) = 1)$ and $(m(p) = \perp \Rightarrow v(p) = 0)$.
Soit H une clause de la forme :
 - $\rightarrow p$, alors $v(p) = 1$ et $v \models H$
 - $p_1, \dots, p_m \rightarrow p$. Si pour tout p_i , $v(p_i) = 1$, alors $v(p) = 1$ (car $v(p) = \top$) et $v \models H$.
Si $\exists p_i, v(p_i) = 0$, alors $v \models C$ (peu importe la valuation $v(p) = 0$).
 - $p_1, \dots, p_m \rightarrow$ Le cas "pour tout p_i , $v(p_i) = 1$ " est impossible (car l'algorithme retourne "F insatisfiable" dans ce cas).
Donc $\exists p_i, v(p_i) = 0$ et $v \models H$.
 - $\rightarrow$. Ce cas n'est pas possible (car l'algo retourne alors "F insatisfiable").
 Donc l'algorithme est correct. ■

Remarque 126.

- Les formules de Horn se prêtent à un traitement algorithmique.
- Alors que toute formule peut être réécrite en une formule logiquement équivalente, en f.n.c, ce n'est pas vraie pour les formules de Horn (cf $p \vee r \vee \neg q \vee \neg s$).

Résolvante et preuve par résolution

Définition 99 (Résolvante).

Soient $C' = C \vee p$ et $D' = D \vee \neg p$ deux clauses.
La clause $C \vee D$ est une **résolvante** de C' et D' .
(par rapport à la variable propositionnelle p , dite **variable pivot**).

Exemple 127.

Les clauses $\neg p \vee q \vee r$ et $p \vee \neg q \vee r$ ont deux résolvantes, respectivement :

- $q \vee r \vee \neg q \vee r$ (par rapport à p)
- $\neg p \vee r \vee p \vee r$ (par rapport à q).

Définition 100 (Système formel de la résolution).

Les formules concernées par système formel de la résolution sont les clauses.
Les règles de déduction sont :

$$\frac{C \vee \neg p \quad D \vee p}{C \vee D} [RES]$$

$$\frac{C \vee l \vee l}{C \vee l} [FACT]$$

avec C et D clauses, l littéral, et p variable propositionnelle.

La règle RES est la **règle de résolution**, et les littéraux p et $\neg p$ sont dits annulés par RES.

La règle FACT permet d'éliminer les répétitions de littéraux (qui peuvent apparaître en considérant $C \vee D$, issue de la règle RES, avec C et D deux clauses quelconques).

Exemple 128.

$$\frac{\frac{\neg p \vee s \quad \neg s \vee l}{\neg p \vee l} \quad \neg l \vee q}{\neg p \vee q}$$

est une déduction de $\neg p \vee q$ à partir des clauses $\{\neg p \vee s, \neg s \vee l, \neg l \vee q\}$.

On note alors $\{\neg p \vee s, \neg s \vee l, \neg l \vee q\} \vdash_{RES} \neg p \vee q$.

Théorème 129 (Correction).

Le système de preuve par résolution est correct.

Autrement dit, pour tout ensemble Γ de clauses, pour toute clause C , si $\Gamma \vdash_{Res} C$ alors $\Gamma \models C$.

Démonstration : Il suffit de montrer que chacune des deux règles RES et FACT sont correctes, i.e. préservent la véracité des formules.

Trivial pour la règle FACT

Considérons la règle RES.

Soit $C \vee \neg p$ et $D \vee p$ deux clauses, et soit v une valuation vérifiant $v \models C \vee \neg p$ et $v \models D \vee p$ (i.e. $v^*(C \vee \neg p) = 1$ et $v^*(D \vee p) = 1$).

Analyse par cas : soit $v(p) = 1$, soit $v(p) = 0$.

- si $v(p) = 1$, alors $v \models C$ (car $v \models C \vee \neg p$ avec $v^*(\neg p) = 0$)
et par conséquent $v \models C \vee D$
- si $v(p) = 0$, alors $v \models D$
et par conséquent $v \models C \vee D$

Donc la règle de résolution préserve la satisfiabilité (simultanée des deux prémisses) ■

Définition 101 (Réfutation).

Soit E un ensemble de clauses.

Une **réfutation** (par résolution) est une déduction par résolution menant à la clause vide $\square$ (ou $\perp$).

On note alors : $E \vdash_{RES} \square$

Théorème 130 (Preuve par réfutation).

S'il existe une réfutation à partir d'un ensemble E de clauses, alors E est insatisfiable.

Démonstration : Conséquence directe du théorème de correction du système de preuve par résolution. ■

Exemple 131.

A partir de l'ensemble

$$\{\neg p \vee s, \neg s \vee l, \neg l \vee q, \neg q, p\}$$

on peut construire la réfutation suivante :

$$\frac{\frac{\frac{\neg p \vee s \quad \neg s \vee l}{\neg p \vee l} \quad \neg l \vee q}{\neg p \vee q} \quad \neg q}{\neg p} \quad p \quad \square$$

Soit $\{\neg p \vee s, \neg s \vee l, \neg l \vee q, \neg q, p\} \vdash_{RES} \square$

Complétude de la réfutation par résolution

Théorème 132.

Soit E est un ensemble de clauses.

Si E est un ensemble insatisfiable de clauses, alors $E \vdash_{RES} \square$.

Définition 102 (Rencontres minimales).

Soit $\mathcal{E}$ un ensemble d'ensembles $C_1, C_2, \dots, C_k$.

Un ensemble J

- **rencontre** $\mathcal{E}$, si $\forall C \in \mathcal{E}, C \cap J \neq \emptyset$
- **rencontre minimalement** $\mathcal{E}$, si tout sous-ensemble strict de J ne rencontre pas $\mathcal{E}$.

On dit alors que J est une rencontre minimale de $\mathcal{E}$.

Remarquons que :

- pour tout ensemble $\mathcal{E}$ d'ensembles C non vides, $\cup C$ rencontre $\mathcal{E}$.
- tout élément de $\mathcal{H} = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$ rencontre minimalement l'ensemble $\mathcal{H}$.
- pour $\mathcal{J} = \{C_n \mid n \in \mathbb{N}\}$ avec $C_n = \{n, n+1, n+2, \dots\}$, $\mathcal{J}$ n'a pas d'ensemble rencontrant minimalement $\mathbb{N}$.

Proposition 133.

Soit J rencontrant $\mathcal{E}$.

Les propriétés suivantes sont équivalentes :

1. J rencontre minimalement $\mathcal{E}$
2. $\forall a \in J, \exists C \in \mathcal{E}, J \cap C = \{a\}$

Démonstration : — $(i) \Rightarrow (ii)$. Supposons que (ii) n'est pas vérifié.

Il existe alors a dans J tel que $\forall C \in \mathcal{E}, \exists c \in C, \{a, c\} \subseteq J \cap C$.

Autrement dit, $J - \{a\}$ rencontre $\mathcal{E}$.

- $(ii) \Rightarrow (i)$. Si a est élément de J , alors $J - \{a\}$ ne rencontre pas $\mathcal{E}$.
(car il existe un élément C de $\mathcal{E}$ tel que $(J - \{a\}) \cap C = \emptyset$)

■

Lemme 134.

Soit X un ensemble dénombrable d'éléments.

Soit (C_i) une famille d'ensemble finis inclus dans X .

L'ensemble $\mathcal{E}$ des ensembles C_i possède une rencontre minimale.

Démonstration : Considérons $x_0, x_1, \dots, x_k, \dots$ une énumération des éléments de X

Considérons la suite J_i suivante :

- $J_0 = X$. J_0 rencontre $\mathcal{E}$.
- J_{i+1} est
 - J_i , si $J_i - \{x_i\}$ ne rencontre pas $\mathcal{E}$
 - $J_i - \{x_i\}$ si $J_i - \{x_i\}$ rencontre $\mathcal{E}$

Par construction tous les ensembles J_i rencontrent $\mathcal{E}$.

Soit $J = \cap_i J_i$.

J rencontre $\mathcal{E}$: comme pour tout $C \in \mathcal{E}$ est fini, il n'est pas possible de retirer tous les éléments de C (le dernier, selon l'énumération, ne peut être retiré, si les autres ont déjà été enlevés)

J rencontre minimalement $\mathcal{E}$: par construction, aucun élément x_i ne peut être retiré. ■

Démonstration : (théorème de complétude)

Démonstration par l'absurde :

Supposons que $\square$ n'est pas dérivable à partir de E ensemble insatisfiable de clauses (et cherchons à construire un modèle v rendant E satisfiable).

Soit Δ l'ensemble de toutes les clauses dérivables à partir de E .

Soit $\mathcal{E}$ l'ensemble des clauses de Δ ne contenant que des littéraux positifs. $\mathcal{E}$ est un ensemble d'ensembles finis, non vides (car $\square$ n'est pas dérivable).

Considérons J une rencontre minimale de $\mathcal{E}$ (existe d'après le lemme précédent).

Soit v le modèle défini par $v(p) = 1$ si p dans J , et $v(p) = 0$ sinon. Par définition $v \models \mathcal{E}$.

Montrons que $v \models \Delta$.

Soit $C \in \Delta$.

Montrons $v \models C$ par induction sur le nombre de littéraux négatifs de C .

— Pas de littéraux négatifs : $C \in \mathcal{E}$ et $v \models C$

— Soit C contenant un littéral négatif $\neg p$.

Si $p \notin J$, alors $v \models C$.

Supposons alors que $p \in J$. Par propriété, il existe $D \in \mathcal{E}$, tel que $D \cap J = \{p\}$ (première propriété).

Considérons la résolvante :

$$R : (C - \{\neg p\}) \cup (D - \{p\})$$

R appartient à Δ et a un littéral négatif de moins que C .

Par hypothèse d'induction, on a $v \models R$. Donc v satisfait l'un des littéraux de R . Cela ne peut être de $D - \{p\}$, car toutes les variables en dehors de J sont évaluées à 0. Donc il existe un littéral de $C - \{\neg p\}$ qui est évalué à 1 par v .

Soit $v \models C - \{\neg p\}$, et donc $v \models C$.

par conséquent $\forall C \in \Delta, v \models C$, comme $E \subseteq \Delta$, E est satisfiable. Contradiction avec l'hypothèse F insatisfiable. ■

SOUS PARTIE 5

Logique des prédicats

19. Introduction

Le calcul propositionnel est mal adapté pour pratiquer la pensée déductive étudiée depuis l'antiquité. Elle donne simplement la structure générale (i.e., son squelette) de tout raisonnement déductif mais sans aucune indication sur les objets manipulés par cette déduction. Par exemple, si l'on veut modéliser le raisonnement suivant :

Tous les hommes sont mortels
Socrate est un homme
Donc, Socrate est mortel

Le calcul propositionnel nous permet simplement au moyen des trois variables propositionnelles p , q et r représentant les trois phrases ci-dessus, de modéliser le raisonnement par la simple structure suivante :

$$\{p, q\} \vdash r$$

Cependant, on ne voit pas les relations sous-jacentes qui lient ces variables propositionnelles. La logique des prédicats du premier ordre répond à ce manque de description en augmentant son lexique linguistique. Maintenant, ce dernier va contenir en plus des variables propositionnelles, des variables d'individu, $x, y, z \dots$ mises pour des objets de nature indéterminée, des fonctionnalités et des relations sur ces objets, ainsi que des quantificateurs existentiel ($\exists$) et universel ($\forall$) pour effectuer des "jugements" sur les objets manipulés. Un tel langage va alors nous permettre de faire une division des propositions en sujets et prédicats. Pour revenir à notre exemple de départ, on pourra alors considérer un symbole de prédicat $H(x)$ qui intuitivement veut dire *x est un homme*, un symbole fonctionnel *socrate* pour désigner l'individu *Socrate* et un symbole de prédicat $M(x)$ pour signifier que *x est mortel*. Le raisonnement précédent se modélisera alors par :

$$\{\forall x, (H(x) \Rightarrow M(x)), H(socrate)\} \vdash M(socrate)$$

où " $\forall$ " se lit *quel que soit*. Maintenant, en spécialisant l'énoncé $\forall x, H(x) \Rightarrow M(x)$ à l'individu "socrate", à partir de la connaissance $H(socrate)$, on peut déduire par Modus-Ponens l'énoncé $M(socrate)$.

Comme pour la logique propositionnelle, la logique des prédicats du premier ordre va suivre dans sa présentation le découpage en une syntaxe, une sémantique et un calcul. Les sections suivantes vont alors présenter ces trois aspects de la logique des prédicats. Après cette présentation, nous aborderons les limitations de la logiques des prédicats du premier ordre dans son pouvoir d'expression mais aussi dans ses possibilités d'automatisation.

20. Logique des prédicats : Syntaxe

20.1. Signatures

Une signature du premier ordre sera donc définie par un ensemble de constantes, de fonctions et de prédicats, et aussi un ensemble de variables pour dénoter des individus génériques sur lesquels on exprimera des propriétés d'existence ou d'universalité.

Définition 103 (Signature).

Une **signature** Σ est un triplet $(\mathcal{F}, \mathcal{R}, \mathcal{V})$ où :

- $\mathcal{F}$ est un ensemble de noms de fonction, chacun muni d'un entier naturel appelé son *arité*. Dans la suite, pour désigner un nom de fonction f munie d'une arité n , nous noterons : f^n . On appelle toute fonction munie d'une arité nulle, une **constante**.
- $\mathcal{R}$ est un ensemble de noms de prédicat, chacun muni d'un entier naturel non nul appelé son *arité*. Dans la suite, pour désigner un nom de prédicat r muni d'une arité n , nous noterons : r^n .
- $\mathcal{V}$ est un ensemble dont les éléments sont appelés des *variables* tel que aucun des noms de constantes, de fonctions, et de prédicats n'appartiennent à cet ensemble.

Comme exemples de signature, on peut considérer :

1. la signature utilisée pour spécifier l'arithmétique usuelle (appelée aussi l'arithmétique de Peano), et définie par une constante 0, un symbole de fonction unaire *succ*, un symbole de fonction binaire $+$ pour désigner la somme entre entiers, et un symbole de prédicat binaire “=” pour dénoter l'égalité usuelle entre entiers. Si nous notons Σ_{Peano} cette signature, nous avons alors : $\Sigma_{Peano} = (\mathcal{F}_{Peano}, \mathcal{R}_{Peano}, \mathcal{V}_{peano})$ où $\mathcal{F}_{Peano} = \{0^0, succ^1, +^2\}$, $\mathcal{R}_{Peano} = \{=^2\}$, et $\mathcal{V}_{peano}$ est l'ensemble des lettres minuscules.
2. la signature utilisée pour spécifier l'arithmétique de Presburger¹. À la différence de la signature Σ_{Peano} , nous avons deux constantes 0 et 1, un unique symbole de fonction $+$ d'arité 2, et deux symboles de prédicat d'arité 2 : “<” pour désigner la relation d'ordre totale sur $\mathbb{N}$, et “=” pour l'égalité entre entiers. On obtient la signature Σ_{Presb} suivante : $\Sigma_{Presb} = (\mathcal{F}_{Presb}, \mathcal{R}_{Presb}, \mathcal{V}_{Preb})$ où $\mathcal{F}_{Presb} = \{0^0, 1^0, +^2\}$, $\mathcal{R}_{Presb} = \{<^2, =^2\}$, et $\mathcal{V}_{Presb}$ est l'ensemble des lettres minuscules.
3. enfin, la signature associée à la théorie des ensembles de Zermelo-Fraenkel (ZF), composée simplement des deux symboles de prédicats binaires (i.e., munies d'une arité 2) : “=” pour dénoter le symbole d'égalité habituel entre ensembles, et “ $\in$ ” pour dénoter l'appartenance.

1. L'arithmétique de Presburger est une théorie décidable, c'est-à-dire comme nous l'avons vu dans la première partie de ce cours, qu'il existe un algorithme pour décider si une formule est valide ou non pour cette théorie. À l'inverse, l'arithmétique usuelle de Peano dont la signature est donnée ci-dessus, a été démontrée indécidable (fameuse conséquence du non moins fameux théorème d'incomplétude de Godel).

On a alors la signature Σ_{ZF} suivante : $\Sigma_{ZF} = (\mathcal{F}_{ZF}, \mathcal{R}_{ZF}, \mathcal{V}_{ZF})$ où $\mathcal{F}_{ZF} = \emptyset$, $\mathcal{R}_{ZF} = \{=^2, \in^2\}$, et $\mathcal{V}_{ZF}$ est l'ensemble des lettres minuscules.

Remarque 135.

Dans la suite, tout symbole de prédicat binaire (i.e., d'arité 2) sera notée sous sa forme infixe.

20.2. Termes et formules

Nous avons tous les ingrédients pour définir la notion de formules pour la logique du premier ordre. Comme on l'a vu dans l'introduction de ce chapitre, la logique des prédicats a pour but de décrire formellement des propriétés sur des individus. Pour désigner ces individus, nous avons alors besoin d'un premier élément syntaxique, les **termes avec variables**. Ces termes avec variables vont être la représentation syntaxique et générique des valeurs potentiellement manipulées dans nos modèles.

Définition 104 (Termes avec variables).

Étant donnée une signature $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$, l'ensemble $T_\Sigma(\mathcal{V})$ des **termes avec variables dans $\mathcal{V}$** de la signature Σ est le plus petit ensemble (au sens de l'inclusion) tel que :

- il contient toutes les variables et les symboles de constantes (c-à-d., l'ensemble $\mathcal{V}$ et les symboles de fonctions d'arité 0),
- pour chaque symbole de fonction $f \in \mathcal{F}$ d'arité $n \neq 0$, et chaque tuple $(t_1, \dots, t_n)$ de $T_\Sigma(\mathcal{V})^n$, $f(t_1, \dots, t_n)$ est un terme de $T_\Sigma(\mathcal{V})$.

On note $T_\Sigma(\emptyset)$ l'ensemble des termes sans variables, appelés aussi **termes clos**.

Nous pouvons maintenant aborder la notion de formules sur une signature Σ . Ces dernières sont définies de façon inductive à partir de l'ensemble de base des formules dites *atomiques*, des connecteurs propositionnels, et des quantificateurs existentiel ($\exists$) et universel ($\forall$).

Définition 105 (Formule atomique).

Étant donnée une signature $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$, une **Σ -formule atomique** est une expression de la forme $r(t_1, \dots, t_n)$ où r est un symbole de prédicats d'arité n et $(t_1, \dots, t_n)$ un tuple de $T_\Sigma(\mathcal{V})^n$.

À partir des formules atomiques, nous pouvons définir les formules plus générales.

Définition 106 (Formule).

Étant donnée une signature Σ dont l'ensemble des variables est $\mathcal{V}$, l'ensemble $For(\Sigma)$ des **Σ -formules bien-formées au-dessus de Σ** est le plus petit ensemble (au sens de l'inclusion) tel que :

- il contient toutes les formules atomiques,
- à chaque fois qu'il contient deux mots φ et ψ , il contient également :

$$\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Rightarrow \psi$$

- à chaque fois qu'il contient un mot φ , contient pour toute variable $x \in \mathcal{V}$:

$$\forall x.\varphi, \exists x.\varphi$$

Exemple 136.

Toujours à partir des trois signatures développées dans la section 20.1, nous pouvons considérer les exemples de formules suivantes :

- pour la signature associée à l'arithmétique de Peano, on a les axiomes suivants :
 - 0 n'est successeur de personne : $\forall x. \neg(succ(x) = 0)$
 - tout individu autre que 0 est successeur de quelqu'un :

$$\forall x. \exists y. (\neg(x = 0) \Rightarrow succ(y) = x)$$

- *succ* est une application injective : $\forall x. \forall y. (succ(x) = succ(y) \Rightarrow x = y)$
- définition de + via les constructeurs *succ* et 0 :
 - $\forall x. (x + 0 = x)$
 - $\forall x. \forall y. (x + succ(y) = succ(x + y))$

Enfin, nous devons ajouter, la formule suivante :

$$(\varphi(x/0) \wedge \forall x. (\varphi(x) \Rightarrow \varphi(x/succ(x)))) \Rightarrow \forall x. \varphi(x)$$

Ce dernier schéma de formules décrit une induction pour la formule φ .^a

- pour la signature associée à l'arithmétique de Presburger, on a les axiomes suivants :
 - 0 est neutre pour l'addition : $\forall n. (n + 0 = n)$
 - “+” est associatif : $n + (m + p) = (n + m) + p$
 - “<” est un ordre strict, total et discret :
 - “<” est un ordre strict et total :
 - Anti-réflexif : $\forall n. \neg(n < n)$
 - transitif : $\forall x. \forall y. \forall z. (x < y \wedge y < z \Rightarrow x < z)$
 - anti-symétrique : $\forall x. \forall y. \neg(x < y \wedge y < x)$
 - total : $\forall x. \forall y. (x < y \vee y < x \vee x = y)$
 - “<” est discret :
 - < n'est pas dense : $\forall x. \exists y. (x < y \wedge \forall z. (x < z \Rightarrow (y < z \vee y = z)))$
 - tout élément excepté 0 a un unique prédécesseur :

$$\forall x. \forall y. (y < x \Rightarrow \exists z. \forall w. (z < x \wedge (z < w \Rightarrow x < w \vee x = w)))$$

- tout élément est plus petit que son successeur direct : $\forall n. n < n + 1$
- pour la signature associée à la théorie des ensembles de ZF, on a parmi tous les axiomes de cette théorie, les formules suivantes :
 - **L'axiome d'extensionnalité**

$$\forall x. \forall y. (\forall z. (z \in x \Leftrightarrow z \in y) \Rightarrow x = y)$$

Cette formule dit simplement que deux ensembles x et y composés des mêmes éléments sont égaux entre eux.

- **L'axiome de la réunion**

$$\forall x. \exists y. \forall z. (z \in y \Leftrightarrow \exists w (w \in x \wedge z \in w))$$

Cette formule assure l'existence d'un ensemble y (qui par l'axiome d'extensionnalité est unique) comme étant la réunion de tous les ensembles de l'ensemble x (c-à-d., $y = \bigcup_{w \in x} w$).

- **L'axiome des parties**

$$\forall x. \exists y. \forall z. (z \in y \Leftrightarrow \forall w (w \in z \Rightarrow w \in x))$$

Cette formule affirme qu'étant donné un ensemble x , il existe un ensemble y dont les éléments sont exactement les parties de x .

^a. Ce principe n'est pas exprimable dans le 1er ordre. Ceci vient du fait que ne pouvons pas quantifier sur les formules au premier ordre (ceci est une particularité de la logique dite du *second ordre*), ce qui nous oblige à considérer une infinité d'axiomes, un pour chaque formule φ .

21. Sémantique

21.1. Modèles ou structures du premier ordre

Un modèle (ou structure) donne un **sens mathématique** aux composantes de notre signature. Il est décrit simplement au moyen d'un ensemble muni d'opérations internes (ou applications) et de relations avec éventuellement ce que l'on a coutume d'appeler des "éléments distingués"¹. L'ensemble caractérisera le **domaine des individus**, les lois internes donneront une signification mathématique (ou sémantique) aux fonctions et les relations aux symboles de prédicats de la signature.

Définition 107 (Modèles).

Étant donnée une signature $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$, un **modèle au dessus de Σ** ou encore un Σ -modèle $\mathcal{M}$, est la donnée d'un ensemble M muni :

- pour chaque symbole de fonction f d'arité n , d'une application $f^{\mathcal{M}} : M^n \rightarrow M$,
- pour chaque symbole de prédicat r d'arité n , d'un sous-ensemble $r^{\mathcal{M}}$ de M^n .

a. $M^n = \underbrace{M \times \dots \times M}_{n \text{ fois}}$ et $M^0 = \{ \mathbb{I} \}$ où $\mathbb{I}$ est neutre pour " $\cdot$ " (c-à-d., $(a_1, \dots, \mathbb{I}, \dots, a_m) = (a_1, \dots, a_m)$)

1. En mathématique, on parle alors de structures.

Exemple 137.

À partir des exemples donnés dans la section 20.1, on peut considérer les structures suivantes :

- pour la signature associée à l'arithmétique de Peano, le modèle $\mathcal{M}_{Peano}$:

$$\mathcal{M}_{Peano} = \langle \mathbb{N}, 0_{\mathbb{N}}, succ_{\mathbb{N}}, +_{\mathbb{N}}, =_{\mathbb{N}} \rangle$$

Le modèle $\mathcal{M}_{Peano}$ est celui auquel on pense comme modèle canonique de l'arithmétique usuelle (et dont les axiomes vont être donnés dans la section 20.2). Cependant, il n'est pas unique ^a. On établira dans la suite ce chapitre, un résultat, le **théorème de Loweinheim-Skolem**, qui confirmera la non-unicité des modèles.

- pour la signature associée à l'arithmétique de Presburger, le modèle $\mathcal{M}_{Presb}$:

$$\mathcal{M}_{Presb} = \langle \mathbb{N}, 0_{\mathbb{N}}, 1_{\mathbb{N}}, +_{\mathbb{N}}, =_{\mathbb{N}}, <_{\mathbb{N}} \rangle$$

- pour la signature de la théorie des ensembles selon ZF , le modèle $\mathcal{M}_{ZF}$ dont le domaine M_{ZF} est défini inductivement par :

$$M_0 = E, \quad M_n = \mathcal{P}(M_{n-1}), \quad M_{ZF} = \bigcup_n \mathcal{P}(M^n)$$

$\mathcal{P}(M)$ dénote l'ensemble des parties de M , E est un ensemble, et muni de l'égalité usuelle entre ensembles pour le symbole de prédicat "=", et de la relation $\in^{\mathcal{M}}$ suivante :

$x \in^{\mathcal{M}} y$ si et seulement s'il existe $n \in \mathbb{N}$ tel que $x \in M_n$, $y \in M_{n+1}$, et $x \in y$ ^b
on a alors : $\forall y \in E, \forall x \in M, x \notin^{\mathcal{M}} y$

^a. Ceci est aussi vrai pour la théorie des ensembles de ZF.

^b. Attention, ici, $\in$ désigne l'appartenance usuelle dans les ensembles. $\in^{\mathcal{M}}$ est le sens mathématique donné au symbole de prédicat " $\in^2$ " de la signature Σ_{ZF} .

21.2. Validation des formules

La syntaxe de la logique des prédicats du premier ordre étant plus riche, la validation des formules sera un peu plus compliquée à définir que celle pour la logique propositionnelle. En effet, avant de donner une valeur de vérité à nos formules, nous devons tout d'abord donner un sens mathématique aux premiers éléments syntaxiques de ces dernières que sont les termes. Ceci va se faire via la notion de *substitution* (ou **interprétation**) des variables dans le domaine d'un modèle $\mathcal{M}$ donné.

Définition 108 (Interprétation).

Étant donnés une signature Σ dont l'ensemble des variables est $\mathcal{V}$ et un Σ -modèle $\mathcal{M}$, une **interprétation** ν est une application de $\mathcal{V}$ dans M .

À partir d'une interprétation $\nu : \mathcal{V} \rightarrow M$, nous pouvons étendre de manière canonique cette dernière en une application $\nu^\sharp : T_\Sigma(\mathcal{V}) \rightarrow M$.

Théorème 138.

Soient une signature Σ dont l'ensemble des variables est $\mathcal{V}$ et $\mathcal{M}$ un Σ -modèle. Étant donnée une interprétation $\nu : \mathcal{V} \rightarrow M$, il existe une unique application $\nu^\sharp : T_\Sigma(\mathcal{V}) \rightarrow M$, extension de ν (i.e. pour tout $x \in \mathcal{V}$, $\nu^\sharp(x) = \nu(x)$).

Démonstration : $\nu^\sharp$ est définie de façon inductive sur tous les termes de $T_\Sigma(\mathcal{V})$ de la façon suivante :

$$\begin{aligned} x \in V &\mapsto \nu(x) \\ f(t_1 \dots, t_n) &\mapsto f^{\mathcal{M}}(\nu^\sharp(t_1), \dots, \nu^\sharp(t_n)) \end{aligned}$$

Elle définit totalement une application de $T_\Sigma(\mathcal{V})$ dans M . Nous concluons alors directement sur l'existence et l'unicité de $\nu^\sharp$. ■

Nous pouvons de même prolonger l'application ν en un prédicat unique, noté $\mathcal{M} \models_\nu$, sur les formules de $For(\Sigma)$ de la façon suivante :

Définition 109 (Relation de satisfaction).

Soient Σ une signature dont l'ensemble des variables est $\mathcal{V}$, $\mathcal{M}$ un Σ -modèle, et φ une Σ -formule. $\mathcal{M}$ **satisfait** φ pour une interprétation $\nu : \mathcal{V} \rightarrow M$ donnée, notée $\mathcal{M} \models_\nu \varphi$, si et seulement si :

- si φ est une formule atomique de la forme $r(t_1, \dots, t_n)$, alors $\mathcal{M} \models_\nu \varphi$ ssi $(\nu^\sharp(t_1), \dots, \nu^\sharp(t_n)) \in r^{\mathcal{M}}$,
- si φ est de la forme $\neg\psi$ alors $\mathcal{M} \models_\nu \varphi$ ssi $\mathcal{M} \not\models_\nu \psi$,
- si φ est de la forme $\psi \wedge \pi$ alors $\mathcal{M} \models_\nu \varphi$ ssi $\mathcal{M} \models_\nu \psi$ et $\mathcal{M} \models_\nu \pi$,
- si φ est de la forme $\psi \vee \pi$ alors $\mathcal{M} \models_\nu \varphi$ ssi $\mathcal{M} \models_\nu \psi$ ou $\mathcal{M} \models_\nu \pi$,
- si φ est de la forme $\psi \Rightarrow \pi$ alors $\mathcal{M} \models_\nu \varphi$ ssi si $\mathcal{M} \models_\nu \psi$ alors $\mathcal{M} \models_\nu \pi$,
- si φ est de la forme $\exists x.\psi$ alors $\mathcal{M} \models_\nu \varphi$ ssi il existe une interprétation ν' qui affecte la même valeur à toutes les variables que ν excepté peut-être pour la variable x (dans ce cas-là, on dit que ν' est *x-équivalente* à ν), telle que $\mathcal{M} \models_{\nu'} \psi$,
- si φ est de la forme $\forall x.\psi$ alors $\mathcal{M} \models_\nu \varphi$ ssi $\mathcal{M} \not\models_\nu \exists x.\neg\psi$.

$\mathcal{M}$ **valide** φ , noté $\mathcal{M} \models \varphi$ si, et seulement si pour toute interprétation $\nu : \mathcal{V} \rightarrow A$, $\mathcal{M} \models_\nu \varphi$.

Exemple 139.

À titre d'exemple, montrons que le modèle défini sur la signature associée à la théorie des ensembles ZF donné en section 21.1 valide l'axiome de la réunion (AR) (cf. section 20.2). Pour cela, supposons une interprétation $\nu : \mathcal{V}_{ZF} \rightarrow A$ quelconque, et montrons que :

$$\mathcal{M} \models^\nu AR$$

Soit ν' une interprétation x -équivalente à ν . on a alors trois cas à considérer :

1. cas où $\nu'(x) \in E$. Il nous suffit alors de choisir n'importe quel interprétation ν'' y -équivalente à ν' telle que $\nu''(y) \in E$. En effet, pour toute interprétation ν''' z -équivalente à ν'' , nous avons :

$$\mathcal{M} \models^{\nu'''} (z \in y \Leftrightarrow \exists w.(w \in x \wedge z \in w))$$

(la relation $\in^{\mathcal{M}}$ n'est pas vérifiée pour les éléments de E)

2. cas où $\nu'(x) \subseteq \mathcal{P}(E)$. Ici, le seul ensemble pour lequel la propriété ci-dessus est vérifiée est l'ensemble vide. Donc, il suffit de choisir une interprétation ν'' y -équivalente à ν' telle que $\nu''(y) = \emptyset$.
3. cas où $\nu'(x) \in M_n$ avec $n \geq 2$. Par définition, $\nu'(x)$ est un sous-ensemble de M_{n-1} . Donc, nous pouvons choisir l'interprétation ν'' y -équivalente à ν' telle que $\nu''(y) = \bigcup_{S \in \nu'(x)} S$. Cet ensemble existe car $\nu'(x)$ est un ensemble dont les éléments sont aussi des ensembles (au sens mathématique du terme). On peut donc faire une union entre eux.

La logique des prédicats a été définie pour donner un fondement aux objets manipulés par les mathématiciens. Dans ce cadre, un ensemble d'hypothèses Γ sur une signature Σ (i.e. $\Gamma \subseteq For(\Sigma)$) est souvent appelée un **système axiomatique**, et $\Gamma^\bullet$ contenant toutes les formules que l'on peut démontrer comme valide pour tous les modèles de Γ , est appelée la **théorie sous-jacente** de Γ .

Définition 110 (Conséquences sémantiques et théorie).

Un Σ -modèle $\mathcal{M}$ valide un ensemble de formules Γ , noté $\mathcal{M} \models \Gamma$ si, et seulement si $\mathcal{M} \models \varphi$ pour tout $\varphi \in \Gamma$.

Une Σ_1 -formule ψ est une **conséquence sémantique** d'un ensemble de formules Γ , notée $\Gamma \models \psi$, si et seulement si, pour tout Σ -modèle $\mathcal{M}$ si $\mathcal{M} \models \Gamma$ alors $\mathcal{M} \models \psi$.

Une **théorie** est tout ensemble de formules Γ tel que $\Gamma = \Gamma^\bullet$.

21.3. Vers une procédure de semi-décision

J. Herbrand est le premier à s'être intéressé à la décidabilité de la logique des prédicats du premier ordre. Il y a en partie répondu en définissant une procédure de semi-décision qui quand elle aboutit assure la validité des formules passées en argument. Malheureusement, il est mort très jeune et donc n'a pas eu le temps de finir ce travail et d'arriver à l'indécidabilité de la logique des prédicats. Ce résultat a été obtenu peu après par A. Church. Néanmoins le travail en logique de J. Herbrand a eu un grand retentissement car il est à la base d'un grand nombre de travaux remarquables tels que la définition de la calculabilité à partir des fonctions récursives ou encore les résultats de complétude de la logique des prédicats du premier ordre établis par K. Godel en 1930.

La démarche de J. Herbrand a été de tenter de montrer la décidabilité de la logique des prédicats en établissant un lien entre cette dernière et la logique propositionnelle. Bien entendu, ceci n'est pas direct et il a fallu au préalable faire des transformations des formules. C'est le sujet de la section suivante.

Formules prénexes et universelles

Ce qui différencie les formules de la logique des prédicats avec celles de la logique propositionnelle est l'utilisation de quantificateurs. Pour lier les formules des prédicats avec celles de la logique propositionnelle, il faut déjà sortir les quantificateurs des formules, et les faire porter au début. La question que l'on se pose alors est de savoir si cela est possible. la réponse est oui comme le montre le résultat suivant :

Définition 111 (Formule prénexe).

Une Σ -formule φ est dite **prénexe** si elle est de la forme :

$$Q_1x_1 \dots Q_nx_n\varphi'$$

où chaque $Q_i \in \{\forall, \exists\}$ et φ' est une formule sans quantificateur.

Ce que l'on peut démontrer est que toute formule du premier ordre peut être transformée en une formule prénexe équivalente (i.e. valide pour le même ensemble de modèle).

Théorème 140.

Toute Σ -formule φ est équivalente (i.e. à la même classe de Σ -modèles qui les valident) à une Σ -formule prénexe ψ .

Démonstration : Par récurrence sur la structure des formules à partir des transformations suivantes :

$$\begin{array}{ll} \neg\forall x\varphi \Leftrightarrow \exists x\neg\varphi & \neg\forall x\varphi \Leftrightarrow \exists x\neg\varphi \\ (\forall x\varphi) @ \psi \Leftrightarrow \forall x(\varphi @ \psi) & (\exists x\varphi) @ \psi \Leftrightarrow \exists x(\varphi @ \psi) \\ \psi @ (\forall x\varphi) \Leftrightarrow \forall x(\psi @ \varphi) & \psi @ (\exists x\varphi) \Leftrightarrow \exists x(\psi @ \varphi) \\ ((\forall x\varphi) \Rightarrow \psi) \Leftrightarrow \exists x(\varphi \Rightarrow \psi) & ((\exists x\varphi) \Rightarrow \psi) \Leftrightarrow \forall x(\varphi \Rightarrow \psi) \\ (\psi \Rightarrow (\forall x\varphi)) \Leftrightarrow \forall x(\psi \Rightarrow \varphi) & (\psi \Rightarrow (\exists x\varphi)) \Leftrightarrow \exists x(\psi \Rightarrow \varphi) \end{array}$$

@ $\in \{\wedge, \vee\}$, x est **libre** (i.e. n'apparaît pas dans la **portée** d'un quantificateur) dans ψ . ■

Par définition de la validation, les quantificateurs universelles en tête d'une formule peuvent être éliminés de la formule sans changer sa valeur de vérité. C'est toute la démarche de J. Herbrand.

Définition 112 (Formule prénexe universelle).

Une Σ -formule prénexe φ est dite **universelle** si l'ensemble de ses quantificateurs sont $\forall$.

Théorème d'Herbrand

Pour démontrer son résultat, il a alors défini un modèle canonique que l'on peut voir comme le plongement de la syntaxe dans la sémantique.

Définition 113 (Modèle de Herbrand).

Soit $\mathcal{M}$ un Σ -modèle. Notons $\mathcal{H}_{\mathcal{M}}$ le Σ -modèle défini par :

- l'ensemble $H_{\mathcal{M}}$ est l'ensemble des termes clos $T_{\Sigma}(\emptyset)$ sur Σ ,
- à toute fonction $f^n \in F$, $f^{\mathcal{H}_{\mathcal{M}}}$ est l'application qui à tout $(t_1, \dots, t_n) \in \prod_{i=1}^n H_{\mathcal{M}}$ associe le terme clos $f(t_1, \dots, t_n)$.
- à toute relation $r^n \in R$, $r^{\mathcal{H}_{\mathcal{M}}} = \{(t_1, \dots, t_n) \mid \mathcal{M} \models r(t_1, \dots, t_n)\}$

Ce type de Σ -modèle est appelé **modèle de Herbrand**.

À partir de là, nous avons le résultat recherché :

Théorème 141 (Théorème de Herbrand).

Soit Γ un ensemble de Σ -formules universelles. Γ a un Σ -modèle si, et seulement si Γ a un modèle de Herbrand.

Démonstration : La condition “si” est claire. Pour la condition “seulement si”, supposons que $\mathcal{M}$ soit un des Σ -modèles qui valident les formules de Γ . Montrons alors que $\mathcal{H}_{\mathcal{M}}$ est aussi un Σ -modèle de Γ . Soit $\varphi \in \Gamma$. Notons Γ_{φ} l'ensemble des formules sans variable obtenu à partir de φ en remplaçant chacune des variables par un terme clos et en éliminant tous les quantificateurs universels. Par construction de $\mathcal{H}_{\mathcal{M}}$, l'équivalence suivante est vraie :

$$\mathcal{H}_{\mathcal{M}} \models \varphi \iff \mathcal{H}_{\mathcal{M}} \models \Gamma_{\varphi}$$

Soit $\psi \in \Gamma_{\varphi}$. Par hypothèse, $\mathcal{M} \models \psi$. ψ étant une formule close, la vérification de $\mathcal{M} \models \psi$ revient à donner une valeur de vérité aux formules atomiques closes de la forme $r(t_1, \dots, t_n)$ apparaissant dans ψ . Mais par construction, pour toute formules atomiques closes $r(t_1, \dots, t_n)$, on a

$$\mathcal{H}_{\mathcal{M}} \models r(t_1, \dots, t_n) \iff \mathcal{M} \models r(t_1, \dots, t_n)$$

On conclut alors $\mathcal{H}_{\mathcal{M}} \models \psi$. ■

Ce théorème est important car il assure que dans le cas de théories universelles, la consistance de ces dernières est équivalente à raisonner sur le modèle de Herbrand. Or, ce dernier est syntaxique. En effet, son ensemble de valeurs n'est ni plus ni moins que les termes sans variables que l'on peut construire sur la signature. Cet ensemble est par définition récursivement énumérable (i.e. que l'on peut énumérer algorithmiquement ses éléments), donc si un sous-ensemble est inconsistant, on peut le calculer par énumération sur les termes clos. Bien sur, si cet ensemble de termes clos est infini, il est clair que nous ne pourrions pas vérifier la consistance d'un ensemble de formules universelles car il faudrait pour cela parcourir l'ensemble des termes ce qui n'est pas effectif. C'est pourquoi la procédure ci-dessous pour savoir si un ensemble Γ de Σ -formules universelles a ou n'a pas de modèle n'est qu'une procédure de semi-décision mais pas récursive :

1. On énumère l'ensemble $\Gamma' = \bigcup_{\varphi \in \Gamma} \Gamma_\varphi$ et on arrête cette énumération dès que l'ensemble des formules déjà énumérées n'a pas de modèle propositionnel.
2. D'après la contraposée du théorème de Herbrand, la procédure s'arrête après avoir trouvé un sous-ensemble fini $\Gamma'' \subseteq \Gamma'$ sans modèle propositionnel si, et seulement si Γ n'a pas de Σ -modèle.
3. Quand la procédure s'arrête sans avoir trouvé un tel sous-ensemble (ce n'est possible qu'avec un ensemble de terme clos fini) alors Γ a un Σ -modèle d'après le théorème précédent.

On peut utiliser cette procédure pour démontrer qu'une formule φ est une conséquence sémantique d'un ensemble consistant de formules universelles Γ . En effet, nous avons le résultat :

Proposition 142.

$\Gamma \models \varphi \iff \Gamma \cup \{\neg\varphi\}$ inconsistant.

À partir du résultat ci-dessus, nous avons alors :

- si $\Gamma \models \varphi$ alors la procédure ci-dessus termine.
- si $\Gamma \not\models \varphi$ alors la procédure ci-dessus ne termine pas.

Skolémisation

La procédure de semi-décision précédente ne marche que pour des formules prénexes universelles. Que pouvons-nous faire alors en présence de quantificateur existentiel? En effet, le théorème de Herbrand n'est plus vrai si l'ensemble Γ ne contient pas que des formules prénexes universelles. Soit Σ la signature composée d'une unique constante a et d'un symbole de relation unaire R , et soit $\Gamma = \{R(a), \exists x. \neg R(x)\}$. Γ a au moins le modèle $\mathcal{M}$ dont $M = \{0, 1\}$ et tel que $a^{\mathcal{M}} = 0$, $0 \in R^{\mathcal{M}}$, et $1 \notin R^{\mathcal{M}}$. Par contre, il n'a pas de modèle de Herbrand. En fait, il y a deux modèles de Herbrand possibles sur la signature Σ . Chacun a pour ensemble $H_{\mathcal{M}} = \{a\}$, et deux possibilités pour l'interprétation de R : $R_1^{\mathcal{M}} = \emptyset$ et $R_2^{\mathcal{M}} = \{a\}$. Mais aucun n'est modèle de Γ .

Néanmoins, étant donnée une formule prénexe φ possédant des quantificateurs existentiels, on peut construire une formule universelle φ' telle que si l'une est satisfiable, l'autre aussi. Le procédé de construction d'une telle formule s'appelle la *skolémisation*.

Définition 114 (Skolémisation).

