Université Paris Diderot - Paris 7
Licence *Sciences et Applications*
Mention *Informatique*

Outils Logiques

Ralf Treinen

Version 6.0 du 25 septembre 2012

Adresse de l'auteur :

Ralf Treinen
Laboratoire Preuves, Programmes et Systèmes (PPS)
Université Paris Diderot - Paris 7
Case 7014
75205 PARIS Cedex 13
France

Email : `treinen@pps.univ-paris-diderot.fr`
WWW : `http://www.pps.univ-paris-diderot.fr/~treinen`

Table des matières

Chapitre 1

Introduction

La logique peut être définie comme *l'étude des règles formelles que doit respecter toute déduction correcte* [**?**]. La logique était à l'origine (c'est-à-dire, dans l'antiquité) une sous-discipline de la philosophie. La logique a continué à intéresser les philosophes, mais elle a aussi trouvé des utilisations dans des autres domaines scientifiques : Dans le 19-ème et le 20-ème siècle elle était utilisée comme base pour une nouvelle fondation de la Mathématique, et depuis le 20-ème siècle comme fondation de l'informatique. Aujourd'hui on peut dire que l'influence de la logique dans les mathématiques est très limitée, tandis que la logique s'est avérée comme absolument indispensable pour l'informatique [**?**]. Dans ce chapitre introductif nous essayons d'expliquer cette importance de la logique pour l'informatique. Certaines des notions logiques et des domaines d'application dans l'informatique mentionnés dans ce chapitre seront abordés dans ce cours d'*outils logiques*, d'autres seront présentés dans des cours qui interviendront plus tard dans un cursus d'informatique.

Avant de parler des applications il faut se faire une idée de ce qu'est la logique. La *Logique* en soit est un domaine scientifique, mais il y a plusieurs *systèmes logiques* qui sont étudiés par les logiciens (en fait on parle souvent simplement d'une *logique* au lieu de dire *système logique*). Dans une première approche on peut dire qu'un système logique consiste en les éléments suivants :

Syntaxe il s'agit d'une définition formelle (c'est-à-dire, mathématique) de l'ensemble des énoncés qui sont considérés dans ce système logique. Ces énoncés sont normalement exprimés dans un langage symbolique (donc, en particulier pas en langue naturelle), et on appelle aussi un énoncé une *formule*. Voici deux exemples de telles formules, ces exemples sont pris des systèmes logiques que nous allons étudier dans ce cours :

$$(x \wedge (y \wedge \neg z))$$

est une formule du premier système logique que nous allons étudier, la *logique propositionnelle*. Les formules de cette logique sont composées de *variables propositionnelles*, ici x, y et z, et d'opérateurs logiques, ici $\wedge$ et $\neg$. Nous reviendrons plus tard à ce que cette formule veut dire, mais pour le moment nous pouvons déjà dire qu'une telle formule ressemble beaucoup à un circuit. La logique propositionnelle est le sujet de la première partie du cours.

Un deuxième exemple d'une formule est

$$\neg x = 0 \wedge x * y = 0$$

Il s'agit ici d'une formule du système de la *logique du premier ordre*. Dans cette logique il y a des variables qui dénotent des valeurs entières (comme x, y), des opérations arithmétiques comme la multiplication $*$, et aussi des opérateurs logiques comme $\wedge$. Cette logique est en fait bien adaptée pour raisonner sur le comportement d'un programme et pour exprimer par exemples des invariants de boucles. Cela sera le sujet de la deuxième moitié du cours ; en fait nous n'allons pas étudier la logique du premier ordre dans toute sa généralité mais seulement la partie de cette logique que nous allons utiliser pour raisonner sur des programmes.

Il est important que la syntaxe soit rigoureusement définie. Nous allons dans ce cours définir les formules à l'aide de définitions inductives (cette notation sera expliquée à l'aide d'exemples). Par « définition rigoureuse » nous entendons en particulier le fait que la définition doit permettre de décider, sans aucun doute, si un texte donné présente une formule et pas.

Sémantique La sémantique définit quand une formule est vraie et quand une formule est fausse. Elle est donnée, au moins pour les systèmes logiques qui nous intéressent ici, par une définition rigoureuse qui fait bien sûr référence à la définition formelle de la syntaxe.

Très souvent, la véracité d'une formule dépend d'un contexte. La nature de ce contexte varie d'un systèmes logique à un autre. Reprenons les deux exemples de la logique propositionnelle et de la logique du premier ordre.

Dans la logique propositionnelle, le contexte spécifie la véracité des variables propositionnelles. Ainsi, la véracité de la formule $(x \wedge (y \wedge \neg z))$ dépend des valeurs des variables propositionnelles x, y, et z. La définition de la sémantique de la logique propositionnelle (que nous allons définir dans le chapitre suivant) dit que cette formule est vraie dans tout contexte dans lequel x et y sont vraies et z est fausse, et que la formule est fausse dans tous les autres contextes.

Dans la logique du premier ordre, le contexte spécifie les valeurs (qui sont des valeurs entières) des variables. La sémantique de la logique du premier ordre (que nous allons définir précisément plus tard) dira que la formule $\neg x = 0 \wedge x * y = 0$ est vraie exactement dans les contextes dans lesquelles la valeur de x est différente de 0, est dans lesquelles l'expression $x * y$ s'évalue à 0. On voit facilement que cela implique que y vaut 0. On dira aussi que la formule $y = 0$ est une *conséquence* de la formule $\neg x = 0 \wedge x * y = 0$ c'est-à-dire que la formule $y = 0$ est vraie (au moins) dans tous les contextes dans lesquelles $\neg x = 0 \wedge x * y = 0$ est vraie. La notation de conséquence est fondamentale en logique, elle existe dans tous les systèmes logiques (pas seulement celui du premier ordre), et nous en parlerons souvent dans ce cours.

Une autre notion importante est la *validité* d'un formule. Tandis que la véracité d'une formule dépend en général du contexte, nous dirons qu'une formule est *valide* si elle vraie dans *tous* les contextes possibles. Les deux formules données au-dessus, par exemple, ne sont pas valides car il y a pour les deux des contextes qui les rendent vraies, mais aussi des contextes qui les rendent fausses. Par contre, la formule

$$\neg x = 0 \vee x * y = 0$$

où le symbole $\vee$ dénote « ou », est vraie comme tous les contextes comme on peut facilement voir en considérant les deux cas $x = 0$ et $x \neq 0$, elle donc valide.

Inférence Il s'agit des règles formelles de raisonnement. Les règles d'inférence disent comment conclure la véracité d'un énoncé si on suppose que certaines hypothèses sont vraies. Plus

généralement, il s'agit de la question comment déterminer (ou faire calculer par un programme) si une formule est vraie ou pas, ou encore si elle valide ou pas. En ce qui concerne la logique propositionnelle nous n'allons pas présenter des règles d'inférence mais donner directement un algorithme qui permet de dire si une formule est valide ou pas. Par contre, un système d'inférence célèbre sera au cœur de la deuxième partie du cours, ce système permettra d'obtenir des énoncés de propriétés de programmes composés à partir d'énoncés qui portent sur des morceaux de programmes plus simples.

Cette présentation des éléments d'un système logique est volontairement simpliste, mais elle suffit pour donner une première idée des systèmes logiques que vous allez rencontrer pendant vos études. Les spécialistes de la logique peuvent nous reprocher qu'il existe des systèmes logiques qui ne répondent pas à cette caractérisation. En particulier, il existe des systèmes logiques pour lesquels la sémantique n'a pas de définition indépendante d'un système d'inférence, mais pour lesquels les règles d'inférence elles-mêmes définissent la sémantique. Notre point de vue est celui de la *logique classique* qui admet une notion de véracité qui vient avant la notion de déduction ; en opposition au point de vue de la *logique intuitionniste* qui met la notion de preuve à la base d'une sémantique.

Revenons à la question : quelle est l'importance de tout ça pour l'informatique ? Voici quelques applications des systèmes logiques :

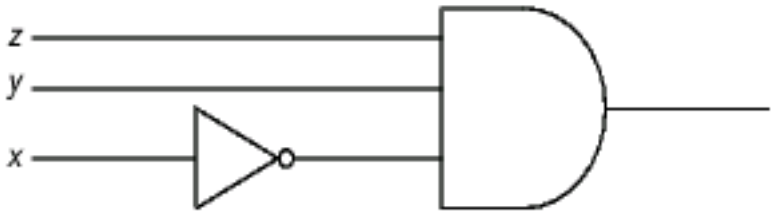
Expressions Booléennes Ces expressions existent dans tous les langages de programmation. L'écriture de ces expressions suit bien sur les règles syntaxiques propres au langage de programmation. Par exemple

Formule logique	Expression en Java
$(x \wedge (y \wedge \neg z))$	`x && (y && (!z))`
$\neg x = 0 \wedge x * y = 0$	`! (x==0) && x*y==0`

Ces expressions Booléennes sont par exemple utilisées comme gardes dans des instructions conditionnelles ou des boucles.

Questions qui nous intéressent dans ce contexte :
- Est-ce qu'on peut remplacer une formule par une formule équivalente qui est moins coûteuse à évaluer ? Cela permettra à un compilateur de simplifier la formule avant d'engendrer le code, et ainsi d'optimiser le temps de calcul.
- Est-ce qu'une formule donnée est toujours vraie ou toujours fausse ? Cela permettra à un compilateur d'omettre un test inutile, et ainsi d'optimiser le temps de calcul et aussi la taille du programme. Un compilateur pourrait aussi utiliser un tel test pour détecter des erreurs de programmation probables : une conditionnelle avec une expression de garde qui est toujours vraie ou toujours fausse est souvent signe d'une erreur de programmation.

Circuits Les circuits sont une réalisation matérielle des formules de la logique propositionnelle, dans le sens que les variables propositionnelles correspondent aux entrées d'un circuit, les opérateurs logiques correspondent aux portes logiques, et la véracité de la formule correspond à la sortie du circuit. Ainsi, la formule $(x \wedge (y \wedge \neg z))$ correspond au circuit suivant :



Les questions qui nous intéressent dans ce contexte sont :
- Construire un circuit (c'est-à-dire, une formule logique) qui réalise une fonction donnée (par exemple un additionneur).
- Déterminer la fonctionnalité d'un circuit donné.
- Est-ce que deux circuits donnés sont équivalents ?
- Minimiser un circuit : étant donné un circuit, construire le plus petit circuit avec la même fonctionnalité.

Vérification et développement de programmes C'est l'application à laquelle nous nous intéresserons dans la deuxième partie du cours. Prenons comme exemple le morceau de programme suivant :

```
z := 0;
i := 0;
  while i ≠ y do
  z := z + x;
  i := i + 1;
done
```

Nous prétendons qu'à la fin de l'exécution de ce morceau de programme, la valeur de la variable z est $y * x$. Comment peut-on se convaincre que c'est effectivement le cas ? Ici, la logique vient à la rescousse car elle nous permet d'exprimer des énoncés (en tant que formules logiques) qui sont vrais quand le programme passe par un certain point de contrôle. On appelle de telles formules des *assertions*, et quand il s'agit d'une assertion à un point de contrôle par lequel l'exécution du programme risque de passer plusieurs fois (par exemple à l'intérieur d'une boucle) un *invariant*. Dans un premier temps on peut simplement écrire des telles assertions comme un commentaire dans le programme (ici nous notons un commentaire entre accolades) :

```
z := 0;
i := 0;
while i ≠ y do
  { z = i * x }
  z := z + x;
  i := i + 1;
done
```

Dans cet exemple, l'invariant est $z = i * x$, et nous prétendons qu'il est vrai chaque fois que le programme passe par le début du corps de la boucle. Nous le montrons par une induction sur le nombre n d'itérations de la boucle :

- À la première exécution du corps de la boucle, donc quand $n = 1$, on a que z vaut 0 et i vaut 0, l'équation $z = i * x$ est donc vraie.
- S'il ne s'agit pas de la première exécution de la boucle, donc quand $n > 1$, alors on peut supposer (par l'hypothèse d'induction) que c'était vrai lors du passage précédent. On note z', i', et x' les valeurs respectives des variables z, i et x au début de l'exécution *précédente* du corps de la boucle. On a donc, par hypothèse d'induction, que

  $$z' = i' * x'$$

  Une analyse de la mise à jour des variables nous donne que

  $$\begin{aligned} z &= z' + x' \\ x &= x' \\ i &= i' + 1 \end{aligned}$$

  En d'autres mots, pendant cette exécution du corps de la boucle on incrément z par x ($=x'$) et i par 1, donc on a augmenté chacun des deux côtés de l'équation par la même valeur x. L'équation $z = i * x$ en suit par un calcul facile.

Finalement, quand on sort de la boucle la condition de garde de la boucle est forcement fausse, c'est-à-dire on a que $i = y$. L'énoncé en suit car $z = y * x$ est une conséquence de $z = i * x$ et $i = y$.

Les assertions dans les programmes sont extrêmement utiles car elles permettent de comprendre, ou d'expliquer à un tiers le fonctionnement d'un programme ; elles ouvrent la voie à la vérification automatique des programmes, et ils apportent même une méthode pour le développement de programmes. Quelques questions pertinentes dans ce contexte sont :

- Comment trouver les bonnes assertions pour un programme donné ?
- Étant donné un programme annoté avec des assertions, comment vérifier que les assertions sont vraies ?

Bases de données C'est la première application qui ne sera pas discutée dans le cadre de ce cours mais que vous allez rencontrer plus tard pendant vos études. Les bases de données modernes (par exemple les systèmes MySQL et Postgres qui sont aujourd'hui omniprésents dans les applications Web) utilisent un langage de requêtes appelé SQL. Dans ce langage, les requêtes sont spécifiées dans une variante notationnelle de la logique du premier ordre, Par exemple, la requête

```
SELECT nom FROM persons WHERE age > 18 AND ( children > 0 OR married = yes)
```

revient à évaluer la formule logique $age > 0 \wedge (children > 0 \vee married = yes)$ sur tous les éléments de la table avec le nom *persons*. Les questions qui nous intéressent dans ce contexte sont :

- Comment évaluer efficacement une requête (formule logique) ?
- Comment transformer une requête pour rendre son évaluation plus efficace ? Cela est une question importante pour des bases de données de très grande taille.

Programmation logique La programmation logique, et plus généralement la programmation logique par contraintes, est un paradigme de programmation fondamentalement différent de la programmation impérative (comme par exemple la programmation en Java). Contrairement à la programmation impérative dans laquelle le programme modifie les valeurs des variables, un programme logique cherche à trouver des valeurs de variables qui résolvent un problème exprimé dans une certaine logique, et cela en appliquant des règles de déduction qui constituent le programme. La programmation logique se prête naturellement à la solution des problèmes combinatoire, comme par exemple des problèmes de planification ou d'ordonnancement, ou encore à résoudre un puzzle du genre de Sudoku.

Démonstration Automatique Il s'agit ici de la preuve de théorèmes par ordinateur, soit de façon complètement automatique, soit dans un système interactif (aussi appelé un *assistant de preuve*) qui assure que la preuve composée par un utilisateur est correcte. Citons ici comme exemple la preuve du théorème des quatre couleurs qui a été formalisée complètement dans l'assistant de preuves Coq par une équipe d'INRIA [**?**].