Soit φ une Σ -formule prénexe $q_1x_1q_2x_2\dots q_nx_n.\psi$. On définit φ^s , appelée **formule de Skolem**, la formule obtenue à partir de φ en éliminant dans φ tous les quantificateurs existentiels $\exists x_i$ et en substituant dans ψ toutes les occurrences de la variables x_i dans la portée du quantificateur existentiel par un terme $f(x_{k1}, \dots, x_{kv})$ où chaque variable x_{kj} pour $j \in \{1, \dots, v\}$ est une variable quantifiée universellement se trouvant avant x_i dans la formule φ , et f^v est un nouveau nom de fonction ajouté à la signature Σ .

Exemple 143.

Pour $\varphi = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5, p(x_1, \dots, x_5)$, la formule de Skolem associée est :

$$\varphi^s = \forall x_2 \forall x_4, p(a, x_2, f(x_2), x_4, g(x_2, x_4))$$

Théorème 144.

Soit φ une Σ -formule prénexe, et φ^s la formule de Skolem associée. φ est satisfaite par un modèle $\mathcal{M}$ si, et seulement si φ^s l'est aussi pour un modèle $\mathcal{M}'$ défini comme $\mathcal{M}$ sur les éléments de la signature Σ .

Démonstration : Par récurrence sur le nombre de quantificateurs existentiels. Quand ce nombre est 0, $\varphi = \varphi^s$ et la propriété est trivialement vérifiée. Supposons que la propriété est vérifiée pour un certain rang n , et vérifions la pour le rang suivant $n+1$. φ est alors de la forme $\forall X \exists y QZ\psi$ où $\forall X$ est une succession de p quantifications universelles. Clairement, $\varphi^s = \varphi'^s$ où $\varphi' = \forall X QZ\psi[x \rightarrow f(X)]$. Montrons alors le *si* de l'équivalence. Supposons un modèle $\mathcal{M}$ de φ . Par l'axiome du choix, on peut définir une application $h : M^p \rightarrow M$ qui à chaque uplet $(a_1, \dots, a_p)$ associe $a \in M$ tel que $\mathcal{M} \models_\iota QZ.\psi$ où $\iota(x_i) = a_i$ pour tout i , $1 \leq i \leq p$, et $\iota(y) = a$. Soit le modèle $\mathcal{M}'$ défini comme $\mathcal{M}$ et tel que $f^{\mathcal{M}'} = h$. On alors $\mathcal{M}' \models \varphi'$ (rappelons que φ' est une formule close). Par hypothèse de récurrence, on en déduit que $\mathcal{M}' \models \varphi'^s$, et donc $\mathcal{M} \models \varphi^s$.

Pour le sens *seulement si*. Supposons un modèle $\mathcal{M}'$ tel que $\mathcal{M}' \models \varphi^s$, i.e. $\mathcal{M}' \models \varphi'^s$. Par hypothèse d'induction, on a donc $\mathcal{M}' \models \varphi'$. Ceci signifie que pour tout ι , on a $\mathcal{M}' \models_\iota QZ, \psi[y \rightarrow f(X)]$. Nous savons ainsi faire correspondre à tout tuple $(\iota(x_1), \dots, \iota(x_p))$ la valeur $\iota(y) = f^{\mathcal{M}'}(\iota(x_1), \dots, \iota(x_p))$ telle que $\mathcal{M} \models_\iota QZ, \psi$ d'où nous déduisons $\mathcal{M} \models \varphi$. ■

Attention : En aucun cas φ et φ^s sont équivalentes (i.e. ont le même ensemble de modèles).

On dispose maintenant d'une semi-procédure de décision pour tester la satisfiabilité de toute formule de la logique des prédicats du premier ordre en utilisant la procédure précédente.

21.4. Indécidabilité de la logique des prédicats du premier ordre

Dans la section précédente, on a vu que de démontrer la consistance d'une théorie logique et donc le fait de décider si une formule était une conséquence sémantique d'une théorie était un problème indécidable en général. Qu'en est-il des tautologies? En fait la réponse est aussi négative. C'est l'objet du théorème qui suit, établi par A. Church.

Théorème 145.

Le problème de décider qu'une formule est une tautologie est indécidable en général.

Démonstration : Nous allons formaliser le problème de la correspondance de Post en un problème de validité d'un ensemble de formule en logique des prédicats du premier ordre. Tout d'abord, on va démontrer que toute instance du problème de correspondance de Post sur un alphabet V de plus de deux lettres, peut se ramener à une autre instance du problème de correspondance de Post (avec même réponse) sur une alphabet à deux lettres. Supposons alors une instance du problème de la correspondance de Post $x, y, I = \{1, \dots, n\}$. L'alphabet V étant fini, on peut numéroter chacune des lettres de V et considérer par exemple leur notation binaire. Il suffit alors dans chacun des mots de x et y de remplacer chacune des lettres par leur équivalent binaire. Ceci nous donne deux autres séquences de mots x' et y' dans $\{0, 1\}$. Il est très facile alors de montrer que :

$$\sigma x = \sigma y \iff \sigma x' = \sigma y'$$

Revenons maintenant à la preuve du théorème. Soit C une instance x, y , et $I = \{1, \dots, n\}$ sur l'alphabet $\{0, 1\}$ du problème de la correspondance de Post. Soit la signature $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$ où :

- $\mathcal{F} = \{c^0, f_0^1, f_1^1\}$
- $\mathcal{R} = \{p^2\}$
- $\mathcal{V} = \{u, v, z\}$

Transformons l'instance du problème de la correspondance de Post ci-dessus en la formule $\varphi_1 \wedge \varphi_2 \Rightarrow \varphi_3$ avec :

- $\varphi_1 = \bigwedge_{i=1, \dots, n} p(f_{x_i}(c), f_{y_i}(c))$ où $f_{x_i}(c) = f_{x_{i1}}(f_{x_{i2}}(\dots(f_{x_{im_i}}(c))\dots))$ (resp. $f_{y_i}(c) = f_{y_{i1}}(f_{y_{i2}}(\dots(f_{y_{im_i}}(c))\dots))$)
pour $x_i = x_{i1}x_{i2} \dots x_{im_i}$ (resp. $y_i = y_{i1}y_{i2} \dots y_{im_i}$)
- $\varphi_2 = \forall u \forall v (p(u, v) \Rightarrow \bigwedge_{i=1, \dots, n} p(f_{x_i}(u), f_{y_i}(v)))$
- $\varphi_3 = \exists z, p(z, z)$

Montrons alors que C a une solution si, et seulement si la formule ci-dessus est une tautologie. Supposons alors que c'est une tautologie. Par définition, ceci veut dire que toute interprétation est un modèle. Considérons alors le modèle $\mathcal{M}$ suivant :

- $M = \{0, 1\}^*$
- $c^{\mathcal{M}} = \varepsilon$ où ε est le mot vide neutre pour la concaténation.
- $f_0^{\mathcal{M}} : \alpha \mapsto 0.\alpha$ et $f_1^{\mathcal{M}} : \alpha \mapsto 1.\alpha$.
- $(\alpha, \beta) \in p^{\mathcal{M}} \iff \exists \sigma \in I^+, \sigma x = \alpha \wedge \sigma y = \beta$

Il est facile de voir que $\mathcal{M} \models \varphi_1 \wedge \varphi_2$, et donc $\mathcal{M} \models \varphi_3$.

Montrons maintenant la réciproque. Supposons alors que C a pour solution $\sigma = i_1 \dots i_p$ telle que $\sigma x = \sigma y = \alpha$. Soit une interprétation quelconque $\mathcal{M}'$. Supposons alors que $\mathcal{M}'$ satisfait φ_1 et φ_2 , et montrons que $\mathcal{M}' \models \varphi_3$. Comme $\mathcal{M}' \models \varphi_1$, nous avons $(f_{x_{i_p}}^{\mathcal{M}'}(\varepsilon), f_{y_{i_p}}^{\mathcal{M}'}(\varepsilon)) \in p^{\mathcal{M}'}$. Donc, en appliquant la formule φ_2 valide aussi pour $\mathcal{M}'$, nous avons aussi $(f_{x_{i_{p-1}}}^{\mathcal{M}'}(f_{x_{i_p}}^{\mathcal{M}'}(\varepsilon)), f_{y_{i_{p-1}}}^{\mathcal{M}'}(f_{y_{i_p}}^{\mathcal{M}'}(\varepsilon))) \in p^{\mathcal{M}'}$. En répétant ce processus $p - 1$ fois, nous obtenons à la fin que $(f_{\sigma x}^{\mathcal{M}'}(\varepsilon), f_{\sigma y}^{\mathcal{M}'}(\varepsilon)) \in p^{\mathcal{M}'}$. ■

21.5. Pouvoir d'expression de la logique des prédicats

Logique des prédicats avec égalité

L'égalité est une relation binaire qui a toute son importance en mathématique mais aussi en informatique. En effet, l'une des applications de la logique des prédicats du premier ordre est

pour montrer la correction des programmes standards et donc vérifier que les résultats rendus par les programmes sont bien ceux attendus. Ceci s'exprimera alors par des formules du premier ordre dont les atomes seront des équations de la forme $f(t_1, \dots, t_n) = t$ où f est une fonction de notre programme, $t_1, \dots, t_n$ les arguments exprimés par des termes du premier ordre, et t le résultat attendu quand f est appliqué au tuple $(t_1, \dots, t_n)$. Pour pouvoir raisonner sur de tels équations, nous devons attacher des propriétés ou axiomes à ce nouveau prédicats. Ces derniers sont naturellement les suivants qui expriment que $=$ est une relation d'équivalence compatible avec les opérations et les relations (on parle alors de **congruence**) :

- $\forall x. x = x$
- $\forall x \forall y. x = y \Rightarrow y = x$
- $\forall x \forall y. \forall z. x = y \wedge y = z \Rightarrow x = z$
- $\forall x_1 \dots \forall x_n \forall y_1 \dots \forall y_n. x_1 = y_1 \wedge \dots \wedge x_n = y_n \Rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$
- $\forall x_1 \dots \forall x_n \forall y_1 \dots \forall y_n. x_1 = y_1 \wedge \dots \wedge x_n = y_n \Rightarrow (r(x_1, \dots, x_n) \Rightarrow r(y_1, \dots, y_n))$

Les deux derniers axiomes sont répétés autant de fois qu'il y a de noms de fonctions et de relations dans la signature.

Par convention, pour toute théorie dans un langage contenant le symbole d'égalité, les axiomes ci-dessus seront implicites dans l'énoncé des (autres) axiomes. On parlera alors de **théorie égalitaire**.

L'égalité peut se traiter de façon plus systématique en transformant les axiomes ci-dessus comme des règles d'inférence. On verra aussi que l'on peut rendre alors plus efficace le raisonnement équationnel en remplaçant les deux derniers axiomes par deux règles d'inférence traduisant la règle de Leibniz connu aussi sous le nom de remplacement d'égale par égale et qui succinctement se traduit par :

1. $\frac{x=y}{t[z/x]=t[z/y]}$
2. $\frac{x=y \quad r(x_1, \dots, x, \dots, x_n)}{r(x_1, \dots, y, \dots, x_n)}$

où $t[z/x]$ (resp. $t[z/y]$) signifie que nous avons remplacé dans le terme t toutes les occurrences de la variable z par x (resp. y). La première règle d'inférence donne alors naissance à une relation binaire dont on prendra la fermeture réflexive et transitive. On parlera de **relation de convertibilité**.

Théorème de Lowenheim et Skolem

La logique des prédicats n'est pas complètement adéquate comme langage pour les mathématiques. Il y a des aspects des mathématiques qu'elle ne permet pas d'axiomatiser. Ceci est une conséquence d'un théorème fameux, le théorème de Lowenheim et Skolem. Son énoncé est le suivant :

Théorème 146 (Théorème de Lowenheim-Skolem).

Soit T une théorie égalitaire sur une signature Σ qui admet au moins un modèle d'une cardinalité $\aleph$. Pour toute cardinalité $\aleph' > \aleph$, il existe un modèle de T dont l'ensemble des valeurs a cette cardinalité.

Démonstration : On ajoute à la signature Σ un ensemble de cardinalité $\aleph'$ de constantes c_i^0 , et à la théorie T l'ensemble infini de formules $\neg(c_i = c_j)$ pour $i \neq j$. On définit ainsi une nouvelle signature Σ' et une nouvelle théorie T' . T étant consistante par définition, il est facile de construire pour

chacun des sous-ensembles finis de T' , un modèle. Donc par le théorème de la compacité, T' a aussi un modèle qui par définition a au moins pour cardinalité $\aleph'$. ■

Corollaire 5.

La logique des prédicats ne permet pas de spécifier des modèles munis d'une cardinalité infinie donnée.

Corollaire 6.

La théorie de l'arithmétique a des modèles non isomorphes à $\mathbb{N}$. Ces modèles s'appellent des **modèles non standards** de l'arithmétique.

Ce dernier corollaire fait que la théorie de Peano n'est pas complète, c'est-à-dire qu'il existe des propriétés valides pour $\mathbb{N}$ mais qui ne sont pas valides pour d'autres modèles non isomorphes à $\mathbb{N}$. Le génie de Gödel est d'avoir été capable de différencier ces différents modèles de l'arithmétique par des formules exprimables dans la logique des prédicats du premier ordre.

22. Calcul pour la logique des prédicats

Comme pour la logique propositionnelle, nous allons traduire la sémantique de la logique des prédicats du premier ordre dans un monde purement symbolique. Les formules du premier ordre étant une extension des formules propositionnelles, il est alors naturel d'étendre les différents calculs de la logique propositionnelle aux quantificateurs et donc à la prise en compte de la notion de variables et de termes. Nous étendrons ici les calculs que nous avons présentés pour la logique propositionnelle.

- Calcul à la Hilbert ;
- D'édution naturelle ;
- Calcul des équents ;
- Résolution ;
- Méthode des tableaux.

Pour le calcul des séquents nous montrerons un résultat de normalisation de preuves fondamental pour la démonstration automatique ou encore le fondement des langages de programmation (principalement fonctionnels), le *théorème de l'élimination des coupures*. Enfin, pour la résolution, nous présenterons son application dans le cadre du langage de programmation logique Prolog.

22.1. Calcul à la Hilbert

Comme précédemment pour la logique propositionnelle, pour rendre plus concis la présentation du calcul associé à la logique des prédicats du premier ordre, nous allons d'abord restreindre l'ensemble des formules aux connecteurs $\neg$ et $\Rightarrow$ et au quantificateur universel $\forall$. Ainsi, par des transformations sémantiquement équivalentes, on peut supprimer les connecteurs $\wedge$ et $\vee$ et le quantificateur $\exists$. On a déjà donné ces transformations pour l'élimination des connecteurs $\wedge$ et $\vee$. Donnons alors juste celle concernant la suppression du quantificateur $\exists$.

$$\exists x\varphi \text{ est sémantiquement équivalent à } \neg\forall x\neg\varphi$$

Avant de donner les règles d'inférence relatives au quantificateur universel $\forall$, deux notions doivent être introduites afin de prendre en compte la notion de variable et le fait qu'elle puisse être substituée par toute valeur symbolique qu'est un terme.

Définition 115 (Variable libre).

Une occurrence de variable x dans une formule φ est **libre** si on ne trouve aucun quantificateur de la forme $\exists x$ ou $\forall x$ dans le chemin de cette occurrence à ce quantificateur lorsque l'on représente la formule φ sous-forme d'arbre. Si une occurrence de la variable x dans φ n'est pas libre alors elle est dite **liée**.

On dit qu'une variable x est **libre** dans une formule φ , et on le note $\varphi(x)$ si elle y a au moins une occurrence libre.

Définition 116 (Substitution).

Étant donnés une formule φ et un terme t , on note $\varphi(x/t)$ la formule obtenue en remplaçant toute occurrence libre de la variable x par t avec la condition qu'aucune des variables de t n'apparaissent comme variable liée dans t .

Définition 117 (Théorème).

Une Σ -formule φ est un **théorème** d'un ensemble de Σ -formules Γ , noté $\Gamma \vdash \varphi$, si et seulement si on peut obtenir l'énoncé $\Gamma \vdash \varphi$ en utilisant un nombre fini de fois les règles de la logique propositionnelle auxquelles on ajoute les règles suivantes :

- **Instantiation** Si $\Gamma \vdash \forall x\varphi$ alors pour tout terme $t \in T_\Sigma(V)$, $\Gamma \vdash \varphi(x/t)$.
- **Renomage** Si $\Gamma \vdash \forall x.\varphi$ alors $\Gamma \vdash \forall y.\varphi(x/y)$
- **Distribution** Si $\Gamma \vdash \forall x(\psi \Rightarrow \varphi)$ alors $\Gamma \vdash \psi \Rightarrow (\forall x.\varphi)$ où x n'a pas d'occurrence libre dans ψ .
- **Généralisation universelle** Si $\Gamma \vdash \varphi$ et x n'est libre dans aucune des formules de Γ , alors $\Gamma \vdash \forall x\varphi$

Correction et complétude**Théorème 147 (Correction).**

Étant donnée une Σ -théorie Γ et une Σ -formule φ , on a :

$$\Gamma \vdash \varphi \implies \Gamma \models \varphi$$

La preuve de complétude pour la logique des prédicats du premier ordre est beaucoup plus compliquée que celle pour la logique propositionnelle. Elle fait appel à des notions profondes de théorie des modèles. Entre autre, elle utilise une autre notion de complétude, la **complétude syntaxique** d'une théorie axiomatique. En effet, il existe deux sens du mot complétude en logique mathématique : la **complétude syntaxique** et la **complétude sémantique**. La complétude syntaxique s'intéresse au pouvoir de démonstration (via la relation $\vdash$) au sein d'une théorie axiomatique Γ .

Définition 118 (Syntaxiquement complète).

Une théorie Γ sera alors **syntaxiquement complète** si elle est consistante et si de plus, l'on peut démontrer toute formule φ ou sa négation (i.e., $\Gamma \vdash \varphi$ ou $\Gamma \vdash \neg\varphi$).

L'intérêt d'une telle notion est qu'elle entraîne la décidabilité de la théorie Γ .¹ La complétude sémantique établit une équivalence entre la notion de dérivabilité et de validité. C'est celle que nous allons démontrer pour la logique des prédicats du premier ordre.

1. En effet, l'ensemble des théorèmes ainsi que des non-théorèmes sont alors récursivement énumérables.

Le résultat de complétude pour la logique des prédicats du premier ordre a d'abord été démontré au départ par K. Godel en 1930, puis généralisé par L. Henkin en 1949. Dans la littérature, nous trouvons principalement la preuve de complétude donnée Par L. Henkin. L'intérêt de cette dernière sur celle établit par K. Godel porte principalement sur deux points :

1. elle considère des langages qui peuvent contenir une infinité de symboles primitifs.
2. elle peut être étendue à des logiques d'ordre supérieur (logique du second ordre, théorie des types,...).

Nous allons donc prouver le résultat suivant :

*si Γ est une théorie consistante alors Γ a un modèle.*²

La preuve est plus complexe que celle donnée pour la logique propositionnelle parce qu'elle passe nécessairement par la construction d'un modèle. Pour cela, nous avons besoin de quelques définitions et lemmes intermédiaires. Tout d'abord, donnons la définition qui va nous permettre de construire notre modèle.

Définition 119 (Axiome de Henkin).

Étant donnée une signature Σ , on appelle **Σ -axiome de Henkin** toute Σ -formule de la forme :

$$\exists x. \varphi \Rightarrow \varphi(x/c)$$

où c est une constante de la signature Σ .

On appelle **Σ -théorie de Henkin** toute Σ -théorie Γ qui contient tous les Σ -axiomes de Henkin.

La preuve de complétude selon L. Henkin se scinde alors en deux parties. Tout d'abord, nous montrons que toute Σ -théorie de Henkin Γ complète syntaxiquement, admet un modèle. Enfin, nous montrons que pour toute Σ -théorie consistante il existe une Σ' -théorie de Henkin complète syntaxiquement.

Lemme 148.

Si Γ est une Σ -théorie de Henkin complète syntaxiquement alors Γ admet un modèle.

Démonstration : Notons $\mathcal{M}$ le Σ -modèle suivant :

- $M = T_{\Sigma}(\emptyset)$,
- pour chaque symbole de la signature Σ , nous donnons la sémantique suivante :
 - pour chaque symbole de constante c de $\mathcal{F}$, nous avons : $c^{\mathcal{M}} = c$,
 - pour chaque symbole de fonction f^n de $\mathcal{F}$, nous posons $f^{\mathcal{M}} : M^n \rightarrow M$ par : $(t_1, \dots, t_n) \mapsto f(t_1, \dots, t_n)$,
 - pour chaque symbole de prédicat R^n de $\mathcal{R}$, nous posons :

$$R^{\mathcal{R}} = \{(t_1, \dots, t_n) \mid \Gamma \vdash R(t_1, \dots, t_n)\}$$

2. Nous rappelons que théorie Γ est consistante s'il existe une formule φ telle que $\Gamma \not\vdash \varphi$.

De part la définition de $\mathcal{M}$, pour toute formule atomique φ , on a : $\Gamma \vdash \varphi \iff \mathcal{M} \models \varphi$. Montrons alors que cette équivalence peut-être étendue à toutes les Σ -formules.

La preuve se fait par induction sur la structure des Σ -formules. Les cas dont le connecteur principal est un connecteur propositionnel ne pose pas de problème particulier. Par conséquent, les seuls cas que nous allons mentionner ici sont ceux faisant intervenir le seul quantificateur “ $\forall$ ” comme symbole de tête.

Supposons que $\Gamma \vdash \forall x.\varphi$. Nous avons aussi pour tout terme t de $T_\Sigma(\emptyset)$: $\Gamma \vdash \varphi(x/t)$ par la règle d’instantiation. Par hypothèse de récurrence, nous pouvons alors écrire : $\mathcal{M} \models \varphi(x/t)$ et ce pour tout terme t de $T_\Sigma(\emptyset)$. Il en découle alors directement : $\mathcal{M} \models \forall x.\varphi$.

Montrons la réciproque par contraposée. Supposons alors que $\Gamma \not\vdash \forall x.\varphi$. Comme Γ est syntaxiquement complète, nous avons alors : $\Gamma \vdash \neg(\forall x.\varphi)$. Nous avons montré dans la section précédente que $\models \neg(\forall x.\varphi) \iff \exists x.\neg\varphi$ donc par modus ponens qui est une règle correcte, nous obtenons alors : $\Gamma \models \exists x.\neg\varphi$. Maintenant, nous savons que Γ est une théorie de Henkin. Nous avons alors : $\Gamma \models \exists x.\neg\varphi \Rightarrow \neg\varphi(x/c)$. D’où par modus ponens, nous pouvons écrire : $\Gamma \models \neg\varphi(x/c)$. Par hypothèse de récurrence, nous obtenons alors : $\mathcal{M} \models \neg\varphi(x/c)$ d’où nous pouvons directement conclure par : $\mathcal{M} \not\models \forall x.\varphi$. ■

La seconde partie de la preuve repose sur une technique bien connue en théorie des modèles : la *méthode des diagrammes*. Cette dernière a déjà utilisée dans la preuve du théorème de Lowenheim-Skolem. Succinctement, elle consiste à enrichir une signature de départ en ajoutant une ensemble de constantes nouvelles en un nombre suffisant afin de pouvoir construire toutes les nouvelles formules utiles (dans notre cas, les axiomes de Henkin) pour résoudre un problème donné.

Lemme 149.

Soit Γ une Σ -théorie consistante. Il existe une signature Σ' contenant Σ et une Σ' -théorie de Henkin Γ' complète syntaxiquement qui contient Γ .

Démonstration : Tout d’abord, on construit inductivement sur $\mathbb{N}$ une nouvelle théorie consistante Γ' contenant Γ sur une signature Σ' . Pour cela, on pose comme point départ, $\Gamma_0 = \Gamma$ et $\Sigma_0 = \Sigma$, et on définit Γ_{n+1} (resp. Σ_{n+1}) à partir de Γ_n (resp. Σ_n) de la façon suivante :

$$\Gamma_{n+1} = \Gamma_n \bigcup \{ \exists x.\varphi \Rightarrow \varphi(x/c_\varphi) \mid \varphi \text{ est une } \Sigma_n\text{-formule} \}$$

$$\Sigma_{n+1} = \Sigma_n \bigcup \{ c_\varphi \mid \varphi \text{ est une } \Sigma_n\text{-formule} \}$$

Maintenant, montrons que Γ_{n+1} est une Σ_{n+1} -théorie consistante. Ceci se fait par récurrence sur les entiers n . Par hypothèse de départ, nous avons bien entendu que Γ_0 est consistante. Pour Γ_{n+1} , raisonnons par l’absurde. Supposons alors que Γ_{n+1} est inconsistante. Cela veut dire qu’il existe un nouvel axiome de Henkin $\exists x.\varphi \Rightarrow \varphi(x/c_\varphi)$ n’apparaissant pas dans Γ_n tel que : $\Gamma_n \vdash_{\Sigma_n} \exists x.\varphi \wedge \neg\varphi(x/c_\varphi)$. Montrons alors que si $\Gamma_n \vdash_{\Sigma_n} \neg\varphi(x/c_\varphi)$ alors $\Gamma_n \vdash_{\Sigma_n} \forall x.\neg\varphi$. Si $\Gamma_n \vdash_{\Sigma_n} \neg\varphi(x/c_\varphi)$, nous avons une suite $(\varphi_1, \dots, \varphi_m)$ dénotant une preuve de $\neg\varphi(x/c_\varphi)$. Substituons alors dans toutes les formules φ_i pour i compris entre 1 et m , la constante c_φ par une nouvelle variable fraîche z (c-à-d., n’apparaissant dans aucune des formules φ_i). Nous obtenons alors une nouvelle suite $(\psi_1, \dots, \psi_m)$ dénotant une preuve de $\neg\varphi(x/z)$ dans Γ_n . Nous avons par la règle de généralisation : $\Gamma_n \vdash_{\Sigma_n} \forall z.\neg\varphi(x/z)$. Or, il est facile de montrer que $\vdash_{\Sigma_n} \forall z.\neg\varphi(x/z) \Rightarrow \forall x.\neg\varphi$ (le faire à titre d’exercice). De ceci, nous déduisons : $\Gamma_n \vdash_{\Sigma_n} \exists x.\varphi \wedge \forall x.\neg\varphi$ ce qui n’est pas possible puisque Γ_n est consistante par hypothèse de récurrence.

On pose alors : $\Gamma' = \bigcup_n \Gamma_n$ et $\Sigma' = \bigcup_n \Sigma_n$. Maintenant, il nous reste à trouver une Σ' -théorie Γ'' complète syntaxiquement. Pour cela, nous allons utiliser un résultat classique en théorie des ensembles : le *lemme de Zorn*.

Considérons l'ensemble S de toutes les Σ' -théories consistantes contenant Γ' . Cet ensemble n'est pas vide car il contient au moins Γ' (nous avons montré ci-dessus que Γ' est elle-même consistante). Maintenant, considérons un sous-ensemble X de S totalement ordonné par inclusion (on appelle cela une chaîne). Posons alors $\bar{\Gamma} = \bigcup_{\Gamma \in X} \Gamma$ et montrons que $\bar{\Gamma}$ est aussi consistante. Pour cela, nous

devons considérer le lemme intermédiaire suivant :

Lemme 150.

Si Γ est une Σ -théorie dont toutes les parties finies sont consistantes, alors Γ est elle-même consistante.

Démonstration : Découle directement du fait que la relation $\vdash$ vérifie la propriété de compacité. ■

À partir du lemme ci-dessus, montrons que $\bar{\Gamma}$ est consistante. Pour cela, supposons le contraire. Il existe un sous-ensemble fini $\bar{\Gamma}'$ de $\bar{\Gamma}$ inconsistant. Par définition de l'ensemble $\bar{\Gamma}$, chaque formule φ de $\bar{\Gamma}$ appartient à une Σ' -théorie consistante Γ_φ de X . Mais, l'ensemble $\{\Gamma_\varphi \mid \varphi \in \bar{\Gamma}'\}$ admet un plus grand élément $\bar{\Gamma}_0$ appartenant à X . Maintenant, l'ensemble de formules $\bar{\Gamma}'$ est un sous-ensemble de $\bar{\Gamma}_0$. Donc, $\bar{\Gamma}_0$ est inconsistant ce qui contredit l'hypothèse de départ.

On a alors que la Σ' -théorie $\bar{\Gamma}$ est consistante. Par définition, elle est un majorant (au sens de l'inclusion) de l'ensemble X dans S . On peut appliquer le lemme de Zorn dont l'énoncé est le suivant :

Lemme 151 (Lemme de Zorn).

Étant donné un ensemble non vide S muni d'une relation d'ordre $\prec$, si tout sous-ensemble S' de S totalement ordonné par $\prec$ possède un élément maximal au sens de $\prec$ dans S , alors, S possède un élément maximal (c-à-d., plus grand que tous ses autres éléments toujours au sens de $\prec$)^a

a. Le lemme de Zorn est l'équivalent de l'axiome de choix vérifiée dans la théorie des ensembles de ZF.

À partir de ce lemme, il existe donc une Σ' -théorie Γ'' maximale au sens de l'inclusion pour S . Pour finir montrons alors que Γ'' est complète syntaxiquement, c'est-à-dire, pour toute Σ' -formule φ soit $\Gamma'' \vdash_{\Sigma'} \varphi$, ou $\Gamma'' \vdash_{\Sigma'} \neg\varphi$. Supposons alors une Σ' -formule φ telle que $\varphi \notin \Gamma''$. Par la propriété de Γ'' d'être maximale, $\Gamma'' \cup \{\varphi\}$ n'est pas consistante. On en déduit alors directement que : $\Gamma'' \vdash \neg\varphi$. ■

Maintenant, nous avons tous les ingrédients pour démontrer le théorème de complétude.

Théorème 152.

Tout Σ -théorie Γ consistante, admet un modèle.

Démonstration : Il suffit d'appliquer le lemme 149 pour trouver une Σ' -théorie de Henkin consistante et complète syntaxiquement. Par le lemme 148, on sait alors qu'il existe un Σ' -modèle $\mathcal{M}$ pour cette

théorie. Le Σ -modèle obtenu en élaguant tous les termes de M faisant intervenir des constantes n'apparaissant pas dans Σ (c-à-d., toutes les constantes utilisées pour les axiomes de Henkin) est alors un modèle de Γ . ■

On sait aussi que le théorème de complétude peut s'énoncer sous cette forme.

Théorème 153 (Complétude).

Si Γ est une Σ -théorie et si φ est une Σ -formule telle que $\Gamma \models \varphi$, alors $\Gamma \vdash \varphi$.

Démonstration : Sans perte de généralité, nous supposons que φ est une formule close, c'est-à-dire que toutes les occurrences de variables apparaissant dans φ sont dans la portée d'un quantificateur. Supposons alors le contraire, c'est-à-dire supposons que $\Gamma \not\vdash \varphi$. Ceci veut dire que $\Gamma \cup \{\neg\varphi\}$ est consistante. Ainsi par le théorème 152, $\Gamma \cup \{\neg\varphi\}$ admet un modèle $\mathcal{M}$. $\mathcal{M}$ est alors un modèle de Γ mais pas de φ . Donc, $\Gamma \not\models \varphi$. ■

22.2. Dédution naturelle

À FAIRE.

22.3. Calcul des séquents

Les règles de déduction

Les règles du calcul des séquents étant défini sur la structure inductive des formules, on retrouve toutes les règles données pour les connecteurs logiques propositionnels en section 17.3. On retrouve aussi les règles structurelles d'affaiblissement et de coupure. Il nous reste alors simplement à ajouter les règles propres aux quantificateurs.

Soit Σ une signature dont $\mathcal{V}$ est l'ensemble des variables. Soient Γ et Δ deux ensembles de formules. Soit φ une formule. Soient x et y deux variables de $\mathcal{V}$. Soit t un terme de $T_\Sigma(\mathcal{V})$.

Règles pour les quantificateurs

$$\frac{\Gamma, \varphi(x/t) \vdash \Delta}{\Gamma, \forall x.\varphi \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, \varphi(x/y)}{\Gamma \vdash \Delta, \forall x.\varphi}$$

Pour la règle de $\forall$ à droite, on impose que la variable y n'est pas libre dans $\Gamma \cup \Delta$

$$\frac{\Gamma, \varphi(x/y) \vdash \Delta}{\Gamma, \exists x.\varphi \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, \varphi(x/t)}{\Gamma \vdash \Delta, \exists x.\varphi}$$

Pour la règle de $\exists$ à gauche, la variable y dans $\Gamma \cup \Delta$

Pour prouver un théorème avec le calcul des séquents, on conserve la représentation en arbre de preuve. Ceci vient du fait même de la définition du calcul des séquents qui cherche à montrer que la formule à prouver ne contient comme sous-formule (à transformation près) que des tautologies, et donc comme sous-formules de bases des axiomes de la forme : $\Gamma, \varphi \vdash \Delta, \varphi$.

Correction et complétude du calcul des séquents

Théorème 154 (Correction).

Tout séquent $\Gamma \vdash \Delta$ dérivable à partir du calcul des séquents est valide.

Démonstration : Comme à son habitude, cette propriété se démontre par récurrence sur la longueur de la preuve. Pour cela, Nous supposons un modèle $\mathcal{M}$ et un séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_m$ quelconques. Le cas de base est celui où le séquent $\Gamma \vdash \Delta$ a été obtenu en une étape. C'est donc une axiome de la forme $\Gamma, \varphi \vdash \Delta, \varphi$ dont il a déjà été montré que c'est une tautologie. Maintenant, nous devons montrer que la validité est préservée au travers des règles d'inférence. Vu le nombre des règles du calcul des séquents, nous n'allons pas prouver cette propriété pour chacune de ces règles mais simplement pour la règle de coupure et celles associées au quantificateur universelle.

coupure

Nous supposons que le séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_m$ a été obtenu par application de la règle de coupure. Nous avons alors deux séquents $\psi_1, \dots, \psi_k \vdash \varphi_1, \dots, \varphi_l, \varphi$ et $\varphi, \psi_{k+1}, \dots, \psi_n \vdash \varphi_{l+1}, \dots, \varphi_m$ avec $k \leq n$ et $l \leq m$ dans l'arbre de preuve du séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_m$. Par hypothèse de récurrence, ces deux séquents sont validés par le modèle $\mathcal{M}$, c'est-à-dire : $\mathcal{M} \models \psi_1 \wedge \dots \wedge \psi_k \Rightarrow \varphi_1 \vee \dots \vee \varphi_l \vee \varphi$ et $\mathcal{M} \models \varphi \wedge \psi_{k+1} \wedge \dots \wedge \psi_n \Rightarrow \varphi_{l+1} \vee \dots \vee \varphi_m$. Par définition, ceci veut dire que pour toute interprétation $\nu : \mathcal{V} \rightarrow M$, nous avons : $\mathcal{M} \models_\nu \psi_1 \wedge \dots \wedge \psi_k \Rightarrow \varphi_1 \vee \dots \vee \varphi_l \vee \varphi$ et $\mathcal{M} \models_\nu \varphi \wedge \psi_{k+1} \wedge \dots \wedge \psi_n \Rightarrow \varphi_{l+1} \vee \dots \vee \varphi_m$. Soit $\nu^+ : \mathcal{V} \rightarrow M$ une interprétation telle que $\mathcal{M} \models_{\nu^+} \psi_1 \wedge \dots \wedge \psi_k \wedge \psi_{k+1} \wedge \dots \wedge \psi_n$. Nous avons alors deux cas à considérer :

1. $\mathcal{M} \models_{\nu^+} \varphi$. Dans ce cas-là, nous avons aussi $\mathcal{M} \models_{\nu^+} \varphi \wedge \psi_{k+1} \wedge \dots \wedge \psi_n$, et donc $\mathcal{M} \models_{\nu^+} \varphi_{l+1} \vee \dots \vee \varphi_m$. Nous avons alors $\mathcal{M} \models_{\nu^+} \varphi_1 \vee \dots \vee \varphi_l \vee \varphi_{l+1} \vee \dots \vee \varphi_m$.
2. $\mathcal{M} \not\models_{\nu^+} \varphi$. Dans ce cas-là, nous avons $\mathcal{M} \models_{\nu^+} \varphi_1 \vee \dots \vee \varphi_l$, d'où nous déduisons directement $\mathcal{M} \models_{\nu^+} \varphi_1 \vee \dots \vee \varphi_l \vee \varphi_{l+1} \vee \dots \vee \varphi_m$.

quantificateur universel

$\forall \vdash$ Nous supposons que le séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_m$ a été obtenu par application de la règle $\forall \vdash$. Nous avons alors un séquent $\psi_1, \dots, \psi_{n-1}, \varphi(x/t) \vdash \varphi_1, \dots, \varphi_m$ dans l'arbre de preuve du séquent $\psi_1, \dots, \psi_n, \forall x. \varphi \vdash \varphi_1, \dots, \varphi_m$. Par hypothèse de récurrence, nous pouvons écrire $\mathcal{M} \models \psi_1 \wedge \dots \wedge \psi_{n-1} \wedge \varphi(x/t) \Rightarrow \varphi_1 \vee \dots \vee \varphi_m$. Par définition, ceci veut dire que pour toute interprétation $\nu : \mathcal{V} \rightarrow M$, $\mathcal{M} \models_\nu \psi_1 \wedge \dots \wedge \psi_{n-1} \wedge \varphi(x/t) \Rightarrow \varphi_1 \vee \dots \vee \varphi_m$. Supposons alors que pour une interprétation $\nu^+ : \mathcal{V} \rightarrow M$ donnée, $\mathcal{M} \models_{\nu^+} \psi_1 \wedge \dots \wedge \psi_{n-1} \wedge \forall x. \varphi$. Nous avons déjà montré pour la correction du calcul précédent pour la logique des prédicats du premier ordre, que la formule $\forall x. \varphi \Rightarrow \varphi(x/t)$ est une tautologie (règle d'instantiation). Nous avons alors $\mathcal{M} \models_{\nu^+} \forall x. \varphi \Rightarrow \varphi(x/t)$, ce qui par définition nous donne, $\mathcal{M} \models_{\nu^+} \varphi(x/t)$. De ceci nous obtenons alors directement, $\mathcal{M} \models_{\nu^+} \psi_1 \wedge \dots \wedge \psi_{n-1} \wedge \varphi(x/t)$, et donc, $\mathcal{M} \models_{\nu^+} \varphi_1 \wedge \dots \wedge \varphi_m$.

$\vdash \forall$ Nous supposons que le séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_m$ a été obtenu par application de la règle $\vdash \forall$. Nous avons alors un séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_{m-1}, \varphi(x/y)$ dans l'arbre de preuve du séquent $\psi_1, \dots, \psi_n, \forall x. \varphi \vdash \varphi_1, \dots, \varphi_m$. Par hypothèse de récurrence, nous pouvons écrire $\mathcal{M} \models \psi_1 \wedge \dots \wedge \psi_n \Rightarrow \varphi_1 \vee \dots \vee \varphi_{m-1} \vee \varphi(x/y)$. Supposons que l'ensemble des variables libres de $\psi_1 \wedge \dots \wedge \psi_n$ et $\varphi_1 \wedge \dots \wedge \varphi_m$ est $\{x_1, \dots, x_k\}$. L'ensemble des variables libres du séquent $\psi_1, \dots, \psi_n \vdash \varphi_1, \dots, \varphi_{m-1}, \varphi(x/y)$ est alors $\{x_1, \dots, x_k, y\}$. Nous pouvons donc écrire par la règle de généralisation, $\mathcal{M} \models \forall x_1. \dots \forall x_k. \forall y. (\neg \psi_1 \vee \dots \vee \neg \psi_n \vee \varphi_1 \vee \dots \vee \varphi_{m-1} \vee \varphi(x/y))$. Comme la variable y n'est pas libre pour $\psi_1, \dots, \psi_n$ et $\varphi_1, \dots, \varphi_{m-1}$, nous pouvons écrire $\mathcal{M} \models \forall x_1. \dots \forall x_k. (\neg \psi_1 \vee \dots \vee \neg \psi_n \vee \varphi_1 \vee \dots \vee \varphi_{m-1} \vee \forall y. \varphi(x/y))$. La suite de la preuve repose sur le lemme simple suivant :

Lemme 155.

Étant données une formule φ et une variable y qui n'apparaît pas libre dans φ , nous avons :
si $\vdash \forall y.\varphi(x/y)$ alors $\forall x.\varphi$

Démonstration : Par la règle de renommage et le théorème de la déduction, nous avons :
 $\forall x.\varphi(x/y)(y/x)$. Mais par définition, dans la formule $\varphi(y/x)$, la variable x n'apparaît plus libre. Donc, $\varphi(x/y)(y/x)$ est φ . ■

Par ce lemme, nous obtenons alors directement : $\mathcal{M} \models \forall x_1 \dots \forall x_k.(\neg\psi_1 \vee \dots \vee \neg\psi_n \vee \varphi_1 \vee \dots \vee \varphi_{m-1} \vee \forall x.\varphi)$ ■

La complétude du calcul des séquents se démontre en deux étapes principales. La première étape consiste à simuler le premier calcul dans le calcul des séquents. Ceci fait l'objet de l'exercice suivant.

À partir de ce résultat, nous obtenons le corollaire suivant :

Corollaire 7.

Pour toute formule φ , nous avons : $\models \varphi$ implique $\vdash \varphi$ admet une preuve à partir du calcul des séquents.

Démonstration : En effet, par le théorème de complétude établi dans la section précédente, nous avons une preuve de φ par le calcul précédent qui par le résultat explicité ci-dessus nous donne une preuve du séquent $\vdash \varphi$ par le calcul des séquents. ■

Mais ceci ne suffit pas à montrer la complétude du calcul des séquents. En effet, ici nous avons simplement ce résultat pour des séquents dont la forme est : $\vdash \varphi$. Il nous reste alors à le montrer pour des séquents de forme plus générale, c'est-à-dire : $\Gamma \vdash \Delta$. Pour cela, nous devons prouver le lemme intermédiaire suivant :

Lemme 156.

Pour $n > 0$ et $m > 0$, les séquents suivants sont prouvables :

1. $\varphi_1, \dots, \varphi_n \vdash \varphi_1 \wedge \dots \wedge \varphi_n$
2. $\psi_1 \vee \dots \vee \psi_m \vdash \psi_1, \dots, \psi_m$

Démonstration : Par récurrence sur les entiers n et m (laissée en exercice). ■

Nous pouvons maintenant montrer le théorème de complétude pour le calcul des séquents dont l'énoncé est le suivant :

Théorème 157.

Tout séquent $\varphi_1, \dots, \varphi_n \vdash \psi_1, \dots, \psi_m$ valide, admet une preuve à partir du calcul des séquents.

Démonstration : $\varphi_1, \dots, \varphi_n \vdash \psi_1, \dots, \psi_m$ est un séquent valide, donc $\varphi_1 \wedge \dots \wedge \varphi_n \Rightarrow \psi_1 \vee \dots \vee \psi_m$ est une formule valide. Par le corollaire ci-dessus, le séquent $\vdash \varphi_1 \wedge \dots \wedge \varphi_n \Rightarrow \psi_1 \vee \dots \vee \psi_m$ admet une preuve à partir du calcul des séquents. Par la règle ($\vdash \Rightarrow$), le séquent $\varphi_1 \wedge \dots \wedge \varphi_n \vdash \psi_1 \vee \dots \vee \psi_m$ admet aussi une preuve à partir du calcul des séquents. En appliquant une fois, le lemme ci-dessus et la règle de coupure, le séquent $\varphi_1, \dots, \varphi_n \vdash \psi_1 \vee \dots \vee \psi_m$ devient prouvable. Il suffit alors d'appliquer encore une fois le lemme ci-dessus pour obtenir que le séquent $\varphi_1, \dots, \varphi_n \vdash \psi_1, \dots, \psi_m$ soit aussi prouvable à partir du calcul des séquents. ■

Élimination des coupures

Un résultat, sûrement un des plus importants de la logique moderne principalement pour ses applications en théorie de la démonstration, a été démontré par H. Gentzen au départ comme un résultat intermédiaire à sa démonstration de la consistance de l'arithmétique. Ce résultat, connu sous le nom de **Hauptsatz** de Gentzen, consiste à montrer que la règle de coupure est redondante. À première vue, ceci semble sans surprise. En effet, la coupure a pour rôle de "modulariser" les preuves en introduisant des lemmes intermédiaires (représentés par la formule φ) afin de rendre plus concis les preuves. Éliminer les coupures revient donc à étendre les preuves en remplaçant les lemmes intermédiaires par leur preuve. Gentzen a donné une preuve constructive de ce résultat (i.e. un algorithme) par des transformations de bouts d'arbre de preuve. L'idée de la preuve est de faire "remonter" les coupures jusqu'aux axiomes où elles s'annulent. Ces transformations ne sont pas très compliquées à définir. La difficulté est de montrer la terminaison du processus, c'est-à-dire montrer qu'au final on obtient toujours en un nombre d'étapes de transformation finies, un arbre de preuve sans coupure équivalent, c'est-à-dire défini à partir du même ensemble d'axiomes et aboutissant à la même conclusion. Commençons alors par donner l'ensemble de ces transformations. Mais avant ça, donnons d'abord quelques notions qui seront utiles à ces transformations. La première notion qui sera importante pour notre processus de transformation est celle de **formule principale** pour une règle. Intuitivement, elle correspond à la formule mise en jeu dans les règles logiques (propositionnelles et pour les quantificateurs). Ainsi, cette notion se définit comme suit :

- $\neg\varphi$ est principale pour $\neg \vdash$ et $\vdash \neg$,
- $\varphi @ \psi$ est principale pour $@ \vdash$ et $\vdash @$ avec $@ \in \{\vee, \wedge, \Rightarrow\}$, et
- $Qx.\varphi$ est principale pour $Q \vdash$ et $\vdash Q$ avec $Q \in \{\forall, \exists\}$.

Afin de définir notre algorithme de transformation par un simple ensemble de règles de transformation, nous introduisons deux nouvelles règles au calcul des séquents :

$$\frac{\Gamma \vdash \Delta}{\sigma(\Gamma) \vdash \sigma(\Delta)} \text{Sub.}$$

où $\sigma : \mathcal{V} \rightarrow T_{\Sigma}(\mathcal{V})$ est une substitution et $\sigma(\Gamma) = \sigma(\varphi_1), \dots, \sigma(\varphi_n)$ quand $\Gamma = \varphi_1, \dots, \varphi_n$.

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'} \text{Aff.}$$

où $\Gamma \subseteq \Gamma'$ et $\Delta \subseteq \Delta'$.

L'idée intuitive d'une telle extension est que dans la preuve habituelle de l'élimination des coupures, ces deux règles sont remplacées par des lemmes intermédiaires. Elles définissent donc un procédé de transformation situé au niveau de la méta-preuve et non plus au niveau de notre algorithme de transformation. L'intérêt ici de considérer ces deux règles est de ramener de façon uniforme cette transformation au niveau de l'algorithme de l'élimination des coupures et donc rendre complète la description de ce dernier. Notons enfin par l'exercice ci-dessus que l'ajout

de ces deux règles n'engendre pas de puissance de preuve supplémentaire par rapport à l'ancien calcul, c'est-à-dire qu'à toute preuve d'une tautologie utilisant ces deux règles et les autres règles du calcul des séquents, on peut associer une preuve équivalente dans le calcul des séquents présenté dans la définition ??, à cette même tautologie.

Ainsi, l'algorithme d'élimination des coupures se définit par un ensemble de règles de transformation que l'on peut regrouper en quatre sous-ensembles de règles de transformation comme suit :

1. Cas où la formule de coupure n'est pas principale pour au moins une des règles directement au-dessus de la coupure. Ici, nous donnons simplement le cas d'une règle à droite d'introduction d'un connecteur propositionnel ou d'un quantificateur, appliquée sur la prémisse gauche de la règle de coupure, tous les autres cas pouvant être ramenés par symétrie à ce cas-là.

$$\frac{\frac{\Gamma_1 \vdash \Delta_1, \varphi}{\Gamma'_1 \vdash \Delta'_1, \varphi} @ \quad \Gamma'_1, \varphi \vdash \Delta'_1}{\Gamma'_1 \vdash \Delta'_1} \text{Coupure} \rightsquigarrow \frac{\frac{\frac{\Gamma_1 \vdash \Delta_1, \varphi}{\Gamma_1, \Gamma'_1 \vdash \Delta_1, \Delta'_1, \varphi} \text{Aff.} \quad \frac{\Gamma'_1, \varphi \vdash \Delta'_1}{\Gamma_1, \Gamma'_1, \varphi \vdash \Delta_1, \Delta'_1} \text{Aff.}}{\Gamma_1, \Gamma'_1 \vdash \Delta_1, \Delta'_1} \text{Coupure}}{\Gamma'_1 \vdash \Delta'_1} @$$

où $@ \in \{\vee, \neg, \Rightarrow, \exists, \forall\}$ et $\Gamma'_1 = \Gamma_1$ à l'exception de la règle $\vdash \neg$ où dans ce cas-là $\Gamma'_1 = \Gamma_1, \neg\psi$ pour une formule ψ donnée.

2. Cas où la formule de la coupure est principale pour les deux règles au-dessus de la coupure. Dans ce cas, nous avons les transformations suivantes :

$$\begin{aligned} & \frac{\frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \varphi' \vdash \Delta}{\Gamma, \varphi \vee \varphi' \vdash \Delta} \vee \vdash \quad \frac{\Gamma \vdash \Delta, \varphi, \varphi'}{\Gamma \vdash \Delta, \varphi \vee \varphi'} \vdash \vee}{\Gamma \vdash \Delta} \text{Coupure} \rightsquigarrow \\ & \frac{\frac{\frac{\Gamma, \varphi \vdash \Delta}{\Gamma, \varphi \vdash \Delta, \varphi'} \text{Aff.} \quad \Gamma \vdash \Delta, \varphi, \varphi'}{\Gamma \vdash \Delta, \varphi'} \text{Coupure} \quad \Gamma, \varphi' \vdash \Delta}{\Gamma \vdash \Delta} \text{Coupure} \\ & \frac{\frac{\Gamma \vdash \Delta, \varphi}{\Gamma, \neg\varphi \vdash \Delta} \vdash \neg \quad \frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \Delta, \neg\varphi} \neg \vdash}{\Gamma \vdash \Delta} \text{Coupure} \rightsquigarrow \frac{\Gamma \vdash \Delta, \varphi \quad \Gamma, \varphi \vdash \Delta}{\Gamma \vdash \Delta} \text{Coupure} \\ & \frac{\frac{\Gamma, \varphi[x/y] \vdash \Delta}{\Gamma, \exists x. \varphi \vdash \Delta} \vdash \exists \quad \frac{\Gamma \vdash \Delta, \varphi[x/t]}{\Gamma \vdash \Delta, \exists x. \varphi} \exists \vdash}{\Gamma \vdash \Delta} \text{Coupure} \rightsquigarrow \frac{\frac{\Gamma, \varphi[x/y] \vdash \Delta}{\Gamma, \varphi[x/y][y/t] \vdash \Delta} \text{Sub.} \quad \Gamma \vdash \Delta, \varphi[x/t]}{\Gamma \vdash \Delta} \text{Coupure} \end{aligned}$$

3. Cas où au moins une des prémisses de la coupure est un axiome.

$$\begin{aligned} & \frac{\frac{}{\Gamma, \varphi \vdash \Delta, \varphi} \text{Ax.} \quad \Gamma \vdash \Delta, \varphi}{\Gamma \vdash \Delta, \varphi} \text{Coupure} \rightsquigarrow \Gamma \vdash \Delta, \varphi \\ & \frac{\frac{}{\Gamma, \varphi, \psi \vdash \Delta, \varphi} \text{Ax.} \quad \Gamma, \varphi \vdash \Delta, \varphi, \psi}{\Gamma, \varphi \vdash \Delta, \varphi} \text{Coupure} \rightsquigarrow \Gamma, \varphi \vdash \Delta, \varphi \end{aligned}$$

Les cas duaux (i.e. la prémisse droite de la coupure est un axiome) se traite de la même façon.

4. Finalement, il reste le cas où les règles Sub. et Aff. ont pour prémisse la conclusion d'une des autres règles du calcul des séquents. Dans ce cas-là, on montre que ces deux règles Sub. et Aff. remontent sur toutes les autres règles (coupure comprise) et s'annulent sous les axiomes.

La relation $\rightsquigarrow$ définit ainsi un ensemble de règles de transformation entre des **pseudo-arbres de preuve**, c'est-à-dire des arbres où l'on n'impose pas dans les membres gauche et droit des règles que les formules apparaissant aux feuilles soient des axiomes. Notre algorithme est alors défini en prenant la fermeture transitive et par contexte de preuve de cette relation de transformation $\rightsquigarrow$ où la **fermeture par contexte de preuve** est définie ci-dessous. Mais avant de donner cette notion de fermeture par contexte de preuve, introduisons d'abord quelques notations qui nous seront utiles pour ce propos mais aussi pour la suite.

Notations.

- On notera $\pi : s$ l'arbre de preuve dont la conclusion est le séquent s .
- On notera $(\pi_1, \dots, \pi_n, s)_\iota$ l'arbre de preuve dont la dernière règle d'inférence appliquée est $\iota \in \{\text{coupure}, \wedge \vdash, \vdash \wedge, \dots\}$ et π_i est l'arbre de preuve menant à au séquent s_i prémisses de la règle ι .
- Utilisant une notation classique de numérotation des chemins des noeuds d'un arbre par une suite finie d'entiers naturels, nous pouvons nous référer aux positions dans un arbre de preuve. Ainsi,
 - étant donné un arbre de preuve π , une **position** dans π est toute suite finie d'entiers naturels ω définissant le chemin menant de la racine au sous-arbre dont la conclusion se trouve à cette position. Le sous-arbre est noté $\pi|_\omega$.
 - étant donné une position $\omega \in \mathbb{N}^*$ dans un arbre de preuve π , $\pi[\pi']_\omega$ est l'arbre de preuve obtenu à partir de π en remplaçant le sous-arbre $\pi|_\omega$ par l'arbre de preuve π' . Bien sûr, les arbres $\pi|_\omega$ et π' ont même racine.
- On appelle **contexte de preuve** tout arbre obtenu par composition des règles du calcul des séquents dont un séquent et un seul aux feuilles n'est pas un axiome. On notera alors $\Rightarrow$ la fermeture par contexte de preuve de $\rightsquigarrow$. Cette nouvelle relation se définit par :
 - $\rightsquigarrow \subseteq \Rightarrow$
 - Si $\pi : s \Rightarrow \pi' : s$ et π'' est un contexte de preuve dont le séquent qui n'est pas un axiome est s et se trouve à la position ω , alors $\pi''[\pi]_\omega \Rightarrow \pi''[\pi']_\omega$.
 - Si $\pi \Rightarrow \pi'$ et il existe deux positions ω et ω' dans π et π' telles que $\pi|_\omega$ et $\pi'|_{\omega'}$ dénotent un même séquent s qui n'est pas un axiome, alors pour tout arbre de preuve $\pi'' : s$, $\pi[\pi'']_\omega \Rightarrow \pi'[\pi'']_{\omega'}$

Notre algorithme de transformation est alors défini par la fermeture transitive $\stackrel{+}{\Rightarrow}$. Il nous reste maintenant à prouver que $\stackrel{+}{\Rightarrow}$ termine et que les arbres de preuve atteints (i.e. ceux qui ne sont plus transformables - aussi appelés arbre en **forme normale**) sont tous les arbres sans coupure. Au regard de l'exhaustivité des règles de transformation ci-dessus, il est clair que les arbres en forme normale sont tous les arbres sans coupure mais aussi sans affaiblissement. Il nous reste alors à prouver le résultat suivant :

Théorème 158.

$\stackrel{+}{\Rightarrow}$ termine.

Démonstration : Pour montrer ce résultat de terminaison, nous allons montrer qu'un ordre de grandeur dans $\mathbb{N}$ sur les arbres de preuve diminue à chaque application d'une règle de $\rightsquigarrow$.

La **longueur** d'un arbre de preuve $\pi = (\pi_1, \dots, \pi_n, \varphi)_\iota$, noté $|\pi|$, est inductivement définie

par $\sum_{i=1}^n |\pi_i|$ si ι est une coupure ou une instance des règles de substitution et d'affaiblissement, et $\sum_{i=1}^n |\pi_i| + 1$ sinon, avec $|\varphi| = 0$ si φ est une feuille. Ainsi, la longueur d'un arbre de preuve est le nombre d'instances de règles apparaissant dans π autres que les coupures, les substitutions et les affaiblissements (i.e. les règles pour lesquelles il existe des transformations afin de les faire remonter ou les éliminer).

On définit l'ordre bien fondé $\geq$ sur les instances des règles d'inférence du calcul des séquents étendu de la façon suivante :

$$Cut > Sub., Aff. > @ \vdash, \vdash @, Ax.$$

où $@ \in \{\vee, \neg, \exists\}$

$$\begin{aligned} & \frac{\Gamma_1, \varphi_1 \vdash \Delta_1}{\Gamma_1 \vdash \Delta_1} \text{Coupure} > \frac{\Gamma_2, \varphi_2 \vdash \Delta_2}{\Gamma_2 \vdash \Delta_2} \text{Coupure} \\ & \iff \\ & |\varphi_1| > |\varphi_2| \\ & \frac{\Gamma_1 \vdash \Delta_1}{\sigma(\Gamma_1) \vdash \sigma(\Delta_1)} \text{Sub.} > \frac{\Gamma_2 \vdash \Delta_2}{\sigma(\Gamma_2) \vdash \sigma(\Delta_2)} \text{Sub.} \\ & \iff \\ & \sum_{\varphi \in \Gamma_1 \cup \Delta_1} |\varphi| > \sum_{\varphi \in \Gamma_2 \cup \Delta_2} |\varphi| \\ & \frac{\Gamma_1 \vdash \Delta_1}{\Gamma_1' \vdash \Delta_1'} \text{Aff.} > \frac{\Gamma_2 \vdash \Delta_2}{\Gamma_2' \vdash \Delta_2'} \text{Aff.} \\ & \iff \\ & \sum_{\varphi \in \Gamma_1 \cup \Delta_1} |\varphi| > \sum_{\varphi \in \Gamma_2 \cup \Delta_2} |\varphi| \end{aligned}$$

Il est aisé de voir que $\geq$ est un ordre bien fondé. On peut donc définir l'application d qui à tout élément plus petit pour l'ordre fait correspondre 0, puis 1 pour les autres éléments plus petits, et ainsi de suite.

Une preuve $\pi = (\pi_1, \dots, \pi_n, s)_\iota$ est dite **maximale** si, et seulement si pour tout i , $1 \leq i \leq n$, π_i est en forme normale mais π ne l'est pas.

Soit $\pi = (\pi_1, \dots, \pi_n, s)_\iota$ une preuve maximale (i.e. que ι est soit une coupure, soit une instance de la substitution ou d'affaiblissement). Alors, on définit le **rang** de π , noté $rk(\pi)$ comme $d(\iota) + |\pi|$.

La preuve du théorème repose sur le lemme intermédiaire suivant :

Lemme 159.

Supposons qu'un séquent s est prouvable et a une preuve $\pi : s$ qui est maximale. Alors, il existe une preuve $\pi' : s$ en forme normale.

Démonstration : La preuve du lemme se fait par induction sur le rang des preuves maximales.

Soit $\pi = (\pi_1, \dots, \pi_n, s)_\iota$ une preuve maximale et soit $(\iota_1, \dots, \iota_n, s)_\iota \rightsquigarrow \pi'$ une règle de transformation qui transforme π en $\bar{\pi}$ (i.e. $\pi \Rightarrow \bar{\pi}$). Soit $\bar{\pi}$ est en forme normale, ou alors il existe une sous-preuve maximale $\pi'' = (\pi_1'', \dots, \pi_n'', \varphi'')_{\iota''}$ in $\bar{\pi}$. On peut observer dans nos transformations, qu'aucune règle autre que la substitution et l'affaiblissement n'a été introduite dans π'' et au moins une a été retirée (en fait elle se trouve en dessous de la règle de coupure, de substitution ou d'affaiblissement). Donc, nous avons $|\pi''| < |\pi|$. Maintenant, pour toutes les règles de transformation dans $\rightsquigarrow$ les instances à la racine des membres gauche et droit respecte

l'ordre introduit par l'application d , nous avons donc $d(\iota) > d(\iota'')$. Nous pouvons donc écrire que $rk(\pi'') \leq rk(\pi) - 1$. Ainsi, par l'hypothèse d'induction, toute sous-preuve maximale de $\bar{\pi}$ peut être transformée en une preuve en forme normale. Ceci nous amène alors à une nouvelle preuve $\bar{\pi}'$ qui peut contenir quelques sous-preuves maximales. Mais en suivant les mêmes étapes que pour π'' , nous pouvons conclure que le rang de ces sous-preuves maximales est plus petit que le rang de π . Ceci nous permet alors de conclure que π peut être transformé en une preuve en forme normale. Ceci termine la preuve du lemme 159. ■

La preuve du théorème 158 se poursuit par induction sur la structure des preuves. Le cas de base est élémentaire et concerne les preuves réduites à des instances d'axiomes.

Maintenant, soit $\pi = (\pi_1, \dots, \pi_n, s)_\iota$ une preuve qui n'est pas en forme normale et n'est pas maximale. Par hypothèse d'induction, chaque π_i peut être transformée en une preuve $\bar{\pi}_i$ en forme normale. Ceci conduit à une preuve $\pi' = (\bar{\pi}_1, \dots, \bar{\pi}_n, s)_\iota$. π' est soit en forme normale, soit maximale. Dans le dernier cas, par le lemme 159, π' peut être transformée en une preuve $\bar{\pi} : s$ en forme normale. Ceci finit la preuve du théorème 158. ■

Le calcul des séquents sans coupure et affaiblissement a des propriétés formelles importantes entre autres pour ses liens avec la démonstration automatique. En effet, si on examine les règles, on remarque que chaque formule apparaissant en conclusion a ses sous-formules directes qui apparaissent déjà en prémisses selon une définition précise de la notion de sous-formule donnée ci-dessous.

Définition 120 (Sous-formule).

L'ensemble des **sous-formules** d'une formule φ contient φ , et

- si φ est $\varphi_1 @ \varphi_2$ avec $@ \in \{\wedge, \vee, \Rightarrow\}$, l'ensemble des sous-formules de φ_1 et φ_2 ,
- si φ est $\neg\psi$, l'ensemble des sous-formules de ψ ,
- si φ est $\exists x.\psi$ ou $\forall x.\psi$, des sous-formules de $\psi[x/t]$ pour tout terme t .

Enfin, les sous-formules d'un séquent sont toutes les sous-formules des formules qui le composent.

Le résultat suivant est immédiat, par simple inspection des règles.

Proposition 160.

Le calcul des séquents sans coupures et sans affaiblissement vérifie la propriété de la sous-formule : les formules figurant dans une dérivation d'un séquent sont des sous-formules de ce séquent.

Ceci a des conséquences immédiates dans le domaine de la démonstration automatique. Si on cherche à construire une dérivation d'un séquent, l'espace de recherche est restreint aux sous-formules du séquent que l'on cherche à prouver. Dans le cas du calcul des séquents pour la logique propositionnelle (on ne considère pas les règles associées aux quantificateurs), cet espace est fini, et l'on prouve ainsi que ce calcul est décidable. Avec les quantificateurs, il y a une infinité de sous-formules dès qu'il y a une infinité de termes (ce qui est souvent le cas).

22.4. Résolution

Problème d'unification

Définition 121.

Soit $T_\Sigma(\mathcal{V})$ un ensemble de termes.

Un **problème d'unification** est de l'une des formes suivantes :

- $\top$
- $\perp$
- $\{s_1 = t_1, \dots, s_n = t_n\}$ avec pour tout i dans $[1, n]$, s_i et t_i termes de $T_\Sigma(\mathcal{V})$.

Toute substitution est solution de $\top$, aucune substitution n'est solution de $\perp$ et σ est solution de $\{s_1 = t_1, \dots, s_n = t_n\}$ si pour tout i dans $[1, n]$, $s_i\sigma = t_i\sigma$.

L'ensemble des solutions d'un problème d'unification P est noté $\mathcal{U}(P)$.

Définition 122.

Deux problèmes d'unification P_1 et P_2 définis sur $T_\Sigma(\mathcal{V})$ sont équivalents si leurs ensembles de solutions $\mathcal{U}(P_1)$ et $\mathcal{U}(P_2)$ sont égaux.

Problème résolu

Définition 123.

Soit P un problème d'unification défini sur $T_\Sigma(\mathcal{V})$. P est en **forme résolue** si P s'écrit sous l'une des formes suivantes :

- $\top$
- $\perp$
- $\{v_1 = t_1, \dots, v_n = t_n\}$ où les v_i sont des variables deux à deux distinctes, et vérifient $v_i \notin \cup_j \text{Var}(t_j)$.

Proposition 161.

Soit $P = \{v_1 = t_1, \dots, v_n = t_n\}$ un problème d'unification en forme résolue, et soit θ la substitution $[v_1 \mapsto t_1, \dots, v_n \mapsto t_n]$.

L'ensemble des solutions de P est égal à l'ensemble des instances de θ (la substitution définie par le problème P), i.e. à l'ensemble :

$$\mathcal{U}(P) = \{\theta\sigma \mid \sigma \text{ substitution} \}$$

Démonstration : Soit σ une solution de P . Montrons que $\sigma = \theta\sigma$.

Soit v_i une variable du problème P , donc appartenant au domaine de θ .

$$\begin{aligned}
v_i(\theta\sigma) &= (v_i\theta)\sigma \\
&= t_i\sigma && \text{par définition de } \theta \\
&= v_i\sigma && \text{car } \sigma \in \mathcal{U}(P)
\end{aligned}$$

Soit w une variable qui n'appartient pas à $Dom(\theta)$.

$$\begin{aligned}
w(\theta\sigma) &= (w\theta)\sigma \\
&= w\sigma && \text{car } w\theta = w
\end{aligned}$$

donc on a bien $\sigma = \theta\sigma$.

Montrons qu'une instance de θ , notée $\theta\sigma$ est solution de P .

$$\begin{aligned}
v_i(\theta\sigma) &= (v_i\theta)\sigma \\
&= t_i\sigma \\
&= t_i\theta\sigma && \text{car } Var(t_i) \cap Dom(\theta) = \emptyset \text{ et donc } t_i\theta = t_i \\
&= t_i(\theta\sigma)
\end{aligned}$$

■

Définition 124.

Soit $P = \{v_1 = t_1, \dots, v_n = t_n\}$ un problème d'unification en forme résolue.
La substitution $[v_1 \mapsto t_1, \dots, v_n \mapsto t_n]$ est appelée **unificateur principal** de P , ou **unificateur plus général** de P .

Règles d'unification

Définition 125.

(Règles d'unification)

Les règles d'unification transforment les problèmes d'unification et sont définies par l'un des schémas ci-dessous :

<i>Trivial</i>	$\frac{\{v = v\} \cup P}{P}$	
<i>Orientation</i>	$\frac{\{t = v\} \cup P}{\{v = t\} \cup P}$	si $v \in \mathcal{V}$ et $t \notin \mathcal{V}$
<i>Decomposition</i>	$\frac{\{f(u_1, \dots, u_n) = f(t_1, \dots, t_n)\} \cup P}{\{u_1 = t_1, \dots, u_n = t_n\} \cup P}$	
<i>Incompatibilité</i>	$\frac{\{f(u_1, \dots, u_n) = g(t_1, \dots, t_m)\} \cup P}{\perp}$	si $f \neq g$
<i>Union</i>	$\frac{\{v = w\} \cup P}{\{v = w\} \cup P[v \mapsto w]}$	si $v, w \in \mathcal{V}$ et $v \in \text{Var}(P)$
<i>Remplacement</i>	$\frac{\{v = t\} \cup P}{\{v = t\} \cup P[v \mapsto t]}$	si $v \in \mathcal{V}$ et $v \in \text{Var}(P)$ et $t \notin \mathcal{V}$ et $v \notin \text{Var}(t)$
<i>Fusion</i>	$\frac{\{v = t, v = u\} \cup P}{\{v = t, t = u\} \cup P}$	si $v \in \mathcal{V}$ et $t, u \notin \mathcal{V}$ et $ t \leq u $ (au sens de la taille des termes)
<i>Test d'occurrence</i>	$\frac{\{v_1 = t_1, \dots, v_n = t_n\} \cup P}{\perp}$	si pour tout $i < n$, t_i (resp. t_n) contient v_{i+1} (resp. v_1) comme sous-terme strict

$P \rightarrow_U P'$ indique que le problème d'unification P a été transformé en le problème P' par l'un des 7 règles ci-dessus.

Théorème 162.

(Equivalence des problèmes d'unification par application des règles) Les règles de transformation transforment un problème d'unification en un problème d'unification équivalent.

Démonstration : En tirant parti que l'égalité des termes est une relation de congruence sur les termes, chaque règle préserve l'ensemble des substitutions qui assurent l'égalité syntaxique des termes.

En ce qui concerne la règle du Test d'occurrence, la prémisse indique que (par transitivité de l'égalité) v_1 est sous-terme strict de lui (en remplaçant successivement v_2 jusqu'à v_n par les termes

t_i associés). Or aucune substitution ne peut unifier une variable avec un terme contenant cette variable comme sous-terme strict. ■

Théorème 163.

(Terminaison) Soit P un problème d'unification. Il n'existe pas de séquence infinie

$$P = P_0 \rightarrow_U P_1 \dots \rightarrow_U P_n \rightarrow_U P_{n+1} \dots$$

telle que P_{n+1} est obtenu à partir de P_n en appliquant une des règles d'unification.

Démonstration : Il suffit de munir l'ensemble des problèmes d'unification d'une relation d'ordre bien fondée.

$P_1 < P_2$ si $\Phi(P_1) <_{lex} \Phi(P_2)$ avec $<_{lex}$ ordre lexicographique défini sur les triplets $\Phi(P_i) = (r_i, M_i, v_i)$ pour $i = 1, 2$ avec

- r_i nombre de variables résolues du problème P_i .
Une variable v est dite résolue pour P_i si v apparaît exactement une fois dans P_i , comme membre gauche d'une équation (et n'apparaît pas dans les autres composantes de P_i , i.e. autres équations ou second membre de l'équation).
- M_i est le multiensemble des tailles des équations avec $|s = t| = \max(|s|, |t|)$ et $|s|$ nombre total de symboles fonctionnels et de variables apparaissant dans s .
 $M' > M$ si M est obtenu à partir de M' en enlevant un élément x et en ajoutant des éléments de taille strictement plus petite.
- v_i nombre d'équations de P_i dont l'un des membres est une variable.

L'ordre lexicographique $<_{lex}$ induit par le triplet (r, M, v) est bien fondé, car il est construit sur les ordres (sur $\mathbb{N}$ et multiensembles sur $\mathbb{N}$) qui sont bien-fondés.

L'application d'une règle d'unification fait décroître selon l'ordre lexicographique : si $P \rightarrow_U P'$, on a $P' <_{lex} P$.

En effet, chaque règle diminue l'une des composantes, de sorte à garantir la décroissance stricte de l'ordre lexicographique.

Trivial	$\geq$	$>$	
Orientation	$\geq$	$\geq$	$>$
Decomposition	$\geq$	$>$	
Incompatibilité	$\geq$	$>$	
Union	$>$		
Remplacement	$>$		
Fusion	$\geq$	$\geq$	$>$
Test d'occurrence	$\geq$	$>$	

Théorème 164.

(Complétude)

Soit P un problème d'unification pour lequel il n'est pas possible d'appliquer une règle d'unification.

Alors P est soit de la forme $\top$ ou $\perp$, soit en forme résolue.

Démonstration : Si P est de la forme $\top$ ou $\perp$, plus aucune règle n'est applicable.

Supposons que P n'est pas réduit à $\top$ ou $\perp$.

Les équations de P contiennent nécessairement une variable comme membre de l'équation. En effet, sinon, les deux membres sont des termes non réduits à une variable (dont le symbole de tête est un symbole fonctionnel) : si les symboles de tête des deux membres sont identiques, la règle Décomposition s'applique, si les symboles de tête des deux membres sont différents, la règle Incompatibilité s'applique.

Toutes les règles contiennent donc au moins une variable comme membre gauche. En effet, si le terme gauche n'est pas une variable, la règle Orientation peut s'appliquer. De plus, comme la règle Fusion ne peut s'appliquer, les variables apparaissant comme membres gauches d'une équation sont deux à deux distinctes.

Considérons v_i l'une des variables apparaissant comme membre gauche d'une équation ($v_i = t_i \in P$ avec $v_i \in \mathcal{V}$).

- v_i ne peut apparaître comme sous-terme strict de t_i avec $v_i = t_i \in P$ car la règle Test d'occurrence s'appliquerait alors ;
 - v_i ne peut être t_i , car sinon la règle Trivial s'appliquerait alors ;
 - si t_i n'est pas une variable, v_i ne peut être un sous-terme strict de l'un des t_j avec $i \neq j$ car sinon, la règle Remplacement pourrait s'appliquer alors (car d'après le premier point v_i n'est pas une variable de t_i) ;
 - si t_i est une variable, notée v'_i , v_i n'apparaît nulle part ailleurs dans P , sinon la règle Union s'appliquerait alors.
- P peut alors se voir comme

$$\{v_1 = t_1, \dots, v_{i-1} = t_{i-1}, v_i = v'_i, v_{i+1} = t_{i+1}, \dots, v_n = t_n\}$$

avec v_i n'apparaissant pas ailleurs dans P .

Puisque les termes t_i ne contiennent aucune des variables $v_1, \dots, v_n$, P est en forme résolue. ■

Substitution principale (ou unificateur principal)

Corollaire 8.

Soit P un problème d'unification et $\mathcal{U}(P)$ l'ensemble des solutions. Alors si $\mathcal{U}(P)$ n'est pas vide, $\mathcal{U}(P)$ admet une substitution principale, unique à renommage près.

Cette substitution est appelée **unificateur principal** de P et est notée $mgu(P)$ (pour *most general unifier*).

Démonstration : Dénотons P' le problème d'unification en forme résolue et équivalent à P . P' est obtenu à partir de P en appliquant les règles d'unification tant que cela est possible (cf résultats précédents). Selon la forme de P' :

- P' est de la forme $\top$: n'importe quelle substitution est solution de P' et de P ($\mathcal{U}(P)$ n'est donc pas vide). La substitution de domaine vide est substitution principale de P .
- P' est de la forme $\perp$, et $\mathcal{U}(P)$ et $\mathcal{U}(P')$ sont vides.
- P' se présente sous la forme $\{v_1 = t_1, \dots, v_n = t_n\}$. Soit θ la substitution $[v_1 \mapsto t_1, \dots, v_n \mapsto t_n]$.

D'après la proposition sur les problèmes en forme résolue, $\mathcal{U}(P')$ est égale à l'ensemble des substitutions, instances de θ . $\mathcal{U}(P)$ n'est pas vide et admet θ comme substitution principale.

Supposons que $\mathcal{U}(P)$ n'est pas vide et admet deux substitutions principales θ et θ' .

Alors il existe deux substitutions ρ et ρ' vérifiant :

$$\theta = \theta' \rho'$$

$$\theta' = \theta \rho$$

On a alors $\theta = \theta\rho\rho'$ (et $\theta' = \theta'\rho'\rho$). $\rho\rho'$ est l'identité sur les variables du codomaine de θ (i.e. les variables apparaissant dans les termes $x\theta$ avec $x \in \text{Dom}(\theta)$). De même, $\rho'\rho$ est l'identité sur les variables du codomaine de θ' . Et θ' est un renommage de θ . ■

Calcul par résolution

En mettant sous forme prénexe et skolem les formules du premier ordre, on peut se ramener à un ensemble de formules du 1er ordre sans quantificateurs (donc implicitement universellement quantifiées), que l'on peut mettre sous forme normale conjonctive, c'est-à-dire comme la conjonction de clauses.

Les clauses du premier ordre sont les formules de la forme

$$C : l_1 \vee l_2 \dots \vee l_n$$

avec l_i littéral, de la forme $r(t_1, \dots, t_n)$ ou $\neg r(t_1, \dots, t_n)$ avec r prédicat, et t_i termes (avec variables).

$\alpha(C)$ (ou $C\alpha$) est la clause obtenue par application de la substitution $\alpha : V \rightarrow T_\Sigma(V)$ à la clause C . Pour $\alpha : V \rightarrow T_\Sigma$ (ensemble des termes clos), $\alpha(C)$ est appelée instance close de C .

Exemple de clause :

$$r(x, f(a), y) \vee p(f(z)) \vee \neg q(x, g(x, y))$$

$$r(c, f(a), f(a)) \vee p(f(g(c, c))) \vee \neg q(c, g(c, f(a)))$$

est une instance close (avec $\alpha = [x \mapsto c, y \mapsto f(a), z \mapsto g(c, c)]$).

Résolution étendue

Définition 126.

Soient deux clauses de la forme $P \cup C$ et $N \cup D$ (ou $P \vee C$ et $N \vee D$) avec :

- P, N, C et D sont des clauses (ensemble de littéraux) ;
- $P \cap C = N \cap D = \emptyset$
- $\text{Var}(C) \cap \text{Var}(D) = \emptyset$ (si ce n'est pas le cas, renommer les variables, car les clauses sont implicitement universellement quantifiées)
- il existe α et β deux substitutions, A formule atomique vérifiant

$$P\alpha = \{A\} \text{ et } N\beta = \{\neg A\}$$

(ou réciproquement)

$C\alpha \cup D\beta$ est une **résolvante étendue** de $P \cup C$ et $N \cup D$ par rapport aux sous-clauses P et N .

Remarque : il n'est pas requis que P et N soient des singletons, mais ils doivent être réduits à un singleton après substitution.

Exemple 165.

Soient les deux clauses

- $C_1 = \{\neg r(x, f(y)), s(x, f(z))\}$
- $C_2 = \{r(w, f(w)), r(f(v), u)\}$

Soit $\alpha = [x \mapsto f(a), y \mapsto f(a), z \mapsto f(a)]$ et $\beta = [w \mapsto f(a), v \mapsto a, u \mapsto f(f(a))]$ deux substitutions.

- $C\alpha = \{\neg r(f(a), f(f(a))), s(f(a), f(f(a)))\}$
- $D\beta = \{r(f(a), f(f(a)))\}$ (singleton)

Une résolvante étendue de C et D est $\{s(f(a), f(f(a)))\}$

(par rapport à $r(f(a), f(f(a)))$)

Définition 127.

(Règles de la résolution étendue $\vdash_{RE}$)

- dérivation de résolvante étendue à partir de clauses (en considérant deux substitutions α et β)
- élimination des littéraux redondants (idem résolution en logique propositionnelle)

$$\frac{P \vee C \quad N \vee D}{C\alpha \vee D\beta} \text{ si } P\alpha = A \text{ et } N\beta = \neg A \text{ et } Var(C) \cap Var(D) = \emptyset$$

$$\frac{P \vee P' \vee C}{P\alpha \vee C\alpha} \text{ si } P\alpha = A \text{ et } P'\alpha = A$$

Théorème 166.

(Correction et complétude de la résolution étendue)

- Si deux clauses C et D ont E pour résolvante étendue, alors $\{C, D\} \models E$
- **Correction de $\vdash_{RE}$**
Pour Γ ensemble de clauses, si $\Gamma \vdash_{RE} E$, alors $\Gamma \models E$
Si $\Gamma \vdash_{RE} \square$, alors Γ est insatisfiable.
- **Complétude du raisonnement par réfutation avec de $\vdash_{RE}$**
Si Γ est un ensemble de clauses insatisfiable, alors $\Gamma \vdash_{RE} \square$

Démonstration : S'établit en se ramenant au cas propositionnel et s'appuyant que Γ (formules universelles) et $ground(\Gamma)$, ensemble des instances closes de Γ sont (in)satisfiables en même temps (modèle de Herbrand). ■

Règle de la résolution

Sans surprise, l'obtention via des moyens algorithmiques des substitutions α et β se fera à l'aide de l'unification.

Définition 128.

Une clause est une **résolvante** des clauses C et D par rapport à $P \subseteq C$ et $N \subseteq D$ si elle est plus générale dans la classe des résolvantes étendues de C et D par rapport à P et N (i.e. si elle s'obtient en appliquant une substitution la plus générale dans l'ensemble des substitutions permettant de calculer une résolvante étendue de C et D par rapport à P et N).

La définition est bien fondée car la donnée de P et de N conduit à la définition d'un problème d'unification.

Proposition 167.

- une résolvante (aux sous-ensembles P et N fixés) est unique à renommage près.
- pour toute résolvante étendue, il existe une résolvante, qui lui est plus générale.

Définition 129.

Règle de la résolution

$$\frac{r(t_1, \dots, t_n) \vee C \quad \neg r(s_1, \dots, s_n) \vee D}{C\sigma \vee D\sigma} \text{ si } \sigma \text{ mgu du problème } \{t_1 = s_1, \dots, t_n = s_n\} \text{ et } \text{Var}(C) \cap \text{Var}(D) = \emptyset$$

$$\frac{r(t_1, \dots, t_n) \vee r(s_1, \dots, s_n) \vee C}{r(t_1, \dots, t_n)\sigma \vee C\sigma} \text{ si } \sigma \text{ mgu du problème d'unification } \{t_1 = s_1, \dots, t_n = s_n\}$$

La résolution est correcte et complète pour la réfutation.
(démonstration non développée ici, mais fondée sur les propriétés de la résolution étendue et le fait qu'une résolvante étendue est une instance d'une résolvante)

Langage Prolog

À FAIRE.

22.5. Méthode des tableaux

À FAIRE.

23. Logique équationnelle et réécriture

23.1. Introduction

Les équations sont naturelles en mathématique et principalement en algèbre, mais aussi dans toutes les sciences fortement mathématisées telles que la physique et l'informatique. Nous pourrions bien entendu traiter l'égalité comme un prédicat binaire dans la logique des prédicats du premier ordre, ce que nous avons d'ailleurs fait quand nous avons présenté la logique des prédicats avec égalité en section 21.5. Le problème est qu'alors nous ne profitons pas du raisonnement naturel et concis classiquement associé à l'égalité qu'est le **remplacement d'égal par égale** et que l'on peut traduire simplement par :

$$\frac{x=y \quad P(x)}{P(y)}$$

où P est une propriété portant sur x et y .

Pour formaliser un tel raisonnement, les logiciens, essentiellement D. Birkhoff, ont défini la **logique équationnelle**. La logique équationnelle peut alors être vue comme une restriction de la logique des prédicats du premier ordre au seul prédicat d'égalité. Cette logique s'est montrée très adaptée pour formaliser les structures algébriques telles que les groupes, les anneaux, les corps, etc., mais aussi les structures de données informatiques telles que les listes, les piles, les files, les arbres, etc. C'est pourquoi, la logique équationnelle est aussi appelée **spécifications algébriques** surtout dans son utilisation pour la modélisation des structures de données et donc des programmes dits standards (i.e. des programmes retournant un résultat en manipulant des structures de données complexes mais des structures de contrôle relativement simples)¹.

23.2. Logique équationnelle

Syntaxe

Comme il est maintenant usuel dans la présentation d'une logique, nous allons commencer par le volet syntaxique de la logique équationnelle. La logique équationnelle étant une restriction de la logique des prédicats du premier ordre, nous allons retrouver les trois notions que sont les signatures, les termes et les formules au dessus de ces signatures. Pour rendre plus simple l'utilisation de la logique équationnelle à la description des structures de données, un ingrédient supplémentaire a été ajouté, les **types**. Ces derniers vont permettre de partitionner l'ensemble des valeurs et spécialiser le profil des fonctions implicitement, c'est-à-dire sans être obligé d'écrire les axiomes permettant un tel découpage. Cette extension peut bien entendu être appliquée de la même façon à la logique des prédicats du premier ordre.

1. Ces programmes se différencient alors de l'autre famille de programmes en interaction avec l'extérieur telles que les systèmes embarqués, les capteurs, les protocoles de communications, etc., qui la plupart du temps ne rendent aucun résultat mais plutôt maintiennent une interaction permanente avec leur environnement. Ces programmes, appelés aussi souvent **systèmes réactifs**, se définissent alors par des structures de données simples (voir inexistantes) mais des structures de contrôle compliquées (e.g. parallélisation et entrelacement de processus).

Définition 130 (Signature).

Une **signature** Σ est un triplet (S, F, V) où :

- S est un ensemble (au sens mathématique du terme) dont les éléments sont appelés **sortes** ou **types**,
- F est un ensemble de noms de fonction, chacun muni d'un profil dans S^+ , et
- V est une famille indexée par S d'ensemble V_s telle que pour chaque $s \in S$, $V_s \cap F = \emptyset$. Pour chaque $s \in S$, les éléments de V_s sont appelés **variables de type** s .

Dans la suite, on préférera noter $f : s_1 \times \dots \times s_n \rightarrow s_{n+1}$ pour désigner un nom de fonction f de F muni du profil $s_1 \dots s_{n+1} \in S^+$. $s_1 \times \dots \times s_n$ est appelé le **domaine** de f et s_{n+1} son co-domaine.

Exemple 168.

Soit **Sig-Arith** la signature suivante :

Types : $S = \{ nat \}$

Fonctions : $F = \{ 0 : \rightarrow nat ,$
 $succ : nat \rightarrow nat ,$ (*passage au successeur*)
 $_ + _ : nat \times nat \rightarrow nat \}$ (*addition de deux entiers*)
 $_ \times _ : nat \times nat \rightarrow nat \}$ (*multiplication de deux entiers*)

$V_{nat} = \{ x, y \}$

La signature **Sig-Arith** sera associée dans la suite à la spécification de l'arithmétique élémentaire.

Exemple 169.

Soit **Sig-Liste** la signature suivante :

Types : $S = \{ liste, elem, nat \}$

Fonctions : $F = \{ [] : \rightarrow liste ,$
 $_ :: _ : elem \times liste \rightarrow liste ,$ (*ajout d'un élément dans la liste*)
 $@ : liste \times liste \rightarrow liste ,$ (*concaténation de deux listes*)
 $long : liste \rightarrow nat ,$ (*longueur d'une liste*)
 $0 : \rightarrow nat ,$
 $succ : nat \rightarrow nat \}$ (*passage au successeur*)

$V_{liste} = \{ l, l' \}, V_{elem} = \{ e \}, V_{nat} = \emptyset$

La signature **Sig-Liste** sera associée dans la suite à la spécification de la structure de données des listes d'éléments.