Intelligence Artificielle Une direction de l'intelligence artificielle consiste à modéliser le processus de raisonnement humain par des déductions logiques. Cela nécessite de logiques très puissantes qui peuvent par exemple modéliser des propriétés temporelles (c'est à dire, qui peuvent modéliser des changement dans le temps), des probabilités, ou des énoncés épistemique qui modélisent les *connaissances* des énoncés.

Autres applications Il y a un nombre de sous-domaines de l'informatique théorique dans lesquels la logique s'est montré fructueuse, comme par exemple la théorie des types, ou encore la complexité descriptive.

Le cours va clore sur une question fondamentale de l'Informatique : Est-ce qu'il existe pour tous les problèmes mathématiques (c'est à dire des problèmes bien posés, pas du genre de la Grande Question sur la Vie, l'Univers et le Reste [**?**]), un programme qui peut répondre correctement à une question posée ? Un exemple d'un tel problème est celui de la validité d'une formule : étant donnée une formule (propositionnelle ou de la logique du premier ordre), est-elle valide ? Nous allons montrer qu'il y a des problèmes mathématiques pour lesquels un tel programme ne peut pas exister par principe, et nous allons exhiber un exemple concret d'un tel problème.

Première partie

Logique Propositionnelle

Chapitre 2

Définition de la logique propositionnelle

2.1 Syntaxe des formules propositionnelles

Les formules de la logique propositionnelle sont composées à partir des *variables propositionnelles* à l'aide des opérateurs propositionnells.

Intuitivement, pour construire une formule propositionnelle on procède comme suit : au départ on dispose d'un ensemble de variables propositionnelles. Une variable propositionnelle est déjà une formule propositionnelle, mais on a aussi le droit de construire des formules plus compliquées par les constructions suivantes :

- $\neg p$, où p est une formule
- $(p \wedge q)$, où p et q sont des formules
- $(p \vee q)$, où p et q sont des formules

Seulement les expressions formées ainsi sont des formules.

Par exemple, x_1, x_2 et x_3 sont des variables propositionnelles, et donc aussi des formules. Par conséquent, $\neg x_1$ est une formule et $(x_2 \wedge x_3)$ est une formule, donc on obtient finalement que $(\neg x_1 \vee (x_2 \wedge x_3))$ est une formule. Par contre, $(x_1 \wedge x_2) \vee \neg$ n'est pas une formule.

Une telle description informelle des formules n'est pas suffisante. Il nous faut plus de rigueur dans la définition car nous aurions plus tard à définir des fonctions sur l'ensemble des formules, et nous allons faire des preuves de propriétés des formules. D'abord nous fixons un ensemble de variables propositionnelles :

$$V := \{x, x_1, x_2, x_3, \ldots, y, y_1, y_2, \ldots, z, z_1, z_2, z_3, \ldots\}$$

Nous admettons également les décorations habituelles des variables propositionnelles comme par exemple x', y''.

Faisons un premier essai d'une définition des formules propositionnelles :

Tentative de définition : *L'ensemble Form des formules propositionnelles est défini comme l'ensemble de chaînes de caractères tel que :*

1. $V \subseteq Form$
2. *Si* $p \in Form$ *alors* $\neg p \in Form$

3. *Si* $p, q \in Form$ *alors* $(p \wedge q) \in Form$
4. *Si* $p, q \in Form$ *alors* $(p \vee q) \in Form$

Nous appelons ces quatre conditions (1) - (4) les *propriétés de clôture des formules propositionnelles*.

Le problème avec cette tentative de définition est qu'elle n'est pas une vraie définition car il y a plusieurs ensembles *Form* qui satisfont la description. L'ensemble des formules propositionnelles dans le sens de la description informelle au-dessus répond à cette description, mais par exemple l'ensemble de toutes les chaînes de caractères y répond également. En fait il y a beaucoup d'exemples d'ensembles qui satisfont cette description (voir le TD).

La racine du problème est que notre tentative de définition prend en compte le cas de base (les variables propositionnelles) et les constructions autorisées ($\neg$, $\wedge$, $\vee$), mais ne prend absolument pas en compte la restriction qui dit qu'une chaîne qui ne peut pas être construite selon les règles n'est pas une formule.

Définition 1 *L'ensemble Form des formules propositionnelles est le plus petit ensemble de chaînes de caractères tel que :*

1. $V \subseteq Form$
2. *Si* $p \in Form$ *alors* $\neg p \in Form$
3. *Si* $p, q \in Form$ *alors* $(p \wedge q) \in Form$
4. *Si* $p, q \in Form$ *alors* $(p \vee q) \in Form$

La différence entre cette définition et la tentative de définition au-dessus est l'ajout de la spécification « *le plus petit* » ensemble. Cela veut dire : quand un ensemble X satisfait les quatre propriétés de clôture des formules propositionnelles, alors $Form \subseteq X$. Pour montrer qu'il s'agit effectivement d'une définition valide il faut montrer deux choses :

1. il existe effectivement (au moins) un tel plus petit ensemble, et
2. cet ensemble est unique.

L'unicité est facile à voir : supposons par l'absurde qu'il y avait deux tels ensembles différents, disons *Form* et $Form'$. Donc, on obtient que $Form \subseteq Form'$ (car *Form* est le plus petit ensemble avec les propriétés de clôture des formules propositionnelles), et aussi que $Form' \subseteq Form$ (car $Form'$ est aussi le plus petit ensemble avec ces propriétés). Par conséquent, $Form = Form'$, ce qui est en contradiction avec l'assomption que $Form \neq Form'$. Absurde.

Nous ne montrerons pas ici pas le premier point (l'existence d'un plus petit ensemble). Remarquons seulement qu'il n'est à priori pas évident que, pour une certaine propriété donnée, un *plus petit* ensemble avec cette propriété existe. Si par exemple la propriété est « l'ensemble contient au moins deux éléments » alors un tel plus petit ensemble n'existe pas. Démonstration : l'ensemble $E = \{0, 1\}$ a bien cette propriété mais il n'est certainement pas le plus petit ensemble avec cette propriété car $\{2, 3\}$ aussi a deux éléments mais $E \not\subseteq \{2, 3\}$. S'il y avait un plus petit ensemble avec cette propriété il faudrait alors qu'il soit un sous-ensemble propre de E, ce qui est une contradiction car dans ce cas cet ensemble a moins que deux éléments.

Nous dirons que la définition 1 est une *définition inductive*. Nous n'allons dans ce cours pas définir la notion d'une définition inductive en général, bien que nous verrons encore d'autres exemples de définitions inductives plus tard dans le cours. Sachez qu'il y a un cadre théorique

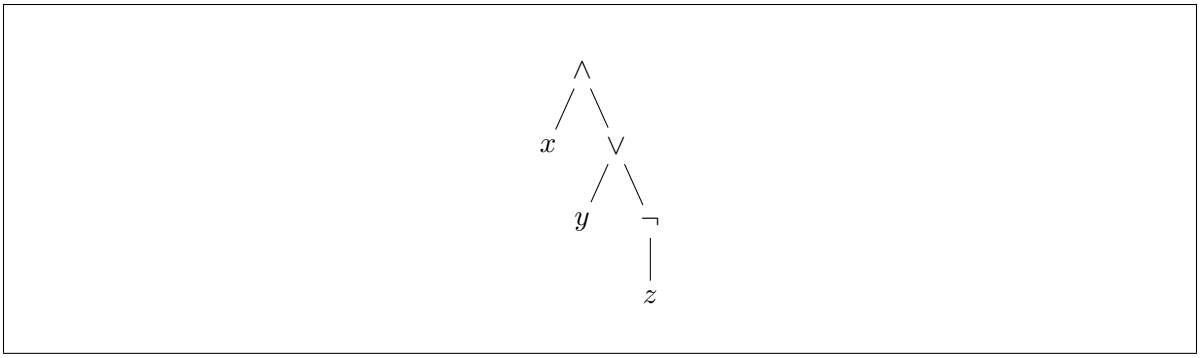


FIGURE 2.1: L'arbre syntaxique de la formule $(x \wedge (y \vee \neg z))$.

pour les définitions inductives qui assure, pour une certaine classe de propriétés, qu'un plus petit ensemble avec cette propriété existe. Il s'agit de la théorie des clauses définies (voir le cours de programmation logique en M1).

La définition de la syntaxe doit aussi permettre d'analyser la structure d'une formule : Sur l'exemple de la formule $(x \wedge (y \vee \neg z))$, l'analyse nous donne que la formule consiste en deux parties qui sont jointes par l'opérateur $\wedge$, la première partie est la variable x, et la deuxième partie consiste elle-même en deux parties qui sont jointes par l'opérateur $\vee$: la variable y, et l'application de l'opérateur $\neg$ à la variable z.

C'est la seule façon de décomposer la formule, elle donne lieu à l'*arbre syntaxique* donné sur la figure 2.1. La définition de la syntaxe doit permettre d'écrire un programme qui lit un texte, et qui construit la structure correspondante quand le texte représente effectivement une formule.

2.2 Preuves par induction

Un intérêt d'une définition inductive d'un ensemble est qu'on peut démontrer des propriétés de cet ensemble par une *induction structurelle*. Encore une fois nous n'allons pas étudier ce principe de preuve dans toute sa généralité. Nous nous contenterons pour l'instant de montrer ce principe sur l'ensemble *Form* des formules propositionnelles.

Nous illustrons d'abord ce principe de preuve sur un exemple simple :

Théorème 1 *Toute formule propositionnelle a le même nombre de parenthèses ouvrantes que de parenthèses fermantes.*

Pour montrer ce théorème il convient d'introduire une petite notation : $|w|_($ dénote le nombre de parenthèses ouvrantes de w, et $|w|_)$ dénote le nombre de parenthèses fermantes de w. Avec cette notation nous pouvons reformuler l'énoncé comme : pour tout $w \in \mathit{Form}$, $|w|_(= |w|_)$.

Démonstration: Soit X l'ensemble des chaînes de caractères qui ont le même nombre de parenthèses ouvrantes que de parenthèses fermantes :

$$X = \{w \mid |w|_(= |w|_)\}$$

Nous montrons d'abord que cet ensemble X satisfait les propriétés de clôture des formules propositionnelles :

1. Il faut montrer que $V \subseteq X$:
   Si $w \in V$ alors $|w|_(= 0$ et $|w|_) = 0$, donc $|w|_(= |w|_)$.
2. Il faut montrer que si $p \in X$ alors $\neg p \in X$:
   Soit $|p|_(= |p|_)$. On a que
   $$|\neg p|_(= |p|_(= |p|_) = |\neg p|_)$$
3. Il faut montrer que si $p, q \in X$ alors $(p \wedge q) \in X$.
   Soient $|p|_(= |p|_)$ et $|q|_(= |q|_)$. On a que
   $$|(p \wedge q)|_(= 1 + |p|_(+ |q|_(= |p|_) + |q|_) + 1 = |(p \wedge q)|_)$$
4. Il faut montrer que si $p, q \in X$ alors $(p \vee q) \in X$.
   Soient $|p|_(= |p|_)$ et $|q|_(= |q|_)$. On a que
   $$|(p \vee q)|_(= 1 + |p|_(+ |q|_(= |p|_) + |q|_) + 1 = |(p \vee q)|_)$$

Donc, X est un ensemble qui satisfait les propriétés de clôture des formules propositionnelles. Or, *Form* est le *plus petit ensemble* qui satisfait les propriétés de clôtures des formules propositionnelles selon la définition de *Form*. Par conséquent, $Form \subseteq X$. □

On peut maintenant définir plus généralement ce principe de preuve pour une propriété, ici appelée P, des formules propositionnelles :

Théorème 2 *Soit P une propriété des chaînes de caractères, Si les énoncés suivants sont vrais :*
– tout élément de V a la propriété P,
– si p satisfait P alors $\neg p$ satisfait P,
– si p et q satisfont P alors $(p \wedge q)$ satisfait P,
– si p et q satisfont P alors $(p \vee q)$ satisfait P,
alors tous les éléments de Form satisfont P.

La preuve reprend simplement l'argument final de la preuve du théorème 1 :

Démonstration: Soit X l'ensemble des chaînes de caractères avec la propriété P. L'ensemble X satisfait les propriétés de clôture des formules propositionnelles (hypothèse du théorème), donc $Form \subseteq X$ car *Form* est le plus petit ensemble de chaînes de caractères qui satisfait ces propriétés. Autrement dit, tout élément de *Form* a la propriété P. □

Dans le cas du théorème 1, la propriété P est d'avoir le même nombre de parenthèses ouvrantes que de parenthèses fermantes.

2.3 Définition de fonctions par récurrence

Comme pour le principe de preuve par induction nous allons illustrer le principe d'une définition d'une fonction par récurrence seulement sur l'exemple des formules propositionnelles.

Tout le monde connaît des définitions de fonctions par une expression close. Par exemple, la fonction qui envoie un argument, qui est un nombre naturel, vers l'argument incrémenté par 1, peut être définie par

$$f(x) := x + 1$$

Dans ce cas, l'expression close est $x + 1$. Pour évaluer la fonction f sur un argument donné, par exemple 17, on remplace dans l'expression de la définition de la fonction f (c.-à-d. $x + 1$) la variable qui sert comme paramètre (c.-à-d. x) par sa valeur (17), et on évalue l'expression qui en résulte :

$$17 + 1 = 18$$

On peut aussi définir de la même façon certaines fonctions sur les formules propositionnelles. Par exemple, la fonction qui envoie son argument, qui est une formule propositionnelle, vers sa négation peut être définie par

$$g(x) := \neg x$$

Il y a peu de fonctions intéressantes sur les formules propositionnelles qui peuvent être définies de cette façon. En particulier il nous faudrait la possibilité de prendre en compte la façon dont une formule est construite (par $\wedge$, par $\vee$, ...), de décomposer l'argument en sous-formules, et éventuellement de faire un calcul sur les sous-formules pour obtenir la valeur envoyée par la fonction. Le principe de définition d'une fonction par récurrence nous donne cette possibilité. Ce principe peut être énoncé comme suit :

Définition d'une fonction sur *Form* par récurrence *On peut définir une fonction avec le domaine Form comme suit :*

1. *on donne le résultat de la fonction appliquée à un élément quelconque de V,*
2. *on donne le résultat de la fonction appliquée à une formule de la forme $\neg p$, en supposant le résultat de la fonction appliquée à p connu,*
3. *on donne le résultat de la fonction appliquée à une formule de la forme $(p \wedge q)$, en supposant le résultat de la fonction appliquée à p et le résultat de la fonction appliquée à q connus.*
4. *on donne le résultat de la fonction appliquée à une formule de la forme $(p \vee q)$, en supposant le résultat de la fonction appliquée à p et le résultat de la fonction appliquée à q connus.*

Illustrons d'abord ce principe par deux exemples. La fonction *length* est définie comme suit :

1. $\mathit{length}(x) = 1$ si $x \in V$
2. $\mathit{length}(\neg p) = 1 + \mathit{length}(p)$
3. $\mathit{length}((p \wedge q)) = 3 + \mathit{length}(p) + \mathit{length}(q)$
4. $\mathit{length}((p \vee q)) = 3 + \mathit{length}(p) + \mathit{length}(q)$

Intuitivement, la fonction *length* calcule le nombre de caractères dans une formule, où on pose que la longueur d'une variable propositionnelle est 1. Pour évaluer cette fonction sur un argument donné on « déplie » la définition de la fonctions selon la structure de l'argument donné. Dans l'exemple suivant nous donnons dans chaque ligne le numéro de la clause (de la définition de la fonction *length*) que nous avons utilisée pour le dépliage :

$$\begin{array}{rcll} \mathit{length}((x_1 \wedge \neg x_2)) & = & 3 + \mathit{length}(x_1) + \mathit{length}(\neg x_2) & (3) \\ & = & 3 + 1 + \mathit{length}(\neg x_2) & (1) \\ & = & 3 + 1 + 1 + \mathit{length}(x_2) & (2) \\ & = & 3 + 1 + 1 + 1 & (1) \\ & = & 6 & \end{array}$$

Repère de notation : récurrence et induction.
- Un ensemble peut être défini par *induction* : on dit comment construire un nouvel élément de l'ensemble à partir des éléments plus primitifs. Il y donc un sens « ascendant ».
- Les fonctions peuvent être définies par *récurrence* : on défini le résultat d'une fonction appliquée sur un argument composé en faisant référence au résultat de la fonction sur des arguments plus simple. Il y a donc un sens « descendant ».
- Finalement, une propriété de tous les éléments d'un ensemble qui est défini par induction est normalement démontrée par *induction structurelle*.