Exemple 170.

Soit **Sig-Pile** la signature suivante :

Types : $S = \{ pile, elem, nat \}$

Fonctions : $F = \{ \begin{array}{l} pile_vide : \rightarrow pile, \text{ (désigne la pile vide)} \\ empiler : elem \times pile \rightarrow pile, \text{ (ajout dans la pile)} \\ depiler : pile \rightarrow pile, \text{ (retire l'élément au sommet)} \\ sommet : pile \rightarrow elem, \text{ (rend le sommet de la pile)} \\ hauteur : pile \rightarrow nat, \text{ (hauteur de la pile)} \\ 0 : \rightarrow nat, \\ succ : nat \rightarrow nat \end{array} \}$ (passage au successeur)

$V_{pile} = \{p\}, V_{elem} = \{e\}, V_{nat} = \emptyset$

La signature **Sig-Pile** sera associée dans la suite à la spécification de la structure de données des piles d'éléments.

Exemple 171.

Soit **Sig-Tab** la signature suivante :

Types : $S = \{ tableau, elem, indice, bool \}$

Fonctions : $F = \{ \begin{array}{l} val_init : \rightarrow elem, \text{ (valeur initiale dans le tableau)} \\ init : \rightarrow pile, \text{ (initialisation du tableau)} \\ _[_] := _ : tableau \times indice \times elem \rightarrow tableau, \text{ (affectation)} \\ _[_] : tableau \times indice \rightarrow elem, \text{ (accès à un élément)} \\ eq? : indice \times indice \rightarrow bool, \text{ (test l'égalité entre indice)} \\ vrai, faux : \rightarrow bool \end{array} \}$ (constantes booléennes)

$V_{tableau} = \{t\}, V_{elem} = \{e, e'\}, V_{indice} = \{i, j\}$

La signature **Sig-Tab** sera associée dans la suite à la spécification de la structure de données des tableaux d'éléments.

Termes du premier ordre

Suivant la même démarche que pour la logique des prédicats du premier ordre, à partir d'une signature Σ , on peut définir l'ensemble des termes avec variables sur cette signature. La définition est sensiblement identique à celle que nous avons donnée pour la logique des prédicats du premier ordre à ceci près que nous devons respecter le typage des fonctions. On va donc définir une partition de l'ensemble des termes indexée par l'ensemble des types S de la signature.

Définition 131 (Termes avec variables).

Soit $\Sigma = (S, F, V)$ une signature. Pour tout $s \in S$, on définit l'ensemble $T_\Sigma(V)_s$ comme le plus petit ensemble au sens de l'inclusion satisfaisant les clauses suivantes :

- $V_s \subseteq T_\Sigma(V)_s$,
- pour tout $f : \rightarrow s \in F$, $f \in T_\Sigma(V)_s$, et
- pour tout $f : s_1 \times \dots \times s_n \rightarrow s \in F$ et pour tout $(t_1, \dots, t_n) \in T_\Sigma(V)_{s_1} \times \dots \times T_\Sigma(V)_{s_n}$, $f(t_1, \dots, t_n) \in T_\Sigma(V)_s$.

On note $T_\Sigma(V) = (T_\Sigma(V)_s)_{s \in S}$ la famille d'ensembles indexée par S , et pour tout $s \in S$, les éléments de $T_\Sigma(V)_s$ sont appelés **terme avec variables de type s** . L'ensemble des termes clos est la famille d'ensembles $(T_\Sigma(\emptyset))_{s \in S}$ où pour chaque type $s \in S$, l'ensemble des variables est l'ensemble vide. On le notera dans la suite simplement T_Σ .

Formules équationnelles

À partir des termes avec variables, nous pouvons maintenant énoncer l'objet de notre syntaxe qui est la définition des formules. Ces dernières se définissent comme pour la logique des prédicats du premier order à ceci près que les formules atomiques se restreignent aux équations. Formellement, une équation se définit de la façon suivante :

Définition 132 (Équations).

Soit $\Sigma = (S, F, V)$ une signature. Une Σ -**équation** est une paire de termes avec variables (t, t') de même type. On la note $t = t'$.

À partir des équations, on peut obtenir les formules plus générales en clôturant l'ensemble des formules bien formées par les connecteurs propositionnels et les quantificateurs en suivant la définition 106.

Comme il est habituel de le faire en logique mathématique, on peut regrouper des formules portant sur une même signature et définir ainsi un système axiomatique. En informatique, on parle alors de **spécification**.

Définition 133 (Spécification).

Une **Spécification** SP est une paire (Σ, Ax) où Σ est une signature et Ax est un ensemble de Σ -formules. Les formules de Ax sont souvent appelées **axiomes** de la spécification SP .

Bien entendu, en pratique, l'ensemble des axiomes d'une spécification est fini.

Exemple 172.

À partir de la signature **Sig-Arith** donnée dans l'exemple 168, on peut associer l'ensemble **Ax-Arith** des axiomes suivant :

Axiomes :

$$\begin{aligned}x + 0 &= x \\x + succ(y) &= succ(x + y) \\x \times 0 &= 0 \\x \times succ(y) &= x + (x \times y)\end{aligned}$$

La paire (**Sig-Arith**,**Ax-Arith**) définit la spécification de l'arithmétique élémentaire **SP-Arith**.

Exemple 173.

À partir de la signature **Sig-Liste** donnée dans l'exemple 169, on peut associer l'ensemble **Ax-Liste** des axiomes suivant :

Axiomes :

$$\begin{aligned}\square @ l &= l \\(e :: l) @ l' &= e :: (l @ l') \\long(\square) &= 0 \\long(e :: l) &= succ(long(l))\end{aligned}$$

La paire (**Sig-Liste**,**Ax-Liste**) définit la spécification de la structure des données des listes d'éléments **SP-Liste**.

Exemple 174.

À partir de la signature **Sig-Pile** donnée dans l'exemple 170, on peut associer l'ensemble **Ax-Pile** des axiomes suivant :

Axiomes :

$$\begin{aligned}depiler(pile_vide) &= pile_vide \\depiler(empiler(e, p)) &= p \\somet(empiler(e, p)) &= e \\hauteur(pile_vide) &= 0 \\hauteur(empiler(e, p)) &= succ(hauteur(p))\end{aligned}$$

La paire (**Sig-Pile**,**Ax-Pile**) définit la spécification de la structure des données des piles d'éléments **SP-Pile**.

Remarquons que la spécification ci-dessus est « incomplète » parce qu'elle ne fixe pas la valeur de la fonction *somet* sur la pile vide.

Exemple 175.

À partir de la signature **Sig-Tab** donnée dans l'exemple 171, on peut associer l'ensemble **Ax-Tab** des axiomes suivant :

Axiomes :

$$\begin{aligned} init[i] &= val_init \\ (t[i] := e)[i] &= e \\ i = j &\Rightarrow (t[i] := e)[j] := e' = t[j] := e' \\ \neg(i = j) &\Rightarrow (t[i] := e)[j] := e' = (t[j] := e')[i] := e \end{aligned}$$

La paire (**Sig-Tab**, **Ax-Tab**) définit la spécification de la structure des données des tableaux d'éléments **SP-Tab**.

Quelques explications sur les axiomes ci-dessus :

- le premier axiome dit que tout tableau initialisé voit le contenu à chacun de ses indices être à la valeur *val_init*.
- Les deux axiomes suivants disent que le contenu d'un tableau à un indice donné est le dernier élément qui y a été affecté. Ainsi, toute affectation à indice donné écrase l'élément contenu précédemment.
- Enfin, le dernier axiome dit que pour deux indices différents l'ordre d'affectation n'importe pas.

Dans la démarche du génie-logiciel (ensemble de techniques et de méthodes permettant de concevoir et valider des logiciels), l'intérêt des formalisme de spécification est de permettre d'écrire des spécifications abstraites, c'est-à-dire s'intéressant au **quoi** du logiciel (*qu'est-ce que le logiciel est supposé faire*) mais pas au **comment** (*comment il est supposé le faire*). Sinon, nous pouvons directement utiliser les langages de programmation qui sont eux-mêmes des langages formels pour raisonner sur les logiciels.

L'avantage d'une spécification abstraite est que souvent elle est plus concise et plus claire, et donc plus facile à manipuler. C'est simplement au travers des étapes successives de raffinement que des choix de décision et d'algorithmes vont être faits permettant d'aborder ces aspects du "comment". Pour illustrer ce propos prenons l'exemple simple suivant :

Exemple 176.

Supposons que nous voulions spécifier le PGCD (Plus Grand Commun Diviseur). Pour cela, nous devons ajouter à la signature **Sig-Arith** les trois fonctions suivantes :

1. $pgcd : nat \times nat \rightarrow nat$ qui calcule le pgcd de deux nombres,
2. $_mod_ : nat \times nat \rightarrow nat$ qui calcule le reste de la division des deux nombres passés en argument, et
3. $_ \geq _ : nat \times nat \rightarrow bool$ qui teste si un nombre est plus grand ou égale qu'un autre.

Il existe plusieurs algorithmes permettant de calculer le pgcd de deux nombres mais tous répondent à la même spécification abstraite représentée par les trois axiomes suivants :

Axiomes :

$$\begin{aligned} n \bmod pgcd(n, m) &= 0 \\ m \bmod pgcd(n, m) &= 0 \\ n \bmod p = 0 \wedge m \bmod p = 0 &\Rightarrow pgcd(n, m) \geq p = vrai \end{aligned}$$

Maintenant, une spécification plus concrète peut-être donnée par l'algorithme d'Euclide. Ce qui donne :

Types : $S = \{ nat, bool \}$

Fonctions : $F = \{ 0 : \rightarrow nat, \text{succ} : nat \rightarrow nat, (\text{passage au successeur})$
 $_ - _ : nat \times nat \rightarrow nat, (\text{soustraction de deux entiers})$
 $pgcd : nat \times nat \rightarrow nat,$
 $> : nat \times nat \rightarrow bool,$
 $eq? : nat \times nat \rightarrow bool,$
 $vrai, faux : \rightarrow bool \}$

$V_{nat} = \{x, y\}$

Axiomes :

$$\begin{aligned} n = m &\Rightarrow pgcd(n, m) = m \\ \neg(n = m) \wedge m > n = vrai &\Rightarrow pgcd(n, m) = pgcd(n, m - n) \\ \neg(n = m) \wedge m > n = faux &\Rightarrow pgcd(n, m) = pgcd(n - m, m) \end{aligned}$$

Une autre spécification concrète plus efficace que l'algorithme d'Euclide est le suivant :

Types : $S = \{ nat, bool \}$

Fonctions : $F = \{ 0 : \rightarrow nat, \text{succ} : nat \rightarrow nat, (\text{passage au successeur})$
 $_mod_ : nat \times nat \rightarrow nat,$
 $pgcd : nat \times nat \rightarrow nat,$
 $> : nat \times nat \rightarrow bool,$
 $eq? : nat \times nat \rightarrow bool,$
 $vrai, faux : \rightarrow bool \}$

$V_{nat} = \{x, y\}$

Axiomes :

$$\begin{aligned} m > n = vrai &\Rightarrow pgcd(n, m) = pgcd(m, n) \\ m > n = faux \wedge n \bmod m = 0 &\Rightarrow pgcd(n, m) = m \\ m > n = faux \wedge \neg(n \bmod m = 0) &\Rightarrow pgcd(n, m) = pgcd(m, n \bmod m) \end{aligned}$$

Ce que nous avons pu constater au travers de l'exemple précédent est que la spécification abstraite et les deux spécifications concrètes définissant des réalisations possibles (i.e. des implantations), sont toutes définies dans le même formalisme ce qui permet d'aborder la correction partielle (on ne s'intéresse pas à la terminaison des programmes ici) des implantations en restant dans un cadre homogène.

Sémantique

Commençons alors par donner un sens mathématique aux signatures. Ce qui définit mathématiquement un type de données est la donnée d'un ensemble muni de lois internes et externes et d'éléments distingués. Par exemple, la structure de données des listes se définit comme l'ensemble de tous les mots finis que l'on peut définir à partir d'un ensemble d'éléments E pris comme alphabet. Cet ensemble est alors muni du mot vide pour désigner la liste vide (c'est l'élément distingué), de la loi externe de concaténation d'une lettre en tête d'un mot pour l'opération ajouter, de la loi interne de concaténation pour concaténer deux listes, et enfin de la fonction qui calcule la taille d'un mot pour la longueur d'une liste.

Ceci est semblable à la définition des structures algébriques usuelles en algèbre telles les groupes, les anneaux, les corps, etc. La différence essentielle entre ces structures algébriques et les types de données est que ces derniers sont définis inductivement. Ils sont même librement engendrés. C'est ce qui nous permet de programmer dessus.

Définition 134 (S-ensemble).

Soit S un ensemble de types. Un **S -ensemble** A est une famille d'ensembles indexée par S , c'est-à-dire $A = (A_s)_{s \in S}$.

Définition 135 (Σ -algèbre).

Soit $\Sigma = (S, F, V)$ une signature. Une **Σ -algèbre** $\mathcal{A}$ est un S -ensemble muni pour chaque nom de fonction $f : s_1 \times \dots \times s_n \rightarrow s \in F$ par une application $f^{\mathcal{A}} : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$.

Ainsi, les Σ -algèbres sont des structures du premier ordre à ceci près que l'ensemble des valeurs est maintenant partitionner par l'ensemble des types S . L'interprétation des variables sera donc des applications $\iota : V \rightarrow A$ respectant le typage (i.e. $\forall s \in S, \iota(V_s) \subseteq A_s$). Comme pour la logique des prédicats, toute interprétation $\iota : V \rightarrow A$ s'étend de façon canonique aux termes avec variables en une application $\iota^\sharp : T_\Sigma(V) \rightarrow A$.

Définition 136 (Satisfaction).

Une Σ -algèbre $\mathcal{A}$ **satisfait** pour une Σ -interprétation ι une formule φ , notée $\mathcal{A} \models_{\iota} \varphi$ si, et seulement si :

- si φ est de la forme $t = t'$, alors $\mathcal{A} \models_{\iota} t = t'$ si et seulement si $\iota^{\#}(t) = \iota^{\#}(t')$.
- Les connecteurs propositionnels et les quantificateurs sont traités comme pour la logique des prédicats.

Une Σ -algèbre $\mathcal{A}$ **valide** une Σ -formule φ , notée $\mathcal{A} \models \varphi$, si et seulement si pour toute Σ -interprétation ι , $\mathcal{A} \models_{\iota} \varphi$.

Ainsi, une Σ -algèbre validera une spécification si elle valide l'ensemble des formules qui la compose. On parle alors de *SP*-algèbre. Formellement, ceci est défini de la façon suivante :

Définition 137 (*SP*-Algèbre).

Soit $SP = (\Sigma, Ax)$ une spécification. Une *SP*-**algèbre** $\mathcal{A}$ est une Σ -algèbre qui valide toutes les formules de Ax (i.e. $\forall \varphi \in Ax, \mathcal{A} \models \varphi$).

Définition 138 (Conséquence sémantique).

Soit SP une spécification. SP valide une Σ -formule φ , noté $SP \models \varphi$, si et seulement si pour toute *SP*-algèbre $\mathcal{A}$, on a : $\mathcal{A} \models \varphi$. φ est alors appelée **conséquence sémantique** de SP . On note $SP^{\bullet}$ l'ensemble de toutes les conséquences sémantiques de SP .

Observons que $SP^{\bullet}$ contient nécessairement les axiomes Ax de SP (i.e. $Ax \subseteq SP^{\bullet}$).

Exemple 177.

Soit la spécification SP suivante :

Type : nat

Fonctions : $\begin{cases} +, \times : nat \times nat \rightarrow nat \\ 2_ : nat \rightarrow nat \\ _{}^2 : nat \rightarrow nat \end{cases}$ multiplication par 2
élévation au carré

Variables : $x, y, z : nat$

Axiomes :

$x + y = y + x$	commutativité de $+$
$x \times y = y \times x$	commutativité de $\times$
$x + (y + z) = (x + y) + z$	associativité de $+$
$x \times (y \times z) = (x \times y) \times z$	associativité de $\times$
$x \times (y + z) = (x \times y) + (x \times z)$	distributivité
$2x = x + x$	multiplication par 2
$x^2 = x \times x$	élévation au carré

Montrons que $SP \models (x + y)^2 = x^2 + 2(x \times y) + y^2$. Soit $\mathcal{A}$ une SP -algèbre quelconque. Soit ι une interprétation des variables qui à x associe n et à y associe m , n et m étant donnés deux entiers quelconques. Montrons alors que $\mathcal{A} \models_{\iota} (x + y)^2 = x^2 + 2(x \times y) + y^2$. Le choix de $\mathcal{A}$ et ι étant quelconque, ceci reviendra à démontrer que $SP \models (x + y)^2 = x^2 + 2(x \times y) + y^2$. Par hypothèse, les égalités suivantes sont vérifiées :

$$\begin{aligned}
 (n +^{\mathcal{A}} m)^2 &= (n +^{\mathcal{A}} m) \times^{\mathcal{A}} (n +^{\mathcal{A}} m) \text{ (élévation au carré)} \\
 (n +^{\mathcal{A}} m) \times^{\mathcal{A}} (n +^{\mathcal{A}} m) &= ((n +^{\mathcal{A}} m) \times^{\mathcal{A}} n) +^{\mathcal{A}} ((n +^{\mathcal{A}} m) \times^{\mathcal{A}} m) \text{ (distributivité)} \\
 ((n +^{\mathcal{A}} m) \times^{\mathcal{A}} n) +^{\mathcal{A}} ((n +^{\mathcal{A}} m) \times^{\mathcal{A}} m) &= \\
 (n \times^{\mathcal{A}} (n +^{\mathcal{A}} m)) +^{\mathcal{A}} (m \times^{\mathcal{A}} (n +^{\mathcal{A}} m)) &= \text{ (commutativité de } \times \text{)} \\
 (n \times^{\mathcal{A}} (n +^{\mathcal{A}} m)) +^{\mathcal{A}} (m \times^{\mathcal{A}} (n +^{\mathcal{A}} m)) &= \\
 (n \times^{\mathcal{A}} n +^{\mathcal{A}} n \times^{\mathcal{A}} m) +^{\mathcal{A}} (m \times^{\mathcal{A}} n +^{\mathcal{A}} m \times^{\mathcal{A}} m) &= \text{ (distributivité)} \\
 (n \times^{\mathcal{A}} n +^{\mathcal{A}} n \times^{\mathcal{A}} m) +^{\mathcal{A}} (m \times^{\mathcal{A}} n +^{\mathcal{A}} m \times^{\mathcal{A}} m) &= \\
 n \times^{\mathcal{A}} n +^{\mathcal{A}} (n \times^{\mathcal{A}} m +^{\mathcal{A}} m \times^{\mathcal{A}} n) +^{\mathcal{A}} m \times^{\mathcal{A}} m &= \text{ (associativité de } + \text{)} \\
 n \times^{\mathcal{A}} n +^{\mathcal{A}} (n \times^{\mathcal{A}} m +^{\mathcal{A}} m \times^{\mathcal{A}} n) +^{\mathcal{A}} m \times^{\mathcal{A}} m &= \\
 n^2 +^{\mathcal{A}} (n \times^{\mathcal{A}} m +^{\mathcal{A}} m \times^{\mathcal{A}} n) +^{\mathcal{A}} m^2 &= \text{ (élévation au carré)} \\
 n^2 +^{\mathcal{A}} (n \times^{\mathcal{A}} m +^{\mathcal{A}} m \times^{\mathcal{A}} n) +^{\mathcal{A}} m^2 &= n^2 +^{\mathcal{A}} (n \times^{\mathcal{A}} m +^{\mathcal{A}} n \times^{\mathcal{A}} m) +^{\mathcal{A}} m^2 \\
 &= \text{ (commutativité de } \times \text{)} \\
 n^2 +^{\mathcal{A}} (n \times^{\mathcal{A}} m +^{\mathcal{A}} n \times^{\mathcal{A}} m) +^{\mathcal{A}} m^2 &= n^2 +^{\mathcal{A}} 2(n \times^{\mathcal{A}} m) +^{\mathcal{A}} m^2 \text{ (multiplication par 2)}
 \end{aligned}$$

Par transitivité de l'égalité ensembliste, nous avons alors : $(n +^{\mathcal{A}} m)^2 = n^2 +^{\mathcal{A}} 2(n \times^{\mathcal{A}} m) +^{\mathcal{A}} m^2$. Ceci clos la preuve.

Le calcul équationnel

Bien que nous avons présenté la logique équationnelle dans son ensemble dans les sections précédentes, ici nous allons nous restreindre au raisonnement purement équationnel, c'est-à-dire que nous allons considérer que nos formules sont de simples équations. L'intérêt est double. Tout d'abord, le calcul n'en sera que plus simple dans sa présentation et l'étendre aux autres connecteurs et quantificateurs est possible par exemple en ajoutant les règles du calcul des séquents moyennant quelques modifications élémentaires. Le second intérêt d'une telle restriction est qu'elle permet dans certains cas de définir une procédure de transformation des équations en des règles dites de réécriture et ainsi définir un algorithme permettant de résoudre le problème du mot (défini ici par la validité d'une équation) posé par un ensemble d'équations (voir la section sur la réécriture ci-dessous).

Définitions

Définition 139 (Substitution).

Soit $\Sigma = (S, F, V)$ une signature. Une **substitution** σ est une famille indexée par S d'applications $\sigma_s : V_s \rightarrow T_\Sigma(V)_s$.

Ainsi, une substitution est une interprétation des variables dans les termes. C'est l'équivalent syntaxique des interprétations de variables dans les algèbres.

Toute substitution $\sigma : V \rightarrow T_\Sigma(V)$ s'étend naturellement aux termes avec variables en une application $\sigma^\# : T_\Sigma(V) \rightarrow T_\Sigma(V)$ défini de la façon suivante :

$$\begin{cases} x \mapsto \sigma(x) \\ f(t_1, \dots, t_n) \mapsto f(\sigma^\#(t_1), \dots, \sigma^\#(t_n)) \end{cases}$$

Cette extension consiste donc à remplacer les variables aux feuilles par leur substitution et conserver tel quel tout le reste.

On peut maintenant donner l'ensemble de nos règles d'inférence. Le calcul présenté ci-dessous est aussi souvent appelé calcul de Birkhoff du nom de son créateur.

Définition 140 (Calcul).

Soit $\Sigma = (S, F, V)$ une signature. Soit t, t', t'' des termes avec variables de $T_\Sigma(V)$.

$$\begin{array}{lll} \text{Sym.} & \frac{t = t'}{t' = t} & \text{Refl.} \quad \frac{}{t = t} \quad \text{Trans.} \quad \frac{t = t' \quad t' = t''}{t = t''} \\ \text{Rempl.} & \frac{t_1 = t'_1 \dots t_n = t'_n}{f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)} & \text{Subst.} \quad \frac{t = t'}{\sigma^\#(t) = \sigma^\#(t')} \end{array}$$

Quelques commentaires sur ce calcul :

- Les trois premières règles du calcul caractérisant respectivement la *symétrie*, le *réflexivité* et la *transitivité*, traduisent le fait que le prédicat égalité est une relation d'équivalence.

- La dernière règle du raisonnement équationnel, appelée communément *remplacement*, exprime que les noms de fonctions agissent comme des applications, c'est-à-dire des relations n -aires qui à tout tuple $(a_1, \dots, a_{n-1})$ associent un unique a_n . Ainsi, appliquer le même nom de fonction $f : s_1 \times \dots \times s_n \rightarrow s$ à deux tuples de termes $(t_1, \dots, t_n)$ et $(t'_1, \dots, t'_n)$ pour lesquels on a démontré pour chaque $i = 1, \dots, n$, que t_i et t'_i dénotaient la même valeur (c-à-d., on a $t_i = t'_i$), doit dénoter la même valeur (c-à-d., on a $f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)$).
- Les variables jouant le rôle de valeur générique, une équation est valide si elle est valide pour toutes les valeurs possibles que peuvent prendre les variables. Dans un cadre syntaxique, les valeurs possibles sont dénotées par des termes avec variables. Ainsi, si nous avons réussi à montrer la validité d'une équation $t = t'$, cette dernière est encore préservée après avoir *substitué* n'importe quelle variable de t et t' par un terme avec variables.

Comme pour la logique propositionnelle et la logique des prédicats, ces règles d'inférence permettent de définir la relation $\vdash$ entre les spécifications équationnelles et les équations, c'est-à-dire étant donnée une spécification $SP = (\Sigma, Ax)$ où Ax est un ensemble de Σ -équations, et une équation $t = t'$, on dira que SP infère $t = t'$ que l'on notera $SP \vdash t = t'$, s'il existe un arbre de preuve construit à partir des équations dans Ax et des règles ci-dessus.

Exemple 178.

Soit la spécification SP suivante :

Type : rel

Fonctions : $\begin{cases} 0 \rightarrow rel \\ +, -, \times : rel \times rel \rightarrow rel \\ -^2 : nat \rightarrow nat & \text{élévation au carré} \\ oppose : rel \rightarrow rel & \text{élément symétrique} \end{cases}$

Variables : $x, y, z : nat$

Axiomes :

$x + y = y + x$	commutativité de +
$x \times y = y \times x$	commutativité de ×
$x + (y + z) = (x + y) + z$	associativité de +
$x \times (y \times z) = (x \times y) \times z$	associativité de ×
$x \times (y + z) = (x \times y) + (x \times z)$	distributivité
$x^2 = x \times x$	élévation au carré
$x + 0 = x$	neutralité à droite
$x + oppose(x) = x$	élément symétrique
$x - y = x + oppose(y)$	

Montrons alors l'assertion suivante à partir du calcul présenté en définition 140 :

$$SP \vdash (a + b) \times (a - b) = a^2 + b^2$$

$$\frac{\frac{\mathcal{A}_1}{(a+b) \times (a-b) = (a^2 - a \times b) + (b \times a - b^2)} \text{Trans.} \quad \frac{\mathcal{A}_2}{(a^2 - a \times b) + (b \times a - b^2) = a^2 + b^2} \text{Trans.}}{(a+b) \times (a-b) = a^2 + b^2}$$

$\mathcal{A}_2$ étant le sous-arbre de preuve à compléter dans l'exercice ?? :

$$\frac{\dots}{(a \times (a - b)) + (b \times (a - b)) = (a^2 - a \times b) = (b \times a - b^2)}$$

et $\mathcal{A}_1$ étant le sous-arbre de preuve (complet et à suivre en exemple) ci-dessous :

$$\frac{\frac{\frac{x \times y = y \times x}{(a+b) \times c = c \times (a+b)} \text{Ax.} \quad \text{Subst.}[x/(a+b), y/c] \quad \frac{\mathcal{A}_3 \quad \mathcal{A}_4}{c \times (a+b) = a \times c + b \times c} \text{Trans.}}{(a+b) \times c = a \times c + b \times c} \text{Subst.}[c/(a-b)]$$

où $\mathcal{A}_3$ est le sous-arbre :

$$\frac{\frac{x \times (y+z) = x \times y + x \times z}{c \times (a+b) = c \times a + c \times b} \text{Ax.} \quad \text{Subst.}[x/c, y/a, z/b]$$

et $\mathcal{A}_4$ est le sous-arbre :

$$\frac{\frac{\frac{x \times y = y \times x}{c \times a = a \times c} \text{Ax.} \quad \text{Subst.}[x/c, y/a] \quad \frac{\frac{x \times y = y \times x}{c \times b = b \times c} \text{Ax.} \quad \text{Subst.}[x/c, y/b]}{c \times a + c \times b = a \times c + b \times c} \text{Remp.+}$$

Correction et complétude

Comme nous en avons l'habitude maintenant, nous devons nous assurer que les transformations du calcul, *a priori* sans aucun sens, reflètent fidèlement le monde réel représenté ici par la sémantique. Il nous faut alors vérifier que tout ce que le calcul permet de prouver doit être vrai "dans le monde réel", c'est-à-dire que toutes les équations que les règles d'inférence permettent d'inférer doivent être satisfaites par les modèles sémantiques (i.e. les algèbres). Cette propriété est la *correction* du calcul. Elle s'exprime formellement par :

Théorème 179 (Correction).

Pour toute spécification $SP = (\Sigma, Ax)$, l'inclusion suivante est vérifiée :

$$SP \vdash t = t' \subseteq SP \models t = t'$$

Démonstration : Ceci se démontre par induction mathématique sur la structure des arbres de preuves. La preuve de ce théorème est plutôt longue, aussi nous allons simplement la montrer pour la règle d'inférence de substitution. Les autres règles d'inférence sont laissées au lecteur à titre d'exercice.

Supposons alors un arbre de preuve de la forme $\frac{\vdots}{\sigma^\#(t) = \sigma^\#(t')}$. Par hypothèse d'induction, nous avons $t = t' \in SP^\bullet$, c'est-à-dire, que pour toute SP -algèbre $\mathcal{A}$ nous avons $\mathcal{A} \models t = t'$. Soit $\mathcal{A}$ une SP -algèbre. Par hypothèse d'induction, $\mathcal{A} \models t = t'$, c'est-à-dire par définition, pour toute interprétation $\iota : V \rightarrow A$, $\iota^\#(t) = \iota^\#(t')$. Soit $\sigma : V \rightarrow T_\Sigma(V)$ une substitution. Par définition, pour tout $\iota : V \rightarrow A$, $\iota^\# \circ \sigma$ est encore une interprétation de $V \rightarrow A$. Donc par hypothèse, $\mathcal{A} \models_{\iota^\# \circ \sigma} t = t'$. Pour la suite de la preuve, nous avons besoin du résultat intermédiaire suivant :

Lemme 180.

Pour toute interprétation $\iota : V \rightarrow A$ et toute substitution $\sigma : V \rightarrow T_\Sigma(V)$, nous avons l'égalité suivante : $(\iota^\# \circ \sigma)^\# = \iota^\# \circ \sigma^\#$.

(Nous laissons la preuve de ce lemme en exercice)

Reprise de la preuve du théorème 179 : À partir du lemme ci-dessus, on a :

$$\begin{aligned} \mathcal{A} \models_{\iota^\# \circ \sigma} t &= t' \\ \Updownarrow \\ \iota^\# \circ \sigma^\#(t) &= \iota^\# \circ \sigma^\#(t') \\ \Updownarrow \\ \mathcal{A} \models_{\iota} \sigma^\#(t) &= \sigma^\#(t') \end{aligned}$$

Ceci clos la preuve pour le cas de la règle d'inférence de substitution. ■

Pour montrer la complétude, définissons la relation binaire $\equiv_{SP}$ sur les termes suivante :

$$t \equiv_{SP} t' \iff SP \vdash t = t'$$

Proposition 181.

$\equiv_S P$ est une relation d'équivalence compatible avec les sortes et les fonctions, c'est-à-dire si $t_1 \equiv_{SP} t'_1, \dots, t_n \equiv_{SP} t'_n$, alors $f(t_1, \dots, t_n) \equiv_{SP} f(t'_1, \dots, t'_n)$.

Démonstration : Simple application des règles du calcul équationnel. ■

Définissons l'algèbre $\mathcal{T}$ suivante :

- $\forall s \in S, T_s = T_\Sigma(V)_{/\equiv_{SP}}$
- $\forall f : s_1 \times \dots \times s_n \rightarrow s \in F, \forall (t_1, \dots, t_n) \in T_\Sigma(V)_{s_1} \times \dots \times T_\Sigma(V)_{s_n}, f^{\mathcal{T}}([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)]$

$\mathcal{T}$ est bien une Σ -algèbre car $\equiv_{SP}$ est compatible avec les fonctions.

Théorème 182 (Complétude).

Pour toute spécification $SP = (\Sigma, Ax)$, l'inclusion suivante est vérifiée :

$$SP \models t = t' \subseteq SP \vdash t = t'$$

Démonstration : Soit $t = t' \in SP^\bullet$. Par définition, pour toute équation eq de SP , on a $SP \vdash eq$, et donc $\mathcal{T} \models eq$. Ainsi, $\mathcal{T} \models t = t'$. Soit $\sigma : x \mapsto x$ la substitution définie comme l'identité sur les variables. L'application $q \circ \sigma$ où q est l'application quotient de $T_\Sigma(V)$ dans $T_\Sigma(V)_{/\equiv_{SP}}$, est une interprétation de V dans $\mathcal{T}$. On a donc $\mathcal{T} \models_{q \circ \sigma} t = t'$ qui est équivalent à écrire $[t] = [t']$, d'où nous déduisons $SP \vdash t = t'$. ■

Une approche de preuve syntaxique plus efficace

Nous pouvons observer que le type de preuve de théorème que l'on peut mener avec le calcul précédemment défini, ne caractérise pas le raisonnement algébrique usuel. En effet, pour établir une équation, nous utilisons plutôt comme type de raisonnement, la stratégie qui consiste à "réduire" les deux expressions de l'égalité à une même troisième. Par exemple, à partir de la spécification SP donnée dans l'exemple 178, pour montrer que l'identité remarquable $(x + y)^2 = x^2 + 2(x \times y) + y^2$ est un théorème de SP , au lieu d'associer un arbre de preuve en suivant les règles du calcul de la définition 140, on préfère plutôt adopter la stratégie suivante :

$$(x + y)^2$$

$$(x + y) \times (x + y)$$

$$((x + y) \times x) + ((x + y) \times y) \qquad x^2 + 2xy + y^2$$

$$(x \times (x + y)) + ((x + y) \times y) \qquad xx + 2xy + y^2$$

$$(x \times (x + y)) + (y \times (x + y)) \qquad xx + 2xy + yy$$

$$xx + xy + (y \times (x + y)) \qquad xx + xy + xy + yy$$

$$xx + xy + yx + yy$$

Ce type de raisonnement est rendu possible par une propriété fondamentale associée au prédicat d'égalité connu sous le nom de *loi de Leibniz*. Usuellement, cette loi s'exprime sous la forme suivante :

$$\textbf{Loi de Leibniz} \frac{x = y \quad P(x)}{P(y)}$$

Si l'on restreint cette loi à la règle de Remplacement et à la règle algébrique de Substitution on la nomme alors usuellement sous le nom de *remplacement d'égale par égale*. Cette adaptation de la loi de Leibniz s'exprime alors de la façon suivante :

$$\frac{C[u] = t \quad u = v}{C[v] = t}$$

où C , appelé *contexte*, est un terme de $T_\Sigma(V \cup \{\square - s\}_{s \in S})$ possédant une unique occurrence d'un $\square_s$, et $C[u]$ est le terme de $T_\Sigma(V)_s$ dans le quel on a substitué l'unique occurrence de $\square_s$ par u .

La règle de remplacement d'égale par égale signifie donc que pour un terme donné, le remplacement de n'importe lequel de ses sous-termes par un terme équivalent, conserve l'égalité. L'idée est alors d'utiliser ce principe pour essayer de réduire un terme par remplacements successifs de ses sous-termes par des termes équivalentes. Cette méthode de preuve équationnelle dans le monde des spécifications algébriques est à la base d'une méthode effective de preuve de théorème que l'on appelle la *réécriture* et que l'on abordera dans le chapitre suivant.

Une notion importante pour ce type de preuve, est celle de *contexte pour une signature* Σ .

Définition 141 (Contexte).

Soit $\Sigma = (S, F, V)$ une signature. Notons $\Sigma_{\square}$ la signature obtenue à partir de Σ en ajoutant à F l'ensemble de constantes $F' = \{\square_s : \rightarrow s \mid s \in S\}$ (c-à-d., $\Sigma_{\square} = (S, F \cup F', V)$). Un Σ -**contexte** C est un terme de $T_{\Sigma_{\square}}(V)$ où apparaît une unique occurrence d'une unique constante $\square_s$. Dans ce cas-là, On dit que C est de **type** s et on le note $C : s$.

Soit $C : s$ un contexte et $t \in T_{\Sigma}(V)_s$. On note $C[t]$ le terme de $T_{\Sigma}(V)$, le terme obtenu à partir de C en substituant l'unique occurrence de la constante $\square_s$ par le terme t .

À partir de là, on peut définir notre nouvelle méthode de preuve.

Définition 142 (Étape de réduction).

Soit $SP = (\Sigma, Ax)$ une spécification. Définissons alors la relation binaire $\rightarrow_{SP}$ sur $T_{\Sigma}(V)$ de la façon suivante :

$$u \rightarrow_{SP} v \iff \begin{cases} \exists s \in S, u, v \in T_{\Sigma}(V)_s, \\ \exists u_1 = v_1 \in Ax, \exists C : s \in T_{\Sigma_{\square}}(V), \exists \sigma : V \rightarrow T_{\Sigma}(V), \\ u = C[\sigma^{\#}(u_1)] \wedge v = C[\sigma^{\#}(v_1)] \end{cases}$$

Notons $\leftrightarrow_{SP}$ la fermeture symétrique de $\rightarrow_{SP}$. Tout élément de $\leftrightarrow_{SP}$ s'appelle une **étape de réduction**.

Une étape de réduction peut donc se schématiser de la façon suivante :

$$\begin{array}{ccc} u & & v \\ & C & \\ & & \\ & u_1 & v_1 \\ & \sigma & \sigma \end{array}$$

Exemple 183.

Dans l'exemple de la réduction de l'identité remarquable ci-dessus, nous avons par exemple l'étape de réduction suivante :

$$u = x \times (x + y) + (y \times (x + y)) \longleftrightarrow_{\emptyset, SP} v = xx + xy + (y \times (x + y))$$

où $u_1 = x \times (y + z)$, $v_1 = xy + xz$, $C = \square_{rel} + (y \times (x + y))$ et $\sigma : \begin{array}{l} x \mapsto x \\ y \mapsto x \\ z \mapsto y \end{array}$

On peut montrer simplement les deux implication suivantes : Pour tous $t, t' \in T_{\Sigma}(V)_s$

1. $t \leftrightarrow_{SP} t' \implies \forall \sigma : V \rightarrow T_\Sigma(V), \sigma^\#(t) \leftrightarrow_{\sigma^\#(\Gamma), SP} \sigma^\#(t')$.
2. $t \leftrightarrow_{SP} t' \implies \forall C : s, C[t] \leftrightarrow_{\Gamma, SP} C[t']$.

Une réduction sera donc une "composition" de réduction. Il nous reste alors à définir ce que l'on entend par "composition".

Définition 143 (Réduction).

Avec les notations et hypothèses de la définition précédente, on note $\overset{+}{\leftrightarrow}_{SP}$ la fermeture transitive de $\leftrightarrow_{SP}$, et $\overset{*}{\leftrightarrow}_{\Gamma, SP}$ la fermeture réflexive de $\overset{+}{\leftrightarrow}_{\Gamma, SP}$.

L'inférence de formules se définit maintenant de la façon suivante :

Définition 144 (Inférence).

Une spécification $SP = (\Sigma, Ax)$ **infère** une Σ -équation $t = t'$ si, et seulement si $t \overset{*}{\leftrightarrow}_{SP} t'$.

Cette nouvelle forme d'inférence a la même puissance de preuve que celle définie dans la définition 140 comme l'atteste le résultat suivant :

Théorème 184.

$$t \overset{*}{\leftrightarrow}_{SP} t' \iff SP \vdash t = t'$$

Démonstration : Le sens $\implies$ de la preuve s'obtient en montrant qu'à partir de tout énoncé $t \overset{*}{\leftrightarrow}_{SP} t'$ on peut construire un arbre de preuve π de conclusion $t = t'$. Ceci s'obtient par récurrence sur la longueur des réductions.

Le cas de base : La réduction est réduite à $t \overset{*}{\leftrightarrow}_{SP} t$. On pose alors $\pi = \frac{}{t=t} \text{Refl.}$

L'étape de récurrence : la réduction $t \overset{*}{\leftrightarrow}_{SP} t'$ a été obtenue à partir des deux réductions $t \overset{*}{\leftrightarrow}_{SP} t''$ et $t'' \overset{*}{\leftrightarrow}_{SP} t'$. Par hypothèse d'induction, il existe deux arbres de preuve π_1 et π_2 de conclusion $t = t''$ et $t'' = t'$, respectivement. Posons alors l'arbre $\pi = \frac{\frac{\pi_1}{t=t''} \quad \frac{\pi_2}{t''=t'}}{t=t'} \text{Trans.}$

Le sens $\impliedby$ se fait par récurrence sur la structure des arbres de preuve.

Le cas de base : L'arbre de preuve est réduit à une racine. Il a donc pour conclusion une formule de la forme $t = t$ ou bien un axiome de la spécification $\Gamma \Rightarrow t = t'$. Dans le premier cas, nous avons alors $t \overset{*}{\leftrightarrow}_{SP} t$. Dans le second cas, nous avons $t \leftrightarrow_{SP} t'$.

Le pas de récurrence : Il faut faire un raisonnement par cas sur la dernière règle d'inférence appliquée dans l'arbre de preuve. Le cas de la règle de transitivité est simple et laissée en exercice. Ici, nous allons simplement le montrer pour la substitution. Supposons donc l'arbre de preuve de la

forme $\pi = \frac{\vdots}{\sigma^\#(\frac{t=t'}{t=t'})}$. Par hypothèse d'induction, nous avons $t \overset{*}{\leftrightarrow}_{SP} t'$, c'est-à-dire il existe $t_1, \dots, t_n \in T_\Sigma(V)$ tel que $t \leftrightarrow_{SP} t_1 \leftrightarrow_{SP} t_2 \leftrightarrow_{SP} \dots \leftrightarrow_{SP} t_n \leftrightarrow_{SP} t'$ et par n étapes de réduction aboutit à $t \overset{*}{\leftrightarrow}_{SP} t'$. Par le résultat établi dans l'exercice ??, on peut écrire $\sigma^\#(t) \leftrightarrow_{SP} \sigma^\#(t_1) \leftrightarrow_{SP} \sigma^\#(t_2) \leftrightarrow_{SP} \dots \leftrightarrow_{SP} \sigma^\#(t_n) \leftrightarrow_{SP} \sigma^\#(t')$. Le cas de la règle de remplacement se traite de la même façon à partir de la second implication ci-dessus. ■

23.3. Réécriture

On a vu dans la section précédente une approche syntaxique plus efficace que le raisonnement équationnel pour montrer qu'une équation est déduite d'un ensemble d'autres équations. Ce procédé consistait à définir la relation d'équivalence $\stackrel{*}{\leftrightarrow}_{SP}$, appelée aussi **relation de convertibilité**. Le problème est que cette relation d'équivalence, bien que plus simple dans son utilisation pour prouver des équations à partir d'autres équations, n'est pas effective, c'est-à-dire qu'elle ne permet pas d'effectuer ces preuves équationnelles de façon automatique. La raison est la fermeture symétrique de la relation qui ne nous donne pas le moyen de décider dans une classe d'équivalence pour la relation de convertibilité, si un terme est plus "canonique" qu'un autre, ce qui nous permettrait d'orienter notre choix. La question que l'on peut se poser alors est de savoir si nous pouvons nous passer de la règle de symétrie? Ceci consisterait alors à orienter les équations. Cependant, cette orientation n'est pas aussi simple à définir. Tout d'abord parce que parfois, il n'existe aucun moyen d'orienter les équations (la logique équationnelle est indécidable), et quand il est possible de le faire, ceci ne se fait pas par une simple orientation mais demande tout un processus de complétion qui peut engendrer un ensemble beaucoup plus grand (voir infini) de règles que d'équations de départ. Ce processus de complétion est à la base de nombreuses méthodes de prototypage rapide de spécifications équationnelles.