Notre deuxième exemple est la fonction $\mathcal{V}$, définie comme suit :

1. $\mathcal{V}(x) = \{x\}$ si $x \in V$
2. $\mathcal{V}(\neg p) = \mathcal{V}(p)$
3. $\mathcal{V}((p \wedge q)) = \mathcal{V}(p) \cup \mathcal{V}(q)$
4. $\mathcal{V}((p \vee q)) = \mathcal{V}(p) \cup \mathcal{V}(q)$

Intuitivement, la fonction $\mathcal{V}$ calcule l'ensemble des variables propositionnelles qui paraissent dans une formule. On évalue cette fonction comme dans le premier exemple par dépliage. Sur l'exemple suivant nous déplions des sous-expressions indépendantes en une seule étape pour aller un peu plus vite :

$$\begin{array}{rcll} \mathcal{V}((x_1 \wedge (x_2 \vee x_3))) & = & \mathcal{V}(x_1) \cup \mathcal{V}((x_2 \vee x_3)) & (3) \\ & = & \{x_1\} \cup \mathcal{V}(x_2) \cup \mathcal{V}(x_3) & (1), (4) \\ & = & \{x_1\} \cup \{x_2\} \cup \{x_3\} & (1), (1) \\ & = & \{x_1, x_2, x_3\} & \end{array}$$

Exercice 1 *Montrer par induction structurelle que pour toute formule $p \in Form$:*

$$card(\mathcal{V}(p)) \leq length(p)$$

où $card(E)$ dénote la cardinalité (c.-à-d. le nombre d'éléments) d'un ensemble E.

Il y a une subtilité derrière ce principe de définition. Une fonction doit toujours associer à un argument donné un seul résultat unique. On doit donc assurer qu'une définition récursive d'une fonction garantie bien cette unicité du résultat. Le problème est que, en général, la même formule pourrait être construite de deux façon différentes, et que la définition de la fonction donne deux valeurs différentes selon la construction considérée. Heureusement ce risque n'existe pas pour les formules de la logique propositionnelle car toute formule peut être construite d'une seule façon, c'est-à-dire il y a un seul arbre syntaxique pour toute formule (il y a derrière cette assertion le *théorème de lecture unique* que nous n'allons pas montrer ici). Sachez que quand on essaye de généraliser les définitions par induction et récurrence il faut donc assurer cette unicité de la construction, sinon on aura le souci supplémentaire de montrer qu'une fonction définie par récurrence est effectivement bien définie.

2.4 Sémantique de la logique propositionnelle

Le rôle de la sémantique est de dire quand une formule est vraie et quand une formule est fausse (c'est encore une simplification qui nous suffit pour notre cours mais qui n'est pas vraie pour certaines logiques exotiques - par exemple il existe des logiques avec plus que deux valeurs de vérités, mais ça dépasse largement le cadre de ce cours). Dans ce cours nous avons choisi de représenter les deux valeurs logiques possibles par 0 et par 1. On trouve dans la littérature aussi des notations différentes, par exemple True et False, tt et ff, et encore des autres. Évidemment 1, True, tt correspondent au fait qu'un énoncé est vrai, et 0, False, ff correspondent au fait qu'un énoncé est faux.

Comme expliqué dans l'introduction, la véracité d'une formule dépend d'un contexte, et pour la logique propositionnelle un contexte est une *affectation* :

Définition 2 *Une* affectation *est une fonction*

$$v : V \to \{0, 1\}$$

Le support *d'une affectation v est défini comme*

$$supp(v) = \{x \in V \mid v(x) = 1\}$$

Nous précisons qu'une affectation est toujours une fonction *totale*, c'est-à-dire une fonction qui à chaque variable associe une valeur. Par exemple, la fonction v_1 qui associe à x la valeur 1, à y la valeur 1, et qui associe à toute autre variable la valeur 0 est une affectation, et son support est $\{x, y\}$. Sont également des affectations la fonction qui associe à chaque variable la valeur 0, ou la fonction qui associe à chaque variable la valeur 1, leur support est respectivement l'ensemble vide et l'ensemble V.

Nous utilisons la notation suivante pour noter des affectations :

$$[x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 1, \ldots, x_n \mapsto 1]$$

est l'affectation qui aux variables $x_1, \ldots, x_n$ associe la valeur 1, et qui associe à toute autre variable la valeur 0. Par exemple, $[x \mapsto 1, y \mapsto 1]$ est l'affectation qui associe à x la valeur 1, à y la valeur 1, et qui associe à toute autre variable la valeur 0 ; et [] est l'affectation qui associe à toute variable la valeur 0. On peut dans cette notation aussi ajouter des valeurs 0 pour certaines variables, comme dans $[x \mapsto 0, y \mapsto 1]$ mais cela n'est pas très utile car 0 est de toute façon la « valeur de défaut ».

Une affectation avec un support infini, comme par exemple la fonction qui associe à toute variable la valeur 1, ne peut évidemment pas être représentée de cette façon. Cela ne nous dérange pas car dans les applications on s'intéresse seulement à un nombre fini de variables. Par ailleurs, on peut toujours faire recours à une notation mathématique si on en a besoin.

Définition 3 *L'interprétation $[\![p]\!]v$ d'une formule p par rapport à l'affectation v est définie*

Repère de notation : évaluation et interprétation.
– En général, une application d'une fonction à des arguments est *évaluée*.
– Plus spécifiquement, une formule propositionelle est *interprétée* par rapport à une affectation.
L'interprétation de p par rapport à v est obtenue par l'évaluation de $[\![p]\!]v$.

par récurrence sur la structure de p :

$$\begin{array}{lcl}
[\![x]\!]v & = & v(x) \\
[\![\neg p]\!]v & = & \left\{ \begin{array}{ll} 0 & si\ [\![p]\!]v = 1 \\ 1 & si\ [\![p]\!]v = 0 \end{array} \right. \\
[\![(p \wedge q)]\!]v & = & \left\{ \begin{array}{ll} 0 & si\ [\![p]\!]v = 0\ ou\ [\![q]\!]v = 0 \\ 1 & si\ [\![p]\!]v = 1\ et\ [\![q]\!]v = 1 \end{array} \right. \\
[\![(p \vee q)]\!]v & = & \left\{ \begin{array}{ll} 0 & si\ [\![p]\!]v = 0\ et\ [\![q]\!]v = 0 \\ 1 & si\ [\![p]\!]v = 1\ ou\ [\![q]\!]v = 1 \end{array} \right.
\end{array}$$

Par exemple, pour l'affectation $v_1 = [y \mapsto 1]$ on a que $[\![x]\!]v_1 = 0$ (car $v_1(x) = 0$) et que $[\![y]\!]v_1 = 1$ (car $v_1(y) = 1$). Par conséquence, $[\![(x \wedge y)]\!]v_1 = 0$ et $[\![(x \vee y)]\!]v_1 = 1$.

Cette définition laisse le libre choix de l'ordre dans lequel les deux sous-formules p et q sont interprétées dans les deux dernières cas de la définition. Le théorème suivant donne une stratégie d'interprétation :

Théorème 3 *Soient p et q des formules propositionnelles et v une affectation, alors*

$$[\![(p \wedge q)]\!]v = \left\{ \begin{array}{ll} 0 & si\ [\![p]\!]v = 0 \\ [\![q]\!]v & si\ [\![p]\!]v = 1 \end{array} \right.$$

$$[\![(p \vee q)]\!]v = \left\{ \begin{array}{ll} 1 & si\ [\![p]\!]v = 1 \\ [\![q]\!]v & si\ [\![p]\!]v = 0 \end{array} \right.$$

Démonstration: Nous démontrons seulement le premier des deux énoncés ; le second se montre de façon analogue. Il y a deux cas, selon la valeur de $[\![p]\!]v$:

Cas $[\![p]\!]v = 0$ **:** Nous avons à montrer que dans ce cas $[\![(p \wedge q)]\!]v = 0$. C'est une conséquence immédiate de la définition 3.

Cas $[\![p]\!]v = 1$ **:** Nous avons à montrer que dans ce cas $[\![(p \wedge q)]\!]v = [\![q]\!]v$. Il y a deux cas, selon la valeur de $[\![q]\!]v$:

Cas $[\![q]\!]v = 0$ **:** Nous avons $[\![(p \wedge q)]\!]v = 0 = [\![q]\!]v$

Cas $[\![q]\!]v = 1$ **:** Nous avons $[\![(p \wedge q)]\!]v = 1 = [\![q]\!]v$

□

Ce théorème nous dit que pour évaluer $[\![(p \wedge q)]\!]v$ on évalue d'abord $[\![p]\!]v$, et selon le résultat obtenu il se peut qu'il n'est plus nécessaire d'évaluer $[\![q]\!]v$, ce qui est bon à savoir quand q est une très grande expression. Remarquez qu'on aurait pu donner une variante du théorème 3 dans laquelle on interprète d'abord q au lieu de p, ou encore des variantes avec des critères plus sophistiqués (par exemple : on commence avec l'interprétation de la formule parmi p, q qui est est la plus petite).

Repère de notation : notions de sémantique. Ne pas confondre les notations :
- Une formule est *vraie* ou *fausse* toujours par rapport à une affectation.
- Une formule peut être *satisfaisable* ou *falsifiable* tout court. Il n'y a pas de « satisfaisable par une affectation ».
- Une formule peut être *valide* ou *contradictoire*.

Définition 4 *Soit p une formule propositionnelle.*
- *On écrit $v \models p$ si $[\![p]\!]v = 1$, et on dit « p est vraie par rapport à v ».*
- *On écrit $v \not\models p$ si $[\![p]\!]v = 0$, et on dit « p est fausse par rapport à v ».*
- *On dit que p est* satisfaisable *s'il existe une affectation v telle que $v \models p$.*
- *On dit que p est* falsifiable *s'il existe une affectation v telle que $v \not\models p$.*
- *On écrit $\models p$ si $v \models p$ pour toute affectation v, et on dit que p est* valide *(ou une* tautologie*).*
- *On dit que p est* contradictoire *si $v \not\models p$ pour toute affectation v.*

Proposition 1 *Une formule p est valide si et seulement si $\neg p$ n'est pas satisfaisable.*

Démonstration: On a la chaîne d'équivalences suivante :

p est valide
ssi $v \models p$ pour toute affectation v
ssi $[\![p]\!]v = 1$ pour toute affectation v
ssi $[\![\neg p]\!]v = 0$ pour toute affectation v
ssi $v \models \neg p$ pour *aucune* affectation v
ssi $\neg p$ n'est pas satisfaisable

□

Le mot « ssi » est une abréviation pour « si et seulement si ».

Proposition 2 *Une formule p est contradictoire si et seulement si $\neg p$ est valide.*

Exercice 2 *Le démontrer.*

Pour savoir si une formule propositionnelle donnée est satisfaisable ou valide il faut donc en principe évaluer la formule sur toutes les affectations possibles, ce qui pose une petit problème : il y a un nombre infini d'affectations possibles car il y a un nombre infini de variables propositionnelles ! Heureusement, il y a un théorème qui nous dit que seulement les variables qui paraissent dans une formule sont pertinentes.

Proposition 3 *Soit p une formule propositionnelle et v_1, v_2 des affectations telles que $v_1(x) = v_2(x)$ pour toute variable $x \in \mathcal{V}(p)$. Alors $[\![p]\!]v_1 = [\![p]\!]v_2$.*

Exercice 3 *Montrer la proposition 3.*

Théorème 4 *Une formule p est*

1. *satisfaisable si et seulement s'il existe une affectation v telle que $supp(v) \subseteq \mathcal{V}(p)$ et $v \models p$.*
2. *valide si et seulement si $v \models p$ pour toute affectation v avec $supp(v) \subseteq \mathcal{V}(p)$.*

Démonstration: Nous montrons ici seulement le premier énoncé, le deuxième est laissé comme exercice.

Si $v \models p$ avec $supp(v) \subseteq \mathcal{V}(p)$ alors p est, par définition satisfaisable.

Si p est satisfaisable il y a une affectation w telle que $w \models p$. Nous construisons une nouvelle affectation v comme suit :

$$v(x) = \begin{cases} w(x) & \text{si } x \in \mathcal{V}(p) \\ 0 & \text{si } x \notin \mathcal{V}(p) \end{cases}$$

On a que $supp(v) \subseteq \mathcal{V}(p)$, et $v \models p$ par proposition 3. □

Exercice 4 *Montrer le deuxième énoncé du théorème 4.*

Il suffit donc de tester les affectations qui ont pour domaine $\mathcal{V}(p)$. L'avantage est qu'il y en un nombre fini, plus exactement il y a 2^n telles affectations quand p contient n variables. C'est donc à priori faisable mais potentiellement coûteux.

Une méthode pour déterminer si une formule p est satisfaisable est donc :

1. Calculer $\mathcal{V}(p)$
2. Engendrer l'ensemble A des affectations v avec $supp(v) \subseteq \mathcal{V}(p)$
3. Évaluer $[\![p]\!]v$ pour toute affectation $v \in A$. Dès qu'on tombe sur un v tel que $[\![p]\!]v = 1$ on sait que p est satisfaisable, si on n'en trouve pas alors p n'est pas satisfaisable.

Une méthode pour déterminer si une formule p est valide est :

1. Calculer $\mathcal{V}(p)$
2. Engendrer l'ensemble A des affectations v avec $supp(v) \subseteq \mathcal{V}(p)$
3. Évaluer $[\![p]\!]v$ pour tout $v \in A$. Dès qu'on tombe sur un v tel que $[\![p]\!]v = 0$ on sait que p n'est pas valide, si on n'en trouve pas alors p est valide.

En pratique on utilise souvent des *tables de vérité* pour déterminer la satisfaisabilité ou la validité d'une formule p. Les colonnes d'une telle table de vérité sont étiquetées par des sous-formules de p : au début (dans les colonnes à gauche) on a toutes les variables propositionnelles qui paraissent dans p, puis les sous-formules de p dans un ordre croissant de complexité, et finalement la formule p elle-même. La table a 2^n lignes où n est le nombre de variables propositionnelles qui paraissent dans la formule. Maintenant, on remplit toutes les cases des colonnes qui correspondent à des variables propositionnelles par toutes les combinaisons possibles de 0 et 1. Il convient de choisir un système pour ne pas oublier une combinaison, par exemple on fait dans la première colonne une suite alternante 01010101..., dans la deuxième colonne une suite alternante à deux pas 00110011..., dans la troisième 00001111..., etc. De cette façon toute ligne correspond à une affectation différente v avec $supp(v) \subseteq \mathcal{V}(p)$. Puis, on rempli toutes les autres colonnes de la table, de gauche vers la droite. Puisqu'on a choisi les étiquettes des colonnes dans un ordre croissant de complexité, on trouve toujours les interprétations des sous-formules d'une formule composée dans des colonnes qui sont déjà remplies.