Systèmes de réécriture et leurs propriétés

Ici, nous allons introduire les concepts qui nous permettront de formuler les conditions sous lesquelles un ensemble d'équations peut être interprétées comme un programme.

Définition

Définition 145 (Système de réécriture).

Soit Σ une signature. Un **système de réécriture** $\mathcal{R}$ sur Σ est une relation binaire $\rightarrow$ sur $T_{\Sigma}(V)$. Chaque couple (t, t') de $\rightarrow$ est appelée une **règle de réécriture**. On la note $t \rightarrow t'$.

Exemple 185.

La fonction d'Ackermann que nous avons présentée dans la partie 1 de ce cours peut être représentée par un système de réécriture.

$$\begin{aligned} \text{Ack}(x+1, 0) &\rightarrow \text{Ack}(x, 1) \\ \text{Ack}(0, x) &\rightarrow x+1 \\ \text{Ack}(x+1, y+1) &\rightarrow \text{Ack}(x, \text{Ack}(x+1, y)) \end{aligned}$$

Exemple 186.

Le calcul formel travaille aussi avec des règles. Voici, le système de réécriture de dérivation formelle par rapport à ξ :

$$\begin{aligned} \delta\xi &\rightarrow 1 \\ \delta a &\rightarrow 0 \\ \delta(u+v) &\rightarrow \delta u + \delta v \\ \delta(uv) &\rightarrow (\delta u)v + u(\delta v) \\ \delta(-u) &\rightarrow \delta - (\delta u) \\ \delta(u-v) &\rightarrow (\delta u) - (\delta v) \\ \delta(u/v) &\rightarrow ((\delta u)v - u(\delta v))/v^2 \quad \delta(\ln u) \rightarrow (\delta u)/u \\ \delta(u^v) &\rightarrow vu^{v-1}\delta u + u^v(\ln u)\delta v \end{aligned}$$

De la même manière que pour la relation de convertibilité, on peut définir une étape de réécriture.

Définition 146 (Étape de réécriture).

Soit $\mathcal{R}$ un système de réécriture. Définissons alors la relation binaire $\rightarrow_{\mathcal{R}}$ sur $T_{\Sigma}(V)$ de la façon suivante :

$$u \rightarrow_{\mathcal{R}} v \iff \begin{cases} \exists s \in S, u, v \in T_{\Sigma}(V)_s, \\ \exists u_1 \rightarrow v_1 \in \mathcal{R}, \exists C : s \in T_{\Sigma_{\square}}(V), \exists \sigma : V \rightarrow T_{\Sigma}(V), \\ u = C[\sigma^{\#}(u_1)] \wedge v = C[\sigma^{\#}(v_1)] \end{cases}$$

Tout élément de $\rightarrow_{\mathcal{R}}$ s'appelle une **étape de réécriture**.

On notera alors $\xrightarrow{+}_{\mathcal{R}}$ (resp. $\xrightarrow{*}_{\mathcal{R}}$, resp. $\xleftrightarrow{*}_{\mathcal{R}}$) la fermeture transitive (resp. réflexive et transitive, resp. réflexive, symétrique et transitive) de $\rightarrow_{\mathcal{R}}$.

Propriétés

Pour pouvoir être utilisé comme un programme ou un système de preuve décidable, les systèmes de réécriture doivent posséder un certain nombre de propriétés.

Définition 147 (Confluence).

Un système de réécriture $\mathcal{R}$ est dit **confluent** si, et seulement si $\mathcal{R} \xleftarrow{*} \circ \xrightarrow{*}_{\mathcal{R}} \subseteq \xrightarrow{*}_{\mathcal{R}}$ où $\circ$ dénote la composition de relation binaire.

Si un système de réécriture $\mathcal{R}$ possède la propriété d'être confluent alors on peut extraire de sa fermeture réflexive et transitive une fonction partielle. Les systèmes de réécriture confluent définissent ainsi un algorithme. En effet, on a le résultat suivant :

Définition 148 (Forme normale).

Soit $\mathcal{R}$ un système de réécriture. Un terme $t \in T_\Sigma(V)$ est une **forme normale** pour $\mathcal{R}$ si, et seulement si, il n'existe aucun autre terme t' de $T_\Sigma(V)$ tel que $t \xrightarrow{*}_{\mathcal{R}} t'$. De plus, t est appelé **forme normale** de t' si, et seulement si $t' \xrightarrow{*}_{\mathcal{R}} t$ et t est une forme normale pour $\mathcal{R}$.

Proposition 187.

Si $\mathcal{R}$ est confluente alors la forme normale de tout terme est unique si elle existe.

Démonstration : Supposons qu'un terme t possède deux formes normales t' et t'' . Par l'hypothèse que $\mathcal{R}$ soit confluente, il existe un terme t_3 tel que $t' \xrightarrow{*}_{\mathcal{R}} t_3$ et $t'' \xrightarrow{*}_{\mathcal{R}} t_3$. Donc, soit nous contredisons l'hypothèse que t' et t'' sont des formes normales, soit $t' = t''$. ■

L'intérêt d'un tel résultat est qu'il définit un moyen de calcul pour générer les différentes valeurs d'une fonction définie par la fermeture réflexive et transitive de la relation de réécriture $\rightarrow_{\mathcal{R}}$ quand cette dernière est confluente. Un autre intérêt des relations de réécriture confluentes est que l'on peut les utiliser pour prouver des équations en raisonnant simplement sur les expressions qui les composent. Dans ce cas-là, on dit que la relation binaire possède la propriété de Church-Rosser.

Définition 149 (Church-Rosser).

Un système de réécriture $\mathcal{R}$ est dit de **Church-Rosser** si, et seulement si $\xrightarrow{*}_{\mathcal{R}} \subseteq \xrightarrow{*}_{\mathcal{R}} \circ \mathcal{R} \xrightarrow{*}$.

Théorème 188.

$\mathcal{R}$ est de Church-Rosser si, et seulement si, il est confluente.

Démonstration : L'implication $\Rightarrow$ est trivialement satisfaite. En effet, toute relation de Church-Rosser entraîne *a fortiori* la confluence.

L'autre sens de l'implication se montre par récurrence sur le nombre de fois qu'apparaît $\leftrightarrow_{\mathcal{R}}$ dans $\xrightarrow{*}_{\mathcal{R}}$. Supposons alors que ce nombre soit n et notons le $t \xrightarrow{n}_{\mathcal{R}} t'$.

si $n = 0$ alors t et t' sont égaux. Dans ce cas-là, comme $\xrightarrow{*}_{\mathcal{R}}$ contient l'égalité, il suffit de poser $t'' = t = t'$.

Sinon, soit $t \leftrightarrow_{\mathcal{R}} t' \xrightarrow{n}_{\mathcal{R}} t''$. Par hypothèse d'induction, il existe t_3 tel que $t' \xrightarrow{*}_{\mathcal{R}} t_3$ et $t'' \xrightarrow{*}_{\mathcal{R}} t_3$. Maintenant, soit $t \rightarrow_{\mathcal{R}} t'$ et dans ce cas-là on a $t \xrightarrow{+}_{\mathcal{R}} t_3$, soit $t \leftarrow_{\mathcal{R}} t'$ et par confluence, il existe t_4 tel que $t \xrightarrow{*}_{\mathcal{R}} t_4$ et $t_3 \xrightarrow{*}_{\mathcal{R}} t_4$. La propriété de Church-Rosser est encore satisfaite. ■

Maintenant, pour que ce procédé soit totalement effectif, il manque encore une propriété fondamentale qui est la terminaison de ce dernier.

Définition 150 (Terminaison).

Un système de réécriture $\mathcal{R}$ **termine** si, et seulement si toute séquence de réduction par la relation $\rightarrow_{\mathcal{R}}$ est finie.

La terminaison d'un système de réécriture est intimement liée à la notion d'ordre bien fondé comme l'établit le résultat suivant :

Proposition 189.

Un système de réécriture $\mathcal{R}$ termine si, et seulement s'il existe un ordre bien fondé $>$ sur $T_{\Sigma}(V)$ qui contient la relation de réécriture $\rightarrow$ de $\mathcal{R}$.

Démonstration : ($\Rightarrow$) Si $\mathcal{R}$ termine par définition il n'existe pas de séquences de réécriture infinie et donc de suites de termes décroissantes par la relation $\rightarrow_{\mathcal{R}}$ non stationnaires. Il suffit alors de poser $> = \rightarrow_{\mathcal{R}}^*$.

($\Leftarrow$) $>$ est une relation bien fondée. Ceci signifie qu'il n'existe aucune suite $(t_i)_{i \in \mathbb{N}}$ non stationnaire. Donc, si pour n'importe quel terme x de cette suite on extrait la sous-suite dont le premier terme est x et l'on extrait la séquence de termes de cette sous-suite, cette dernière est forcément finie. ■

La proposition 189 bien qu'élémentaire, est importante car elle énonce que pour montrer la terminaison d'un système de réécriture, il suffit de trouver un ordre bien fondé qui contiendrait la relation de réécriture sous-jacente. Comme il peut être très difficile de montrer la terminaison d'un système de réécriture (ce problème étant je le rappelle non-décidable et donc non trivial), un grand nombre de recherche ont été effectuées pour définir des ordres bien fondés particuliers. Dans ce cadre, on peut citer tous les ordres lexicographiques sur les chemins. Nous ne présenterons pas ces derniers dans ce cours car ils font appels à des notions compliquées qui sortent du cadre de ce cours.

La propriété de confluence d'une relation d'ordre n'est pas toujours aisée à obtenir. Il en existe une version plus locale qui considère seulement un divergence simple (i.e. une seule application de la relation sur chaque branche de la divergence) :

Définition 151 (Locale confluence).

Soit $\mathcal{R}$ un système de réécriture. $\mathcal{R}$ est dit **localement confluent** si, et seulement si $\mathcal{R} \leftarrow \circ \rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}}^* \circ \mathcal{R} \leftarrow^*$.

Bien entendu, la confluence locale est plus faible que la confluence, comme en atteste le contre-exemple suivant défini sur l'ensemble $E = \{a, b, c, d\}$.

$$a \longleftarrow b \begin{array}{c} \xrightarrow{\quad} \\ \xleftarrow{\quad} \end{array} c \longrightarrow d$$

Cette relation est localement confluyente car toute divergence locale à partir de a (resp. b) est joignable en c (resp. d). En revanche, elle n'est pas confluyente : par exemple on peut, à partir de b , atteindre a et d qui ne sont pas joignables.

De là, nous avons le résultat remarquable :

Théorème 190 (Lemme du diamant : Newman, 1942).

Soit $\mathcal{R}$ un système de réécriture qui termine. Alors, $\mathcal{R}$ est confluent si, et seulement s'il est localement confluent.

Démonstration : — $(\Rightarrow)$ Hypothèse : $\mathcal{R}$ est confluent.

Montrons qu'il est localement confluent.

Pour tous x, y et z tels que $(x \xrightarrow{*}_{\mathcal{R}} y)$ et $(x \xrightarrow{*}_{\mathcal{R}} z)$, il existe un terme x' tel que $(y \xrightarrow{*}_{\mathcal{R}} x')$ et $(z \xrightarrow{*}_{\mathcal{R}} x')$. A fortiori, pour tous x, y et z tels que $(x \rightarrow_{\mathcal{R}} y)$ et $(x \rightarrow_{\mathcal{R}} z)$, il existe un terme x' de E tel que $(y \xrightarrow{*}_{\mathcal{R}} x')$ et $(z \xrightarrow{*}_{\mathcal{R}} x')$.

— $(\Leftarrow)$ Hypothèse : $\mathcal{R}$ est localement confluent.

En utilisant le raisonnement par induction mathématique, il nous faut caractériser le prédicat P et la relation $\prec$. Puisque nous souhaitons montrer la confluence de $\mathcal{R}$, nous choisissons le prédicat P suivant :

$P(x)$ si, et seulement si pour tous x_1 et x_2 tels que $(x \xrightarrow{*}_{\mathcal{R}} x_1)$ et $(x \xrightarrow{*}_{\mathcal{R}} x_2)$, il existe x' tel que $(x_1 \xrightarrow{*}_{\mathcal{R}} x')$ et $(x_2 \xrightarrow{*}_{\mathcal{R}} x')$. Le système de réécriture $\mathcal{R}$ terminant, nous prenons donc $\prec$ égale à $\xrightarrow{*}_{\mathcal{R}}^{-1}$ (pour n'utiliser que $\prec$).

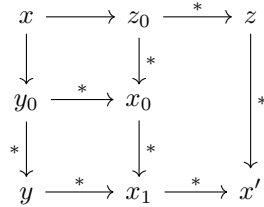
Supposons que, pour tout y tel que $(y \prec x)$, on a $P(y)$. A-t-on $P(x)$?

Cette question se traduit par : soient u et v tels que $(x \xrightarrow{n}_{\mathcal{R}} u)$ et $(x \xrightarrow{m}_{\mathcal{R}} v)$; existe-t-il x' tel que $(u \xrightarrow{*}_{\mathcal{R}} x')$ et $(v \xrightarrow{*}_{\mathcal{R}} x')$?

— Si $n = 0$, x' est v et si $m = 0$, x' est u .

— Sinon, il existe u_1 et v_1 tels que $(x \rightarrow_{\mathcal{R}} u_1)$, $(u_1 \xrightarrow{*}_{\mathcal{R}} u)$ et $(x \rightarrow_{\mathcal{R}} v_1)$, $(v_1 \xrightarrow{*}_{\mathcal{R}} v)$. Par hypothèse, $\mathcal{R}$ est localement confluent : il existe donc t tel que $(u_1 \xrightarrow{*}_{\mathcal{R}} t)$ et $(v_1 \xrightarrow{*}_{\mathcal{R}} t)$. Par hypothèse d'induction, puisque $(u_1 \prec x)$, on a $P(u_1)$; il existe donc t' tel que $(u \xrightarrow{*}_{\mathcal{R}} t')$ et $(t \xrightarrow{*}_{\mathcal{R}} t')$. De même, puisque $(v_1 \prec x)$, on a $P(v_1)$ et il existe x' tel que $(t' \xrightarrow{*}_{\mathcal{R}} x')$ et $(v \xrightarrow{*}_{\mathcal{R}} x')$.

La preuve du sens $\Leftarrow$ peut se résumer par le dessin suivant :



Un algorithme de décision

À partir des systèmes de réécriture confluents et qui terminent, on peut définir une procédure de décision permettant de montrer qu'une équation est ou non une conséquence de la théorie sous-jacente au système de réécriture. Cette procédure est la suivante :

Inputs Un système de réécriture $\mathcal{R}$ et une équation $u = v$ pour laquelle on veut montrer $u \xleftrightarrow{*}_{\mathcal{R}} v$

Initialization $S; Tmp = \{u\}$, $S', Tmp' = \{v\}$, et $answer = false$.

Loop while $\neg answer$ do :

- 1) choisir $u' \in Tmp$ et $Tmp = Tmp \setminus \{u'\}$;
- 2) $Tmp := Tmp \cup \{u'' \mid u' \rightarrow_{\mathcal{R}} u''\}$;

- 3) $S := S \cup Tmp$;
- 4) choisir $v' \in Tmp'$ et $Tmp' = Tmp' \setminus \{v'\}$;
- 5) $Tmp' := Tmp' \cup \{v'' \mid v' \rightarrow_{\mathcal{R}} v''\}$;
- 6) $S' := S' \cup Tmp'$;
- 7) $answer = (S \cap S' \neq \emptyset)$

end of loop

Output $return(answer)$

La procédure ci-dessus définit un parcours en largeur de la théorie sous-jacente au système de réécriture $\mathcal{R}$.

Les systèmes de réécriture confluents (et donc de Church-Rosser) et qui terminent peuvent être utilisés pour répondre au problème de la validité d'une équation.

Théorème 191.

Soit $\mathcal{R}$ un système de réécriture. Si $\mathcal{R}$ est confluent, alors $\Gamma \vdash u = v$ où $\Gamma = \{u = v \mid u \rightarrow v \in \mathcal{R} \vee v \rightarrow u \in \mathcal{R}\}$ si, et seulement si la procédure ci-dessus termine et répond oui. Si de plus, $\mathcal{R}$ termine, alors le problème de la validité d'une équation est décidable.

Démonstration : Le théorème 191 est composé de deux propriétés. Prouvons la première définie par l'équivalence suivante :

$\Gamma \vdash u = v$ si, et seulement si la procédure termine et répond vraie

Le sens *seulement si* est trivialement prouvé par le point 7) de la procédure.

Pour le sens *si*, le système de réécriture $\mathcal{R}$ étant confluent, si $\Gamma \vdash u = v$, nous savons qu'il existe un w tel que $u \xrightarrow{*}_{\mathcal{R}} w \xleftarrow{*}_{\mathcal{R}} v$. Chacune des dérivations $u \xrightarrow{*}_{\mathcal{R}} w$ et $v \xrightarrow{*}_{\mathcal{R}} w$ est finie, donc, le terme w sera à moment donné atteint par l'algorithme, la procédure faisant un parcours en largeur de la théorie sous-jacente.

La seconde propriété est donnée par l'implication :

si $\mathcal{R}$ est confluent et termine, alors le problème de la validité d'une équation est décidable

Comme $\mathcal{R}$ termine et est confluent, les formes normales de u et de v seront obtenues par la procédure ci-dessus en un temps fini. Ainsi, pour toute équation $u = v$, la procédure peut répondre en un temps fini si l'énoncé $\Gamma \vdash u = v$ est vraie ou fausse. ■

Décider de la confluence

Nous avons vu dans la section précédente que la propriété de Church-Rosser est équivalente à la propriété de confluence importante pour faire des règles de réécriture un algorithme. Nous avons aussi montré que la propriété de confluence est équivalente à une propriété plus simple quand le système de réécriture termine (propriété importante pour rendre l'algorithme sous-jacent effectif). Nous allons montrer ici que si la confluence locale est mise en échec, alors il existe des règles de réécriture dites **superposables**, appelées aussi **paires critiques**. L'intérêt de ces paires critiques est qu'elles sont calculables quand l'ensemble des règles de réécriture est fini.

Paires critiques

Dans cette section, nous allons définir la notion de paire critique. Mais avant cela donnons quelques notions qui seront utiles à ce propos. Tout d'abord la notion de position au sein d'un terme. Cette notion sera sensiblement équivalente à celle que nous avons définie dans les arbres de preuves lors de la preuve de l'élimination des coupures dans le calcul des séquents.

Notations. Utilisant une notation classique de numérotation des chemins des noeuds d'un arbre par un suite finie d'entiers naturels, nous pouvons nous référer aux positions dans un terme. Ainsi,

- étant donné un terme t , une **position** dans t est toute suite finie d'entiers naturels ω définissant le chemin menant de la racine au sous-terme dont la conclusion se trouve à cette position. Le sous-terme est noté $t|_\omega$.
- étant donnée une position $\omega \in \mathbb{N}^*$ dans un terme t , $t[t']_\omega$ est le terme obtenu à partir de t en remplaçant le sous-terme $t|_\omega$ par le terme t' .

Ceci nous permet alors de redéfinir une étape de réécriture sans passer par la notion de contexte. En effet, on pourra dire maintenant que $u \rightarrow_{\mathcal{R}} v$ si, et seulement s'il existe une règle $u' \rightarrow v'$, une substitution σ et une position ω telles que $\sigma^\#(u|_\omega) = \sigma^\#(u')$ et $v = u[\sigma^\#(v')]_\omega$.

Définition 152 (Superposable).

Soient t et t' deux termes. On dit que t et t' sont **superposables** à la position ω si, et seulement si $t|_\omega$ n'est pas une variable et $t|_\omega$ et t' sont unifiables (au sens défini dans la section sur la résolution pour la logique du premier ordre).

Une règle de réécriture $u \rightarrow v$ est **superposable** à une autre règle $u' \rightarrow v$ à l'occurrence ω si, et seulement si u est superposable à u' à ω . Pour tout unificateur σ de $u|_\omega$ et u' , on a les deux réductions $\sigma^\#(u) \rightarrow_{\mathcal{R}} \sigma^\#(v)$ et $\sigma^\#(u) \rightarrow_{\mathcal{R}} \sigma^\#(u)[\sigma^\#(v')]_\omega$.

Définition 153 (Paire critique).

Le couple $(\theta^\#(v), \theta^\#(u)[\theta^\#(v')]_\omega)$ est une **paire critique** où θ est un unificateur principal de $u|_\omega$ et u' .

Ainsi, les paires critiques peuvent être vues comme un signe d'ambiguïté du système de réécriture.

Exemple 192.

La règle d'associativité $(xy)z \rightarrow x(yz)$ se superpose à elle-même à la position 1, xy et $(x'y')z'$ étant unifiable ($x \mapsto x'y'$ et $y \mapsto z'$), donnant les deux réductions :

$$\begin{aligned} ((x'y')z')z &\rightarrow_{\mathcal{R}} (x'y')(z'z) \\ ((x'y')z')z &\rightarrow_{\mathcal{R}} (x'(y'z'))z \end{aligned}$$

et donc la paire critique $((x'y')(z'z), (x'(y'z'))z)$.

Théorème 193.

Si $v \mathcal{R} \leftarrow u \rightarrow \mathcal{R} w$, alors

- soit il existe z tel que $v \xrightarrow{*} \mathcal{R} z \mathcal{R} \xleftarrow{*} w$,
- soit il existe une paire critique (z, z') , un contexte $C[_]$ et une substitution σ tel que $v = C[\sigma(z)]$ et $w = C[\sigma(z')]$.

Démonstration : Supposons que $u \rightarrow \mathcal{R} v$ a été réécrit à partir de la règle $u_1 \rightarrow v_1$, de la substitution σ_1 et de la position ω_1 (i.e. $v = u[\sigma_1(v_1)]_{\omega_1}$, et $u \rightarrow \mathcal{R} w$ a été réécrit à partir de la règle $u_2 \rightarrow w_2$ de la substitution σ_2 et du contexte C_2 . Ici, deux cas sont à considérer selon la nature de ω_1 et ω_2 .

- Si ω_1 n'est pas un préfixe de ω_2 et inversement (i.e. que ces deux positions sont indépendantes), alors il existe dans v (resp. w) une position ω'_1 et une substitution σ'_1 (resp. ω'_2 et σ_2) telles que $\sigma_1^{\#}(v_{\omega'_1}) = \sigma_1^{\#}(u_2)$ (resp. $\sigma_2^{\#}(w_{\omega'_2}) = \sigma_2^{\#}(u_1)$). Nous avons alors $v \rightarrow \mathcal{R} [\sigma_1^{\#}(w_2)]$, $w \rightarrow \mathcal{R} C_2'[\sigma_2^{\#}(v_2)]$, et $C_1'[\sigma_1^{\#}(w_2)] = C_2'[\sigma_2^{\#}(v_2)]$.
- Supposons que parmi ω_1 et ω_2 , l'un est préfixe de l'autre par exemple ω_1 est préfixe de ω_2 . Pour simplifier la preuve, nous allons aussi supposer que $\omega_1 = 0$.² On a donc $u = \sigma_1^{\#}(u_1) \rightarrow \mathcal{R} \sigma_1^{\#}(v_1)$ et $u \rightarrow \mathcal{R} u[\sigma_2^{\#}(w_2)]_{\omega_2}$. Il y a deux cas selon la nature de ω_2 .
 - Si ω_2 est aussi une position dans u_1 . Supposons alors que les règles $u_1 \rightarrow v_1$ et $u_2 \rightarrow w_2$ n'ont aucune variable commune (on peut toujours renommer les variables pour que cette propriété soit vérifiée). On a alors que $\sigma_2^{\#}(\sigma_1^{\#}(u_1)_{|\omega_2}) = \sigma_2^{\#}(u_2)$. Donc, la substitution $\sigma = \sigma_2 \circ \sigma_1$ est un unificateur de $\sigma_1^{\#}(u_1)_{|\omega_2}$ et u_2 . Par hypothèse, les règles $u_1 \rightarrow v_1$ et $u_2 \rightarrow w_2$ sont superposables ce qui entraîne la paire critique $(\theta^{\#}(v_1), u[\theta^{\#}(w_2)]_{\omega_2})$. θ étant un unificateur principal, il existe alors nécessairement une substitution μ telle que $\sigma = \mu \circ \theta$.
 - Si ω_2 n'est pas une position dans u_1 . Ceci veut dire alors qu'il existe deux positions ω'_2 et ω''_2 telles que $\omega_2 = \omega'_2.\omega''_2$, $u_{|\omega'_2} = x$ où x est une variable et $\sigma_2^{\#}(\sigma_1^{\#}(x)_{|\omega_2}) = \sigma_2^{\#}(u_2)$. Là, on a alors deux chemins pour arriver à un même terme. Dans le premier, on réécrit $\sigma_1^{\#}(u)$ en $\sigma_1^{\#}(v_1)$ puis on applique autant de fois la règle $u_2 \rightarrow w_2$ au différent sous-terme $\sigma_1^{\#}(x)$ apparaissant dans $\sigma_1^{\#}(v_1)$. Dans le second cas, on commence par appliquer la seconde règle $u_2 \rightarrow w_2$ sur le terme $\sigma_1^{\#}(u_1)$, puis on applique la première règle $u_1 \rightarrow v_1$ à partir de la substitution $\sigma_1^{\#}(x) = \sigma(x)[\sigma_2^{\#}(w_2)]_{\omega_2}$. Dans les deux cas, on aboutira à un même terme réécrit. ■

Corollaire 9.

Un système de réécriture est localement confluent si, et seulement si toute paire critique est joignable.

En combinant avec le lemme de Newman et le corollaire précédent, on peut ramener l'étude de la confluence d'un système de réécriture qui termine à l'étude de la confluence des paires critiques. Ainsi, le calcul des unificateurs principaux étant décidable et pour un ensemble de règles de réécriture fini l'ensemble des paires critiques étant aussi fini, décider de la confluence d'un système de réécriture qui termine est définissable algorithmiquement.

2. Pour le cas plus général, il suffira de considérer un contexte C à ajouter à l'ensemble de la preuve ci-dessous.

Complétion de Knuth-Bendix

Le calcul des paires critiques étant décidable, D. Knuth et P.-B. Bendix ont eu alors l'idée de s'en servir pour essayer de "prototyper" un ensemble d'équations en un programme permettant à la fois d'implanter l'ensemble d'équations de départ mais aussi, par le fait que la propriété de Church-Rosser est équivalente à la confluence, d'avoir une procédure effective pour montrer qu'une équation est la conséquence (un théorème) de l'ensemble de départ. La méthode qu'ils ont définie permettant d'engendrer à partir d'un ensemble d'équations, un ensemble de règles de réécriture équivalent (au sens du pouvoir de preuve) est appelée **méthode de complétion de Knuth-Bendix**.

La méthode de complétion

Cette méthode de complétion est décrite par un ensemble de règle d'inférence. Ces règles d'inférence travaillent sur des couples $(\Gamma, \mathcal{R})$ où Γ est l'ensemble des équations encore à orienter et $\mathcal{R}$ est le système de réécriture jusque là obtenu. La procédure de complétion de Knuth-Bendix a alors en entrée

- Un ensemble d'équations Γ , et
- un ordre bien-fondé sur les termes $\succ$.³

La première étape de la procédure consiste à définir le couple $(\Gamma_0, \mathcal{R}_0)$ tel que $\Gamma_0 = \Gamma$ et $\mathcal{R}_0 = \emptyset$.

La méthode de complétion est alors définie par l'ensemble des règles d'inférence suivant :

DÉDUIRE	$\frac{\Gamma, \mathcal{R}}{\Gamma \cup \{u=v\}, \mathcal{R}}$	si $u \mathcal{R} \leftarrow u' \rightarrow_{\mathcal{R}} v$
SUPPRESSION	$\frac{\Gamma \cup \{u=u\}, \mathcal{R}}{\Gamma, \mathcal{R}}$	
ORIENTER1	$\frac{\Gamma \cup \{u=v\}, \mathcal{R}}{\Gamma, \mathcal{R} \cup \{u \rightarrow v\}}$	si $u \succ v$
ORIENTER2	$\frac{\Gamma \cup \{u=v\}, \mathcal{R}}{\Gamma, \mathcal{R} \cup \{v \rightarrow u\}}$	si $v \succ u$
SIMPLIFIER	$\frac{\Gamma \cup \{u=v\}, \mathcal{R}}{\Gamma \cup \{u'=v'\}, \mathcal{R}}$	si $u \xrightarrow{*}_{\mathcal{R}} u'$ et $v \xrightarrow{*}_{\mathcal{R}} v'$

La règle DÉDUIRE ajoute une équation qui peut être déduite des pics locaux de $\mathcal{R}$. L'ensemble de ces pics locaux est souvent infini, mais on a vu dans la section précédente que l'étude de la confluence de cet ensemble pouvait se ramener à étudier la confluence des paires critiques dont l'ensemble est décidable. L'application de la règle DÉDUIRE est alors décidable. Ceci est aussi vrai de façon évidente pour les trois autres règles. Enfin, l'orientation des équations étant faite selon l'ordre bien-fondé $\succ$, le système de réécriture obtenue à chaque étape termine. L'application de la règle SIMPLIFIER est aussi décidable et effective. Tout ceci fait que la procédure ci-dessus définit bien un algorithme permettant de transformer un ensemble d'équations en un ensemble de règle de réécriture. Cependant, on ne peut assurer qu'on atteindra toujours lors de cette complétion un système de réécriture ne possédant plus de paires critiques. En ce sens, cet algorithme n'aboutit pas toujours. En revanche quand il aboutit, on assure alors que le système de réécriture obtenu a le même pouvoir de preuve que l'ensemble d'équations de départ. C'est le sujet de la section suivante.

3. La connaissance de cet ordre bien-fondé est la seule façon d'assurer la complétion du système de réécriture final si on l'obtient.

Correction de la méthode

Les preuves que nous pouvons mener à partir d'un couple $(\Gamma, \mathcal{R})$ sont un mélange de preuve équationnelle et de réécriture. Ceci amène alors à la notion de **preuve mixte** définie par :

Définition 154 (Preuve mixte).

Pour tout couple $(\Gamma, \mathcal{R})$, notons $\xleftrightarrow{*}_{\Gamma, \mathcal{R}}$ la relation binaire sur $T_{\Sigma}(V)$ définie comme la fermeture réflexive, symétrique et transitive de la relation $\rightarrow_{\mathcal{R}} \cup \leftrightarrow_{\Gamma}$.

Nous allons maintenant montrer par les résultats qui suivent que la méthode de complétion présentée dans la section précédente est correcte, c'est-à-dire que chaque étape de complétion

- n'augmente pas la puissance de preuve,
- préserve la terminaison, et
- réduit la complexité des preuves mixtes selon un ordre que nous allons définir qui consiste à "tendre" de plus en plus vers des preuves par réécriture en transformant de plus en plus d'effluences locales (i.e. des éléments de $\leftarrow \circ \rightarrow$) en des "vallées" (i.e. des éléments de $\xrightarrow{*} \circ \xleftarrow{*}$).

Notations 194.

Si $\frac{\Gamma, \mathcal{R}}{\Gamma', \mathcal{R}'}$ est une instance d'une des règles d'inférence de la méthode de complétion, alors nous la noterons dans la suite $(\Gamma, \mathcal{R}) \vdash (\Gamma', \mathcal{R}')$ pour désigner cette inférence.

Commençons alors par montrer que les règles d'inférence de la méthode de complétion de Knuth-Bendix sont correctes, c'est-à-dire qu'elles ne changent pas à chaque étape la théorie logique sous-jacente (i.e. l'ensemble des preuves mixtes que l'on peut construire).

Théorème 195.

Si $(\Gamma_1, \mathcal{R}_1) \vdash (\Gamma_2, \mathcal{R}_2)$ alors $\xleftrightarrow{*}_{\Gamma_1, \mathcal{R}_1} = \xleftrightarrow{*}_{\Gamma_2, \mathcal{R}_2}$.

Démonstration : Ceci est triviale pour les 4 premières règles. Pour la règle SIMPLIFIER, nous pouvons remplacer chaque instance $u \leftrightarrow_{\Gamma} v$ dans toute preuve mixte de $\xleftrightarrow{*}_{\Gamma_1, \mathcal{R}_1}$ par $u \xrightarrow{*}_{\Gamma_1, \mathcal{R}_1} u' \leftrightarrow_{\Gamma_2, \mathcal{R}_2} v' \xleftarrow{*}_{\Gamma_1, \mathcal{R}_1} v$. Comme $R_1 = R_2$, on a aussi $u \xrightarrow{*}_{\Gamma_2, \mathcal{R}_2} u'$ et $v' \xleftarrow{*}_{\Gamma_2, \mathcal{R}_2} v$, et donc $u \xleftrightarrow{*}_{\Gamma_2, \mathcal{R}_2} v$. ■

De plus, tous les systèmes de réécriture que l'on engendre avec les règles d'inférence de la méthode de complétion terminent.

Théorème 196.

Pour tout couple $(\Gamma, \mathcal{R})$ obtenu à chaque étape de la méthode de complétion, le système de réécriture $\mathcal{R}$ termine.

Démonstration : Les seules règles où l'on augmente la taille des systèmes de réécriture sont ORIENTER1 et ORIENTER2. Or, dans chacun de ses deux règles, les équations sont orientées selon l'ordre bien fondé $\succ$. ■

Enfin chaque étape de la méthode de complétion définit des transformations des preuves mixtes. Ici, les sous-preuves réduites sont de deux types :

1. les effluences maximales (i.e. les éléments de $\mathcal{R} \xleftarrow{*} \circ \xrightarrow{*} \mathcal{R}$ pour un couple $(\Gamma, \mathcal{R})$), et
2. les preuves mixtes contenant au moins une instance de la forme $u \leftrightarrow_{\Gamma} v$ pour un couple $(\Gamma, \mathcal{R})$.

Si l'algorithme de complétion aboutit (i.e. il engendre un couple $(\Gamma, \mathcal{R})$ où $\Gamma = \emptyset$) alors par les théorèmes 195 et 196 et 191, nous avons une procédure de décision pour la théorie équationnelle de départ. Comme il est d'usage, une condition suffisante doit être imposée pour assurer que la méthode de complétion "fait tout pour aboutir" au système de réécriture recherché. Cette condition est naturelle quand nous avons des choix non-déterministes sur les équations $u = v$ à orienter ou à supprimer. L'idée sous-jacente est que tout choix accessible n'est pas repoussé indéfiniment. Autrement dit, toute équation apparaissant dans un Γ_i sera ultérieurement supprimée, normalisée ou orientée, et toute paire critique créée par la suite sera ultérieurement transférée dans la partie équationnelle, puis à son tour supprimée, normalisée ou orientée. Nous parlons alors d'*équité*. Ici, l'équité se définit :

Définition 155 (Équité).

Soit $((\Gamma_i, \mathcal{R}_i))_{i \geq 0}$ une chaîne telle que $(\Gamma_0, \mathcal{R}_0) \vdash (\Gamma_1, \mathcal{R}_1) \vdash \dots$. Nous disons alors qu'une équation est **persistante** si, et seulement si elle apparaît dans tous les Γ_i au-delà d'un certain rang.

Nous noterons $\Gamma_{\infty} = \bigcup_{i \geq 0} \bigcap_{j \geq i} \Gamma_j$ l'ensemble des équations persistantes, et par $R_{\infty} =$

$\bigcup_{i \geq 0} \bigcap_{j \geq i} R_j$ l'ensemble des règles de réécriture persistantes.

La chaîne $((\Gamma_i, \mathcal{R}_i))_{i \geq 0}$ est **équitable** si, et seulement si aucune équation $u = v$ est persistante et toute effluence locale est à un moment ou à un autre transformée en une équation.

Les règles d'inférence peuvent alors être utilisées pour définir un ordre sur les preuves mixtes et ainsi, normaliser de plus en plus les preuves mixtes en des preuves par réécriture à chaque étape de la méthode de complétion. Ici, la normalisation des preuves consiste essentiellement à remplacer des effluences locales par des vallées équivalentes. Bien sur, ceci n'est pas possible si l'ensemble des équations à orienter n'est pas vide. L'idée est alors d'utiliser les règles de complétion comme des transformations de preuves permettant d'écrire des preuves de plus en plus "normales" (i.e. des preuves où l'on a réduit le nombre d'effluences maximales). Pour cela, nous devons alors montrer en quoi les règles d'inférence de la méthode de complétion font décroître la complexité des preuves mixtes. Notons $>$ l'ordre sur les preuves mixtes défini par :

- DÉDUIRE : $u \mathcal{R} \leftarrow t \rightarrow_{\mathcal{R}} v > u \leftrightarrow_{\Gamma} v$
- EFFLUENCE : $u \mathcal{R} \leftarrow t \rightarrow_{\mathcal{R}} v > u \xrightarrow{*}_{\mathcal{R}} w \mathcal{R} \xleftarrow{*} v$
- ORIENTER :
- $u \leftrightarrow_{\Gamma} v > u \rightarrow v$

- $u \leftrightarrow_{\Gamma} v > v \rightarrow u$
- SIMPLIFIER :
- $u \leftrightarrow_{\Gamma} v > u \xrightarrow{*}_{\mathcal{R}} u' \leftrightarrow_{\Gamma} v$
- $u \leftrightarrow_{\Gamma} v > u \leftrightarrow u' \mathcal{R} \xleftarrow{*} v$

alors la fermeture $\gg$ de $>$ par contexte de preuve mixte et transitivité est bien-fondée en utilisant l'ordre défini comme l'extension de l'ordre de réduction $\succ$ aux multi-ensembles⁴ et aux multi-ensemble de ces multi-ensembles quand la complexité des preuves mixtes est définie par :

- *complexité des preuves élémentaires* : $\|u \rightarrow_{\mathcal{R}} v\| = \{\{u\}\}$ and $\|u \leftrightarrow_{\Gamma} v\| = \{\{u, v\}\}$
- *complexité des preuves mixtes* : les multi-ensembles de la complexité de ses preuves élémentaires.

Quand l'équité est assurée, la méthode de complétion normalise alors les preuves mixtes en des preuves par réécriture.

Théorème 197.

Si la chaîne $((\Gamma_i, \mathcal{R}_i))_{i \geq 0}$ est équitable, alors pour toute preuve mixte π obtenue à une étape i , il existe une preuve par réécriture équivalente (i.e. aboutissant à la preuve de la même équation) dans $\mathcal{R}_{\infty} = \bigcup_{i \geq 0} \bigcap_{j \geq i} \mathcal{R}_j$.

Démonstration : Pour prouver le théorème 197, nous raisonnons par induction bien-fondée sur $\gg$.

Soit $\pi : u \xrightarrow{*}_{\Gamma_i, \mathcal{R}_i} v$ une preuve mixte. Par définition, si cette preuve n'est pas une preuve par réécriture (i.e. une vallée de la forme $u \xrightarrow{*}_{\mathcal{R}} w \mathcal{R} \xleftarrow{*} v$), alors il existe au moins une étape d'induction de la forme $u' \leftrightarrow_{\Gamma_i} v'$, ou bien une effluence locale $u \rightarrow_{\mathcal{R}} w \mathcal{R} \leftarrow v$. Par équité, $u' \leftrightarrow_{\Gamma_i} v'$ aura disparue à une étape $j_1 > i$, et l'effluence locale sera transformée en une équation à une étape $j_2 > i$. Dans tous les cas, la preuve mixte π de départ est transformée en une nouvelle preuve mixte π' à une étape j tel que $j > i$ and $\pi \gg \pi'$. Par hypothèse d'induction, il existe une étape $k > j$ et preuve par réécriture $\pi'' : u \xrightarrow{*}_{\Gamma_k, \mathcal{R}_k} v$. Comme les règles de réécriture ne sont jamais retirées lors des étapes de complétion, nous avons $\mathcal{R}_{\infty} = \mathcal{R}_{\omega}$. ■

4. Soit E un ensemble. Un **multi-ensemble** sur E est une application $M : E \rightarrow \mathbb{N}$. $M(x)$ est le nombre d'occurrence de x dans M , pour $x \in E$. Les éléments d'un multi-ensemble M sont tous les éléments $x \in E$ tels que $M(x) \neq 0$. Un multi-ensemble est **fini** s'il n'a qu'un nombre fini d'éléments.

24. Logique de Hoare

24.1. Correction totale et partielle d'un algorithme

L'analyse d'algorithme s'intéresse à trois points fondamentaux :

1. Montrer que l'algorithme est correcte, c'est-à-dire montrer que l'algorithme fait bien ce que l'on attend de lui ;
2. Montrer qu'il termine sur toute entrée ;
3. Quand il termine, quantifier son temps d'exécution.

Les deux premiers points traitent de la correction de l'algorithme, partielle pour le premier point et totale pour le second. Le dernier point traite de la complexité des algorithmes. Ici, nous allons nous intéresser aux deux premiers points, c'est-à-dire que nous allons tenter de répondre à la question : Comment aborder de façon *scientifique* la correction des algorithmes ? On a déjà vu que la terminaison d'algorithmes est un problème indécidable. Qu'en est-il de sa correction partielle ? En fait, c'est aussi un problème indécidable. Il suffit d'appliquer le théorème de Rice. En effet, si l'on accepte la thèse de Turing/Church, à tout algorithme, il existe une machine de Turing $\mathcal{M}$. Maintenant, il est clair que cette propriété de correction partielle est une propriété non triviale des machines de Turing, et donc par le théorème de Rice, c'est une propriété indécidable. Alors comment établir la correction des algorithmes ? Dans les sections suivantes, nous allons donner des outils formels permettant de répondre à une telle question.

Terminaison des algorithmes

Nous pouvons faire une observation simple. Un algorithme termine toujours si :

- il ne comporte que des boucles For ;
- il n'inclut pas des fonctions (mutuellement) récursives ;
- il n'effectue pas d'opérations illégales (division par zéro, déréférencement d'un pointeur nul, etc.)

Cet ensemble caractérise l'ensemble des algorithmes dit primitifs récursifs. Le problème est que nous avons vu qu'il existait des algorithmes qui ne sont pas primitifs récursifs mais récursifs (e.g. la fonction d'Ackermann). Ces algorithmes récursifs demandent nécessairement l'utilisation de boucles While ou de fonctions récursives. Assurer la terminaison consiste donc à montrer que les données traitées par ces boucles et ces fonctions récursives aboutissent toujours à la condition d'arrêt. Ainsi, étant donnée une boucle While ou une fonction récursive, nous pouvons définir une relation binaire sur les données en entrée respectant l'ordre des itérations dans la boucle ou des appels récursifs selon le cas. Ainsi, montrer la terminaison consiste à montrer que cette relation ne possède pas de descente infinie, c'est-à-dire des suites de données qui n'aboutiraient pas à la condition d'arrêt, en d'autres mots que la relation est bien fondée. Par exemple, si nous considérons l'algorithme suivant :

Nous continuons ce processus jusqu'à obtenir un développement où seuls apparaissent que des entiers compris entre 0 et $b - 1$. Ainsi, sur notre exemple, nous devons encore procéder à une itération car un 3 demeure. On obtient alors :

$$266 = 2^{2^{2^1+1}} + 2^{2^1+1} + 2^1$$

Notons cette opération d'itération en base b appliquée à un entier positif n par $it_b(n)$. Ainsi, nous avons $it_2(266) = 2^{2^{2^1+1}} + 2^{2^1+1} + 2^1$. Notons maintenant $d_b(n)$ le nombre en base 10 après avoir appliqué l'opération $it_b(n)$ et remplacé tous les entiers b apparaissant dans le développement $it_b(n)$ par $b + 1$. Ainsi, pour notre exemple, $d_2(266) = 3^{3^{3^1+1}} + 3^{3^1+1} + 3^1$ dont l'écriture en base 10 a 38 chiffres.