Exemple : Soit p la formule $((x_1 \wedge x_2) \vee (x_3 \wedge \neg x_2))$. La table de vérité est comme suit. Nous indiquons dans cet exemple sur la ligne *Calcul* comment les entrées sont calculées, normalement

les lignes *Numéro* et *Calcul* sont omises dans une table de vérité.

Formule	x_1	x_2	x_3	$\neg x_2$	$(x_1 \wedge x_2)$	$(x_3 \wedge \neg x_2)$	$((x_1 \wedge x_2) \vee (x_3 \wedge \neg x_2))$
Numéro	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Calcul				$\neg$(2)	(1) $\wedge$ (2)	(3) $\wedge$ (4)	(5) $\vee$ (6)
	0	0	0	1	0	0	0
	1	0	0	1	0	0	0
	0	1	0	0	0	0	0
	1	1	0	0	1	0	1
	0	0	1	1	0	1	1
	1	0	1	1	0	1	1
	0	1	1	0	0	0	0
	1	1	1	0	1	0	1

La formule p est donc satisfaisable, mais pas valide.

Quel est le temps de calcul de notre algorithme sur une formule avec n variables ? Il y a 2^n affectations possibles qu'il faut dans le pire des cas toutes essayer avant d'en trouver une qui rend la formule vraie. Ça nous donne donc un temps d'exécution qui est au moins exponentiel dans le nombre de variables. C'est faisable pour des très petites formules comme celle dans l'exemple au-dessus, mais c'est absolument inacceptable pour les formules gigantesques issues par exemple de la vérification des circuits qui peuvent avoir des milliers de variables.

2.5 Raccourcis syntaxiques

Notre définition de la syntaxe des formules propositionnelles (définition 1) est assez stricte : la définition exige des parenthèses autour de toute application d'un opérateur binaire, et elle ne permet pas des applications d'un seul opérateur à plus que deux arguments à la fois. Ces restrictions sont en pratique peu commodes pour écrire des formules. Les règles d'écriture des expressions booléennes dans les langages de programmation, par exemple, sont plus libérales. Pour cette raison nous autorisons dans la suite une notation un peu plus libérale des formules propositionnelles :

– On a le droit d'enchaîner des applications de l'opérateur $\wedge$:

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$$

ainsi que de l'opérateur $\vee$:

$$(p_1 \vee p_2 \vee \ldots \vee p_n)$$

– On se permet d'omettre la paire de parenthèses qui est autour de la formule *entière*.

Il s'agit ici des *raccourcis* d'écriture, pas d'une modification de la définition des formules ! Une formule en écriture raccourcie peut toujours être réécrite dans la syntaxe stricte selon définition 1 :

– Remplacer

$$(p_1 \wedge p_2 \wedge x_3 \wedge \ldots \wedge p_n)$$

par

$$(p_1 \wedge (p_2 \wedge (p_3 \ldots \wedge p_n) \ldots))$$

et pareil pour les chaînes $\vee$.

– Mettre une paire de parenthèses extérieures autour de la formule si nécessaire
Ainsi,

$$(x_1 \wedge x_2 \wedge x_3) \vee y_1 \vee (z_1 \wedge z_2 \wedge z_3 \wedge z_4)$$

s'écrit en syntaxe stricte comme

$$\Big((x_1 \wedge (x_2 \wedge x_3)) \vee \Big(y_1 \vee (z_1 \wedge (z_2 \wedge (z_3 \wedge z_4)))\Big)\Big)$$

où nous avons agrandi un peu les parenthèses des applications de $\vee$ pour mieux faire paraître la structure.

2.6 Références et remarques

Nous avons ici choisi de définir les affectations comme des fonctions totales. On trouve aussi parfois des présentations de la logique propositionnelle qui permettent comme affectations aussi des fonctions partielles, c'est-à-dire des fonctions qui peuvent laisser la valeur de certaines variables non définie.

On peut aussi se poser la question de savoir pourquoi nous avons défini les formules propositionnelles si strictement avec toutes ces parenthèses. La raison est qu'une définition stricte est avantageuse quand on fait des preuves de propriétés des formules, ou quand on définit des fonctions par récurrence sur les formules. Dans la syntaxe stricte il n'y a que quatre cas assez simples (variable, $\neg p$, $(p \wedge q)$, $(p \vee q)$). Si on avait choisi la syntaxe libérale comme définition de la syntaxe on aurait des cas plus compliqués à considérer, comme par exemple $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, ce qui risque de nécessiter une deuxième induction sur le nombre n. Notre façon de définir les formules propositionnelles combine les deux avantages : d'une part des cas simples dans les preuves par induction et définitions des fonctions par récurrence, d'autre part une certaine souplesse dans l'écriture des formules en pratique.

En fait on pourrait être encore plus libéral et aussi permettre l'utilisation des *priorités* entre opérateurs, comme c'est souvent le cas dans les langages de programmation. Les priorités permettent encore plus d'économie dans l'utilisation des parenthèses. Dans ce cours nous avons choisi de ne pas considérer les priorités entre opérateurs, une raison pour ce choix est que l'utilisation des priorités rende la tâche d'analyse des formules par un programme beaucoup plus difficile. Ce problème, et sa solution, est un sujet du cours *Analyse Syntaxique* de l'année L3.

Est-ce qu'il y a un algorithme qui est plus rapide que la méthode des tables de vérité, par exemple un algorithme qui peut déterminer la satisfaisabilité d'une formule avec n variables dans un temps de calcul qui est un polynôme en n (par exemple n^3) ? Il s'agit ici de la question ouverte la plus célèbre de l'informatique ; les informaticiens cherchent une réponse à cette question depuis plus que 30 ans. Il y a des algorithmes qui trouvent des réponses avec une étonnante rapidité pour des classes de formules particulières, mais la question est s'il y a un algorithme qui peut répondre rapidement pour *n'importe quelle* formule qu'on lui présente. La majorité des informaticiens pensent aujourd'hui qu'un tel algorithme ne peut pas exister, mais à ce jour personne n'a réussi à le démontrer. Les questions de ce type sont l'objet d'étude d'un domaine important de l'Informatique : la théorie de la *Complexité*.

Chapitre 3

Lois de la logique propositionnelle

3.1 Conséquences logiques et équivalences

Définition 5 *Soient p et q des formules propositionnelles.*
- *On dit que q est une* conséquence *de p, et on écrit $p \models q$, si pour toute affectation v telle que $v \models p$ on a aussi que $v \models q$.*
- *On dit que p et q sont* équivalentes, *et on écrit $p \models\!\models q$, si $p \models q$ et $q \models p$.*

Donc, deux formules p et q sont équivalentes si et seulement si $[\![p]\!]v = [\![q]\!]v$ pour toute affectation v. Parfois on s'intéresse aussi à la notion d'une conséquence de tout en ensemble de formules :

Définition 6 *Soient $p \in Form$ une formule et $T \subseteq Form$ un ensemble de formules. On dit que p est une* conséquence *de T, noté $T \models p$, si pour toute affection v telle que $v \models q$ pour tout $q \in T$ on a aussi que $v \models p$.*

Exercice 5
1. *Montrer que $\{q\} \models p$ ssi $q \models p$*
2. *Montrer que si $T \models p$ et $T \subseteq S$ alors $S \models p$*
3. *Que veut dire que $\emptyset \models p$?*

On utilise les tables de vérité pour savoir si une formule est conséquence d'une autre ou si deux formules sont équivalentes. La formule q est une conséquence de p si pour toute ligne de la table de vérité où il y a 1 dans la colonne de p il y aussi 1 dans la colonne de q ; les formules p et q sont équivalentes quand le contenu de la colonne de p est le même que le contenu de la colonne de q.

Par exemple, $x \wedge (\neg x \vee y) \models y$ car

x	y	$\neg x$	$\neg x \vee y$	$x \wedge (\neg x \vee y)$	y
0	0	1	1	0	0
1	0	0	0	0	0
0	1	1	1	0	1
1	1	0	1	1	1

Les formules $\neg(x \wedge y)$ et $\neg x \vee \neg y$ sont équivalentes :

x	y	$x \wedge y$	$\neg(x \wedge y)$	$\neg x$	$\neg y$	$\neg x \vee \neg y$
0	0	0	1	1	1	1
1	0	0	1	0	1	1
0	1	0	1	1	0	1
1	1	1	0	0	0	0

3.2 Propriétés des opérateurs logiques

On observe d'abord quelques propriétés fondamentales des opérateurs $\wedge$, $\vee$ et $\neg$:

Proposition 4 *On a les équivalences suivantes :*

1. $x \wedge x \models\!\!\!\models x$
2. $x \wedge y \models\!\!\!\models y \wedge x$
3. $x \wedge (y \wedge z) \models\!\!\!\models (x \wedge y) \wedge z$
4. $x \vee x \models\!\!\!\models x$
5. $x \vee y \models\!\!\!\models y \vee x$
6. $x \vee (y \vee z) \models\!\!\!\models (x \vee y) \vee z$
7. $\neg\neg x \models\!\!\!\models x$

On dit aussi que les opérateurs $\wedge$ et $\vee$ sont *idempotents* (1 et 4), *commutatifs* (2 et 5), et *associatifs* (3 et 6). Il s'agit ici de propriétés importantes d'opérations mathématiques en général. Par exemple, les opérations d'addition et de multiplication des entiers sont commutatifs et associatifs, mais pas idempotent.

Exercice 6 *Dire pour chacun des opérateurs (sur les entiers) suivants s'il est idempotent, commutatif, associatif :*

1. *le maximum de deux entiers, défini par* $x \barwedge y = \text{maximum}(x, y)$
2. *la moyenne de deux entiers, défini par* $x \bowtie y = \frac{x+y}{2}$
3. *l'opérateur de* projection gauche, *défini par* $x \triangleleft y = x$

Démonstration: Par table de vérité pour les trois premier équivalences, les trois équivalences pour $\vee$ et la dernière équivalence pour la double négation se montrent de la même façon.

1. $x \wedge x \models\!\!\!\models x$:

x	$x \wedge x$
0	0
1	1

2. $x \wedge y \models\!\!\!\models y \wedge x$:

x	y	$x \wedge y$	$y \wedge x$
0	0	0	0
1	0	0	0
0	1	0	0
1	1	1	1

3. $x \wedge (y \wedge z) \models\!\dashv (x \wedge y) \wedge z$:

x	y	z	$y \wedge z$	$x \wedge (y \wedge z)$	$x \wedge y$	$(x \wedge y) \wedge z$
0	0	0	0	0	0	0
1	0	0	0	0	0	0
0	1	0	0	0	0	0
1	1	0	0	0	1	0
0	0	1	0	0	0	0
1	0	1	0	0	0	0
0	1	1	1	0	0	0
1	1	1	1	1	1	1

□

3.3 Obtenir des tautologies et des équivalences par instanciation

Maintenant on souhaite généraliser les équivalences de la proposition 4 à des formules quelconques, par exemple $p \wedge p \models\!\dashv p$, $p \wedge q \models\!\dashv q \wedge p$ pour toutes formules propositionnelles p et q. Le théorème de substitution au-dessous (théorème 5) nous permet d'obtenir à partir d'une seule équivalence une telle loi qui est valide pour toute formule p, q, ...

Définition 7 *Soit x une variable propositionnelle et p une formule propositionnelle. La fonction de* substitution de x par p, *noté* $[x/p]$: *Form* $\to$ *Form, est définie par récurrence comme suit (l'application de cette fonction à un argument q est notée $q[x/p]$) :*

1. $y[x/p] = \begin{cases} p & si\ x = y \\ y & si\ x \neq y \end{cases}$
2. $(\neg q)[x/p] = \neg(q[x/p])$
3. $(q_1 \wedge q_2)[x/p] = (q_1[x/p]) \wedge (q_2[x/p])$
4. $(q_1 \vee q_2)[x/p] = (q_1[x/p]) \vee (q_2[x/p])$

Par exemple, si

$$p \quad = \quad z \vee \neg y$$

alors

$$(x \wedge \neg x \wedge y)[x/p] \quad = \quad (z \vee \neg y) \wedge \neg(z \vee \neg y) \wedge y$$

Cette définition se généralise facilement à une *substitution simultanée* $q[x_1/p_1, \ldots, x_n/p_n]$ pour le cas où les $x_1, \ldots, x_n$ sont toutes des variables différentes. Par exemple, si

$$\begin{aligned} p_1 &= (y_1 \wedge \neg y_2) \\ p_2 &= (z_1 \vee (z_2 \wedge z_3)) \end{aligned}$$

alors on a que

$$(x_1 \wedge x_2)[x_1/p_1, x_2/p_2] = (y_1 \wedge \neg y_2) \wedge (z_1 \vee (z_2 \wedge z_3))$$

Attention, on n'a pas toujours que $q[x_1/p_1, x_2/p_2] = (q[x_1/p_1])[x_2/p_2]$. Un contre-exemple est obtenu on choisissant $q = x_1$, $p_1 = (x_2 \wedge x_2)$, et $p_2 = (x_3 \vee x_3)$:

$$\begin{array}{rcl} x_1[x_1/(x_2 \wedge x_2), x_2/(x_3 \vee x_3)] & = & (x_2 \wedge x_2) \\ (x_1[x_1/(x_2 \wedge x_2)])[x_2/(x_3 \vee x_3)] & = & (x_2 \wedge x_2)[x_2/(x_3 \vee x_3)] \\ & = & ((x_3 \vee x_3) \wedge (x_3 \vee x_3)) \end{array}$$

La notion de substitution existe aussi au niveau des affectations :

Définition 8 *Soit v une affectation, x une variable propositionnelle, et $b \in \{0, 1\}$. Alors $v[x/b]$ est l'affectation définie comme suit :*

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

Cette définition se généralise aussi à une *substitution simultanée* $v[x_1/b_1, \ldots, x_n/b_n]$ pour le cas où les $x_1, \ldots, x_n$ sont toutes des variables différentes. Par exemple,

$$[x \mapsto 0, y \mapsto 1][y/0, z/1] = [x \mapsto 0, y \mapsto 0, z \mapsto 1]$$

La proposition suivante met le lien entre les deux notions de substitution.

Proposition 5 *Pour toute formule q, variables différentes $x_1, \ldots, x_n$, formules $p_1, \ldots, p_n$, et affectation v on a que*

$$[\![q[x_1/p_1, \ldots, x_n/p_n]]\!]v = [\![q]\!](v[x_1/[\![p_1]\!]v, \ldots, x_n/[\![p_n]\!]v])$$

En d'autres mots : On obtient le même résultat

1. quand on substitue les x_i par les p_i correspondant dans la formule q et puis évalue la formule ainsi obtenue par rapport à l'affectation v ;
2. quand on évalue d'abord les formules p_i une par une par rapport à l'affectation v, puis on met à jour dans l'affectation v les valeurs des variables x_i par l'interprétation des p_i, et on évalue la formule q originale par rapport à la nouvelle affectation.