Définissons alors les suites suivantes : Soit n un entier positif

- $g_1(n) = n$;
- $g_p(n) = d_p(g_{p-1}(n)) - 1$

Proposition 198.

Pour un entier positif donné n , la séquence $(g_1(n), g_2(n), \dots, g_p(n), \dots)$ est toujours finie. Plus précisément, si nous écrivons l'algorithme qui génère cette séquence, ce dernier termine toujours quelque soit l'entier positif n passé en entrée, c'est-à-dire : $\exists p, g_p(n) = 0$.

Démonstration : Pour démontrer ce résultat, nous devons utiliser un outil puissant de la théorie des ensembles, les *ordinaux* introduits par G. Cantor. La suite des ordinaux est une continuation des entiers naturels. Ainsi, ajoutons à l'ensemble des entiers naturels un nouvel élément ω , et étendons l'ordre classique $\leq$ à ce nouvel ensemble en posant que pour tout $n \in \mathbb{N}, n < \omega$; Continuons ce processus en ajoutant le nouvel élément $\omega + 1$ tel que $\omega < \omega + 1$, puis $\omega + 2, \dots, \omega \times 2, \omega \times 2 + 1, \dots, \omega^2, \omega^2 + 1, \dots, \omega^\omega, \dots, \omega^{\omega^\omega}$, etc.

Il est facile de voir que tout sous-ensemble de cet ensemble possède toujours un unique élément minimal. Ainsi, cet ensemble des ordinaux est bien fondé.

Établissons alors une correspondance entre la suite $(g_1(n), g_2(n), \dots, g_p(n), \dots)$ et une autre suite $(\bar{g}_1(n), \bar{g}_2(n), \dots, \bar{g}_p(n), \dots)$ où pour tout $b \geq 2$, $\bar{g}_b$ est la fonction définie par : $\bar{g}_b(n) = \bar{d}_b(g_{b-1}(n))$ et $\bar{d}_b(n)$ est l'ordinal obtenu après avoir appliqué $it_b(n)$ et remplacé tous les entiers b par l'ordinal ω .

Les ordinaux vérifiant les propriétés usuelles de l'arithmétique sur les entiers, on en déduit facilement que pour tout $b \geq 2$, la fonction $\bar{d}_b$ est strictement croissante.

Il est aussi facile de voir par construction que pour tout $b \geq 2$, on a $\bar{d}_{b+1} \circ d_b = \bar{d}_b$. On a alors les relations suivantes :

$$\begin{aligned} \bar{g}_{b+1}(n) = \bar{d}_{b+1}(g_b(n)) &= \bar{d}_{b+1}(d_b(g_{b-1}(n)) - 1) \\ &< \bar{d}_{b+1}(d_b(g_{b-1}(n))) = \bar{d}_b(g_{b-1}(n)) \end{aligned}$$

L'ensemble des ordinaux défini précédemment étant bien fondé, la suite $(\bar{g}_1(n), \bar{g}_2(n), \dots, \bar{g}_p(n), \dots)$ ne peut pas être infinie et donc il en est de même de la suite $(g_1(n), g_2(n), \dots, g_p(n), \dots)$. En effet, $\bar{g}_p(n)$ ne peut valoir 0 que si $g_p(n)$ lui-même vaut 0. Ainsi, il existe nécessairement un $p \in \mathbb{N}$ tel que $g_p(n) = 0$. ■

Correction partielle

La correction partielle d'un programme consiste à montrer que le programme **fait bien ce que l'on attend de lui** quelque soit les données en entrée. "Faire ce que l'on attend de lui" se définit par une propriété portant sur l'ensemble des données en entrée. Et donc, si $\mathcal{P}$ est la spécification de cette propriété, on doit montrer : $\forall \vec{x}, \mathcal{P}(\vec{x})$.

Si l'on a déjà montré la terminaison du programme, on dispose alors d'une relation bien fondée $\mathcal{R}$ sur les entrées de l'algorithme. On peut alors appliquer le principe d'induction mathématique de la façon suivante :

$$\forall \vec{x}, \mathcal{P}(\vec{x}) \iff [\forall \vec{x}', (\vec{x}, \vec{x}') \in \mathcal{R} \Rightarrow \mathcal{P}(\vec{x}') \Rightarrow \mathcal{P}(\vec{x})]$$

Si nous reprenons notre exemple du calcul du pgcd, deux propriétés naturelles définissent la correction de l'algorithme :

1. Le pgcd de n et m divise à la fois n et m . Formellement, ceci s'exprime par la propriété :

$$\mathcal{P}_1 \equiv \forall (n, m) \in \mathbb{N} \times \mathbb{N}, n \bmod \text{pgcd}(n, m) = 0 \wedge m \bmod \text{pgcd}(n, m) = 0$$

2. Le pgcd de n et m est le plus grand des diviseurs de n et m , i.e.

$$\mathcal{P}_2 \equiv \forall (n, m) \in \mathbb{N} \times \mathbb{N}, \forall p \in \mathbb{N}, n \bmod p = 0 \wedge m \bmod p = 0 \Rightarrow \text{pgcd}(n, m) \geq p$$

Proposition 199.

L'algorithme PGCD satisfait les propriétés $\mathcal{P}_1$ et $\mathcal{P}_2$.

Démonstration : Ces deux propriétés se démontrent par induction à partir de la relation bien fondée $\mathcal{R}$. Commençons par démontrer $\mathcal{P}_1$.

Cas de base. Le couple (n, m) satisfait la condition $n \bmod m = 0$. Selon la définition de $PGCD$, on a : $PGCD(n, m) = m$. Trivialement, $\mathcal{P}_1(n, m)$ est satisfaite.

Étape de récurrence. Ici deux cas sont à considérer selon la définition de $\mathcal{R}$.

1. $n < m$. Selon la définition de $PGCD$, on a $PGCD(n, m) = PGCD(m, n)$. Maintenant, par hypothèse d'induction, $\mathcal{P}_1(m, n)$ est satisfaite. On en déduit alors simplement que $\mathcal{P}_1(n, m)$ l'est aussi.
2. $n > m \wedge n \bmod m \neq 0$. Selon la définition de $PGCD$, on a $PGCD(n, m) = PGCD(m, n \bmod m)$. Maintenant, par hypothèse d'induction, $\mathcal{P}_1(m, n \bmod m)$ est satisfaite. De là, on déduit simplement que $m \bmod PGCD(n, m) = 0$. Il nous reste alors à le montrer pour n . Par définition, nous avons : $m \bmod PGCD(n, m) = 0 \Rightarrow (\exists q_1, m = PGCD(n, m) \times q_1)$. Nous avons aussi : $n \bmod m \neq 0 \Rightarrow \exists q_2, n = m \times q_2 + (n \bmod m)$. Enfin, par hypothèse d'induction, on sait aussi que : $\exists q_3 \in \mathbb{N}, n \bmod m = PGCD(n, m) \times q_3$. De là, on déduit que $n = PGCD(n, m) \times q_1 \times q_2 + PGCD(n, m) \times q_3 = PGCD(n, m) \times (q_1 \times q_2 + q_3)$.

Montrons maintenant la propriété $\mathcal{P}_2$.

Cas de base. Le couple (n, m) satisfait la condition $n \bmod m = 0$. Selon la définition de $PGCD$, on a : $PGCD(n, m) = m$. Trivialement, $\mathcal{P}_2(n, m)$ est satisfaite.

Étape de récurrence. Ici deux cas sont à considérer selon la définition de $\mathcal{R}$.

1. $n < m$. Selon la définition de $PGCD$, on a $PGCD(n, m) = PGCD(m, n)$. Maintenant, par hypothèse d'induction, $\mathcal{P}_2(m, n)$ est satisfaite. On en déduit alors simplement que $\mathcal{P}_2(n, m)$ l'est aussi.

2. $n > m \wedge n \bmod m \neq 0$. Selon la définition de $PGCD$, on a $PGCD(n, m) = PGCD(m, n \bmod m)$. Maintenant, par hypothèse d'induction, $\mathcal{P}_2(m, n \bmod m)$ est satisfaite. Il nous reste alors à montrer que $(n \bmod m) \bmod p = 0$. Par définition, nous avons : $n \bmod m \neq 0 \Rightarrow (\exists q, r \in \mathbb{N}, n = m \times q + r)$. Nous avons aussi : $\exists q_1, q_2 \in \mathbb{N}, n = p \times q_1 \wedge m = p \times q_2$. Enfin, par hypothèse d'induction, on sait que : $\exists q_3 \in \mathbb{N}, n \bmod m = PGCD(n, m) \times q_3$. De là, on déduit que $p \times q_1 = p \times (q_2 \times q) + r$. On a alors, $r = p \times (q_1 - q_2 \times q)$. Trivialement, on a $p \times q_1 > p \times (q_2 \times q)$ et donc $q_1 > q_2 \times q$. ■

24.2. Logique de Hoare : définition

Dans la section précédente, nous avons vu des outils mathématiques pour montrer la correction partielle et totale d'un algorithme. La méthode de preuve sous-jacente à ces outils mathématiques a deux inconvénients :

1. Les preuves de correction partielle menées sont rigoureuses mais pas formelles. Elles ne peuvent donc pas être certifiées, c'est-à-dire vérifiées mécaniquement.
2. Elle impose la terminaison des programmes.

Pour contourner ces deux problèmes, une logique a été définie, la **logique de Hoare**³. La logique de Hoare est une extension de la logique des prédicats. En ce sens, elle se définit aussi par une syntaxe, une sémantique et un calcul pour mener des preuves formelles. Elle est une extension de la logique des prédicats parce que de tout algorithme nous pouvons faire une structure du premier ordre. En effet, si nous reprenons notre exemple de l'algorithme du PGCD, nous avons la structure $\mathcal{M}$ suivante sur la signature $\Sigma = (\{0^0, mod^2, pgcd^2\}, \{Bool^1, Nat^1, <^2, =^2, \geq^2\})$:

- $M = \mathbb{B} \cup \mathbb{N}$;
- $0^{\mathcal{M}} = 0_{\mathbb{N}}$;
- $mod^{\mathcal{M}} : (n, m) \mapsto r$ t.q. $\exists q \in \mathbb{N}, n = m \cdot q + r$;
- $pgcd^{\mathcal{M}} : (n, m) \mapsto \begin{cases} pgcd^{\mathcal{M}}(m, n) & \text{si } n < m \\ m & \text{si } n \bmod^{\mathcal{M}} m = 0 \\ pgcd^{\mathcal{M}}(m, n \bmod^{\mathcal{M}} m) & \text{sinon} \end{cases}$
- $Bool^{\mathcal{M}} = \mathbb{B}$, et $Nat^{\mathcal{M}} = \mathbb{B}$;⁴
- $<^{\mathcal{M}} = \{(n, m) \mid n, m \in \mathbb{N} \text{ and } n <_{\mathbb{N}} m\}$;
- $=^{\mathcal{M}} = \{(n, n) \mid n\}$;
- $\geq^{\mathcal{M}} = \{(n, m) \mid n, m \in \mathbb{N} \text{ and } n \geq_{\mathbb{N}} m\}$;

De plus, les propriétés que nous voulons démontrer sur les algorithmes s'expriment très bien dans la logique des prédicats. Si nous reprenons notre exemple de l'algorithme PGCD, nous avons montré trois propriétés qui, exprimées en logique des prédicats, donnent :

- Le $pgcd$ est un diviseur des nombres passés en argument.

$$\varphi_1 \equiv \forall x. \forall y, x \bmod pgcd(x, y) = 0 \wedge y \bmod pgcd(x, y) = 0$$

3. Tony Hoare est un chercheur britannique inventeur de cette logique. Il a reçu le Turing award (équivalent de la médaille Fields ou du prix Nobel en informatique théorique).

4. La logique des prédicats que nous avons présentée est mono-type. Ces deux prédicats permettent de différencier les valeurs de M qui sont des Booléens de celles qui sont des entiers. Il existe des extensions de la logique des prédicats multi-types (voir à titre d'exemple le chapitre sur la logique équationnelle) où les signatures possèdent un ensemble supplémentaire S contenant les types manipulés, et l'arité des fonctions n'est plus un simple entier mais définie à partir de ces types. Par exemple, la signature de Peano contiendrait le type $S = \{Nat\}$, et la fonction $+$ aurait pour profil $Nat \times Nat \rightarrow Nat$.

— Le pgcd est le plus grand diviseur

$$\varphi_2 \equiv \forall x. \forall y. \forall z. x \bmod z = 0 \wedge y \bmod z = 0 \Rightarrow \text{pgcd}(x, y) \geq z$$

Pour l'algorithme PGCD, montrer sa correction revient à montrer l'énoncé suivant :

$$\{x > 0 \wedge y > 0\} \text{PGCD} \{\varphi_1 \wedge \varphi_2\}$$

De façon plus générale, étant donné un algorithme $\mathcal{A}$, montrer sa correction revient à montrer des énoncés de la forme :

$$\{\psi\} \mathcal{A} \{\varphi\}$$

où ψ et φ sont des formules de la logique des prédicats. La formule ψ , appelée **Précondition**, pose des conditions sur les données en entrée de l'algorithme, et φ , appelée **post-condition**, pose des conditions sur les résultats rendus par l'algorithme.

Langage While

Nous définissons un langage de programmation simple pour exprimer nos algorithmes. Ce langage est identique à celui présenté en section 4.3 à ceci près, pour être dans l'esprit de la logique mathématique de ne pas imposer un modèle des données mais d'être multi-modèles, que les données manipulées par l'algorithme seront fournies par les signatures Σ . Ainsi, aucune structure de données a priori n'est supposée. Rappelons ici sa définition.

Définition 156 (Langage While).

Soit $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$ une signature de la logique des prédicats. Cette signature est le langage pour exprimer les données manipulées par l'algorithme. Une **instruction** est :

- une **affectation** de la forme $x := t$ où $x \in \mathcal{V}$, et $t \in T_{\Sigma}(\mathcal{V})$,
- une **condition** de la forme IF φ THEN α ELSE β où $\varphi \in \text{For}(\Sigma)$, et α et β des programmes.
- une **boucle** de la forme WHILE φ DO α END où $\varphi \in \text{For}(\Sigma)$, et α un programme.

Un **programme** sur Σ est toute séquence finie de la forme $\alpha_1; \alpha_2; \dots; \alpha_n$ où chaque α_i ($i = 1, \dots, n$) est une instruction.

Pour montrer la correction d'un programme, on va exprimer des assertions de la forme :

Définition 157 (Assertions de Hoare).

Soit Σ une signature. Une **assertion** est une suite de symbole de la forme :

$$\{\psi\} P \{\varphi\}$$

où $\psi, \varphi \in \text{For}(\Sigma)$, et P un programme.

La sémantique intuitive d'une l'assertion $\{\psi\} P \{\varphi\}$ est "Si l'état de la mémoire satisfait ψ et P termine, alors, après exécution, l'état de la mémoire satisfait φ ".

Sémantique

À la différence de la section 4.3, nous allons donner ici une sémantique relationnelle compatible avec notre approche multi-modèles. De la même manière que dans la section 4.3, les programmes sont des "transformateurs d'états".

Définition 158 (États).

Soit $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$ une signature. Soit $\mathcal{M}$ un Σ -modèle. Un **état** est une interprétation des variables du programme dans $\mathcal{M}$, i.e. une application $\iota : \mathcal{V} \rightarrow M$. L'ensemble des états est donc l'ensemble des applications $M^{\mathcal{V}}$.

Définition 159 (Transformateurs d'états).

Soit Σ une signature. Soit $\mathcal{M}$ un Σ -modèle. Soit P un programme (sur Σ). Notons $\llbracket P \rrbracket_{\mathcal{M}}$ la relation binaire sur $M^{\mathcal{V}}$ définie inductivement sur la structure de P de la façon suivante :

- $(\iota, \iota') \in \llbracket x := t \rrbracket_{\mathcal{M}} \Leftrightarrow (\forall y \neq x \in V, \iota'(y) = \iota(y) \wedge \iota'(x) = \iota^{\#}(t))$
- $(\iota, \iota') \in \llbracket \text{IF } \varphi \text{ THEN } \alpha \text{ ELSE } \beta \rrbracket_{\mathcal{M}} \Leftrightarrow \begin{cases} \mathcal{M} \models_{\iota} \varphi \wedge (\iota, \iota') \in \llbracket \alpha \rrbracket_{\mathcal{M}} \\ \vee \\ \mathcal{M} \not\models_{\iota} \varphi \wedge (\iota, \iota') \in \llbracket \beta \rrbracket_{\mathcal{M}} \end{cases}$
- $(\iota, \iota') \in \llbracket \text{WHILE } \varphi \text{ DO } \alpha \text{ END} \rrbracket_{\mathcal{M}} \Leftrightarrow \begin{cases} \exists \iota_0, \dots, \iota_n \in M^{\mathcal{V}}, \\ \iota = \iota_0 \\ \forall i \in [0, n-1], \begin{cases} \mathcal{M} \models_{\iota_i} \varphi, \\ (\iota_i, \iota_{i+1}) \in \llbracket \alpha \rrbracket_{\mathcal{M}}, \\ \mathcal{M} \not\models_{\iota_n} \varphi, \text{ et} \\ \iota_n = \iota' \end{cases} \end{cases}$
- $(\iota, \iota') \in \llbracket \alpha_1; \dots; \alpha_n \rrbracket_{\mathcal{M}} \Leftrightarrow \begin{cases} \exists \iota_0, \dots, \iota_n \in M^{\mathcal{V}}, \\ \iota = \iota_0, \iota' = \iota_n, \text{ et} \\ \forall i \in [0, n-1], (\iota_i, \iota_{i+1}) \in \llbracket \alpha_{i+1} \rrbracket_{\mathcal{M}} \end{cases}$

Définition 160 (Validation).

Soit Σ une signature. Soit $\mathcal{M}$ un Σ -modèle. Soit P un programme sur Σ . Soit $\{\psi\} P \{\varphi\}$ une assertion. On dira que cette assertion est **correcte pour un état** $\iota \in M^{\mathcal{V}}$, notée $\models_{\iota} \{\psi\} P \{\varphi\}$, si $\mathcal{M} \models_{\iota} \psi$ et pour tout $\iota' \in M^{\mathcal{V}}$ telle que $(\iota, \iota') \in \llbracket P \rrbracket_{\mathcal{M}}$, on a $\mathcal{M} \models_{\iota'} \varphi$. On dira que cette assertion est **valide**, notée $\models \{\psi\} P \{\varphi\}$, si elle est correcte pour tous les états $\iota \in M^{\mathcal{V}}$.

Calcul de Hoare

Comme pour toute logique, nous pouvons définir un calcul défini par un ensemble de règles d'inférence qui traduit syntaxiquement le comportement des différents éléments du langage. Ce dernier va être comme il est d'usage, défini par induction sur la structure des programmes.

Comme nous allons utiliser deux types de preuves, celles que l'on peut mener dans la logique des prédicats avec n'importe lequel des calculs présentés dans le chapitre précédent, nous noterons $\vdash_{FOL}$ les preuves menées dans la logique des prédicats, et $\vdash_H$ celles que nous mènerons avec le calcul présenté ici.

Définition 161 (Calcul de Hoare).

Soit Σ une signature. Une assertion $\{\psi\} P \{\varphi\}$ est **dérivable**, noté $\vdash_H \{\psi\} P \{\varphi\}$, si on peut l'obtenir en utilisant un nombre fini de fois les règles suivantes :

- *Programme vide* : $\vdash_H \{\varphi\} \varepsilon \{\varphi\}$
- *Affectation* : $\vdash_H \{\varphi(x/t)\} x := t \{\varphi\}$ à condition que $\vdash_{FOL} \varphi(x/t)$;
- *Test* : si $\vdash_H \{\psi \wedge C\} \alpha \{\varphi\}$ et $\vdash_H \{\psi \wedge \neg C\} \beta \{\varphi\}$ alors $\vdash_H \{\psi\} IF C THEN \alpha ELSE \beta \{\varphi\}$
- *Boucle* : si $\vdash_H \{\varphi \wedge C\} \alpha \{\varphi\}$, alors $\vdash_H \{\varphi\} WHILE C DO \alpha END \{\varphi \wedge \neg C\}$
 φ s'appelle **invariant de boucle** ;
- *Composition d'instruction* : si pour tout i , $0 < i \leq n$, $\vdash_H \{\varphi_{i-1}\} \alpha_i \{\varphi_i\}$ alors $\vdash_H \{\varphi_0\} \alpha_1; \dots; \alpha_n \{\varphi_n\}$
- *Raffinement* : si $\vdash_{FOL} \psi \Rightarrow \psi'$, $\vdash_H \{\psi'\} P \{\varphi'\}$ et $\vdash_{FOL} \varphi' \Rightarrow \varphi$ alors $\vdash_H \{\psi\} P \{\varphi\}$

Par rapport à la méthode de preuve que nous avons présentée dans la section précédente, les preuves menées avec ce calcul ont pour avantages qu'une seule application de la règle pour la boucle While permet de conclure sur sa correction partielle. Entre autres, on peut conclure sur la correction partielle d'une boucle While même quand elle ne termine pas. Maintenant, elle a aussi pour inconvénient qu'il faut "inventer" l'invariant de boucle.

Exemple 200.

Cherchons à prouver l'assertion suivante :

$$\vdash_H \{x = a \wedge y = b\} z := x; x := y; y := z \{x = b \wedge y = a\}$$

1. Par la règle d'affectation on a $\vdash_H \{x = a \wedge y = b \wedge x = a\} z := x \{x = a \wedge y = b \wedge z = a\}$
2. De la même manière, $\vdash_H \{y = b \wedge y = b \wedge x = a\} x := y \{x = b \wedge y = b \wedge z = a\}$
3. Enfin, $\vdash_H \{x = b \wedge z = a \wedge z = a\} y := z \{x = b \wedge y = a \wedge z = a\}$
4. Maintenant, on peut montrer facilement les tautologies suivantes :
 - $\vdash_{FOL} x = a \wedge y = b \Rightarrow x = a \wedge y = b \wedge x = a$
 - $\vdash_{FOL} x = a \wedge y = b \wedge z = a \Rightarrow y = b \wedge y = b \wedge x = a$
 - $\vdash_{FOL} x = b \wedge y = b \wedge z = a \Rightarrow y = b \wedge y = b \wedge x = a$
 - $\vdash_{FOL} x = b \wedge y = a \wedge z = a \Rightarrow x = b \wedge y = a$
5. En appliquant 4 fois la règle de raffinement on obtient l'assertion que l'on cherche à prouver.

Ce calcul est à la fois correcte et complet, c'est-à-dire que nous avons l'équivalence :

Théorème 201.

$$\vdash_H \{\psi\} P \{\varphi\} \iff \models \{\psi\} P \{\varphi\}$$

Démonstration : Le sens $\Rightarrow$ se fait comme d'habitude par induction sur la structure des preuves.

Le sens $\Leftarrow$ est comme toujours un peu plus compliqué, et repose sur la notion de plus faible précondition que nous aborderons dans la prochaine section. Ici, je ne donne qu'une esquisse de la preuve. Étant donné un programme P et une post-condition φ , on note $wp(P, \varphi)$ la plus faible précondition, c'est-à-dire une formule ψ telle que $\{\psi\} P \{\varphi\}$ est une assertion valide, et pour toute formule ψ' telle que $\models \{\psi'\} P \{\varphi\}$, on a $\vdash_{FOL} \psi' \Rightarrow \psi$. Ainsi, on a pour toute formule ψ telle que $\vdash_H \{\psi\} P \{\varphi\}$ que $\vdash_{FOL} \psi \Rightarrow wp(P, \varphi)$. Il nous reste alors à montrer l'assertion suivante : $\vdash_H \{wp(P, \varphi)\} P \{\varphi\}$. Ceci se prouvera alors par induction structurelle sur la structure de P (voir le théorème 205). ■

Calcul de précondition la plus faible

Comme on l'a déjà dit dans la preuve de la complétude du calcul de Hoare, étant donné un programme P et une post-condition φ , la plus faible précondition est une formule ψ telle que $\models \{\psi\} P \{\varphi\}$, et pour toute formule ψ' telle que $\models \{\psi'\} P \{\varphi\}$, on a $\vdash_{FOL} \psi' \Rightarrow \psi$.

Ainsi, le calcul de Hoare définit un *problème logique* : Étant donné une précondition ψ , un programme P et une post-condition φ , est-ce que $\vdash_H \{\psi\} P \{\varphi\}$?

Le calcul de la précondition la plus faible *évaluera une fonction* : Étant donné un programme P et une post-condition φ , il faudra trouver l'*unique* formule ψ qui est la précondition la plus faible pour P et φ .

La fonction wp se définit par induction structurelle sur la structure des programmes. Les cas autres que la boucle sont assez directs :

- $wp(x := t, \varphi) = \varphi(x/t)$
- $wp(\alpha; \beta, \varphi) = wp(\alpha; wp(\beta, \varphi))$
- $wp(\text{IF } \varphi \text{ THEN } \alpha \text{ ELSE } \beta, \varphi) = (\varphi \Rightarrow wp(\alpha, \varphi)) \wedge (\neg\varphi \Rightarrow wp(\beta, \varphi))$

On a aussi la règle alternative suivante pour la conditionnelle :

$$wp(\text{IF } \varphi \text{ THEN } \alpha \text{ ELSE } \beta, \varphi) = (\varphi \wedge wp(\alpha, \varphi)) \vee (\neg\varphi \wedge wp(\beta, \varphi))$$

(Les 2 règles sont équivalentes car $\vdash_{FOL} ((b \Rightarrow p) \wedge (\neg b \Rightarrow q)) \Leftrightarrow ((b \wedge p) \vee (\neg b \wedge q))$)

Le cas de la boucle While est plus compliqué. En effet, supposons que nous avons une boucle While et une post-condition φ . Nous cherchons alors une précondition ψ qui est la plus faible et telle que :

- elle implique φ après l'exécution de la boucle,
- et donc tel que son calcul garantisse la terminaison de la boucle.

Le problème est que la terminaison est un problème indécidable, et donc trouver une telle précondition est aussi indécidable, i.e. il n'existe pas d'algorithme permettant de calculer $wp(\text{WHILE } C \text{ DO } \alpha \text{ END})$ dans tous les cas. Maintenant, ceci ne signifie pas que ce problème n'est pas "traitable". Trivialement, on a :

$$\text{WHILE } C \text{ DO } \alpha \text{ ENDO} = \text{IF } \neg C \text{ THEN } \varepsilon \\ \text{ELSE } \alpha ; \text{WHILE } C \text{ DO } \alpha \text{ ENDO}$$

On a alors :

$$wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi) = \begin{cases} (\neg C \wedge \varphi) \\ \vee \\ wp(\alpha, wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)) \end{cases}$$

On va montrer que ce calcul revient à calculer le petit point fixe d'une fonction définie sur les formules $For(\Sigma)$ pour Σ une signature donnée. Bien sûr, ceci demande au préalable de faire de l'ensemble $For(\Sigma)$ un treillis complet. Ceci est possible car $(For(\Sigma), \Rightarrow)$ est un treillis où $\wedge$ et $\vee$ donne respectivement les bornes supérieures et inférieures de deux formules ψ et φ pour l'ordre défini par l'implication $\Rightarrow$. Ce treillis a pour élément minimal $\perp$ définissant l'ensemble de toutes les antilogies, et élément maximal $\top$ définissant l'ensemble de toutes les tautologies. Maintenant, $For(\Sigma)$ ne contient que des conjonctions et des disjonctions finies. Or, pour avoir la propriété de complétude, nous devons considérer des conjonctions et des disjonctions infinies de formules. Notons $D(\Sigma)$ l'ensemble des formules défini comme $For(\Sigma)$ à ceci près que les conjonctions et les disjonctions infinies de formules sont acceptées. Maintenant, $(D(\Sigma), \Rightarrow)$ est un treillis complet.

Étant donnée une boucle $\text{WHILE } C \text{ DO } \alpha \text{ ENDO}$, définissons la fonction $f : D(\Sigma) \rightarrow D(\Sigma)$ par :

$$f : \psi \mapsto (\neg C \wedge \varphi) \vee wp(\alpha, \psi)$$

On a alors : $f(wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)) = wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)$. Ce qui revient à calculer un point fixe, en fait le plus petit par définition des boucles While (voire la sémantique dénotationnelle). Mais, f possède-t-elle un tel point fixe, i.e. $wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi) = \bigvee_{k \in \mathbb{N}} f^k(\perp)$ (une boucle while est équivalente à la disjonction des exécutions de $k \in \mathbb{N}$ tours de boucle) ? Oui, si f est continue.

Proposition 202.

f est continue sur $D(\Sigma)$.

Démonstration : La preuve repose sur le fait que $wp(P, \psi \vee \varphi) \Leftrightarrow wp(P, \psi) \vee wp(P, \varphi)$. ■

Ainsi, on a $wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi) = lfp(f)$ où $lfp(f)$ est le plus petit point fixe de f ,⁵ et donc calculer $wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)$ est équivalent à la disjonction des exécutions de k tours de la boucle, $k \in \mathbb{N}$, i.e.

$$wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi) = \exists k (k \geq 0 \wedge P_k)$$

où P_k est défini inductivement :

- $P_0 = \neg C \wedge \varphi$
- $P_{k+1} = C \wedge wp(\alpha, P_k)$

5. "lfp" pour *least fixpoint*.

La formule $wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)$ n'est alors valide que quand la boucle termine.

Proposition 203.

Soit $\mathcal{M}$ un Σ -modèle. Soit $\iota \in M^{\mathcal{V}}$ un état. Alors, $\mathcal{M} \models_{\iota} wp(P, \varphi)$ si, et seulement si pour tout $\iota' \in M^{\mathcal{V}}$ tel que $(\iota, \iota') \in \llbracket P \rrbracket_{\mathcal{M}}$, $\mathcal{M} \models_{\iota'} \varphi$.

Démonstration : Ceci se prouve par induction structurelle sur la structure de P . ■

Proposition 204.

Si $\models \{\psi\} P \{\varphi\}$, alors $\vdash_{FOL} \psi \Rightarrow wp(P, \varphi)$.

Démonstration : Soit $\mathcal{M}$ un Σ -modèle interprétant les données manipulées par P . Soit $\iota \in M^{\mathcal{V}}$ un état tel que $\mathcal{M} \models_{\iota} \psi$. Si $\mathcal{M} \models_{\iota} \psi$, on a par hypothèse que pour tout $\iota' \in M^{\mathcal{V}}$ tel que $(\iota, \iota') \in \llbracket P \rrbracket_{\mathcal{M}}$, $\mathcal{M} \models_{\iota'} \varphi$, et donc on a $\mathcal{M} \models_{\iota} wp(P, \varphi)$ par la proposition 203. L'on déduit alors que $\mathcal{M} \models wp(P, \varphi) \Rightarrow \psi$, et par complétude de la logique des prédicats, on a $\vdash_{FOL} wp(P, \varphi) \Rightarrow \psi$. ■

Théorème 205.

$$\vdash_H \{wp(P, \varphi)\} P \{\varphi\}$$

Démonstration : La preuve se fait par induction structurelle sur la structure de P . On montre le cas de la boucle While. On veut montrer que

$$\vdash_H \{wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)\} \text{WHILE } C \text{ DO } \alpha \text{ ENDO } \{\varphi\}$$

Notons $\psi = wp(\text{WHILE } C \text{ DO } \alpha \text{ ENDO}, \varphi)$. Par hypothèse d'induction, on a $\vdash_H \{wp(\alpha, \psi)\} \alpha \{\psi\}$. Montrons alors que $\vdash_{FOL} \psi \wedge \neg C \Rightarrow \varphi$, ce qui revient aussi à montrer que $\models \psi \wedge \neg C \Rightarrow \varphi$ par complétude de la logique des prédicats. Soit $\mathcal{M}$ un Σ -modèle pour les données, et soit $\iota : \mathcal{V} \rightarrow M$ un état tels que $\mathcal{M} \models_{\iota} \psi \wedge \neg C$. Ainsi, on a $\mathcal{M} \not\models_{\iota} C$ et $\mathcal{M} \models_{\iota} \psi$, et donc nécessairement par la proposition 203 on a $\mathcal{M} \models_{\iota} \varphi$ (on a nécessairement que $(\iota, \iota) \in \llbracket \text{WHILE } C \text{ DO } \alpha \text{ ENDO} \rrbracket_{\mathcal{M}}$). Ainsi, on a bien $\vdash_{FOL} \psi \wedge \neg C \Rightarrow \varphi$.

Montrons maintenant que $\vdash_{FOL} \psi \wedge C \Rightarrow wp(\alpha, \psi)$. Soit $\mathcal{M}$ un Σ -modèle pour les données, et soit $\iota : \mathcal{V} \rightarrow M$ un état tels que $\mathcal{M} \models_{\iota} \psi \wedge C$. Comme $\mathcal{M} \models_{\iota} \psi$, par la proposition 203, on a pour tout $\iota' \in M^{\mathcal{V}}$ tel que $(\iota, \iota') \in \llbracket \text{WHILE } C \text{ DO } \alpha \text{ ENDO} \rrbracket_{\mathcal{M}}$ que $\mathcal{M} \models_{\iota'} \varphi$, et donc $\mathcal{M} \models_{\iota} wp(\alpha, \psi)$.

On a par hypothèse d'induction $\vdash_H \{wp(\alpha, \psi)\} \alpha \{\psi\}$, et on vient de montrer que $\vdash_{FOL} \{\psi \wedge C\} \alpha \{\psi\}$. Ainsi, par la règle de la boucle While, on a $\vdash_H \{\psi\} \text{WHILE } C \text{ DO } \alpha \text{ ENDO } \{\psi \wedge \neg C\}$, d'où nous déduisons par la règle de raffinement que $\{\psi\} \text{WHILE } C \text{ DO } \alpha \text{ ENDO } \{\varphi\}$. ■

SOUS PARTIE 6

Logique modale

25. Introduction

Comme la logique des prédicats, la logique modale est une extension de la logique propositionnelle pour raisonner à partir de structures relationnelles. La logique modale a une longue tradition mais au départ plutôt philosophique. En effet, très tôt, la philosophie s'est intéressée à formaliser le raisonnement pour mettre fin aux disputes (*disputatio*) et démontrer sans possibilité de remise en cause toute propriété (telle que par exemple l'existence de dieu) en s'interdisant tout regard direct sur le réel. C'est l'approche rationaliste de la scolastique. Il fut alors observé, à la différence du raisonnement mathématique usuel où la vérité d'une propriété est indépendante de tout élément extérieur, qu'il existe une multitude de propositions dont la valeur de vérité peut changer selon l'environnement. C'est le fameux exemple de la proposition "Napoléon est mort". Cette proposition n'est vraie qu'après 1821 et fausse avant. Pendant longtemps, ce type de raisonnement n'était l'affaire que des philosophes et intéressait peu voire pas du tout les logiciens (mathématiciens). Depuis les années 70, ceci a beaucoup changé, et ceci est principalement dû à l'émergence de l'informatique. En effet, la logique modale est la logique la plus simple pour étudier les propriétés des structures relationnelles. Or, les structures relationnelles que l'on appelle plus communément des automates sont des abstractions formelles du comportement des programmes. Ainsi, la logique modale devient adaptée pour raisonner sur le comportement des systèmes informatiques et établir leur correction. Elle a aussi reçu une très forte attention de la communauté de l'informatique théorique, car elle est aussi adaptée pour raisonner sur les systèmes dits intelligents (*Intelligence Artificielle symbolique*) tels que les systèmes multi-agents ou encore la linguistique computationnelle (dont les modèles sont aussi des structures relationnelles).

Comme la logique des prédicats, la logique modale est une extension de la logique propositionnelle par ajout de deux modalités $\Box$ et $\Diamond$ dont les interprétations sont multiples :

- **Standard.** $\Box\varphi$ signifie φ est nécessairement vraie.
- **Épistémique.** $\Box\varphi$ signifie l'agent sait φ ou bien pour un système avec plusieurs agents $\Box_i\varphi$ l'agent i sait φ .
- **Prouvabilité.** $\Box\varphi$ signifie φ est prouvable.
- **Temporel.** $\Box\varphi$ signifie φ est toujours vraie.
- **Dynamique.** $\Box_\pi\varphi$ signifie qu'après avoir exécuté le programme π , φ est vraie.

Nous ne donnons que l'interprétation de la modalité $\Box$, l'interprétation de $\Diamond$ pourra s'exprimer par la dualité :

$$\neg\Box\varphi \iff \Diamond\neg\varphi$$

Comme pour les deux précédentes logiques, la logique modale va suivre dans sa présentation le découpage en une syntaxe, une sémantique et un calcul. Les sections suivantes vont alors présenter ces trois aspects de la logique modale.

26. Logique modale : syntaxe et sémantique

26.1. Syntaxe

Les signatures pour la logique modale sont les signatures de la logique propositionnelle, c'est-à-dire définies par un ensemble de variables propositionnelles P . Dans la suite, on supposera que cet ensemble est dénombrable.

Définition 162 (Formules modales).

Soit P une signature.

L'ensemble des **formules modales** sur P , notée $ForMod(P)$, est le plus petit sous-ensemble de $(P \cup \{\Rightarrow, \neg, \wedge, \vee, \Box, \Diamond, (,)\})^*$ construit selon les règles suivantes :

- toute variable propositionnelle de P est dans $ForMod(P)$;
- si $\varphi, \psi \in ForMod(P)$, alors $\varphi @ \psi \in ForMod(P)$ avec $@ \in \{\Rightarrow, \wedge, \vee\}$;
- si $\varphi \in ForMod(P)$, alors $\neg\varphi \in ForMod(P)$;
- si $\varphi \in ForMod(P)$, alors $M \varphi \in ForMod(P)$ avec $M \in \{\Box, \Diamond\}$.

Les notions de sous-formules, de variable d'une formule et de substitution se définissent comme pour la logique propositionnelle.

26.2. Sémantique : modèle et satisfaction

Les modèles de la logique modale sont des structures relationnelles appelés communément *Modèle de Kripke* du nom de leur créateur S. Kripke. Au départ, ce type de modèle a été défini pour donner une sémantique à la logique intuitioniste (logique où l'on refuse le tiers exclus).

Définition 163 (Structure de Kripke).

Soit P une signature.

Une **structure de Kripke** $\mathcal{M}$ sur P appelée aussi modèle de Kripke sur P ou plus simplement P -modèle, est un triplet (Q, R, ν) où :

- Q est un ensemble dont les éléments sont appelés des **états** ;
- $R \subseteq Q \times Q$ est une relation binaire sur Q appelée **relation d'accessibilité** ;
- $\nu : P \rightarrow 2^Q$ est une application appelée **valuation**.

Quand on oublie la valuation ν , les structures de Kripke sont appelées des *cadres*. Ainsi, un modèle de Kripke $\mathcal{M} = (Q, R, \nu)$ est dit **basé** sur le cadre (Q, R) .

Définition 164 (Relation de satisfaction).

Soit P une signature. Soit $\varphi \in \text{Formod}(P)$ une formule. Soit $\mathcal{M} = (Q, R, \nu)$ un P -modèle. Soit $q \in Q$ un état. $\mathcal{M}$ **satisfait** φ pour q , noté $\mathcal{M} \models_q \varphi$ si, et seulement si :

- si $\varphi \in P$, alors $\mathcal{M} \models_q \varphi$ ssi $q \in \nu(\varphi)$;
- si φ est de la forme $\Box\psi$, alors $\mathcal{M} \models_q \varphi$ ssi pour tout $q' \in Q$ tel que $(q, q') \in R$, on a $\mathcal{M} \models_{q'} \psi$;
- si φ est de la forme $\Diamond\psi$, alors $\mathcal{M} \models_q \varphi$ ssi il existe $q' \in Q$ tel que $(q, q') \in R$ tel que $\mathcal{M} \models_{q'} \psi$;
- le cas des connecteurs propositionnels $\Rightarrow, \neg, \wedge, \vee$ sont traités comme pour la logique propositionnelle et la logique des prédicats.

$\mathcal{M}$ **valide** φ , noté $\mathcal{M} \models \varphi$, si, et seulement si pour tout état $q \in Q$, $\mathcal{M} \models_q \varphi$.

Une formule φ est **valide** pour un cadre (Q, R) si elle est valide dans tous les modèles basés sur ce cadre.

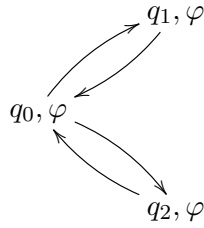
Nous pouvons aussi définir la sémantique d'une formule à partir d'un modèle $\mathcal{M}$ comme une application $\llbracket \mathcal{M} \rrbracket(_) : \text{Formod}(P) \rightarrow 2^Q$ de la façon suivante :

- $\llbracket \mathcal{M} \rrbracket(p) = \nu(p)$ pour $p \in P$;
- $\llbracket \mathcal{M} \rrbracket(\Box\varphi) = \{q \in Q \mid \forall q' \in Q, (q, q') \in R \Rightarrow q' \in \llbracket \mathcal{M} \rrbracket(\varphi)\}$;
- $\llbracket \mathcal{M} \rrbracket(\Diamond\varphi) = \{q \in Q \mid \exists q' \in Q, (q, q') \in R \wedge q' \in \llbracket \mathcal{M} \rrbracket(\varphi)\}$;
- $\llbracket \mathcal{M} \rrbracket(\varphi \wedge \psi) = \llbracket \mathcal{M} \rrbracket(\varphi) \cap \llbracket \mathcal{M} \rrbracket(\psi)$;
- $\llbracket \mathcal{M} \rrbracket(\varphi \vee \psi) = \llbracket \mathcal{M} \rrbracket(\varphi) \cup \llbracket \mathcal{M} \rrbracket(\psi)$;
- $\llbracket \mathcal{M} \rrbracket(\varphi \Rightarrow \psi) = (Q \setminus \llbracket \mathcal{M} \rrbracket(\varphi)) \cup \llbracket \mathcal{M} \rrbracket(\psi)$;
- $\llbracket \mathcal{M} \rrbracket(\neg\varphi) = Q \setminus \llbracket \mathcal{M} \rrbracket(\varphi)$;

Exemple 206.

On peut montrer que la formule $\Box\varphi \Rightarrow \varphi$ (et de façon duale la formule $\varphi \Rightarrow \Diamond\varphi$) est vraie pour les modèles de Kripke dont la relation d'accessibilité est réflexive. En effet, soit un modèle de Kripke $\mathcal{M} = (Q, R, \nu)$ et soit $q \in Q$ un état. Supposons que $\mathcal{M} \models_q \Box\varphi$. Ceci signifie que pour état $q' \in Q$ si $(q, q') \in R$, alors $\mathcal{M} \models_{q'} \varphi$. Maintenant, R étant réflexive, on a $(q, q) \in R$, et donc on en déduit que $\mathcal{M} \models_q \varphi$.

La question que l'on peut se poser est : Existe-t-il une structure de Kripke munie d'une relation non réflexive qui satisfait $\Box\varphi \Rightarrow \varphi$? La réponse est oui. Nous avons par exemple le modèle suivant :



De ceci, nous déduisons que la formule $\Box\varphi \Rightarrow \varphi$ n'est pas une tautologie de la logique modale. Elle le devient si nous restreignons les modèles aux structures de Kripke dont la relation d'accessibilité est réflexive.

À l'inverse quand nous nous restreignons aux cadres, nous pouvons montrer que la formule $\Box\varphi \Rightarrow \varphi$ n'est valide que pour les cadres (Q, R) où R est une relation réflexive.

Différentes logiques modales sont caractérisées par des types de formules valides pour des classes particulières de cadres.

Définition 165 (Propriétés d'un cadre).

Nous disons qu'un cadre (Q, R) est :

- Réflexif (resp. symétrique, transitif, linéaire) si R est réflexive (resp. symétrique, transitive, linéaire).
- Sériel si pour tout $q \in Q$, il existe $q' \in Q$ tel que $(q, q') \in R$.

On peut alors définir les logiques modales suivantes :

Logique	Condition sur le cadre	Axiomes
K	aucune condition	$\Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$
D	Sériel	$\Box\varphi \Rightarrow \Diamond\varphi$
T	Réflexive	$\Box\varphi \Rightarrow \varphi$
B	Réflexive et symétrique	$\Box\varphi \Rightarrow \varphi$ $\varphi \Rightarrow \Box\Diamond\varphi$
S4	Réflexive et transitive	$\Box\varphi \Rightarrow \Box\Diamond\Box\Diamond\varphi$
S5	Réflexive, symétrique et transitive	$\Box\varphi \Rightarrow \varphi$ $\Diamond\varphi \Rightarrow \Box\Diamond\varphi$

Dans la littérature, la formule $\Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ est appelée *Schéma de Kripke*. Toute logique modale $\mathcal{L}$ qui satisfait cette formule est dite *normale*. Toutes les logiques modales que nous considérons dans ce cours telles que toutes les logiques définies ci-dessus, sont des logiques normales.

Remarque 207 (Important).

Dans la suite, on appellera logique modale, que l'on notera abstraitement $\mathcal{L}$, tout sous-ensemble de $ForMod(P)$ pour une signature P donnée, telle que $\mathcal{L}$ contient toutes les conséquences sémantiques des axiomes ci-dessus^a. Par exemple, la logique K est l'ensemble des formules $\{\Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)\}^\bullet$.

a. La notion de conséquence sémantique est définie comme pour les logiques propositionnelle et des prédicats. Ainsi, étant donné un ensemble de formules Γ , on note $\Gamma^\bullet = \{\varphi \mid \forall \mathcal{M}, \mathcal{M} \models \Gamma \Rightarrow \mathcal{M} \models \varphi\}$.

26.3. Propriété de modèle fini et décidabilité

Nous allons montrer dans cette section un résultat important de la logique modale propositionnelle qui montre que l'ensemble des logiques ci-dessus (K, D, T , etc.) sont décidables (i.e. il existe un algorithme pour décider de la validité ou non d'une formule).

Propriété de modèle fini

Définition 166 (Propriété de modèle fini).

Une logique modale $\mathcal{L}$ possède la propriété de modèle fini si pour toute formule φ qui n'est pas valide dans $\mathcal{L}$, il existe un modèle de Kripke fini $\mathcal{M}$ dans lequel φ est falsifiable (i.e. pour un état donné q de $\mathcal{M}$, on $\mathcal{M} \not\models_q \varphi$).

Pour montrer qu'une classe de modèles de Kripke possède la propriété de modèle fini, nous allons définir une procédure qui transforme un modèle infini en un modèle fini. Cette procédure repose sur un quotient des modèles. Ce quotient se définit à partir de la relation d'équivalence sur les états suivante : Soit $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke, et soit Γ un ensemble de formules de $ForMod(P)$ où P est une signature

$$q \sim_\Gamma q' \iff (\forall \varphi \in \Gamma, \mathcal{M} \models_q \varphi \iff \mathcal{M} \models_{q'} \varphi)$$

Proposition 208.

Si Γ est fini, alors $|Q_{/\sim_\Gamma}| \leq 2^{|\Gamma|}$ où $|_|$ dénote la cardinalité de l'ensemble passé en argument.

Démonstration : Définissons l'application $\iota : Q_{/\sim_\Gamma} \rightarrow 2^\Gamma$ par : $[q]_\Gamma \mapsto \{\varphi \in \Gamma \mid \mathcal{M} \models_q \varphi\}$. Par définition, ι est injective et donc nous pouvons directement conclure que $|Q_{/\sim_\Gamma}| \leq 2^{|\Gamma|}$. ■

Étant donnés un modèle de Kripke $\mathcal{M} = (Q, R, \nu)$ sur une signature P et un ensemble de formules $\Gamma \subseteq \text{Formod}(\Gamma)$, on peut définir le modèle quotient $\mathcal{M}_\Gamma = (Q', R', \nu')$ pour $\sim_\Gamma$ de la façon suivante :

- $Q' = Q / \sim_\Gamma$;
- $\forall p \in P, [q]_\Gamma \in \nu'(p) \iff q \in \nu(p)$;
- $\forall q, q' \in Q, ([q]_\Gamma, [q']_\Gamma) \in R' \iff (\exists q_1 \in [q]_\Gamma, \exists q'_1 \in [q']_\Gamma, (q_1, q'_1) \in R)$.

Proposition 209.

Pour tous $q, q' \in Q$, si $([q]_\Gamma, [q']_\Gamma) \in R'$, $\Diamond \varphi \in \Gamma$, et $\mathcal{M} \models_{q'} \varphi$, alors $\mathcal{M} \models'_q \Diamond \varphi$.

Démonstration : Par hypothèse, nous avons $([q]_\Gamma, [q']_\Gamma) \in R'$. Par définition, il existe $q_1 \in [q]_\Gamma$ et $q'_1 \in [q']_\Gamma$ tels que $(q_1, q'_1) \in R$. Maintenant, par l'hypothèse que $\mathcal{M} \models_{q'} \varphi$, on a aussi $\mathcal{M} \models_{q'_1} \varphi$, et donc $\mathcal{M} \models_{q_1} \Diamond \varphi$, d'où l'on déduit par définition de la relation d'équivalence $\sim_\Gamma$ que $\mathcal{M} \models_q \Diamond \varphi$. ■

Dans la littérature, les modèles satisfaisant les trois propriétés du quotient plus celle de la proposition ci-dessus, sont appelés des *filtrations* de $\mathcal{M}$. Ainsi, $\mathcal{M}_\Gamma$ est une filtration particulière. En fait, c'est la plus fine.

Définition 167 (Fermé par sous-formule).

Un ensemble de formules Γ est dit **fermé par sous-formules** si pour toute formule $\varphi \in \Gamma$, $SF(\varphi) \subseteq \Gamma$.

L'application SF est définie comme pour la logique propositionnelle en prenant en compte les formules modales, i.e. $SF(\Box \varphi) = \{\Box \varphi\} \cup SF(\varphi)$ and $SF(\Diamond \varphi) = \{\Diamond \varphi\} \cup SF(\varphi)$.

Proposition 210.

Soit $\Gamma \subseteq \text{Formod}(P)$ un ensemble de formules fermé par sous-formule. Soit $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke sur P . Alors, on a :

$$\forall \varphi \in \Gamma, \forall q \in Q, \mathcal{M} \models_q \varphi \iff \mathcal{M}_\Gamma \models_{[q]_\Gamma} \varphi$$

Démonstration : La preuve se fait par induction sur la structure de la formule φ . L'hypothèse d'induction est applicable parce que l'ensemble Γ est fermé par sous-formule. ■

Théorème 211 (Propriété de modèle fini).

Chaque formule φ satisfiable dans un modèle de Kripke est satisfiable dans un modèle de Kripke fini.

Démonstration : φ étant satisfiable, il existe un modèle de Kripke $\mathcal{M}$ et un état q de $\mathcal{M}$ tel que $\mathcal{M} \models_q \varphi$. Posons $\Gamma = SF(\varphi)$. Par définition, Γ est un ensemble fini et fermé par sous-formule. La conclusion est une conséquence directe des propositions 208 et 210. ■

Nous retrouvons la définition de propriété de modèle fini parce que la satisfiabilité est équivalente à la notion de falsifiabilité dans le sens suivant : φ est satisfiable si, et seulement si $\neg\varphi$ est falsifiable.

Décidabilité de la validation

Montrons tout d'abord que la validation est décidable pour les modèles de Kripke finis.

Proposition 212.

Soit $\mathcal{M} = (Q, R, \nu)$ un modèle fini de Kripke sur une signature P . Soit $\varphi \in ForMod(P)$. L'énoncé $\mathcal{M} \models \varphi$ est décidable.

Démonstration : Soit l'algorithme suivant : soit $\varphi_1, \dots, \varphi_n$ la suite de toutes les sous-formules de φ dans l'ordre de leur taille (et donc $\varphi = \varphi_n$). Pour tout i , $1 \leq i \leq n$, étiquetons chaque état $q \in Q$ par φ_i si $\mathcal{M} \models_q \varphi_i$, et par $\neg\varphi_i$ sinon. Enfin, on retourne *True* si chacun des états q de Q est étiqueté par φ . ■

Trivialement, cet algorithme termine après $|SF(\varphi)|$ étapes.

Définition 168 (Finiment axiomatisable).

Une logique modale $\mathcal{L}$ est **finiment axiomatisable** s'il existe un ensemble fini de formules Γ de $ForMod(P)$ tel que $\mathcal{L} = \Gamma^\bullet$.

Les logiques modales K, D, T, etc. sont toutes axiomatisables.

Théorème 213 (Décidabilité).

Soit $\mathcal{L}$ une logique modale finiment axiomatisable et qui a la propriété de modèle fini. Alors, $\mathcal{L}$ est décidable.

Démonstration : Nous allons montrer que $\mathcal{L}$ et $ForMod(P) \setminus \mathcal{L}$ sont récursivement énumérables. Tout d'abord, par hypothèse, on sait qu'il existe un ensemble fini de formules Γ tel que $\mathcal{L} = \Gamma^\bullet$.

Montrons tout d'abord que $\mathcal{L}$ est récursivement énumérable. Nous avons vu qu'une preuve à partir d'un ensemble de formules Γ est une séquence finie de formules $\varphi_1, \dots, \varphi_n$ telle que pour chaque i , $1 \leq i \leq n$, φ_i est soit une formule de Γ , soit obtenue par application de règles d'inférence à partir de formules précédentes. Ceci entraîne directement que l'ensemble des théorèmes obtenus à partir d'un ensemble de formules est récursivement énumérable. Nous verrons dans la suite, que toutes les logiques modales ci-dessus possèdent un calcul correcte et complet, et ainsi l'ensemble $\mathcal{L}$ est récursivement énumérable.

Montrons maintenant que $ForMod(P) \setminus \mathcal{L}$ est récursivement énumérable. Pour cela, on se donne un ensemble $\mathcal{C}$ de cadres finis (c'est la propriété de modèle fini qui nous permet de nous restreindre à un tel ensemble). Soit $p_1, p_2, \dots$ une énumération des variables propositionnelles de P . Définissons deux ensembles L et Φ récursivement sur $n \in \mathbb{N}$ de la façon suivante :

- $L_0 = \Phi_0 = \emptyset$;
 - pour tout $n > 0$,
 - $L_n = \{\mathcal{M} = (Q, R, \nu) \mid (Q, R) \in \mathcal{C} \wedge \nu : \{p_1, \dots, p_n\} \rightarrow 2^Q \wedge \mathcal{M} \models \Gamma\}$
 - $\Phi_n = \{\varphi \mid P_\varphi \subseteq \{p_1, \dots, p_n\} \wedge (\exists k \leq n, \exists \mathcal{M} \in L_k, \mathcal{M} \not\models \varphi)\}$.
- où P_φ est l'ensemble des variables propositionnelles apparaissant dans φ .

Ainsi, avec l'ensemble Φ , on obtient une définition récursive des formules de $ForMod(P)$ qui ne sont pas valides pour $\mathcal{C}$. Soit $\varphi \in \Phi$. Par définition, il existe un modèle de Kripke fini $\mathcal{M}$ construit à partir de $\mathcal{C}$ tel que $\mathcal{M} \not\models \varphi$. Par construction, il existe un $n \in \mathbb{N}$ tel que $\mathcal{M} \in L_n$. Si $P_\varphi \subseteq \{p_1, \dots, p_m\}$, ceci signifie que φ va apparaître dans Φ après $\max(m, n)$ étapes. ■

Corollaire 10.

Les logiques modales K, D, T, B, S4, et S5 sont décidables.

Démonstration : Elles sont trivialement finiment axiomatisables et elles possèdent la propriété de modèle fini. ■

26.4. Traduire la logique modale dans la logique des prédicats

La logique modale comme la logique des prédicats sont des extensions de la logique propositionnelle à la fois syntaxiquement mais aussi sémantiquement. Maintenant, on peut se poser la question du lien qui lie ces deux logiques ? On va montrer que la logique modale caractérise un sous-ensemble de la logique des prédicats. En effet, nous avons la traduction suivante : Étant donnée une signature P , on associe la signature de la logique des prédicats $\Sigma = (\mathcal{F}, \mathcal{R}, \mathcal{V})$ où

- $\mathcal{F} = \emptyset$;
- $\mathcal{R} = \{p^1 \mid p \in P\} \cup \{R^2\}$;
- $\mathcal{V}$ est un ensemble dénombrable de variables.

Soit $x \in \mathcal{V}$ une variable. On définit l'application $\Theta_x : ForMod(P) \rightarrow For(\Sigma)$ par induction sur la structure des formules modales de la façon suivante :

- $\Theta_x(p) = p(x)$;
- $\Theta_x(\neg\varphi) = \neg\Theta_x(\varphi)$;
- $\Theta_x(\varphi \wedge \psi) = \Theta_x(\varphi) \wedge \Theta_x(\psi)$;
- $\Theta_x(\varphi \vee \psi) = \Theta_x(\varphi) \vee \Theta_x(\psi)$;
- $\Theta_x(\varphi \Rightarrow \psi) = \Theta_x(\varphi) \Rightarrow \Theta_x(\psi)$;
- $\Theta_x(\Diamond\varphi) = \exists y. R(x, y) \wedge \Theta_y(\varphi)$;
- $\Theta_x(\Box\varphi) = \forall y. R(x, y) \Rightarrow \Theta_y(\varphi)$;

où y est une variable fraîche.

Remarquons que pour toute formule modale φ , $\Theta_x(\varphi)$ a une unique variable libre x . Cette unique variable libre x dénote l'état courant. De la même manière que pour les formules, il est facile de transformer tout modèle de Kripke en un Σ -modèle. Pour $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke, on associe le Σ -modèle que l'on note aussi $\mathcal{M}$, suivant :

- l'ensemble des valeurs $M = Q$;
- pour tout prédicat $p^1 \in \mathcal{R}$, $p^{\mathcal{M}} = \nu(p)$;
- $R^{\mathcal{M}} = R$.

On introduit la notation suivante : Soit $\mathcal{M}$ un modèle de Kripke. Soit $\varphi \in \text{Formod}(P)$ une formule modale. La notation $\mathcal{M} \models_{[x \mapsto q]} \Theta_x(\varphi)$ signifie la satisfaction de la formule $\Theta_x(\varphi)$ de $\text{Form}(\Sigma)$ par le Σ -modèle $\mathcal{M}$ à partir de l'interprétation de variable ι qui à x associe la valeur q (l'interprétation des autres variables apparaissant dans $\Theta_x(\varphi)$ importe peu car elles sont toutes liées).

Théorème 214.

Pour tout modèle de Kripke $\mathcal{M}$ sur une signature P , tout état q de $\mathcal{M}$, et toute formule modale $\varphi \in \text{Formod}(P)$, on a :

$$\mathcal{M} \models_q \varphi \iff \mathcal{M} \models_{[x \mapsto q]} \Theta_x(\varphi)$$

Démonstration : La preuve se fait par induction sur la structure de φ . La preuve est assez directe et ne pose aucune difficulté. Nous ne donnons ici à titre d'illustration que la preuve pour la modalité $\Box$.

$$\begin{aligned} \mathcal{M} \models_q \Box \psi &\iff \forall q' \in Q, (q, q') \in R \Rightarrow \mathcal{M} \models_{q'} \psi \\ &\iff \forall q' \in Q, (q, q') \in R \Rightarrow \mathcal{M} \models_{[y \mapsto q']} \Theta_y(\psi) \text{ (hypothèse de récurrence)} \\ &\iff \mathcal{M} \models_{[x \mapsto q]} \Theta_x(\varphi) \end{aligned}$$

On a bien sûr le corollaire suivant :

Corollaire 11.

Pour tout modèle de Kripke $\mathcal{M}$ sur une signature P , et toute formule modale $\varphi \in \text{Formod}(P)$, on a :

$$\mathcal{M} \models \varphi \iff \mathcal{M} \models \forall x, \Theta_x(\varphi)$$

26.5. Bissimulation

Introduction

Une bisimulation est une relation entre deux modèles de Kripke permettant de les comparer. Cette notion a reçu un très fort intérêt en informatique théorique, du fait que les modèles de Kripke et leurs avatars et les logiques modales sont les outils formels pour modéliser et raisonner sur les systèmes dits réactifs. L'intérêt de la formalisation des systèmes réactifs à partir des modèles de Kripke est de permettre une vérification automatique de propriétés (souvent exprimées en logique temporelle) telles que par exemple l'absence de blocage. On parle alors de *model-checking*.¹ Les algorithmes de vérification que nous pouvons définir se caractérisent souvent par

1. Ces algorithmes de model-checking ont déjà été présentés dans le cadre du cours "Modélisation et vérification formelle de systèmes complexes à logiciels prépondérants" en ST5).

un parcours du graphe sous-jacent au modèle. Or, il n'est pas rare en pratique de manipuler des modèles de Kripke comprenant un nombre très important d'états. Par exemple, un programme (même petit) manipule souvent une centaine de variables sous 32 bits (voir 64), ce qui donne un nombre d'états possibles plus important que le nombre de particules dans l'univers, et donc rend rédhibitoire l'utilisation de tels algorithmes pour vérifier des propriétés sur son comportement. On appelle ce problème, *explosion combinatoire*. Le phénomène d'explosion combinatoire a deux sources distinctes : la taille du modèle augmente exponentiellement soit à cause du nombre de variables manipulées, soit à cause du nombre de composants des systèmes dans le cas où les systèmes sont concurrents. Pour répondre à ce problème de l'explosion combinatoire, il est important de réduire la taille des systèmes manipulés. Les bis simulations peuvent permettre une telle réduction.

Définition

Définition 169 (Bissimulation).

Soient $\mathcal{M}_1 = (Q_1, R_1, \nu_1)$ et $\mathcal{M}_2 = (Q_2, R_2, \nu_2)$ deux modèles de Kripke définis sur une même signature P . Une **bissimulation** B est une relation sur $Q_1 \times Q_2$ telle que si $(q_1, q_2) \in B$ alors les conditions suivantes sont satisfaites :

- **Invariance.** $\forall p \in P, q_1 \in \nu_1(p) \iff q_2 \in \nu_2(p)$.
- **Zig.** Pour tout $q'_1 \in Q_1$ tel que $(q_1, q'_1) \in R_1$, il existe $q'_2 \in Q_2$ tel que $(q_2, q'_2) \in R_2$ et $(q'_1, q'_2) \in B$;
- **Zag.** Pour tout $q'_2 \in Q_2$ tel que $(q_2, q'_2) \in R_2$, il existe $q'_1 \in Q_1$ tel que $(q_1, q'_1) \in R_1$ et $(q'_1, q'_2) \in B$.

Quand $(q_1, q_2) \in B$, on dit alors que q_1 et q_2 sont **bissimilaires** et que B est une **bissimulation** pour $\mathcal{M}_1$ et $\mathcal{M}_2$ qui lie q_1 et q_2 . Diagrammiquement, ceci donne :

$$\begin{array}{ccc} q_1 & \xrightarrow{B} & q_2 \\ R_1 \downarrow & & \downarrow R_2 \\ q'_1 & \xrightarrow{B} & q'_2 \end{array}$$

Enfin, on définit la relation $\approx \subseteq Q_1 \times Q_2$ par $q_1 \approx q_2$ si, et seulement s'il existe une **bissimulation** B pour $\mathcal{S}_1$ et $\mathcal{S}_2$ qui lie q_1 et q_2 .

Les bis simulations ont des propriétés remarquables qui font tout leur intérêt. Tout d'abord, la propriété de bis similarité est clos par union.

Proposition 215.

Avec les notations de la définition 169, pour tout $q_1 \in Q_1$ et tout $q_2 \in Q_2$ bis similaires, $\approx$ est la plus large bis simulation qui lie q_1 et q_2 .

Démonstration : Par définition de $\approx$, elle est trivialement une bis simulation. Maintenant, étant définie comme l'union de toutes les bis simulations, elle est forcément la plus large. ■

Quand les systèmes réactifs sont modélisés par des modèles de Kripke, une notion importante est celle de *trace* qui représente une exécution possible d'un système. Les traces vont être des chemins infinis en suivant la relation d'accessibilité R . Elle sont infinies car une des caractéristiques des systèmes réactifs est d'être toujours "en veille" pour répondre à tout stimuli extérieur (l'arrêt d'un système réactif est un "bug").

Définition 170 (Trace).

Soit $\mathcal{M} = (Q, R, \nu)$. Une **trace** pour $\mathcal{M}$ est toute séquence d'états $(q_0, q_1, \dots, q_n, \dots)$ telle que pour tout $i \in \mathbb{N}$, $(q_i, q_{i+1}) \in R$.

L'intérêt des bis simulations vient alors du résultat suivant :

Proposition 216 (Co-induction).

Soient $\mathcal{M}_1 = (Q_1, R_1, \nu_1)$ et $\mathcal{M}_2 = (Q_2, R_2, \nu_2)$ deux modèles de Kripke. Supposons $q_0^1 \in Q_1$ et $q_0^2 \in Q_2$ tels que $q_0^1 \approx q_0^2$, alors pour toute trace $(q_0^1, q_1^1, \dots, q_n^1, \dots)$ dans $\mathcal{M}_1$, il existe une trace $(q_0^2, q_1^2, \dots, q_n^2, \dots)$ dans $\mathcal{M}_2$ telle que pour tout $j \in \mathbb{N}$, $q_j^1 \approx q_j^2$. L'inverse est aussi vraie.

Démonstration Nous allons construire une trace $(q_0^2, q_1^2, \dots, q_n^2, \dots)$ par récurrence sur $\mathbb{N}$. Le cas de base est trivial car nous avons comme hypothèse que $q_0^1 \approx q_0^2$. Supposons pour tout j , $0 \leq j \leq n$, $q_j^1 \approx q_j^2$. Montrons alors comment choisir q_{n+1}^2 tel que $q_{n+1}^1 \approx q_{n+1}^2$. Par hypothèse de récurrence, nous savons que $q_n^1 \approx q_n^2$. De plus, par hypothèse, nous avons $(q_n^1, q_{n+1}^1) \in R_1$. Donc, en appliquant la condition **Zig** des bis simulations, nous savons qu'il existe $q \in Q_2$ tel que $(q_n^2, q) \in R_2$ et $q_{n+1}^1 \approx q$. Il suffit alors de poser $q_{n+1}^2 = q$. Étant donnée une trace $(q_0^2, q_1^2, \dots, q_n^2, \dots)$, la construction d'une exécution $(q_0^1, q_1^1, \dots, q_n^1, \dots)$ à partir de q_0^1 est identique. $\square$

Cette proposition définit un type de preuve dual à la preuve par induction structurale que l'on appelle *preuve par co-induction*. De la même manière que la preuve par induction est valide pour des objets définis comme les plus petits points fixes de fonctions continues sur un treillis complet, la preuve par co-induction (qui revient à définir une bis simulation) est valide pour les objets définis comme les plus grands points fixes de fonctions continues sur un treillis complet. Ici le treillis considéré est celui des bis simulations définies entre $\mathcal{M}_1$ et $\mathcal{M}_2$ munie de l'inclusion comme relation d'ordre.

Une bis simulation peut porter sur un même modèle de Kripke $\mathcal{M} = (Q, R, \nu)$. Dans ce cas-là, on montre alors que la plus large bis simulation $\approx$ est une relation d'équivalence.

Théorème 217.

Si l'on considère toutes les bis simulations qui lient les états Q d'un même modèle $\mathcal{M}$, alors la plus large bis simulation $\approx$ est une relation d'équivalence sur Q .

Démonstration : Il est facile de montrer les trois points suivants :

1. $\{(q, q) \mid q \in Q\}$ est une bisimulation ;
2. si B est une bisimulation sur $\mathcal{M}$, alors B^{-1} est aussi une bisimulation sur $\mathcal{M}$;
3. si B et B' sont deux bisimulations sur $\mathcal{M}$, alors $B \cdot B'$ est une bisimulation sur $\mathcal{M}$. ■

On peut alors définir le modèle quotient. Ce dernier se définit comme le modèle le plus fin de la section 26.3. Ainsi, étant donné un modèle $\mathcal{M} = (Q, R, \nu)$, on définit le modèle quotient $\mathcal{M}_{\approx} = (Q', R', \nu')$ de la façon suivante :

- $Q' = Q /_{\approx}$;
- pour tout $q \in Q$, $\lambda'([q]_{\approx}) = \lambda(q)$ (ceci est correcte car pour tout $q \approx q'$, $\lambda(q) = \lambda(q')$)
- $([q]_{\approx}, [q']_{\approx}) \in R' \iff \exists q_1 \in [q]_{\approx}, \exists q'_1 \in [q']_{\approx}, (q_1, q'_1) \in R$

Étant donné un modèle de Kripke fini $\mathcal{M}$, on peut calculer la relation $\approx$. En effet, on peut donner une autre définition des bisimulations à partir de la notion de fonction monotone dans les treillis complets. Étant donné un modèle de Kripke $\mathcal{M} = (Q, R, \nu)$ sur une signature P , le treillis que nous allons considérer est $(2^S, \subseteq)$ où $S = \{(q, q') \in Q \times Q \mid \forall p \in P, q \in \nu(p) \iff q' \in \nu(p)\}$. On a vu que ce treillis est complet.

Définition 171.

Soit $B \subseteq S$ (B est une relation binaire sur Q). Notons $\mathcal{F}(B)$ l'ensemble défini par :

- $(q, q') \in \mathcal{F}(B)$ ssi
- $\forall q_1 \in Q, (q, q_1) \in R \implies (\exists q'_1 \in Q, (q', q'_1) \in R \wedge (q_1, q'_1) \in B)$
 - $\forall q'_1 \in Q, (q', q'_1) \in R \implies (\exists q_1 \in Q, (q, q_1) \in R \wedge (q_1, q'_1) \in B)$

Il est facile de montrer que la fonction $\mathcal{F}$ est monotone. On peut alors appliquer notre premier théorème de point fixe et poser $\approx = \bigcup \{B \in 2^S \mid B \subseteq \mathcal{F}(B)\}$. On peut aussi montrer que $\mathcal{F}$ est inf-continue (i.e. elle préserve les bornes inférieures de n'importe quel sous-ensemble de 2^S), et donc par le second théorème des points fixe, on a : $\exists n \in \mathbb{N}, \approx = \mathcal{F}^n(S)$. Ce dernier résultat donne lieu à l'algorithme suivant :

- *En entrée.* Un modèle de Kripke fini $\mathcal{M} = (Q, R, \nu)$ sur une signature P elle-même finie.
- *Problème.* Trouver le modèle de Kripke qui est obtenu en remplaçant chaque état $q \in Q$ dans $\mathcal{M}$ par sa classe d'équivalence $[q]_{\approx}$.
- On commence par partitionner l'ensemble des états en regroupant tous les états q et q' tels que $\forall p \in P, q \in \nu(p) \iff q' \in \nu(p)$.
- Étant donnée une partition Π des états du modèle de départ, pour chaque classe d'équivalence $[q]_{\Pi}$ dans Π , on partitionne la classe $[q]_{\Pi}$ telle que deux états q_1 et q_2 sont dans la même sous-classe si, et seulement s'il existe un état $q'_1 \in [q']_{\Pi}$ tel que $(q_1, q'_1) \in R$ alors il existe $q'_2 \in [q']_{\Pi}$ tel que $(q_2, q'_2) \in R$, et inversement. On note Π' la nouvelle partition obtenue.
- Si $\Pi = \Pi'$ alors le processus s'arrête et retourne Π , sinon on pose $\Pi := \Pi'$ et on recommence.

Théorème 218.

La partition Π calculée par l'algorithme ci-dessus est la plus grande bisimulation sur $\mathcal{M}$.

Démonstration : L'ensemble des états du modèle étant fini, l'algorithme ci-dessus termine. Montrons alors que la partition Π obtenue est la plus grande bissimulation. Montrons tout d'abord que c'est une bissimulation. Supposons alors le contraire, c'est-à-dire, supposons qu'il existe $q, q' \in \Pi_1$, Π_1 étant une partition de Π , tel qu'il existe $q_1 \in \Pi_2$, Π_2 étant aussi une partition de Π , avec $(q, q_1) \in R$, mais :

1. soit il n'existe pas de $q_2 \in \Pi_2$ tel que $(q', q_2) \in R$,
2. ou il existe un état $q_2 \in Q$ tel que $(q', q_2) \in R$ mais $q_2 \notin \Pi_2$.

Dans les deux cas, par l'algorithme ci-dessus, la partition aurait subi un découpage tel que q et q' appartiendraient à deux sous-partitions différentes.

Le fait que la dernière partition Π obtenue par l'algorithme est la plus grande bissimulation, est une conséquence directe du fait que l'algorithme implante le calcul du plus grand point fixe. ■

Logique modale et bissimulation

Théorème de Hennessy-Milner

Nous avons défini un type de relation (les bissimulations) entre les modèles de Kripke dont l'intérêt quand elles sont définies sur un même modèle est de permettre de réduire le nombre d'états et donc la complexité des algorithmes de vérification de propriétés. Bien entendu, ceci n'a de sens que si deux modèles bissimilaires sont équivalents en pouvoir de preuve, c'est-à-dire que l'ensemble des propriétés validées par chacun des modèles est identique. On dit qu'ils sont *logiquement équivalents*. Dans cette section, on va tout d'abord montrer que deux modèles de Kripke $\mathcal{M}$ et $\mathcal{M}'$ bissimilaires pour deux états $q \in Q$ et $q' \in Q'$ donnés, satisfont le même ensemble de formules modales pour ces deux états q et q' . Formellement, ceci se traduit par le résultat suivant :

Théorème 219.

Soit $\mathcal{M} = (Q, R, \nu)$ et $\mathcal{M}' = (Q', R', \nu')$ deux modèles de Kripke sur une même signature P . Soit $q \in Q$ et $q' \in Q'$ deux états tels que $q \approx q'$. Alors, pour toute formule $\varphi \in \text{ForMod}(P)$, on a :

$$\mathcal{M} \models_q \varphi \iff \mathcal{M}' \models_{q'} \varphi$$

Démonstration Ceci se prouve par induction structurelle sur la structure de la formule φ . □

La question que l'on peut se poser maintenant est de savoir si l'inverse est vrai. La réponse est non dans le cas général. Cependant, le résultat n'est pas complètement négatif car nous pouvons montrer que dans le cas restreint des modèles de Kripke à *image finie* (ce qui est souvent le cas en pratique pour tous les modèles représentant un système réactif réel), l'inverse est vrai. Commençons d'abord par définir ce que nous entendons par modèle à *image finie*.

Définition 172 (Modèle à image finie).

Un modèle de Kripke $\mathcal{M} = (Q, R, \nu)$ est à **image finie** si pour état $q \in Q$, l'ensemble $\{q' \in Q \mid (q, q') \in R\}$ est fini.

Dans ce cas-là, on a le résultat recherché, c'est-à-dire que deux modèles logiquement équivalents pour deux états donnés q et q' font que q et q' sont bissimilaires.

Théorème 220.

Soit $\mathcal{M} = (Q, R, \nu)$ et $\mathcal{M}' = (Q', R', \nu')$ deux modèles de Kripke à image finie sur une même signature P . Soit $q \in Q$ et $q' \in Q'$ deux états. Si pour toute formule φ , on a :

$$\mathcal{M} \models_q \varphi \Leftrightarrow \mathcal{M}' \models_{q'} \varphi$$

alors, $q \approx q'$.

Démonstration : Pour cela, nous allons démontrer que la relation $\equiv \subseteq Q \times Q'$ définie par :

$$q \equiv q' \iff (\forall \varphi \in \text{Formod}(P), \mathcal{M} \models_q \varphi \Leftrightarrow \mathcal{M}' \models_{q'} \varphi)$$

est elle-même une bissimulation et donc est incluse dans $\approx$. Supposons alors $q \equiv q'$. Par hypothèse, on a trivialement que pour tout $p \in P$, $q \in \nu(p) \Leftrightarrow q' \in \nu'(p)$. Montrons alors la condition **Zig**. Supposons un indice $q_1 \in Q$ tel que $(q, q_1) \in R$. Nous devons montrer qu'il existe un indice $q'_1 \in Q'$ tel que $(q', q'_1) \in R'$ et $q_1 \equiv q'_1$. Supposons le contraire, i.e. il n'existe pas un tel q'_1 . Par le fait qu' $(q, q_1) \in R$, on a $\mathcal{M} \models_q \langle a \rangle \text{ true}$. Par hypothèse, on a alors aussi $\mathcal{M}' \models_{q'} \langle a \rangle \text{ true}$. Ainsi, l'ensemble $U = \{q'_1 \mid (q', q'_1) \in R'\}$ n'est pas vide. Puisque $\mathcal{M}'$ est à image finie, l'ensemble U est de cardinalité finie. Maintenant, pour avoir supposé le contraire, pour tout $q'_1 \in U$, il doit exister une formule ψ telle que $\mathcal{M} \models_{q_1} \psi$ mais $\mathcal{M}' \not\models_{q'_1} \psi$. De là, on peut déduire alors que : $\mathcal{M} \models_q \langle a \rangle (\psi_1 \wedge \dots \wedge \psi_n)$ et $\mathcal{M}' \not\models_{q'} \langle a \rangle (\psi_1 \wedge \dots \wedge \psi_n)$ où n est la cardinalité de U . Mais ceci contredit le fait que $q \equiv q'$. Pour montrer la condition **Zag**, nous utilisons le même raisonnement mais dans l'autre sens, utilisant le fait que $\mathcal{M}$ et $\mathcal{M}'$ sont à image finie. ■

On voit bien dans la preuve ci-dessus l'utilité pour les modèles d'être à image finie. En effet, sans cette condition nous n'aurions jamais pu construire la formule $\langle a \rangle (\psi_1 \wedge \dots \wedge \psi_n)$ car l'ensemble des ψ_i aurait été de cardinalité infinie. Or, les formules modales sont toutes de taille finie. Il existe une généralisation du résultat ci-dessus mais qui justement demande de manipuler des formules de taille infinie, difficilement exploitable en pratique.

Théorème de Van Benthem

Nous avons vu que la logique modale est un fragment de la logique des prédicats. Est-ce que ce fragment peut-être caractérisé ? ou plus précisément, quelles sont les formules des prédicats qui sont équivalentes à la traduction des formules modales ? On peut montrer que la logique modale modulo la traduction donnée en section 26.4 correspond exactement au fragment de la logique des prédicats invariant par bissimulation. Définissons cette notion d'invariance par bissimulation.

Définition 173 (Invariance par bisimulation).

Soit $\varphi \in \text{For}(\Sigma)$ une formule de la logique des prédicats avec exactement une unique variable libre x (Σ est la signature du 1er ordre définie dans la section 26.4). φ est dite **invariante par bisimulation** si pour chaque paire de modèles de Kripke $\mathcal{M} = (Q, R, \nu)$ et $\mathcal{M}' = (Q', R', \nu')$, chaque paire d'états $q \in Q$ et $q' \in Q'$ tel que $q \approx q'$, on a :

$$\mathcal{M} \models_{[x \mapsto q]} \varphi \iff \mathcal{M}' \models_{[x \mapsto q']} \varphi$$

Nous avons alors le théorème suivant, connu sous le nom du théorème de Van Benthem du nom de son inventeur.

Théorème 221 (Théorème de Van Benthem).

Soit $\varphi \in \text{For}(\Sigma)$ une formule de la logique des prédicats avec exactement une unique variable libre x . Les deux énoncés suivants sont équivalents :

1. φ est logiquement équivalente à $\Theta_x(\psi)$ pour quelques formules modales ψ .
2. φ est invariante par bisimulation.

Démonstration : Le sens 1. $\Rightarrow$ 2. est le théorème 219.

Pour montrer 2. $\Rightarrow$ 1., supposons que φ est invariante par bisimulation. Soit l'ensemble $\Gamma = \{\Theta_x(\psi) \mid \psi \in \text{ForMod}(P) \wedge \{\varphi\} \models \Theta_x(\psi)\}$. Plaçons nous dans la signature Σ' définie comme Σ à ceci près que maintenant x est une constante. Prouvons dans cette signature Σ' que $\Gamma \models \varphi$. Soit $\mathcal{M}$ un Σ' -modèle de Kripke tel que $\mathcal{M} \models \Gamma$. Montrons alors que $\mathcal{M} \models \varphi$ pour la signature Σ' . Soit $\Delta = \{\Theta_x(\psi) \mid \mathcal{M} \models \psi\}$. Par construction, on a $\Gamma \subseteq \Delta$. Maintenant, montrons que $\Delta \cup \{\varphi\}$ est satisfiable. Supposons le contraire. On a donc $\Delta \models \neg\varphi$, et donc par le théorème de la compacité (valable pour la logique des prédicats), il existe un sous-ensemble fini $\Delta' = \{\Theta_x(\psi_1), \dots, \Theta_x(\psi_n)\}$ de Δ tel que $\Delta' \models \neg\varphi$. Mais alors, $\varphi \models \Theta_x(\psi')$ où $\psi' = \neg\psi_1 \vee \dots \vee \neg\psi_n$, et donc $\Theta_x(\psi') \in \Gamma$ ce qui contredit le fait que $\Gamma \subseteq \Delta$. Ainsi, si l'on pose $q = x^{\mathcal{M}}$ pour la signature Σ' , on a pour la signature Σ , qu'il existe un Σ -modèle $\mathcal{M}''$ tel que $\mathcal{M}'' \models_{[x \mapsto q]} \varphi$. Donc, il existe un Σ -modèle $\mathcal{M}'$ que l'on peut supposer fini par la propriété de modèle fini tel que $\mathcal{M}' \models_{[x \mapsto x^{\mathcal{M}'}]} \Delta$ et $\mathcal{M}' \models_{[x \mapsto x^{\mathcal{M}'}]} \varphi$. Ainsi on a $x^{\mathcal{M}} \equiv x^{\mathcal{M}'}$ où $\equiv$ est la relation définie dans la preuve du théorème 220. De là, comme φ est invariante par bisimulation, on déduit que $\mathcal{M} \models \varphi$, et donc $\Gamma \models \varphi$ pour la signature Σ' . Par le théorème de la compacité pour la logique des prédicats, il existe alors un sous-ensemble de formules Ψ dans Γ tel que à la fois $\varphi \models \Psi$ et $\Psi \models \varphi$. Si $\Psi = \{\Theta_x(\psi_1), \dots, \Theta_x(\psi_m)\}$, φ est logiquement équivalente à $\Theta_x(\psi_1 \wedge \dots \wedge \psi_m)$. ■

27. Le μ -calcul

27.1. Introduction

La logique modale permet d'exprimer des propriétés locales sur les états telles que par exemple tout état n'est pas bloquant, i.e. qu'on peut toujours passer une transition :

$$\Diamond True$$

Par contre, la logique Modale ne peut pas exprimer des possibilités durables ou inévitables à long terme telles qu'il existe au moins une exécution infinie, une propriété indésirable n'arrivera jamais (propriété de sécurité), une propriété désirée doit nécessairement se produire à un moment, etc. La raison pour laquelle la logique modale ne peut pas exprimer de telles propriétés est que ceci demande de raisonner sur les exécutions et non plus simplement sur les transitions. Le problème est que les exécutions pour un modèle de Kripke sont de taille quelconque voire infinie (voir la notion de trace en section 26.5). Cependant, elles sont toujours de taille finie ou dénombrable. Un moyen pour permettre d'écrire des formules associées à de telles exécutions est d'utiliser des schémas récurrents sur les transitions. Comme on l'a vu dans la première partie de ce cours, un schéma récurrent par plus petit point fixe va nous permettre de manipuler des exécutions de taille finie, et un schéma récurrent par plus grand point fixe caractérisera des exécutions infinies. C'est ce que nous allons faire ici en introduisant une nouvelle logique très expressive, le μ -calcul. L'idée du μ -calcul est de pouvoir quantifier sur des parties de l'ensemble des états Q pour permettre le calcul du plus petit ou du plus grand point fixe. En effet, supposons le système très simple



Une propriété que vérifie le modèle ci-dessus est qu'il peut effectuer une transition sur q éternellement. Mais comment écrire une telle propriété? Sémantiquement, nous voulons que cette propriété φ vérifie l'équation suivante :

$$\llbracket \mathcal{M} \rrbracket(\varphi) = \Diamond_{\mathcal{M}}(\llbracket \mathcal{M} \rrbracket(\varphi))$$

où $\Diamond_{\mathcal{M}} : 2^Q \rightarrow 2^Q$ est l'application définie par : $X \mapsto \{q \in Q \mid \exists q' \in Q, (q, q') \in R \wedge q' \in X\}$. De façon similaire, on définit l'application $\Box_{\mathcal{M}} : 2^Q \rightarrow 2^Q$ par : $X \mapsto \{q \in Q \mid \forall q' \in Q, (q, q') \in R \Rightarrow q' \in X\}$.

$\llbracket \mathcal{M} \rrbracket(\varphi)$ est alors un point fixe de $\Diamond_{\mathcal{M}}$. L'équation se divise en deux inclusions :

1. $\Diamond_{\mathcal{M}}(\llbracket \mathcal{M} \rrbracket(\varphi)) \subseteq \llbracket \mathcal{M} \rrbracket(\varphi)$
2. $\llbracket \mathcal{M} \rrbracket(\varphi) \subseteq \Diamond_{\mathcal{M}}(\llbracket \mathcal{M} \rrbracket(\varphi))$

Toute solution X à l'équation ci-dessus devra alors satisfaire :

1. si $q' \in X$ et $(q, q') \in R$, alors $q \in X$;

2. si $q \in X$, alors il existe $q' \in X$ tel que (q, q') .

où R est la relation d'accessibilité de $\mathcal{M}$.

Triviallement, X peut être soit $\emptyset$, soit $\{q\}$ qui sont le plus petit et le plus grand point fixe de $\Diamond_{\mathcal{M}}$. Bien sûr, ici c'est le plus grand point fixe qui nous intéresse, le plus petit point fixe dans l'exemple donné calculerait les états à partir duquel on ne peut mener que des exécutions finies en suivant la transition.

On a vu dans la première partie de ce cours que ces plus petit et plus grand points fixes sont calculables quand les fonctions concernées sont monotones (ici $\Diamond_{\mathcal{M}}$). La proposition ci-dessous nous assure que tous les connecteurs des logiques modales à l'exception de la négation et de l'implication, sont monotones pour l'inclusion.