Démonstration: Nous donnons ici la preuve pour le cas $n = 1$, c'est-à-dire nous montrons

$$[\![q[x/p]]\!]v = 1 \text{ ssi } [\![q]\!](v[x/[\![p]\!]v]) = 1$$

La preuve se fait par induction structurelle :

1. Variable : il y a deux cas :
   (a) la variable x :

$[\![x[x/p]]\!]v = 1$	
ssi $[\![p]\!]v = 1$	par définition 7
ssi $(v[x/[\![p]\!]v])(x) = 1$	par définition 8
ssi $[\![x]\!](v[x/[\![p]\!]v]) = 1$	par définition 3

(b) une variable $y \neq x$:

$[\![y[x/p]]\!]v = 1$
ssi $[\![y]\!]v = 1$ par définition 7
ssi $v(y) = 1$ par définition 3
ssi $v[x/[\![p]\!]v](y) = 1$ par définition 8
ssi $[\![y]\!](v[x/[\![p]\!]v]) = 1$ par définition 3

2. Négation :

$[\![(\neg q)[x/p]]\!]v = 1$
ssi $[\![\neg(q[x/p])]\!]v = 1$ par définition 7
ssi $[\![q[x/p]]\!]v = 0$ par définition 3
ssi $[\![q]\!](v[x/[\![p]\!]v]) = 0$ par hypothèse d'induction
ssi $[\![\neg q]\!](v[x/[\![p]\!]v]) = 1$ par définition 3

3. Conjonction :

$[\![(q_1 \wedge q_2)[x/p]]\!]v = 1$
ssi $[\![q_1[x/p] \wedge q_2[x/p])]\!]v = 1$ par définition 7
ssi $[\![q_1[x/p]]\!]v = [\![q_2[x/p]]\!]v = 1$ par définition 3
ssi $[\![q_1]\!](v[x/[\![p]\!]v]) = [\![q_2]\!](v[x/[\![p]\!]v]) = 1$ par hypothèse d'induction
ssi $[\![q_1 \wedge q_2]\!](v[x/[\![p]\!]v]) = 1$ par définition 3

4. Disjonction : similaire au cas précédent.

□

Théorème 5 *Soit q une tautologie, $x_1, \ldots, x_n$ des variables propositionnelles différentes, et $p_1, \ldots, p_n$ des formules propositionnelles. Alors*

$$q[x_1/p_1, \ldots, x_n/p_n]$$

est aussi une tautologie.

Démonstration: Nous devrons montrer que pour toute affectation v,

$$[\![q[x_1/p_1, \ldots, x_n/p_n]]\!]v = 1$$

Or, d'après proposition 5,

$$[\![q[x_1/p_1, \ldots, x_n/p_n]]\!]v = [\![q]\!](v[x_1/[\![p_1]\!]v, \ldots, x_n/[\![p_n]\!]v])$$

Puisque q est une tautologie,

$$[\![q]\!](v[x_1/[\![p_1]\!]v, \ldots, x_n/[\![p_n]\!]v]) = 1$$

puisque $[\![q]\!]v' = 1$ pour *toute* affectation v'. □

Il y a aussi une variante de ce théorème qui nous permet d'obtenir de nouvelles équivalences à partir d'une équivalence connue :

Théorème 6 *Soient q_1, q_2 deux formules telles que $q_1 \models\!\models q_2$, $x_1, \ldots, x_n$ des variables propositionnelles différentes, et $p_1, \ldots, p_n$ des formules propositionnelles. Alors*

$$q_1[x_1/p_1, \ldots, x_n/p_n] \models\!\models q_2[x_1/p_1, \ldots, x_n/p_n]$$

Exercice 7 *Montrer le théorème 6.*

Le théorème suivant donne quelques lois de la logique propositionnelle :

Théorème 7 *On a les équivalences suivantes pour toutes les formules p, q, r :*

$p \wedge p \models\!\!\!\models p$	*Loi d'idempotence de la conjonction*
$p \wedge q \models\!\!\!\models q \wedge p$	*Loi de commutativité de la conjonction*
$p \wedge (q \wedge r) \models\!\!\!\models (p \wedge q) \wedge r$	*Loi d'associativité de la conjonction*
$p \vee p \models\!\!\!\models p$	*Loi d'idempotence de la disjonction*
$p \vee q \models\!\!\!\models q \vee p$	*Loi de commutativité de la disjonction*
$p \vee (q \vee r) \models\!\!\!\models (p \vee q) \vee r$	*Loi d'associativité de la disjonction*
$\neg\neg p \models\!\!\!\models p$	*Loi de la double négation*

Démonstration: Conséquence immédiate de la proposition 4 et du théorème 6. □

Le théorème suivant donne des lois qui mettent plusieurs opérateurs logiques en relation :

Théorème 8 *On a les équivalences suivantes pour toutes les formules p, q, r :*

$(p \wedge q) \vee r \models\!\!\!\models (p \vee r) \wedge (q \vee r)$	*Première loi de distributivité*
$(p \vee q) \wedge r \models\!\!\!\models (p \wedge r) \vee (q \wedge r)$	*Seconde loi de distributivité*
$\neg(p \wedge q) \models\!\!\!\models \neg p \vee \neg q$	*Première loi de de Morgan*
$\neg(p \vee q) \models\!\!\!\models \neg p \wedge \neg q$	*Seconde loi de de Morgan*

Démonstration: Comme la preuve du théorème 7. Les équivalences à la base des deux lois de distributivité seront montrées en TD. L'équivalence utilisée pour la preuve de la première loi de de Morgan était montré sur la page 28. □

3.4 Obtenir des équivalences par mise sous contexte

Nous avons donc une première méthode pour obtenir des lois de la logique propositionnelle, c'est de substituer dans des équivalences préalablement établies des variables par des formules quelconques. Une deuxième méthode est comme suit : On part d'une équivalence préalablement établie $p \models\!\!\!\models q$ et on construit une nouvelle équivalence en remplaçant dans une formule quelconque une variable, disons x, une fois par p et une fois par q. Les deux formules obtenues sont aussi équivalentes. Cette méthode se généralise à des substitutions simultanées :

Théorème 9 *Soient $p_1 \models\!\!\!\models q_1, \ldots, p_n \models\!\!\!\models q_n$, p une formule, et $x_1, \ldots, x_n \subseteq \mathcal{V}(p)$ des variables différentes. Alors*

$$p[x_1/p_1, \ldots, x_n/p_n] \models\!\!\!\models p[x_1/q_1, \ldots, x_n/q_n]$$

Démonstration: Il faut montrer que pour toute affectation v on a que

$$[\![p[x_1/p_1, \ldots, x_n/p_n]]\!]v = [\![p[x_1/q_1, \ldots, x_n/q_n]]\!]v$$

Puisque $p_i \models\!\!\!\models q_i$ pour tout i on a aussi que $[\![p_i]\!]v = [\![q_i]\!]v$ pour tout i. On obtient la chaîne d'égalités suivante :

$$
\begin{array}{lll}
 & [\![p[x_1/p_1,\ldots,x_n/p_n]]\!]v & \\
= & [\![p]\!](v[x_1/[\![p_1]\!]v,\ldots,x_n/[\![p_n]\!]v]) & \text{par proposition 5} \\
= & [\![p]\!](v[x_1/[\![q_1]\!]v,\ldots,x_n/[\![q_n]\!]v]) & \text{car } [\![p_i]\!]v = [\![q_i]\!]v \text{ pour tout } i \\
= & [\![p[x_1/q_1,\ldots,x_n/q_n]]\!]v & \text{par proposition 5}
\end{array}
$$

□

Nous savons par exemple, selon le théorème 8, que

$$
\begin{array}{ccc}
\underbrace{\neg(y_1 \wedge y_2)}_{p_1} & \models\!\!\!\models & \underbrace{\neg y_1 \vee \neg y_2}_{q_1} \\
\underbrace{\neg(y_1 \vee y_2)}_{p_2} & \models\!\!\!\models & \underbrace{\neg y_1 \wedge \neg y_2}_{q_2}
\end{array}
$$

Avec la formule $p = (\neg x_1 \wedge x_2)$ on obtient la nouvelle équivalence suivante :

$$
\underbrace{\neg\neg(y_1 \wedge y_2) \wedge \neg(y_1 \vee y_2)}_{p[x_1/p_1,x_2/p_2]} \models\!\!\!\models \underbrace{\neg(\neg y_1 \vee \neg y_2) \wedge (\neg y_1 \wedge \neg y_2)}_{p[x_1/q_1,x_2/q_2]}
$$

3.5 Autres opérateurs booléens

Il est souvent pratique d'utiliser d'autre opérateurs booléens que les trois opérateurs $\neg$, $\wedge$, $\vee$ que nous avons considérés jusqu'à maintenant. Ces opérateurs sont définis comme des *abréviations*. La définition de la syntaxe des formules reste donc inchangée, mais on autorise dans la suite pour l'écriture des formules les abréviations suivantes :

Opérateur	Nom	Définition
$\rightarrow$	Implication	$x \rightarrow y \quad = \quad \neg x \vee y$
$\leftrightarrow$	Équivalence	$x \leftrightarrow y \quad = \quad (\neg x \vee y) \wedge (\neg y \vee x)$
`True`	Constante vrai	`True` $\quad = \quad x \vee \neg x$
`False`	Constante faux	`False` $\quad = \quad x \wedge \neg x$
$\oplus$	Ou exclusif	$x \oplus y \quad = \quad (x \wedge \neg y) \vee (\neg x \wedge y)$
$\uparrow$	Nand	$\mathrm{x} \uparrow y \quad = \quad \neg(x \wedge y)$
$\downarrow$	Nor	$\mathrm{x} \downarrow y \quad = \quad \neg(x \vee y)$

« Nand » et « Nor » sont des noms anglais (contraction de Not-And et Not-Or), et on dit parfois « xor » à la place de « Ou exclusif ». On pourrait aussi imaginer des opérateurs booléens avec plus que deux arguments.

Par exemple,

$$(x \oplus y) \rightarrow (x \vee y)$$

est une abréviation pour

$$\neg((x \wedge \neg y) \vee (\neg x \wedge y)) \vee (x \vee y)$$

Ces nouveaux opérateurs nous permettent d'écrire des nouvelles lois de la logique propositionnelle, comme par exemple :

$\texttt{True} \wedge p \models\!\dashv p$	`True` est l'élément neutre de la conjonction
$\texttt{False} \vee p \models\!\dashv p$	`False` est l'élément neutre de la disjonction
$\texttt{False} \wedge p \models\!\dashv \texttt{False}$	`False` est l'élément nul de la conjonction
$\texttt{True} \vee p \models\!\dashv \texttt{True}$	`True` est l'élément nul de la disjonction
$p \to q \models\!\dashv \neg q \to \neg p$	Loi de la contraposition
$p \leftrightarrow q \models\!\dashv (p \to q) \wedge (q \to p)$	
$p \leftrightarrow q \models\!\dashv \neg p \leftrightarrow \neg q$	
$p \to (q \to r) \models\!\dashv (p \wedge q) \to r$	

Toutes ces lois sont montrées facilement à l'aide de tables de vérité.

Ces abréviations sont utiles mais pas strictement nécessaires car on peut toujours les remplacer par leur définition. Dans ce contexte on peut se poser la question : est-ce que l'ensemble des opérateurs booléens dans la définition de la syntaxe de la logique propositionnelle (définition 1) était le seul, ou même le bon choix ? Une première remarque est qu'on aurait pu définir la logique propositionnelle seulement avec les opérateurs $\neg$ et $\wedge$ car on peut exprimer $\vee$ par ces deux opérateurs :

$$\begin{array}{rcll} x \vee y & \models\!\dashv & \neg\neg(x \vee y) & \text{Loi de la double négation} \\ & \models\!\dashv & \neg(\neg x \wedge \neg y) & \text{Seconde loi de de Morgan} \end{array}$$

On aurait également pu choisir les opérateurs $\neg$ et $\vee$ pour la définition de la logique propositionnelle car on peut de façon analogue définir $\wedge$ par $\neg$ et $\vee$. Nous avons suivi dans notre choix la définition traditionnelle de la logique. Ce choix a l'avantage de faire paraître la « dualité » entre la conjonction $\wedge$ et la disjonction $\vee$ qu'on peut constater dans les lois de la logique propositionnelle : Remarquer la symétrie entre $\wedge$ et $\vee$ dans les théorèmes 7 et 8.

On peut aller plus loin et se poser la question : aurait-on pu définir la logique propositionnelle avec un autre choix d'opérateurs parmi $\neg$, $\wedge$, $\vee$ et ceux qui sont dans la liste des opérateurs au-dessus (ou encore des autres opérateurs qui ne sont pas dans cette liste) ? Un choix possible est par exemple $\neg$ et $\to$ car on peut définir $x \vee y$ par $\neg x \to y$: $\neg x \to y$ est une abréviation pour $\neg\neg x \vee y$, ce qui est équivalent à $x \vee y$ par la loi de la double négation. Par contre, le choix des opérateurs $\wedge$ et $\vee$ ne fait pas l'affaire (ce qui n'est pas complètement évident). Il peut aussi être intéressant de savoir s'il y a un opérateur qui est tout seul suffisant pour exprimer tout ce qu'on peut exprimer avec $\neg$, $\wedge$ et $\vee$ (la réponse est « oui »). Ces questions seront discutées en TD.

Finalement on peut se poser la question si notre choix d'opérateurs $\neg$, $\wedge$, $\vee$ est suffisant pour exprimer tout ce qu'on peut souhaiter exprimer. On peut formaliser cette question comme suit : Est-ce qu'on peut pour tout nombre naturel n, et pour toute fonction f

$$f : \underbrace{\{0,1\} \times \ldots \times \{0,1\}}_{n \text{ fois}} \to \{0,1\}$$

trouver une formule propositionnelle p, avec $\mathcal{V}(p) \subseteq \{x_1, \ldots, x_n\}$ qui « réalise » f, c'est-à-dire

$$[\![p]\!][x_1 \mapsto b_1, \ldots, x_n \mapsto b_n] = f(b_1, \ldots, b_n)$$

pour toutes les valeurs booléennes $b_1, \ldots, b_n \in \{0,1\}$?

La réponse est « oui » : On peut représenter chaque choix des n arguments par une formule, par exemple le choix $c = (1,0,0,1) \in \{0,1\}^4$ est représenté par la formule $p_c = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4$.

Cette formule est vraie dans une affectation v si et seulement si $v(x_1) = 1, v(x_2) = 0, v(x_3) = 0$, et $v(x_4) = 1$. Maintenant on peut construire la formule p comme la disjonction de toutes les formules p_c pour les n-uplets c pour lesquelles f donne le résultat 1.

Exemple : Soit f la fonction à trois arguments qui envoie 1 si et seulement si un des ses arguments est 0 et deux de ses arguments sont 1. Autrement dit, f envoie 1 exactement pour les arguments $(0, 1, 1)$, $(1, 0, 1)$, et $(1, 1, 0)$. La formule correspondante est alors

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

On dit que l'ensemble d'opérateurs $\{\neg, \wedge, \vee\}$ est *fonctionnellement complet*. D'après la discussion de cette section, les ensembles $\{\neg, \wedge\}$, $\{\neg, \vee\}$ et $\{\neg, \rightarrow\}$ sont également fonctionnellement complets.

Proposition 6 *Les deux énoncés suivants sont équivalents pour toutes formules propositionnelles p et q :*

1. $p \models q$
2. $\models p \rightarrow q$

Exercice 8 *Montrer proposition 6.*

Attention, proposition 6 ne veut pas dire que « $p \models q$ » et « $p \rightarrow q$ » sont la même chose. En fait les deux sont d'une nature différente : Le deuxième est une formule propositionnelle, tandis que le premier est un énoncé qui a comme sujet la relation entre deux formules propositionnelles.

3.6 Références et remarques

Les axiomes de la proposition 4 furent trouvées par le mathématicien et philosophe *George Boole* (1815 - 1865). Les lois de de Morgan sont nommées après le logicien Britannique *Augustus de Morgan* (1806 - 1871).

Chapitre 4

Formes normales dans la logique propositionnelle

4.1 La notion d'une forme normale

Nous avons vu dans la section précédente qu'il existe un grand nombre de possibilités pour exprimer une formule dans une forme équivalente. En fait, pour toute formule p il y a une infinité de formules équivalentes : p, $p \wedge p$, $p \wedge p \wedge p$, ..., et d'autres pour lesquelles l'équivalence peut être beaucoup moins évidente.

On peut donc se poser des questions : a-t-on vraiment besoin de toute cette multitude de possibilités d'exprimer la même chose ? Ne serait-il pas mieux d'exiger une forme « standardisée » des formules ? Quand les informaticiens travaillent avec des expressions symboliques (comme par exemple des formules propositionnelles, mais il y a encore des autres) ils définissent souvent un tel standard, appelé une « forme normale ». Nous allons étudier plusieurs notions de formes normales des formules propositionnelles dans cette section. Toute notion de forme normale a ses propriétés spécifiques, mais on attend habituellement d'une notion de forme normale qu'on peut transformer n'importe quelle formule p (ou en général, n'importe quelle expression symbolique) en une formule équivalente en forme normale, appelée une *forme normale de p*.

4.2 Forme normale de négation

Notre première notion de forme normale restreint l'utilisation des négations dans les formules :

Définition 9 *Une formule est en* forme normale de négation *si elle ne contient pas d'applications de l'opérateur $\neg$ sauf des applications à des variables propositionnelles.*

Par exemple, $\neg x \wedge \neg y \wedge z$ et $(\neg x \vee y) \wedge (\neg z \vee x)$ sont en forme normale de négation, mais $\neg(x \wedge y)$ et $\neg\neg y$ ne le sont pas.

Nous donnons un algorithme pour transformer une formule en une formule équivalente en forme normale de négation. Cet algorithme est donné dans la forme de règles de *réécriture* :

$$\neg\neg X \quad \rightarrow \quad X \tag{4.1}$$

$$\neg(X \wedge Y) \quad \rightarrow \quad (\neg X \vee \neg Y) \tag{4.2}$$
$$\neg(X \vee Y) \quad \rightarrow \quad (\neg X \wedge \neg Y) \tag{4.3}$$

Chacune des ces règles est une règle de transformation. On peut appliquer une telle règle à n'importe quel endroit d'une formule, c'est-à-dire soit à la formule entière (on parle dans ce cas aussi d'une application *à la racine du terme*), ou on peut l'appliquer seulement à une sous-formule. Par exemple, la première règle (4.1) permet les transformations suivantes :
- Transformer la formule $\neg\neg(x_1 \wedge x_2)$ en $x_1 \wedge x_2$, en appliquant la règle à la formule entière ;
- Transformer la formule $(x_1 \vee (\neg\neg x_2))$ en $(x_1 \vee x_2)$, en appliquant la règle de transformation seulement à la sous-formule $\neg\neg x_2$.

Remarquez que les règles de transformation sont écrites avec des variables en majuscules X, Y et pas avec des variables en minuscules x, y comme les variables propositionnelles. La raison est qu'il s'agit ici de variables différentes : les variables propositionnelles dénotent des valeurs de vérité, tandis que les variables utilisées dans les règles de transformation dénotent des formules propositionnelles. Pour donner une analogie, pensez à un programme qui manipule des formules propositionnelles, comme par exemple les programmes en Java que vous allez développer en TP et pour votre projet : les variables de votre programme Java sont d'une nature complètement différente que les variables propositionnelles des formules.

Pour transformer une formule en forme normale de négation en lui applique les règles de transformation autant que possible. Par exemple,

	$\neg(x \wedge (y \vee \neg z))$	
se transforme en	$\neg x \vee \neg(y \vee \neg z)$	par règle (4.2)
se transforme en	$\neg x \vee (\neg y \wedge \neg\neg z)$	par règle (4.3)
se transforme en	$\neg x \vee (\neg y \wedge z)$	par règle (4.1)

Est-ce que ce processus se termine toujours ? Ce n'est pas évident, un processus défini par une application itérée d'une transformation peut a priori boucler, ou faire grossir le terme infiniment.

Imaginons d'abord un système de transformation qui consiste en un seule règle : $\neg\neg X \rightarrow X$. Il est facile de montrer que ce système termine : Chaque application de cette règle enlève deux occurrences du symbole $\neg$ de la formule. Si la formule de départ a n occurrences du symbole $\neg$ alors on peut lui appliquer la règle au plus $\frac{n}{2}$ fois.

L'argument de terminaison est donc qu'il y a une mesure (le nombre d'occurrences du symbole $\neg$) qui décroît à chaque transformation. Puisque la mesure ne peut pas passer au-dessous de 0 cela garantie la terminaison.

Comment appliquer cette idée au système complet des règles 4.1, 4.2 et 4.3 ? Le problème est que les règles 4.2 et 4.3 font *accroître* le nombre de $\neg$ au lieu de le faire décroître. Compter le nombre de $\wedge$, ou le nombre de $\vee$, dans la formule ne permet pas non plus de démontrer la terminaison car chacun de ces nombres peut accroître ou décroître, selon la règle précise appliquée. Finalement, la taille de la formule peut également accroître, au lieu de décroître : ces le cas avec les règles 4.2 et 4.3.

ne fait pa ni la taille de la formule. La solution est la proposition suivante :

Proposition 7 *Soit $\phi : Form \rightarrow \mathbb{N}$ définie par récurrence :*
- $\phi(x) = 1$, *où* $x \in V$
- $\phi((p_1 \wedge p_2)) = \phi(p_1) + \phi(p_2)$
- $\phi((p_1 \vee p_2)) = \phi(p_1) + \phi(p_2)$

– $\phi(\neg p) = (\phi(p))^2 + 1$

Alors pour toute formule $p \in Form$: Si p se transforme en q par une application d'une des règles 4.1, 4.2 or 4.3 alors $\phi(p) > \phi(q)$.

L'originalité de la fonction ϕ est l'utilisation d'une expression quadratique dans le cas de la négation.

Exercice 9 *Montrer par induction pour la fonction ϕ de la proposition 7 que $\phi(p) \geq 1$ pour toute formule p.*

Démonstration: On a que $\phi(p) \geq 1$ pour toute formule p (voir exercice 9).

La preuve est faite par une induction sur p. Pour un p donné il faut montrer que *si* p se transforme en q par une application d'une règle alors $\phi(p) > \phi(q)$. En particulier si aucune règle de transformation s'applique à p alors il n'y a rien montrer. Il ne faut pas oublier que dans le cas général il y a plusieurs possibilités pour appliquer une règle : il faut prendre en considération toutes ces possibilités.

– Si $p \in V$ alors aucune transformation n'est possible et il n'y a rien à montrer.
– Si $p = (p_1 \wedge p_2)$ alors il peut exister plusieurs possibilités pour appliquer une règle, cela dépend de p_1 et de p_2. Chaque transformation se fait soit sur p_1 soit sur p_2, il n'est pas possible d'appliquer une règle à la racine de p car toute règle commence sur le symbole $\neg$. Considérons le cas où la transformation se fait en p_1 (le cas pour p_2 est analogue). On a donc que p_1 se transforme en q_1, et $p = (p_1 \wedge p_2)$ se transforme par conséquent en $q = (q_1 \wedge p_2)$. On obtient

$$\begin{array}{lll} & \phi((p_1 \wedge p_2)) & \\ = & \phi(p_1) + \phi(p_2) & \text{par définition de } \phi \\ > & \phi(q_1) + \phi(p_2) & \text{car } \phi(p_1) > \phi(q_1) \text{ par hypothèse d'induction} \\ = & \phi((q_1 \wedge p_2)) & \text{par définition de } \phi \end{array}$$

– Le cas $p = (p_1 \vee p_2)$ est analogue au cas précédent.
– Si $p = \neg p_1$ il y a quatre cas à considérer :

1. La transformation se fait sur p_1 (qui est transformé en q_1), on conclut comme au-dessus en appliquant l'hypothèse d'induction :

$$\begin{array}{lll} & \phi(\neg p_1) & \\ = & (\phi(p_1))^2 + 1 & \text{par définition de } \phi \\ > & (\phi(q_1))^2 + 1 & \text{car } \phi(p_1) > \phi(q_1) \text{ par hypothèse d'induction et } \phi(q_1) \geq 1 \\ = & \phi(\neg q_1) & \text{par définition de } \phi \end{array}$$

2. Il s'agit d'une application de la règle 4.1 à la racine : On a donc que $p = \neg\neg p_2$, et p est transformé en p_2 :

$$\begin{array}{lll} & \phi(\neg\neg p_2) & \\ = & (\phi(\neg p_2))^2 + 1 & \text{par définition de } \phi \\ = & ((\phi(p_2))^2 + 1)^2 + 1 & \text{par définition de } \phi \\ = & \phi(p_2)^4 + 2\phi(p_2)^2 + 1 + 1 & \\ > & \phi(p_2) & \end{array}$$

3. Il s'agit d'une application de la règle 4.2 à la racine : On a donc $p = \neg(p_2 \wedge p_3)$, qui est transformé en $(\neg p_2 \vee \neg p_3)$:

$$
\begin{array}{lll}
 & \phi(\neg(p_2 \wedge p_3)) & \\
= & \phi((p_2 \wedge p_3))^2 + 1 & \text{par définition de } \phi \\
= & (\phi(p_2) + \phi(p_3))^2 + 1 & \text{par définition de } \phi \\
= & (\phi(p_2))^2 + 2\phi(p_2)\phi(p_3) + (\phi(p_3))^2 + 1 & \\
> & (\phi(p_2))^2 + 1 + (\phi(p_3))^2 + 1 & \text{car } \phi(p_2) \geq 1 \text{ et } \phi(p_3) \geq 1 \\
= & \phi(\neg p_2) + \phi(\neg p_3) & \text{par définition de } \phi \\
= & \phi((\neg p_2 \vee \neg p_3)) & \text{par définition de } \phi
\end{array}
$$

4. Il s'agit d'une application de la règle 4.3 à la racine : analogue au cas précédent. □

Le théorème suivant dit qu'on obtient toujours à la fin de ce processus une formule équivalente en forme normale de négation. Remarquez que cette description du processus laisse souvent des libertés de choix car nous n'avons pas spécifié quoi faire quand il y a plusieurs des règles qui peuvent s'appliquer à une formule, comme par exemple pour la formule

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

Le théorème ne fait pas d'hypothèse sur la stratégie d'application des règles dans de tels cas. L'avantage d'une description si générale d'une méthode de transformation est qu'un programmeur a le choix d'implanter la stratégie qui lui semble la meilleure (parce qu'elle est la plus facile à mettre en œuvre, ou parce qu'une certaine stratégie mène plus rapidement au résultat qu'une autre).

Théorème 10 *Pour toute formule il existe une formule équivalente en forme normale de négation.*

Démonstration: Nous utilisons le système de transformation décrit au-dessus. La démonstration du fait que ce système nous permet d'obténir la forme normale de négation d'une formule donnée quelconque consiste en trois parties :

1. Le processus de transformation s'arrête toujours : pour une formule p donnée on arrive toujours après un nombre fini de transformations sur une formule qui ne peut plus être transformée.
2. La formule qu'on obtient à la fin est équivalente à la formule de départ.
3. La formule qu'on obtient à la fin est en forme normale de négation.

La première propriété est une conséquence de la proposition 7 : À chaque transformation la mesure ϕ décroît strictement. Cette mesure ne peut jamais descendre au-dessous de 1, si la formule de départ p a la mesure $\phi(p) = n$ alors on peut au maximum appliquer $n - 1$ transformations.

Pour la deuxième propriété il suffit de remarquer que toute règle transforme une formule donnée en une autre formule qui lui est équivalente. On a donc à la fin d'une séquence de transformations une formule qui est équivalente à la formule de départ. Formellement il s'agit ici bien entendu d'une preuve par induction sur la longueur de la suite de transformations. Pour voir que chacune des trois règles transforme une formule en une formule équivalente il suffit de se rapporter aux théorèmes 7 et 8.

Finalement, la formule qu'on obtient à la fin est une formule p sur laquelle aucune des trois règles de transformation s'applique (dans le cas contraire on n'est pas à la fin du processus). Supposons pour l'absurde que la formule p n'est pas en forme normale de négation. Il y a

donc en p une occurrence du symbole de négation qui n'est pas appliquée à une variable propositionnelle. Ils restent trois possibilités pour la sous-formule qui commence sur cette négation :

1. C'est une formule de la forme $\neg\neg q$: la règle (4.1) s'applique, contradiction
2. C'est une formule de la forme $\neg(q_1 \wedge q_2)$: la règle (4.2) s'applique, contradiction
3. C'est une formule de la forme $\neg(q_1 \vee q_2)$: la règle (4.3) s'applique, contradiction □

4.3 Forme disjonctive normale

Notre deuxième notion de forme normale, après la forme normale de négation, est la forme disjonctive normale.

Définition 10 *1. Un* littéral *est soit une variable propositionnelle, soit la négation d'une variable propositionnelle.*

2. Une clause conjonctive *est soit la constante* `True`*, soit un littéral, soit une conjonction d'aux moins deux littéraux.*

3. Une formule est en forme disjonctive normale *si elle est soit la constante* `False`*, soit une clause conjonctive, soit une disjonction d'aux moins deux clauses conjonctives.*

On écrit parfois plus brièvement DNF, c'est l'abréviation du nom anglais *disjonctive normal form.*

Par une « conjonction d'aux moins deux littéraux » nous entendons une formule de la forme

$$(l_1 \wedge (l_2 \wedge (l_3 \wedge \ldots) \ldots))$$

où chacun des l_i est un littéral, et par une « disjonction d'aux moins deux clauses conjonctives » une formule de la forme

$$(c_1 \vee (c_2 \vee (c_3 \vee \ldots) \ldots))$$

où chacun des c_i est une clause conjonctive. Quand on parle de clauses conjonctives et de formes disjonctives normales on utilise la syntaxe libérale introduite en section 2.5, et on inclut les constantes `True` et `False` dans la syntaxe. En fait on peut simplifier la définition si on admet que la disjonction (la conjonction) d'une seule formule est la formule elle même, et que la disjonction de zéro formules est `False` et la conjonction de zéro formules est `True`. La justification pour cette dernière convention est que la conjonction d'une formule w avec zéro d'autre formules doit donner une formule équivalente à w. Par conséquent, la conjonction de zéro formules est `True` car $w \wedge$ `True` est équivalent à w. De la même façon, la disjonction d'une formule w avec une disjonction de zéro d'autre formules doit donner une formule équivalente à w, la disjonction de zéro formules est donc `False` car $w \vee$ `False` est équivalent à w. Ou autrement dit, la conjonction de zéro formules est l'élément neutre de la conjonction (`True`) et la disjonction de zéro formules est l'élément neutre de la disjonction (`False`). On trouve des conventions similaires dans les Mathématiques, en arithmétique par exemple on dit que la somme de zéro nombres est 0 (l'élément neutre de l'addition) et que le produit de zéros nombres est 1 (l'élément neutre de la multiplication).