Proposition 222.

Soient $\mathcal{M} = (Q, R, \nu)$ un système. Soient $X, Y \subseteq Q$. Si $X \subseteq Y$, alors :

- $Z \cap X \subseteq Z \cap Y$
- $Z \cup X \subseteq Z \cup Y$
- $\Box_{\mathcal{M}}(X) \subseteq \Box_{\mathcal{M}}(Y)$
- $\Diamond_{\mathcal{M}}(X) \subseteq \Diamond_{\mathcal{M}}(Y)$

Il est facile de voir que la négation n'est pas monotone. En effet, quand $X \subseteq Y$, on a $Q \setminus Y \subseteq Q \setminus X$. On déduit alors la non monotonie de l'implication à partir de l'équivalence $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$.

27.2. Syntaxe et sémantique

On ajoute alors à la logique modale, deux quantificateurs nouveaux, μ et ν , permettant de calculer respectivement les plus petit et plus grand point fixe. Ces nouveaux quantificateurs vont porter sur des variables, appelées *variables de point fixe*, dont l'interprétation seront des parties de l'ensemble des états (i.e. des éléments de 2^Q). On a alors la définition suivante :

Définition 174 (Syntaxe du μ -calcul).

Soit P un ensemble de variables propositionnelles et X un ensemble de variables dites de point fixe. Notons $\mathcal{MC}(P, X)$ l'ensemble défini inductivement par :

- $P \subset \mathcal{MC}(P, X)$;
- $X \subseteq \mathcal{MC}(P, X)$;
- si $\varphi, \psi \in \mathcal{MC}(P, X)$, alors $\varphi @ \psi \in \mathcal{MC}(P, X)$ où $@ \in \{\wedge, \vee, \Rightarrow\}$;
- si $\varphi \in \mathcal{MC}(P, X)$, alors $\neg\varphi \in \mathcal{MC}(P, X)$;
- si $\varphi \in \mathcal{MC}(P, X)$, alors $\Box\varphi \in \mathcal{MC}(P, X)$;
- si $\varphi \in \mathcal{MC}(P, X)$ et $x \in X$, alors $@x.\varphi \in \mathcal{MC}(P, X)$ où $@ \in \{\mu, \nu\}$ et tel que la variable x apparait positivement dans φ (on suppose que toutes les occurrences de la variable x sont libres dans φ , i.e. elles ne sont pas dans la portée d'un des quantificateurs μ ou ν) i.e.
 - si φ est *true* ou $p \in P$, x est **positif** dans φ ;
 - si $\varphi = y$ avec $y \in X$, alors x est **positif** dans φ ssi $x = y$;
 - si $\varphi = \varphi_1 \wedge \varphi_2$ ou $\varphi = \varphi_1 \vee \varphi_2$, alors x est **positif** dans φ ssi x est positif à la fois dans φ_1 et φ_2 ;
 - si $\varphi = \varphi_1 \Rightarrow \varphi_2$, alors x est **positif** dans φ ssi x n'est pas positif dans φ_1 ou x est positif dans φ_2 ;
 - si $\varphi = \neg\varphi'$ alors x est **positif** dans φ ssi x n'est pas positif dans φ' ;
 - si $\varphi = @y.\varphi'$ avec $@ \in \{\mu, \nu\}$ ($y \neq x$ par hypothèse), alors x est **positif** dans φ ssi x est positif dans φ' .

On dira qu'une formule est **close** quand toutes les variables de point fixe sont dans la portée d'un quantificateur μ ou ν .

Là encore, on peut définir la sémantique du μ -calcul de deux façons, en définissant la relation $\models$ entre les modèles de Kripke et les formules ou par un calcul des ensembles d'états. Comme précédemment, nous allons donner les deux définitions ici. Ce qui va changer avec la définition 164 est l'introduction d'une nouvelle donnée, l'interprétation des variables de point fixe. Cette interprétation est définie par une application $\delta : X \rightarrow 2^Q$ où Q est l'ensemble des états du modèle de Kripke.

Définition 175 (Sémantique).

Soit $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke sur P . Soit $\varphi \in \mathcal{MC}(P, X)$ une formule. Soit $\delta : X \rightarrow 2^Q$ une interprétation des variables de point fixe. On dit que $\mathcal{M}$ **satisfait** pour $q \in Q$ et δ la formule φ , noté $\mathcal{M} \models_{q, \delta} \varphi$, si, et seulement si : on notera $\delta[Q'/x]$ l'interprétation $\delta' : X \rightarrow 2^Q$ qui à x associe l'ensemble d'états Q' et pour tout $y \neq x$ associe $\delta(y)$

- si $\varphi = x \in X$, alors $\mathcal{M} \models_{q, \delta} \varphi$ ssi $q \in \delta(x)$;
- si $\varphi = \nu x. \varphi'$, alors $\mathcal{M} \models_{q, \delta} \varphi$ ssi il existe $Q' \subseteq Q$ tel que $q \in Q'$ et pour tout $q' \in Q'$, $\mathcal{M} \models_{q', \delta[Q'/x]} \varphi'$;
- si $\varphi = \mu x. \varphi'$, alors $\mathcal{M} \models_{q, \delta} \varphi$ ssi pour tout $Q' \subseteq Q$, si $\{q' \in Q \mid \mathcal{M} \models_{q', \delta[Q'/x]} \varphi'\} \subseteq Q'$ alors $q \in Q'$;
- les autres types de formules sont traitées de façon usuelle.

On dit que $\mathcal{M}$ **valide** φ , noté $\mathcal{M} \models \varphi$, si, et seulement si pour tout $q \in Q$ et tout $\delta : X \rightarrow \mathcal{P}(X)$, $\mathcal{M} \models_{q, \delta} \varphi$.

Remarquons que pour toute formule close φ , $\mathcal{M} \models \varphi \iff \forall q \in Q, \mathcal{M} \models_{q, \emptyset} \varphi$ où $\emptyset : X \rightarrow \mathcal{P}(X)$ est l'application qui à tout $x \in X$ associe l'ensemble vide $\emptyset$.

Proposition 223.

μ et ν sont duales, i.e. que l'équivalence suivante est satisfaite :

$$\neg \nu x. \varphi \equiv \mu x. \neg \varphi'$$

où φ' est la formule obtenue à partir de φ en substituant toutes les occurrences libres de x par $\neg x$.

La définition calculatoire de la sémantique est alors donnée par une famille d'applications $\llbracket \mathcal{M} \rrbracket_\delta : \mathcal{MC}(P, X) \rightarrow 2^Q$ indexée par les interprétations $\delta : X \rightarrow 2^Q$ dont chacune se définit par induction structurelle sur les formules de la façon suivante :

- $\llbracket \mathcal{M} \rrbracket_\delta(x) = \{q \in Q \mid q \in \delta(x)\}$;
- $\llbracket \mathcal{M} \rrbracket_\delta(\nu x. \varphi) = \bigcup \{Q' \subseteq Q \mid Q' \subseteq \llbracket \mathcal{M} \rrbracket_{\delta[Q'/x]}(\varphi)\}$;
- $\llbracket \mathcal{M} \rrbracket_\delta(\mu x. \varphi) = \bigcap \{Q' \subseteq Q \mid \llbracket \mathcal{M} \rrbracket_{\delta[Q'/x]}(\varphi) \subseteq Q'\}$.

27.3. Quelques exemples de formules à point fixe

La logique modale ne permet pas d'exprimer des propriétés fondamentales sur les systèmes telles que :

- **Accessibilité.** *Une certaine situation peut être atteinte*, e.g.
 - le compteur x peut prendre la valeur 0 (i.e. il existe un état atteignable où $x = 0$) ;
 - le point final du programme peut être atteint.
- **Sureté.** *Quelque chose de mauvais n'arrive jamais*, e.g.
 - chaque fois que j'utilise un *unlock*, j'ai utilisé un *lock* avant ;
 - Chaque fois que j'accède à mon compte, j'ai entré le bon code au préalable ;

- Quand la pré-condition du programme est respectée et que le programme termine alors la post-condition est respectée.
- **Vivacité.** *Quelque chose de bon finira par arriver*, e.g.
 - Quand une impression est lancée, elle finira par s'achever ;
 - Quand un message est envoyé, il finira par être reçu ;
 - Quand la pré-condition du programme est respectée, alors le programme termine et la post-condition est respectée.
- **Équité.** *Quelque chose se répètera infiniment souvent*, e.g., si un processus demande son exécution, il finira par l'avoir.
- **Absence de blocage.** *Le système ne se bloque pas*, e.g.,
 - (non blocage total) Il existe au moins une exécution infinie du système ;
 - (non blocage partiel) Il n'existe pas d'état bloquant.

Le μ -calcul par l'introduction de schémas récurrents au travers des quantificateurs μ et ν permet d'exprimer les propriétés ci-dessus.

- **Accessibilité.** Si nous notons φ la situation que nous souhaiterions atteindre, l'accessibilité s'exprime alors par la formule :

$$\mu x.(\varphi \vee \Diamond x)$$

Cette formule calcule l'ensemble des exécutions finies dont le dernier état vérifie la formule φ .

- **Sureté.** Soit φ la formule dont on veut qu'elle n'arrive jamais. La sureté s'exprime alors :

$$\nu x.(\neg \varphi \wedge \Diamond x)$$

Cette formule calcule l'ensemble des exécutions dont tous les états vérifient $\neg \varphi$.

- **Vivacité.** La vivacité est le dual de la sureté.
- **Absence de blocage.** L'absence de blocage s'exprime par le fait que toutes les exécutions peuvent être continuées par application d'une action. Ceci s'exprime par la formule suivante :
 - **non blocage total.** $\nu x. \Diamond x$
 - **non blocage partiel.** $\nu x.(\Diamond true \wedge \Box x)$

28. Calculs pour la logique modale

Comme pour les deux logiques précédentes, les logiques modales normales ont des calculs associés. Comme pour la logique des prédicats, la logique étant une extension de la logique propositionnelle, il est naturel d'étendre les différents calculs présentés pour la logique propositionnelle. Nous étendrons ici les trois calculs suivants :

- Calcul à la Hilbert ;
- Calcul des séquents ;
- Méthode des tableaux.

28.1. Calcul à la Hilbert

Comme précédemment pour la logique propositionnelle, pour rendre plus concis la présentation du calcul associé à la logique des prédicats du premier ordre, nous allons d'abord restreindre l'ensemble des formules aux connecteurs $\neg$ et $\Rightarrow$ et à la modalité $\Box$. Ainsi, par des transformations sémantiquement équivalentes, on peut supprimer les connecteurs $\wedge$ et $\vee$ et la modalité $\Diamond$. On a déjà donné ces transformations pour l'élimination des connecteurs $\wedge$ et $\vee$. Donnons alors juste celle concernant la suppression de la modalité $\Diamond$.

$\Diamond\varphi$ est sémantiquement équivalent à $\neg\Box\neg\varphi$

Pour définir la calcul à la Hilbert pour les logiques modales normale, nous modifions la règle de synthèse et ajoutons les règles suivantes :

Définition 176.

Une formule modale $\varphi \in ForMod(P)$ est un **théorème** d'un ensemble formules modales $\Gamma \subseteq ForMod(P)$, noté $\Gamma \vdash \varphi$, si et seulement si on peut obtenir l'énoncé $\Gamma \vdash \varphi$ en utilisant un nombre fini de fois les règles de la logique propositionnelle (modulo la modification de la règle de synthèse) auxquelles on ajoute les règles suivantes :

- **Synthèse.** Si $\Gamma \vdash \varphi \Rightarrow \psi$, alors $\Gamma \cup \{\varphi\} \vdash \psi$;
- **Normalité.** Si $\Gamma \vdash \Box(\varphi \Rightarrow \psi)$, alors $\Gamma \vdash \Box\varphi \Rightarrow \Box\psi$;
- **Nécessité.** Si $\Gamma \vdash \varphi$, alors $\Gamma \vdash \Box\varphi$.

Ce calcul définit la logique K . Pour définir les autres logiques modales D, T, B, S4 et S5, il faut ajouter les axiomes adéquates. Par exemple, pour la logique modale T, il faudrait ajouter la règle : Si $\Gamma \vdash \Box\varphi$, alors $\Gamma \vdash \varphi$.

Théorème 224 (Correction).

$$\Gamma \vdash \varphi \Longrightarrow \Gamma \models \varphi$$

Démonstration : Ceci se prouve par induction sur la structure des preuves. Nous allons simplement démontrer cette propriété de correction pour les trois règles ci-dessus.

- *Synthèse.* Supposons que $\Gamma \models \varphi \Rightarrow \psi$. Soit $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke qui valide l'ensemble $\Gamma \cup \{\varphi\}$. Soit $q \in Q$ un état. Montrons que $\mathcal{M} \models_q \psi$. Par hypothèse, $\mathcal{M} \models_q \varphi$, et donc par hypothèse de récurrence, on a aussi que $\mathcal{M} \models_q \psi$.
- *Normalité.* Supposons que $\Gamma \models \Box(\varphi \Rightarrow \psi)$. Soit $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke qui valide l'ensemble des formules de Γ . Soit q un état de $\mathcal{M}$ tel que $\mathcal{M} \models_q \Box\varphi$. Ceci signifie que pour tout état $q' \in Q$ tel que $(q, q') \in R$, on a $\mathcal{M} \models_{q'} \varphi$. Maintenant par hypothèse de récurrence, $\mathcal{M} \models_{q'} \Box(\varphi \Rightarrow \psi)$, et donc $\mathcal{M} \models_{q'} \varphi \Rightarrow \psi$, d'où l'on peut déduire que $\mathcal{M} \models_{q'} \psi$. Ceci nous permet alors de conclure que $\mathcal{M} \models_q \Box\psi$.
- *Nécessité.* Supposons que $\Gamma \models \varphi$. Soit $\mathcal{M} = (Q, R, \nu)$ un modèle de Kripke qui valide l'ensemble des formules Γ . Par hypothèse, $\mathcal{M} \models \varphi$. Soit $q \in Q$ un état. Par hypothèse pour tout $q' \in Q$ tel que $(q, q') \in R$, on a $\mathcal{M} \models_{q'} \varphi$, ce qui nous permet de conclure que $\mathcal{M} \models_q \Box\varphi$. ■

L'inverse de la règle de synthèse dont l'énoncé est la suivant :

$$\Gamma \cup \{\varphi\} \vdash \psi \implies \Gamma \vdash \varphi \Rightarrow \psi$$

n'est pas valide en logique modale. Si elle était, elle permettrait, associée à la règle de nécessité, de montrer que la formule $\varphi \Rightarrow \Box\varphi$ est un théorème, ce qui bien entendu n'est pas vrai. Ainsi, le théorème de la déduction qui est valide pour la logique propositionnelle et la logique des prédicats ne l'est plus pour la logique modale. Cependant, pour démontrer la complétude du calcul, nous verrons que nous aurons besoin de cette propriété. Comment alors la réobtenir ? Pour répondre à cette question, nous devons restreindre la notion d'inférence de la façon suivante :

Définition 177 (Dérivation locale).

Soit $\Gamma \subseteq \text{Formod}(P)$ un ensemble de formules modales, soit $\varphi \in \text{Formod}(P)$ une formule modale. $\Gamma \vdash \varphi$ si, et seulement s'il existe un ensemble fini $\{\varphi_1, \dots, \varphi_n\} \subseteq \Gamma$ tel que $\vdash \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi$.

Proposition 225 (Théorème de la déduction).

$$\Gamma \cup \{\varphi\} \vdash \psi \iff \Gamma \vdash \varphi \Rightarrow \psi$$

Démonstration : Le sens $\Leftarrow$ est juste une application de la règle de synthèse. Montrons le sens $\Rightarrow$. Supposons alors que $\Gamma \cup \{\varphi\} \vdash \psi$. Par définition de l'inférence, ceci signifie qu'il existe $\{\varphi_1, \dots, \varphi_n\} \subseteq \Gamma$ tel que $\vdash \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi$. Deux cas doivent être considérés :

1. Il existe i , $1 \leq i \leq n$, tel que $\varphi_i = \varphi$. Il est facile de montrer en logique propositionnelle que $(p \Rightarrow q \Rightarrow r) \Leftrightarrow q \Rightarrow p \Rightarrow r$. Ainsi, on a aussi $\vdash \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi \Rightarrow \psi$. En appliquant la règle du Modus Ponens, on a alors $\Gamma \vdash \varphi \Rightarrow \psi$.
2. Pour tout i , $1 \leq i \leq n$, $\varphi_i \neq \varphi$. Par monotonie de l'implication, on peut écrire $\vdash \varphi \Rightarrow \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \psi$, et donc pour les mêmes raisons que précédemment, on a aussi $\vdash \varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \varphi \Rightarrow \psi$, d'où nous pouvons conclure que $\Gamma \vdash \varphi \Rightarrow \psi$ par Modus Ponens. ■

À partir de cette définition de la dérivation locale, nous retrouvons l'ensemble les autres propriétés standards associées à la preuve de théorèmes, la déductibilité ou encore la consistance.

Proposition 226.

1. $\vdash \varphi$ ssi pour tout $\Gamma \subseteq \text{ForMod}(P)$, $\Gamma \vdash \varphi$.
2. Si $\Gamma \vdash \Gamma'$ et $\Gamma' \vdash \varphi$, alors $\Gamma \vdash \varphi$.
3. Si $\varphi \in \Gamma$, alors $\Gamma \vdash \varphi$.
4. Si $\Gamma \vdash \varphi$ et $\Gamma \subseteq \Delta$, alors $\Delta \vdash \varphi$.
5. $\Gamma \vdash \varphi$ ssi il existe un sous-ensemble fini Δ de Γ tel que $\Delta \vdash \varphi$.
6. Γ est consistant^a ssi il n'existe aucune formule $\varphi \in \text{ForMod}(P)$ telle que nous ayons à la fois $\Gamma \vdash \varphi$ et $\Gamma \vdash \neg\varphi$.
7. $\Gamma \cup \{\varphi\}$ est consistant ssi $\Gamma \not\vdash \neg\varphi$.
8. $\Gamma \vdash \varphi$ ssi $\Gamma \cup \{\neg\varphi\}$ est consistant.

^a. Nous rappelons qu'un ensemble de formules est consistant quand il existe une formule φ telle que $\Gamma \not\vdash \varphi$.

Pour la complétude du calcul, nous allons tout d'abord démontrer le résultat indépendamment d'un ensemble de formules Γ , c'est-à-dire que nous allons démontrer :

$$\vdash \varphi \implies \models \varphi$$

Ce résultat de complétude est aussi appelé *complétude faible*. La *complétude forte* considère un ensemble de formules Γ .

La preuve de complétude que nous présentons ici suit la méthode de Henkin que nous avons déjà vu dans la preuve de complétude pour la logique des prédicats. Nous allons donc construire un modèle dont l'ensemble des conséquences sémantiques est identique aux tautologies que l'on peut démontrer par le calcul.

Définition 178 (Modèle canonique).

Soit P une signature. Le **modèle canonique** sur P est le modèle de Kripke $\mathcal{M}_c = (Q_c, R_c, \nu_c)$ défini par :

- Q_c est l'ensemble de tous les ensembles Γ de formules syntaxiquement complet^a
- $R_c \subseteq Q_c \times Q_c$ est la relation binaire définie par :

$$\forall \Gamma, \Gamma' \in Q_c, (\Gamma, \Gamma') \in R_c \iff (\forall \varphi \in \text{ForMod}(P), \Box\varphi \in \Gamma \Rightarrow \varphi \in \Gamma')$$

- $\forall p \in P, \nu_c(p) = \{\Gamma \in Q_c \mid p \in \Gamma\}$.

^a. Nous rappelons qu'un ensemble $\Gamma \subseteq \text{ForMod}(P)$ est syntaxiquement complet, s'il est consistant et il n'existe pas d'ensemble consistant de formules qui contienne Γ , et donc pour chaque formule $\varphi \in \text{ForMod}(P)$, on a soit $\varphi \in \Gamma$, soit $\neg\varphi \in \Gamma$.

Proposition 227 (Lemme de Lindenbaum).

Soit $\Gamma \subseteq \text{ForMod}(P)$ un ensemble consistant de formules modales. Il existe un ensemble syntaxiquement complet de formules $\bar{\Gamma} \subseteq \text{ForMod}(P)$ qui contient Γ .

Démonstration : Soit $S = \{\Gamma' \subseteq \text{ForMod}(P) \mid \Gamma' \text{ est consistant et } \Gamma \subseteq \Gamma'\}$. L'ensemble $(S, \subseteq)$ est trivialement inductif. Donc, par le lemme de Zorn, S a un élément maximal $\bar{\Gamma}$. Par définition de S , $\bar{\Gamma}$ est consistant et contient Γ . De plus, il est syntaxiquement complet. Sinon, il existe une formule $\varphi \in \text{ForMod}(P)$ tel $\varphi \notin \bar{\Gamma}$. Comme $\bar{\Gamma}$ est maximal, ceci signifie que $\bar{\Gamma} \cup \{\varphi\}$ est inconsistent, et donc $\bar{\Gamma} \cup \{\neg\varphi\}$ est consistant. Comme $\bar{\Gamma}$ est maximal, nous pouvons conclure que $\neg\varphi \in \bar{\Gamma}$. ■

un résultat identique avait déjà été démontré dans la preuve de complétude du calcul à la Hilbert pour la logique des prédicats.

Lemme 228.

$$\Box\varphi \in \Gamma \iff \forall \Gamma' \in Q_c, (\Gamma, \Gamma') \in R_c \Rightarrow \varphi \in \Gamma'$$

Démonstration : Le sens $\Rightarrow$ est une conséquence directe de la définition de R_c . Pour montrer le sens $\Leftarrow$, supposons que $\Box\varphi \notin \Gamma$. Nous avons alors besoin de montrer qu'il existe $\Gamma' \in Q_c$ tel que $(\Gamma, \Gamma') \in R_c$ et $\varphi \notin \Gamma'$. Nous avons alors les équivalences suivantes :

$$\begin{aligned} \exists \Gamma' \in Q_c, (\Gamma, \Gamma') \in R_c \wedge \varphi \notin \Gamma' &\iff \exists \Gamma' \in Q_c, (\Gamma, \Gamma') \in R_c \wedge \neg\varphi \in \Gamma' \\ &\iff \exists \Gamma' \in Q_c, \{\psi \mid \Box\psi \in \Gamma\} \subseteq \Gamma' \wedge \neg\varphi \in \Gamma' \\ &\iff \exists \Gamma' \in Q_c, \{\psi \mid \Box\psi \in \Gamma\} \cup \{\neg\varphi\} \subseteq \Gamma' \end{aligned}$$

Par le lemme de Lindenbaum, il suffit de montrer que $\{\psi \mid \Box\psi \in \Gamma\} \cup \{\neg\varphi\}$ est un ensemble consistant. Supposons le contraire. Ceci signifie qu'il existe un ensemble fini de formules $\psi_1, \dots, \psi_n$ tel que $\{\psi_1, \dots, \psi_n\} \vdash \varphi$ pour quelques $\Box\psi_i \in \Gamma$. Par la théorème de la déduction, nous avons alors $\vdash \psi_1 \Rightarrow \dots \Rightarrow \psi_n \Rightarrow \varphi$. Par la règle de nécessité, on a alors $\vdash \Box\psi_1 \Rightarrow \dots \Rightarrow \Box\psi_n \Rightarrow \Box\varphi$. Comme chaque $\Box\psi_i \in \Gamma$, on en déduit alors que $\Box\varphi \in \Gamma$ d'où contradiction. ■

Théorème 229.

$$\forall \Gamma \in Q_c, \forall \varphi \in \text{ForMod}(P), \mathcal{M}_c \models_{\Gamma} \varphi \iff \varphi \in \Gamma$$

Démonstration : La preuve se fait par induction sur la structure de φ . Le cas de base ainsi que celui des connecteurs propositionnels $\Rightarrow$ et $\neg$ se prouvent assez simplement. Le cas un peu compliqué est celui où φ est de la forme $\Box\psi$. Montrons cette étape de la preuve. Montrons alors : $\mathcal{M}_c \models_{\Gamma} \Box\psi \iff \Box\psi \in \Gamma$. Nous avons les étapes suivantes :

$$\begin{aligned} \mathcal{M}_c \models_{\Gamma} \Box\psi &\iff \forall \Gamma', (\Gamma, \Gamma') \in R_c \Rightarrow \mathcal{M}_c \models_{\Gamma'} \psi \\ &\iff \forall \Gamma', (\Gamma, \Gamma') \in R_c \Rightarrow \mathcal{M}_c \models_{\Gamma'} \psi \text{ (hypothèse d'induction)} \\ &\iff \Box\psi \in \Gamma \text{ (Lemme 228)} \end{aligned}$$

Corollaire 12.

$$\Gamma \vdash \varphi \iff \mathcal{M}_c \models_{\Gamma} \varphi$$

Théorème 230 (Complétude faible).

$$\models \varphi \iff \vdash \varphi$$

Démonstration : Supposons que $\not\models \varphi$. Ceci signifie que $\{\neg\varphi\}$ est consistant. Par le lemme de Lindenbaum, il existe un ensemble $\{\neg\varphi\}$ syntaxiquement complet qui contient la formule $\neg\varphi$, et donc par le théorème 229, nous avons $\mathcal{M}_c \models_{\{\neg\varphi\}} \neg\varphi$, i.e. $\mathcal{M}_c \not\models_{\{\neg\varphi\}} \varphi$, et donc $\mathcal{M}_c \not\models \varphi$. ■

De là, nous pouvons démontrer la complétude forte. Notons déjà que la complétude faible énonce que chaque formule valide est démontrable. Ainsi, si un singleton de formules $\{\varphi\}$ est consistant, la formule $\neg\varphi$ n'est pas démontrable, et donc pas valide, c'est-à-dire pas satisfaisable pour un modèle de Kripke et un état de ce modèle. La complétude forte va généraliser ce lien entre consistance et satisfaisable. On peut redéfinir la notion de complétude forte de la façon suivante :

Définition 179 (Complétude forte).

Une logique modale $\mathcal{L}$ est fortement complète si tout ensemble de formules consistant est satisfaisable.

Cette définition rejoint l'approche de K. Godel pour démontrer la complétude de la logique des prédicats.

Proposition 231.

Une logique modale $\mathcal{L}$ est fortement complète si, et seulement si pour tout ensemble de formules Γ on a :

$$\Gamma \models \varphi \implies \Gamma \vdash \varphi$$

Démonstration : Ceci vient directement des deux équivalences suivantes :

1. $\Gamma \models \varphi$ est équivalent à $\Gamma \cup \{\neg\varphi\}$ n'est pas satisfaisable.
2. $\Gamma \vdash \varphi$ est équivalent à $\Gamma \cup \{\neg\varphi\}$ est inconsistent. ■

Théorème 232 (Complétude forte).

$$\Gamma \models \varphi \implies \Gamma \vdash \varphi$$

Démonstration : Si $\Gamma \not\vdash \varphi$, alors $\Gamma \cup \{\neg\varphi\}$ est consistant. Par le lemme de Lindenbaum, il existe un ensemble $\bar{\Gamma}$ syntaxiquement complet qui contient Γ . Nous avons alors $\mathcal{M}_c \models_{\bar{\Gamma}} \neg\varphi$, et donc $\mathcal{M}_c \not\models_{\bar{\Gamma}} \varphi$. Ainsi, $\Gamma \cup \{\varphi\}$ n'est pas satisfaisable. ■

28.2. Calcul des séquents

Les logiques modales acceptent aussi un calcul des séquents. Comme pour la logique des prédicats, ces calculs sont obtenus en ajoutant des règles pour les modalités (ici, uniquement la modalité $\Box$). Les règles que nous ajoutons dépendent de la logique modale considérée. Ici, nous donnons les règles pour les logiques K, T, S4 et S5.

— **Logique K.** on ajoute la règle

$$\frac{\Gamma \vdash \varphi}{\Box\Gamma \vdash \Box\varphi}$$

— **Logique T.** on ajoute la règle

$$\frac{\Gamma, \varphi \vdash \Delta}{\Gamma, \Box\varphi \vdash \Delta}$$

— **Logique S4.** on ajoute les règles

$$\frac{\Gamma, \varphi \vdash \Delta}{\Gamma, \Box\varphi \vdash \Delta}$$

$$\frac{\Box\Gamma \vdash \varphi}{\Box\Gamma \vdash \Box\varphi}$$

— **Logique S5.** on ajoute les règles

$$\frac{\Gamma, \varphi \vdash \Delta}{\Gamma, \Box\varphi \vdash \Delta}$$

$$\frac{\Box\Gamma \vdash \Box\Delta, \varphi}{\Box\Gamma \vdash \Box\Delta, \Box\varphi}$$

Il n'est pas très difficile de montrer que chacune des règles ci-dessus est correcte. On peut aussi montrer que chacun des calculs est complet. Il suffit de montrer que chacun des calculs des séquents simule le calcul à la Hilbert associé. Pour la logique K, il est facile de montrer que la nouvelle règle simule la règle de Nécessité.

Nous avons le résultat d'élimination des coupure pour K, T et S4. Par exemple si nous considérons la logique modale K, nous avons la transformation suivante :

$$\frac{\frac{\Gamma \vdash \varphi}{\Box\Gamma \vdash \Box\varphi} \quad \frac{\Gamma, \varphi \vdash \psi}{\Box\Gamma, \Box\varphi \vdash \Box\psi}}{\Box\Gamma \vdash \Box\psi} \text{Coupure} \rightsquigarrow \frac{\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash \psi}{\Gamma \vdash \psi} \text{Coupure}}{\Box\Gamma \vdash \Box\psi}$$

L'élimination des coupures n'est pas valide pour S5. La raison est que certaines tautologies de S5 nécessitent l'utilisation de la coupure.

Proposition 233.

Le séquent $\varphi \vdash \Box \neg \Box \neg \varphi$ est prouvable dans S5, mais n'accepte pas de preuve sans coupure.

Démonstration : Nous avons bien la preuve suivante :

$$\frac{\frac{\frac{\Box \neg \varphi \vdash \Box \neg \varphi}{\vdash \Box \neg \varphi, \neg \Box \neg \varphi}}{\vdash \Box \neg \varphi, \Box \neg \Box \neg \varphi} \quad \frac{\frac{\varphi \vdash \varphi}{\neg \varphi, \varphi \vdash}}{\Box \neg \varphi, \varphi \vdash} \text{Coupure}}{\varphi \vdash \Box \neg \Box \neg \varphi}$$

Maintenant, supposons que $\varphi \vdash \Box \neg \Box \neg \varphi$ admet une preuve sans coupure. Considérons alors la dernière inférence de la preuve. En regardant les règles du calcul des séquents pour S5 (à l'exception de la règle de coupure bien sûr), nous pouvons voir que seules les règles structurelles sont possibles. En répétant ceci, nous pouvons voir que cette preuve sans coupure ne peut contenir que des séquents de la forme $\varphi \vdash, \vdash \Box \neg \Box \neg \varphi, \varphi, \dots, \varphi \vdash \Box \neg \Box \neg \varphi, \dots, \Box \neg \Box \neg \varphi$. Or ceci n'aboutit jamais à un axiome du calcul (i.e. un séquent de la forme $\Gamma, \varphi \vdash \varphi, \Delta$), ce qui est une contradiction. ■

28.3. Méthode des tableaux

À FAIRE.

29. Application : raisonnement spatial qualitatif

Les approches topologiques appliquées au raisonnement spatial qualitatif décrivent des relations entre des régions spatiales. Deux modèles ont émergé dans la littérature pour formaliser ces relations topologiques entre des entités spatiales :

1. Region Connection Calculus (RCC-8) qui décrit abstraitement des régions (dans un espace Euclidien ou dans un espace topologique) par les relations possibles qui les lient entre elles ;
2. Dimensionally Extended 9-Intersection Model (DE-9IM) est un modèle topologique utilisé pour décrire les relations entre deux régions spatiales.

29.1. RCC-8

RCC-8 est une théorie fondée sur une relation primitive de connectivité C . À partir de cette relation élémentaire C , plusieurs autres relations binaires peuvent être définies, parmi lesquelles huit ont été identifiées comme étant d'une importance particulière :

- **Déconnection** DC . $DC(X, Y)$ signifie que la région X est déconnectée de la région Y ;
- **Connexion externe** EC . $EC(X, Y)$ signifie que X est connectée de façon externe à Y ;
- **Chevauchement partiel** PO . $PO(X, Y)$ signifie que X et Y ont une intersection commune ;
- **Partie propre tangentielle (resp. inverse)** TPP (resp. TPP_i). $TPP(X, Y)$ (resp. $TPP_i(X, Y)$) signifie que X (resp. Y) est une partie tangentielle propre de Y (resp. de X) ;
- **Partie propre non-tangentielle (resp. inverse)** $NTPP$ (resp. $NTPP_i$). $NTPP(X, Y)$ (resp. $NTPP_i(X, Y)$) signifie que X (resp. Y) est une partie non-tangentielle de Y (resp. de X) ;
- **Égalité** EQ . $EQ(X, Y)$ signifie que X et Y sont des régions identiques.

Ici, étant donné un modèle de Kripke $\mathcal{M} = (Q, R, \nu)$, les éléments de Q sont des entités spatiales (i.e. des régions), et les formules sont des combinaisons de telles entités. Ce que l'on observe à la lecture des propriétés ci-dessus est que le modèle RCC-8 est un modèle qui permet de quantifier sur les entités spatiales. C'est pourquoi dans la littérature, le modèle RCC-8 a surtout été spécifié comme une théorie du premier ordre, c'est-à-dire par une théorie axiomatique de la logique des prédicats. Ici, nous proposons de formaliser ces relations dans le cadre de la logique modale. L'intérêt d'une telle formalisation est de rendre les formules plus concises. Pour permettre cette formalisation, nous devons considérer que la logique modale vérifie les deux axiomes suivants :

1. **Anti-extensivité de $\Box$** . $\Box\varphi \Rightarrow \varphi$;
2. **Extensivité de $\Diamond$** . $\varphi \Rightarrow \Diamond\varphi$;

L'intérêt de ces deux propriétés est de voir la modalité $\Box$ comme une érosion et la modalité $\Diamond$ comme une dilatation de régions¹.

Pour permettre de quantifier sur les états, nous introduisons deux nouveaux opérateurs U et E . Ainsi, nous avons deux types de formules modales supplémentaires :

1. Si $\varphi \in ForMod(P)$, alors $U\varphi \in ForMod(\varphi)$;
2. Si $\varphi \in ForMod(P)$, alors $E\varphi \in ForMod(\varphi)$;

et dont la signification sémantique est la suivante :

- $\mathcal{M} \models_q U\varphi$ iff $\forall q' \in Q, \mathcal{M} \models_{q'} \varphi$;
- $\mathcal{M} \models_q E\varphi$ iff $\exists q' \in Q, \mathcal{M} \models_{q'} \varphi$;

De là, on a la formalisation suivante des relations ci-dessus :

- $C(X, Y) : A(\varphi \wedge \psi)$;
- $DC(X, Y) : U(\neg(\varphi \wedge \psi))$;
- $EC(X, Y) : \neg(\varphi \wedge \psi)$ et $E(\Diamond\varphi \wedge \psi)$ et $E(\varphi \wedge \Diamond\psi)$;
- $PO(X, Y) : E(\varphi \cap \psi)$ et $E(\varphi \wedge \neg\psi)$ et $A(\neg\varphi \wedge \psi)$;
- $TPP(X, Y) : \varphi \Rightarrow \psi$ et $E(\Diamond\varphi \wedge \neg\psi)$;
- $NTPP(X, Y) : \varphi \Rightarrow \psi$ et $\varphi \Rightarrow \Box\psi$;
- $EQ(X, Y) : \varphi \Leftrightarrow \psi$.

où φ et ψ sont des formules qui définissent respectivement les régions X et Y . Ainsi, les formules suivantes traduisent :

- Pour EC . Les deux régions X et Y ne s'intersectent pas mais dès qu'une des deux est dilatée (par l'opérateur $\Diamond$) alors l'intersection devient non-vide.
- Pour TPP . X est inclus dans Y mais la dilatation de X (représentée par $\Diamond\varphi$) ne l'est plus.
- Pour $NTPP$. X est inclus dans Y et le reste même si l'on érode la région Y .

Les autres relations sont naturelle dans leur traduction.

29.2. Le modèle 9-intersection

Le modèle 9-intersection transforme les relations topologiques entre deux entités spatiales X et Y en un problème de topologie générale. Ainsi, les relations topologiques entre deux entités spatiales se définiront en terme d'intersection de frontières, d'intérieur et d'extérieur de X et Y . Ainsi, ce modèle capture l'ensemble des relations topologiques à partir de l'intersection de ces trois types de notions topologiques que sont les calculs des frontières, des intérieurs et des extérieurs. Ces 3×3 types d'intersections peuvent être représentés par la matrice suivante :

$$\begin{pmatrix} \delta X \cap \delta Y & \delta X \cap Y^o & \delta X \cap Y^- \\ X^o \cap \delta Y & X^o \cap Y^o & X^o \cap Y^- \\ X^- \cap \delta Y & X^- \cap Y^o & X^- \cap Y^- \end{pmatrix}$$

où $_^o$, $_^-$ et $\delta_$ dénotent l'intérieur, l'extérieur et la frontière, respectivement. La formalisation de ce modèle demande au préalable une interprétation topologique des modalités $\Box$ et $\Diamond$.

1. Ces notions d'érosion et de dilatation sont deux opérations usuelles dans le traitement d'images. La théorie mathématique qui modélise le comportement de ces deux opérations est la morphologie mathématique.

Sémantique topologique

Jusqu'à présent, les modalités $\Box$ et $\Diamond$ étaient interprétées de façon relationnelle (i.e. dans les modèles de Kripke). Ici, nous allons les interpréter topologiquement. Ceci demande d'abord de définir une nouvelle classe de modèle, les *topos*-modèles.

Définition 180 (Topos-modèle).

Soit P une signature. Un **topos-modèle** $\mathcal{M}$ sur P est un espace topologique (X, τ) muni d'une application $\nu : P \rightarrow 2^X$. On le notera $\mathcal{M} = (X, \tau, \nu)$.

Définition 181 (Satisfaction des formules).

Soit P une signature. Soit $\mathcal{M} = (X, \tau, \nu)$ un topos-modèle sur P . Soit $\varphi \in \text{Formod}(P)$ une formule modale. Soit $x \in X$ un élément de l'ensemble X . $\mathcal{M}$ **satisfait** φ pour x , noté $\mathcal{M} \models_x \varphi$, se définit par induction sur la structure de φ de la façon suivante :

- $\mathcal{M} \models_x \Box \varphi$ ssi il existe un ouvert $O \in \tau$ tel que $x \in O$ et pour tout $y \in O$, on a $\mathcal{M} \models_y \varphi$;
- $\Diamond \varphi$ est équivalent à $\neg \Box \neg \varphi$;
- le cas des variables propositionnelles et des connecteurs se traitent de façon usuelle.

$\mathcal{M}$ **valide** φ si, et seulement si pour tout $x \in X$, $\mathcal{M} \models_x \varphi$.

Ainsi, les modalités $\Box$ et $\Diamond$ sont interprétées par les notions topologiques d'intérieur et de fermeture, respectivement.

Proposition 234.

$\Box$ est anti-extensive, i.e. qu'il satisfait l'axiome $\Box \varphi \Rightarrow \varphi$.

Démonstration : Soit $\mathcal{M} = (X, \tau, \nu)$ un topos-modèle, et soit $x \in X$ tel que $\mathcal{M} \models_x \Box \varphi$. Ceci signifie qu'il existe un ouvert $O \in \tau$ tel que $x \in O$ et pour tout $y \in O$, $\mathcal{M} \models_y \varphi$. Ainsi, on a $\mathcal{M} \models_x \varphi$. ■

Bien sûr, par dualité, nous avons aussi que $\Diamond$ est extensive, i.e. qu'il vérifie l'axiome $\varphi \Rightarrow \Diamond \varphi$.

De la même manière, on peut montrer que le schéma de Kripke, et la règle de nécessité sont aussi valides pour cette logique modale topologique. Nous obtenons ainsi que cette logique est correcte pour le calcul à la Hilbert présenté en section 28.1. On peut montrer aussi qu'il est complet. Ceci demande simplement de modifier la notion de modèle canonique pour obtenir un modèle canonique topologique.

Définition 182 (Modèle canonique topologique).

Soit P une signature. Le **modèle canonique** sur P est le modèle $\mathcal{M}_c = (X_c, \tau_c, \nu_c)$ défini par :

- Q_c est l'ensemble de tous les ensembles Γ de formules syntaxiquement complet ;
- τ_c est l'ensemble généré par union arbitraire des ensembles dans $\mathcal{B} = \{\widehat{\Box\varphi} \mid \varphi \in \text{ForMod}(P)\}$ où $\widehat{\varphi} = \{\Gamma \in Q_c \mid \varphi \in \Gamma\}$;
- $\forall p \in P, \nu_c(p) = \{\Gamma \in Q_c \mid p \in \Gamma\}$.

Proposition 235.

$\mathcal{B}$ forme une base pour la topologie.

Démonstration : Il est assez facile de montrer que $\widehat{\Box(\varphi \wedge \psi)} = \widehat{\Box\varphi} \wedge \widehat{\Box\psi}$. Ainsi, $\mathcal{B}$ est fermé par intersection finie. Il nous reste à montrer que pour chaque $\Gamma \in Q_c$, il existe un ensemble $U_\varphi \in \mathcal{B}$ tel que $\Gamma \in U_\varphi$. Soit φ une tautologie. La règle de nécessité implique que $\Box\varphi \in \Gamma$ pour tout $\Gamma \in Q_c$. Ainsi, $Q_c = \widehat{\Box\varphi}$. ■

Le reste de la preuve pour montrer la complétude du calcul dans le cadre des topos-modèles est identique à celle donnée en section 28.1.

Formalisation du modèle 9-intersection

Soient deux régions X et Y représentées par deux formules modales φ et ψ . On a donc :

- Leur intérieur se définit par $\Box\varphi$ et $\Box\psi$;
- leur extérieur se définit par $\neg\Diamond\varphi$ et $\neg\Diamond\psi$;
- leur frontière se définit par $\varphi \wedge \neg\Box\varphi$ et $\psi \wedge \neg\Box\psi$

On peut alors formaliser la matrice précédente en conséquence. Par exemple, l'intersection $\delta X \cap \delta Y$ s'écrira $(\varphi \wedge \neg\Box\varphi) \wedge (\psi \wedge \neg\Box\psi)$, ou encore $X^\circ \cap \delta Y$ s'écrira $\Box\varphi \wedge (\psi \wedge \neg\Box\psi)$.

Index

Composition de fonctions, 9
Correction et complétude d'un système formel, 25

Dédution, 22
Domaine d'une fonction, 8

Élément minimal, 13
Ensemble de termes, 20
Ensemble des preuves d'un système formel, 24
Ensembles définis par induction, 16

Fermeture d'une relation binaire, 8
Fonction totale, 9
Fonctions, 8

Groupe, 11

Monoïde, 10
Multi-ensemble, 9

Produit cartésien, 6

Relation bien-fondée, 14
Relation d'inférence, 22
Relation de préordre et d'ordre, 13
Relations n-aires, 6

Schéma d'induction, 18
Suite stationnaire, 13
Système formel, 21

Théorèmes d'un système formel, 23