Avec cette convention, on peut donc dire :

1. Une *clause conjonctive* est une conjonction de zéro ou plus littéraux.
2. Une formule est en *forme disjonctive normale* si elle est la disjonction de zéro ou plus clauses conjonctives.

Évidemment toute formule en forme disjonctive normale est aussi en forme normale de négation, mais l'inverse n'est pas le cas. Par exemple, $x \wedge (\neg y \vee \neg z)$ est en forme normale de négation mais pas en forme disjonctive normale.

Pour transformer une formule en une formule équivalente en forme disjonctive normale on commence d'abord à la mettre en forme normale de négation selon la procédure décrite dans la preuve du théorème 10. Puis, on transforme la formule obtenue en forme normale de disjonction à l'aide du système de réécriture suivant :

$$X \wedge (Y \vee Z) \rightarrow (X \wedge Y) \vee (X \wedge Z) \quad (4.4)$$
$$(X \vee Y) \wedge Z \rightarrow (X \wedge Z) \vee (Y \wedge Z) \quad (4.5)$$
$$(X \wedge Y) \wedge Z \rightarrow X \wedge (Y \wedge Z) \quad (4.6)$$
$$(X \vee Y) \vee Z \rightarrow X \vee (Y \vee Z) \quad (4.7)$$

Par exemple,

	$x_1 \wedge (\neg(y_1 \vee y_2) \wedge \neg\neg(z_1 \vee z_2))$	
se transforme en	$x_1 \wedge (\neg(y_1 \vee y_2) \wedge (z_1 \vee z_2))$	par règle 4.1
se transforme en	$x_1 \wedge ((\neg y_1 \wedge \neg y_2) \wedge (z_1 \vee z_2))$	par règle 4.3
se transforme en	$x_1 \wedge (((\neg y_1 \wedge \neg y_2) \wedge z_1) \vee ((\neg y_1 \wedge \neg y_2) \wedge z_2))$	par règle 4.4
se transforme en	$x_1 \wedge ((\neg y_1 \wedge \neg y_2 \wedge z_1) \vee (\neg y_1 \wedge \neg y_2 \wedge z_2))$	par règle 4.6 (2 fois)
se transforme en	$(x_1 \wedge \neg y_1 \wedge \neg y_2 \wedge z_1) \vee (x_1 \wedge \neg y_1 \wedge \neg y_2 \wedge z_2)$	par règle 4.5

Proposition 8 *Soit $\psi : Form \rightarrow \mathbb{N}$ définie par récurrence :*
- *$\psi(x) = 2$ si $x \in V$*
- *$\psi((p_1 \wedge p_2)) = (\psi(p_1))^2 * \psi(p_2)$*
- *$\psi((p_1 \vee p_2)) = 2\psi(p_1) + \psi(p_2) + 1$*
- *$\psi(\neg p) = \psi(p)$*

Alors pour toute formule $p \in Form$: Si p se transforme en q par une application d'une des règles 4.4 à 4.7 alors $\psi(p) > \psi(q)$.

Exercice 10 *Montrer la proposition 8.*

Théorème 11 *Pour toute formule il existe une formule équivalente en forme disjonctive normale.*

Démonstration: Grâce au théorème 10 il existe pour toute formule une formule équivalente en forme normale de négation. Nous appliquons les règles de transformation 4.4 à 4.7 autant que possible à cette formule en forme normale de négation. Le reste de la preuve suit le même schéma que la preuve du théorème 10 : Nous devrons montrer que :

1. Le processus de transformation s'arrête toujours : pour une formule p donnée on arrive toujours après un nombre fini de transformations sur une formule qui ne peut plus être transformée.

2. La formule qu'on obtient à la fin est équivalente à la formule de départ.
3. La formule qu'on obtient à la fin est en forme disjonctive normale.

La terminaison est une conséquence de la proposition 7.

Nous avons vu dans la preuve du théorème 10 que la preuve du deuxième énoncé repose sur le fait que chacune des règles transforme toujours une formule en une formule équivalente. Pour les règles (4.1) à (4.3) nous l'avons déjà montré dans la preuve du théorème 10, pour les règles (4.4) et (4.5) c'est une conséquence directe des lois de distributivité (théorème 8), et pour les règles (4.6) et (4.7) une conséquence directe des deux lois d'associativité (théorème 7).

Il reste à montrer le troisième énoncé : Si on ne peut plus appliquer une des règles alors la formule est en forme disjonctive normale. Supposons par l'absurde que la formule p n'est pas en forme disjonctive normale. Il y a les possibilités suivantes :

1. p n'est pas en forme normale de négation : ce n'est pas possible car aucune des règles 4.4 à 4.7 peut transformer une formule en forme normale de négation en une formule qui n'est pas en forme normale de négation.
2. p contient une sous-formule $(p_1 \wedge p_2) \wedge p_3$ ou $(p_1 \vee p_2) \vee p_3$: Une des règles (4.6) ou (4.7) s'applique alors, contradiction.
3. Il y a en p une disjonction au-dessous d'une conjonction, c'est-à-dire p contient une sous-formule de la forme $(p_1 \wedge (p_2 \vee p_3))$ ou de la forme $((p_1 \vee p_2) \wedge p_3)$: Une des règles (4.4) ou (4.5) s'applique alors, contradiction. □

Attention, la transformation d'une formule en forme disjonctive normale risque de faire croître la taille de la formule de façon exponentielle. Par exemple, la mise en forme disjonctive normale de la formule

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2)$$

donne

$$\begin{array}{ll} & (x_1 \wedge x_2) \\ \vee & (x_1 \wedge y_2) \\ \vee & (y_1 \wedge x_2) \\ \vee & (y_1 \wedge y_2) \end{array}$$

et la mise en forme disjonctive normale de

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge (x_3 \vee y_3)$$

donne

$$\begin{array}{ll} & (x_1 \wedge x_2 \wedge x_3) \\ \vee & (x_1 \wedge x_2 \wedge y_3) \\ \vee & (x_1 \wedge y_2 \wedge x_3) \\ \vee & (x_1 \wedge y_2 \wedge y_3) \\ \vee & (y_1 \wedge x_2 \wedge x_3) \\ \vee & (y_1 \wedge x_2 \wedge y_3) \\ \vee & (y_1 \wedge y_2 \wedge x_3) \\ \vee & (y_1 \wedge y_2 \wedge y_3) \end{array}$$

et en général la mise en forme disjonctive normale de la formule

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \ldots \wedge (x_n \vee y_n)$$

donne une formule avec 2^n clauses conjonctives, chaque clause consistant en n littéraux.

Un avantage d'une forme disjonctive normale est qu'on peut voir facilement si la formule est satisfaisable ou pas.

Théorème 12 *Une formule en forme disjonctive normale*

$$(c_1 \vee c_2 \vee \ldots \vee c_n)$$

est satisfaisable si et seulement s'il y a un i tel que la clause c_i ne contient pas à la fois une variable x et aussi sa négation.

Démonstration: Évidemment une formule en forme disjonctive normale est satisfaisable si et seulement si elle contient au moins une clause conjonctive qui est satisfaisable (donc, il faut déjà que la formule contienne au moins une clause pour être satisfaisable). Il reste à montrer qu'une clause conjonctive

$$c = (l_1 \wedge l_2 \wedge \ldots \wedge l_m)$$

est satisfaisable si et seulement si elle ne contient pas à la fois un littéral x et aussi sa négation $\neg x$.

- Si c contient un littéral x et aussi $\neg x$ alors évidemment c n'est pas satisfaisable.
- Si c ne contient pas à la fois un littéral x et aussi $\neg x$ alors c est satisfait par l'affectation v définie par

$$v(x) = \begin{cases} 1 & \text{si } x \text{ est un littéral en } c \\ 0 & \text{si } \neg x \text{ est un littéral en } c, \text{ ou si } x \notin \mathcal{V}(c) \end{cases}$$

□

Nous avons expliqué à la fin de la section 2.4 qu'on connaît à ce jour aucun algorithme « efficace », c'est-à-dire dont le temps d'exécution ne croit pas de façon exponentielle avec la taille de la formule, pour décider la satisfaisabilité d'une formule. Or, le critère énoncé dans le théorème 12 est très simple à vérifier par un seul passage de la formule de gauche à droite. Est-ce qu'il n'y a pas une contradiction avec la difficulté perçue du problème de décider la satisfaisabilité d'une formule ? L'explication de se paradoxe apparent est très simple : Étant donnée une formule propositionnelle quelconque il faut d'abord la transformer en forme disjonctive normale, avant d'appliquer le critère du théorème 12 à la formule obtenue. Or, cette transformation implique un « gonflement » exponentiel de la taille de la formule, et l'exécution d'un algorithme efficace sur la formule de taille exponentielle se traduit alors en fin de compte toujours en un algorithme qui est exponentiel par rapport à la taille de la formule de départ.

4.4 Équivalences entre formules en forme disjonctive normale

On s'intéresse à la question suivante :

> Question 1 : Est-il possible que deux formules *différentes* en forme disjonctive normale soient équivalentes ?

La réponse à la question 1 est « oui » car les deux formules peuvent déjà différer dans l'ordre des littéraux dans les clauses conjonctives, ou dans l'ordre des clauses conjonctives dans la formule entière. Par exemple, les trois formules suivantes sont toutes en forme disjonctive normale, elles sont équivalentes mais elles sont toutes syntaxiquement différentes :

$$(x \wedge y) \vee (\neg z \wedge \neg y)$$
$$(y \wedge x) \vee (\neg z \wedge \neg y)$$
$$(\neg z \wedge \neg y) \vee (y \wedge x)$$

On pourrait dire que les différences entre ces trois formules ne sont pas vraiment essentielles, et on peut raffiner la question :

> Question 2 : Est-il possible que deux formules en forme disjonctive normale *avec plus de différence que simplement l'ordre des littéraux ou l'ordre des clauses*, soient équivalentes ?

La définition suivante va nous permettre de formuler un critère pour l'équivalence de deux formes disjonctives normales :

Définition 11 *Une clause conjonctive* $l_1 \wedge \ldots \wedge l_n$ subsume *une clause conjonctive* $k_1 \wedge \ldots \wedge k_m$ *si pour tout* i *avec* $1 \leq i \leq n$ *il existe un* j *avec* $1 \leq j \leq m$ *tel que* $k_j = l_i$.

Autrement dit, une clause c subsume une clause d si c est « incluse » en d à l'ordre des littéraux et à des éventuelles répétitions près. Par exemple, $x \wedge \neg y$ subsume $z \wedge x \wedge \neg y$, et aussi $z \wedge \neg y \wedge z \wedge x$. On a évidemment que

Proposition 9 *Si la clause conjonctive* c *subsume la clause conjonctive* d *alors* $d \models c$.

Par exemple $x \wedge \neg y \wedge z \models x \wedge \neg y$.

Théorème 13 *Soient* $c_1, \ldots, c_n$ *des clauses conjonctives. Si* c_j *subsume* c_i *pour* $i \neq j$ *alors*

$$c_1 \vee \ldots \vee c_{i-1} \vee c_i \vee c_{i+1} \vee \ldots \vee c_n \models\!\!\dashv c_1 \vee \ldots \vee c_{i-1} \vee c_{i+1} \vee \ldots \vee c_n$$

Autrement dit, on obtient une formule équivalente quand on supprime dans une forme disjonctive normale une clause qui est subsumée par une autre clause. Les clauses subsumées sont redondantes. La preuve est une simple application de la proposition 9. Par exemple,

$$(x \wedge \neg y) \vee (\neg x \wedge y) \vee (x \wedge \neg y \wedge z_1 \wedge \neg z_2) \models\!\!\dashv (x \wedge \neg y) \vee (\neg x \wedge y)$$

La réponse à la question 2 est que c'est toujours possible, comme on a vu sur l'exemple précédent. On peut donc essayer de raffiner encore la question :

> Question 3 : Est-il possible que deux formules en forme disjonctive normale avec plus de différence que simplement l'ordre des littéraux ou l'ordre des clauses, *et où aucune clause subsume une autre dans la même formule*, soient équivalentes ?

La réponse est toujours « oui ». Par exemple, les deux formules suivantes sont en forme disjonctive normale, équivalentes, mais leur différence syntaxique dépasse l'ordre des littéraux ou des clauses, et on ne peut pas supprimer une clause qui subsume une autre :

$$\neg x \vee (x \wedge \neg y) \tag{4.8}$$
$$\neg y \vee (y \wedge \neg x) \tag{4.9}$$

En fait, les deux formules sont équivalentes à

$$(\neg x \wedge y) \vee (\neg x \wedge \neg y) \vee (x \wedge \neg y) \tag{4.10}$$

ce qui est donc encore une formule en forme disjonctive normale et qui est équivalente aux formules (4.8) et (4.10), mais complètement différente. On voit par exemple l'équivalence de (4.8) et (4.10) comme suit :

$$\begin{array}{lll} & \neg x \vee (x \wedge \neg y) & \\ \models\!\models & (\neg x \wedge (y \vee \neg y)) \vee (x \wedge \neg y) & \texttt{True} \text{ est l'élément neutre de } \wedge \\ \models\!\models & (\neg x \wedge y) \vee (\neg x \wedge \neg y) \vee (x \wedge \neg y) & \text{de Morgan} \end{array}$$

En fait il y a encore une autre formule en forme disjonctive normale qui est équivalente a (4.8) à (4.10) et qui est encore plus petite :

$$\neg x \vee \neg y \tag{4.11}$$

Pour montrer cette équivalence nous avons besoin du critère de subsomption (théorème 13) :

$$\begin{array}{lll} & \neg x \vee \neg y & \\ \models\!\models & \neg x \vee (\neg y \wedge (x \vee \neg x)) & \texttt{True} \text{ est l'élément neutre de } \wedge \\ \models\!\models & \neg x \vee (\neg y \wedge x) \vee (\neg y \wedge \neg x) & \text{de Morgan} \\ \models\!\models & \neg x \vee (x \wedge \neg y) & \text{théorème 13, car } \neg x \text{ subsume } \neg y \wedge \neg x \end{array}$$

Il faut bien sûr préciser qu'est-ce qu'on entend par « plus petit » : Dans le contexte de cette section, c'est-à-dire pour la comparaison des formules en forme disjonctive normale, en compare les formules simplement par le nombre de littéraux qui paraissent dedans (si le même littéral parait plusieurs fois on compte toutes ces occurrences). Par exemple, les formules (4.8) et (4.9) ont chacune 3 littéraux, et la formule (4.10) a 6 littéraux.

On vois facilement qu'il n'y a pas de formule équivalente en forme disjonctive normale qui est encore plus petite, sauf $\neg y \vee \neg x$ qui est simplement une variante de (4.11) par commutativité. Une formule en forme disjonctive normale est dite *minimale* quand il n'y a pas d'autre formule en forme disjonctive normale qui est plus petite et qui lui est équivalente. Dans notre exemple on obtient donc que la forme disjonctive normale *minimale* est unique à des applications de commutativité et associativité près.

Le calcul d'une forme disjonctive normale minimale est d'une grande importance dans le domaine d'*architecture d'ordinateurs* où on s'intéresse au problème de réaliser un circuit d'une certaine fonctionnalité à moindre coût. L'algorithme le plus connu pour le calcul d'une forme disjonctive normale minimale porte le nom de ses deux inventeurs *Quine* et *McCluskey*.

On peut donc encore raffiner notre question :

> Question 4 : est-il possible que deux formules *minimales* en forme disjonctive normale avec plus de différence que simplement l'ordre des littéraux ou l'ordre des clauses, sont équivalentes ?

La réponse est toujours « oui »comme on peut voir sur l'exemple de la formule

$$(x \wedge y) \vee (\neg x \wedge \neg y) \vee (x \wedge \neg y \wedge z) \tag{4.12}$$

Cette formule est équivalente aux deux formules plus petites en forme disjonctive normale

$$(x \wedge y) \vee (\neg x \wedge \neg y) \vee (x \wedge z) \tag{4.13}$$
$$(x \wedge y) \vee (\neg x \wedge \neg y) \vee (\neg y \wedge z) \tag{4.14}$$

Ces deux formules ne sont pas de simples variantes par application de commutativité et d'associativité. De plus, il n'y a pas de formule équivalente en forme disjonctive normale qui soit plus petite.

Exercice 11 *Soit $p = (x \wedge y) \vee (\neg x \wedge \neg y) \vee (x \wedge z)$. Montrer*

1. *Pour aucune des formules q parmi x, $\neg x$, y, $\neg y$, z, $\neg z$ on a que $q \models p$.*
2. *Il n'y a aucune formule en forme disjonctive normale équivalente à p qui contient une clause avec un seul littéral.*
3. *Il n'y a aucune formule en forme disjonctive normale équivalente à p qui a moins que 6 littéraux.*

4.5 Forme conjonctive normale

La forme conjonctive normale est définie de façon analogue à la forme disjonctive normale :

Définition 12
1. *Une* clause disjonctive *est une disjonction de zéro ou plus littéraux.*
2. *Une formule est en* forme conjonctive normale *(abrégé CNF) si elle est la conjonction de zéro ou plus clauses disjonctives.*

Par exemple, $(x \vee y) \wedge (\neg z \vee y \vee \neg z)$ est en forme conjonctive normale. Remarquez qu'il y a des formules qui sont à la fois en forme disjonctive normale et en forme conjonctive normale, par exemple $(x \vee \neg y \vee z)$ qu'on peut voir comme une seule clause disjonctive et donc une formule en forme conjonctive normale ; ou comme une disjonction de trois clauses conjonctives qui chacune consiste en un seul littéral, et donc comme une formule en forme disjonctive normale.

Théorème 14 *Pour toute formule il existe une formule équivalente en forme conjonctive normale.*

La procédure est analogue à celle de la mise en forme disjonctive normale, sauf que les deux premières règles de transformation (4.4) et (4.5) sont à remplacer par

$$X \vee (Y \wedge Z) \quad \rightarrow \quad (X \vee Y) \wedge (X \vee Z) \tag{4.15}$$
$$(X \wedge Y) \vee Z \quad \rightarrow \quad (X \vee Z) \wedge (Y \vee Z) \tag{4.16}$$

On obtient également un correspondant au théorème 12 pour les formules en forme conjonctive normale :

Théorème 15 *Une formule en forme conjonctive normale*

$$(d_1 \wedge d_2 \wedge \ldots \wedge d_n)$$

est valide si et seulement si pour tout i il existe une variable x telle que la clause d_i contient à la fois le littéral x et aussi sa négation $\neg x$.

La notion de subsomption se présente comme suit pour des clauses disjonctives :

Définition 13 *Une clause disjonctive $l_1 \vee \ldots \vee l_n$* subsume *une clause disjonctive $k_1 \vee \ldots \vee k_m$ si pour tout i avec $1 \leq i \leq n$ il existe un j avec $1 \leq j \leq m$ tel que $k_j = l_i$.*

Par exemple, $x \vee \neg y$ subsume $x \vee \neg y \vee z$.

Proposition 10 *Si la clause disjonctive c subsume la clause disjonctive d alors $c \models d$.*

Remarquez que le sens de la conséquence est inverse à celui de la proposition 9.
Par exemple $x \vee \neg y \models x \vee \neg y \vee z$.

Théorème 16 *Soient $c_1, \ldots, c_n$ des clauses disjonctives. Si c_j subsume c_i pour $i \neq j$ alors*

$$c_1 \wedge \ldots \wedge c_{i-1} \wedge c_i \wedge c_{i+1} \wedge \ldots \wedge c_n \models\!\!\models c_1 \wedge \ldots \wedge c_{i-1} \wedge c_{i+1} \wedge \ldots \wedge c_n$$

Exercice 12 *Montrer Proposition 10 et Théorème 16.*

Ce théorème dit que, pour les formules en forme disjonctive normale, une clause subsumée est redondante et peut être supprimée. C'est donc pareil que pour les formules en forme disjonctive normale.

4.6 Références et remarques

Il y des autres questions intéressantes qu'on peut se poser sur les systèmes de transformation par règles, comme par exemple les systèmes donnés ci-dessus. Par exemple, si on a plusieurs possibilités d'appliquer une règle de transformation à une formule, est-ce qu'on arrive toujours à la fin sur la même formule ? Un exemple intéressant dans ce contexte est

$$\neg\neg(x \wedge y)$$

On peut soit appliquer la règle (4.1) et obtenir la formule $(x \wedge y)$ qui est en forme normale, soit appliquer la règle (4.2) à la position de la deuxième négation :

	$\neg\neg(x \wedge y)$	
se transforme en	$\neg(\neg x \vee \neg y)$	par règle (4.2)
se transforme en	$\neg\neg x \wedge \neg\neg y$	par règle (4.3)
se transforme en	$x \wedge y$	par règle (4.1) (deux fois)

Donc dans ce cas le résultat est le même. Est-ce toujours le cas ? La réponse est « oui » pour ce système de transformation, et on dit que le système est *confluent*. Mais ce n'est pas évident, et il y a des systèmes de règles qui n'ont pas cette propriété. Cette question de confluence, ainsi que des techniques de preuves de terminaison (comme la proposition 7), est étudiée dans le domaine des *systèmes de réécriture*, un sous-domaine de la *démonstration automatique*.

Chapitre 5

Satisfaisabilité de formes conjonctives normales

Nous avons vu dans la section précédente qu'il est très facile de vérifier la satisfaisabilité d'une formule donnée en forme disjonctive normale (théorème 12), ou encore de vérifier la validité d'une formule donnée en forme conjonctive normale (théorème 15). Malheureusement il y a beaucoup d'applications où on a un problème différent à résoudre : on a donné au départ une formule en forme conjonctive normale, ou au moins dans une forme qui peut être transformée facilement en forme conjonctive normale, et on souhaite décider de la *satisfaisabilité* de la formule. On peut par exemple penser à des problèmes de planification où il y a un grand nombre de contraintes à satisfaire en même temps (c'est donc une grande conjonction), et chaque contrainte consiste en un certain nombre d'alternatives possibles (autrement dit, des disjonctions).

Il est donc important d'avoir une bonne méthode pour décider de la satisfaisabilité d'une formule en forme conjonctive normale. On pourrait bien sûr transformer la formule en forme disjonctive normale et puis appliquer le critère du théorème 12 — malheureusement la transformation en forme disjonctive normale risque de « gonfler » la formule de façon exponentielle. On pourrait aussi utiliser la méthode des tables de vérité vue en section 2.4 et essayer toutes les affectations possibles (pour les variables qui paraissent dans la formule). Cette méthode de tables de vérité mérite d'être appelée « force brute » si on pense au nombre d'affectations qu'il faut essayer pour des formules d'une taille très modeste : Avec seulement 300 variables propositionnelles on a $2^{300} \approx 10^{90}$ affectations à essayer, c'est déjà un milliard fois le nombre estimé de particules élémentaires dans l'univers connu (les estimation de ce dernier nombre sont entre 10^{79} et 10^{81}).

5.1 Vers un algorithme efficace

Il nous faut donc une méthode plus intelligente qui évite autant que possible une énumération des affectations. Une telle méthode a été proposée en 1962 par les chercheurs américains Martin Davis, Hilary Putnam, George Logemann et Donald Loveland [**?**] et est depuis connu sous le nom *DPLL*. Parfois on parle aussi simplement de la méthode *Davis-Putnam* ou *DP* car c'est le nom d'un prédécesseur de cette méthode proposé par ces deux chercheurs en 1960 [**?**].

Nous allons développer l'algorithme DPLL en plusieurs étapes en partant de l'algorithme des tables de vérité. L'algorithme complet est résumé en section 5.2.

Hypothèses sur la forme des clauses. Nous dirons qu'une variable x paraît avec *polarité positive* dans une clause disjonctive quand la clause est de la forme

$$l_1 \vee \ldots \vee x \vee \ldots \vee l_n$$

et avec *polarité négative* quand la clause est de la forme

$$l_1 \vee \ldots \vee \neg x \vee \ldots \vee l_n$$

A priori une variable peut paraître à la fois avec polarité positive et avec polarité négative dans une clause, comme par exemple la variable x dans la clause

$$y_1 \vee x \vee y_2 \vee \neg x \vee y_3$$

Une telle clause disjonctive qui contient à la fois un littéral x et sa négation $\neg x$ est évidemment une tautologie, elle n'importe pas pour la satisfaisabilité de la formule entière et on peut la supprimer. De même, si une clause contient deux occurrences du même littéral, on peut supprimer une des deux occurrences, la formule résultante est équivalente à la formule de départ grâce à la loi d'idempotence de la disjonction (théorème 7). Exemple : on peut remplacer

$$x_1 \vee \neg x_2 \vee x_3 \vee \neg x_2$$

par la clause simplifiée

$$x_1 \vee \neg x_2 \vee x_3$$

On peut donc dans la suite supposer que dans chacune des clauses disjonctives aucune variable apparaît deux fois car si une variable apparaît deux fois dans une clause alors soit les deux occurrences sont d'une polarité différente et on peut ignorer la clause car elle est une tautologie, soit les deux occurrences sont de la même polarité et on peut simplifier la clause en supprimant le littéral qui paraît en deux copies dans la clause.

Engendrer et tester des solutions candidat partielles. Le principe de l'algorithme présenté dans la section 2.4 est d'engendrer des affectations, donc des candidats pour une solution au problème, et de les tester une par une sur la formule donnée. On appelle cette technique « générer et tester ». Il s'agit d'une approche simple qui a l'avantage de séparer l'algorithme en deux parties distinctes : la génération des solutions candidats (dans notre cas, des affectations qui ont pour support une partie des variables qui paraissent dans la formule), et le test si une solution candidat est vraiment une solution (dans notre cas, évaluer la formule par rapport à l'affectation choisie). Or il se peut qu'on peut détecter qu'une formule n'est pas satisfaisable en choisissant des valeurs pour seulement quelques unes des variables qui paraissent dans la formule. Voici un exemple :

$$x \wedge (\neg x \vee y_1 \vee \neg y_2 \vee y_3) \wedge \neg x \wedge (y_2 \vee \neg y_4 \vee y_5) \tag{5.1}$$

Cette formule a 6 variables, avec la méthode des tables de vérité il y a donc $2^6 = 64$ affectation à engendrer et à tester. Il se trouve qu'on peut voir plus facilement que cette formule n'est pas

satisfaisable en essayant *seulement* les valeurs possibles de x : Si x vaut 0 c'est la première clause qui est évaluée comme 0, et si x vaut 1 c'est la troisième clause qui est évaluée comme 0. Cette formule n'est par conséquent pas satisfaisable.

Il est donc intéressant de ne pas engendrer des solutions candidat complètes mais de choisir des valeurs de variables une par une, et pour chaque choix d'évaluer « partiellement » la formule et voir si on ne peut pas déjà donner une réponse pour ce choix sans essayer les valeurs possibles pour les variables qui restent. Pour mettre cette nouvelle approche en œuvre il faut préciser cette notion d'évaluation partielle d'une formule par rapport à la valeur choisie d'une variable. D'abord pour une clause disjonctive d, une variable x et une valeur booléenne $b \in \{0,1\}$:

$$d[x/b] = \begin{cases} \texttt{True} & \text{si } b = 1 \text{ et x apparait avec polarité positive en } d \\ \texttt{True} & \text{si } b = 0 \text{ et x apparait avec polarité négative en } d \\ d \setminus \{\neg x\} & \text{si } b = 1 \text{ et x apparait avec polarité négative en } d \\ d \setminus \{x\} & \text{si } b = 0 \text{ et x apparait avec polarité positive en } d \\ d & \text{si } x \text{ ne parait pas en } d. \end{cases}$$

Ici nous avons utilisé la notation $d \setminus l$, où l est un littéral de la clause disjonctive l, pour la clause obtenue à partir de l en supprimant le littéral l. Par exemple, si $d = (x \vee \neg y \vee z)$ alors $d \setminus \{\neg y\} = (x \vee z)$. Voici des exemples pour chacun des cinq cas qui apparaissent dans la définition d'une évaluation partielle d'une formule :

$$\begin{aligned} (x \vee \neg y \vee z)[x/1] &= \texttt{True} \\ (\neg x \vee \neg y \vee z)[x/0] &= \texttt{True} \\ (\neg x \vee \neg y \vee z)[x/1] &= (\neg y \vee z) \\ (x \vee \neg y \vee z)[x/0] &= (\neg y \vee z) \\ (y_1 \vee \neg y_2 \vee z)[x/1] &= (y_1 \vee \neg y_2 \vee z) \end{aligned}$$

Puis, on définit la notion d'évaluation partielle d'une formule $c = (d_1 \wedge \ldots \wedge d_n)$ en forme conjonctive normale : $c[x/b]$ est définie comme la conjonction de toutes les clauses $d_i[x/b]$ qui sont différentes de `True` ; on écrit :

$$c[x/b] = \bigwedge_{\substack{1 \le i \le n \\ d_i[x/b] \neq \texttt{True}}} d_i[x/b]$$

Pour l'exemple (5.1) on obtient

$$\begin{array}{lcllllll} c & = & x & \wedge & (\neg x \vee y_1 \vee \neg y_2 \vee y_3) & \wedge & \neg x & \wedge & (y_2 \vee \neg y_4 \vee y_5) \\ c[x/0] & = & \texttt{False} & & & & & \wedge & (y_2 \vee \neg y_4 \vee y_5) \\ c[x/1] & = & & & (y_1 \vee \neg y_2 \vee y_3) & \wedge & \texttt{False} & \wedge & (y_2 \vee \neg y_4 \vee y_5) \end{array}$$

Nous rappelons que `False` est une abréviation pour la clause disjonctive vide, si on supprime de la clause x le littéral x on obtient la clause vide notée `False`.

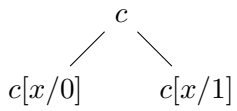
La propriété de cette notion d'évaluation partielle est que

$$[\![c[x/b]]\!]v = [\![c]\!](v[x/b])$$

pour toute affectation v et valeur booléenne b, et on a bien sûr que

$$c \models\!\!\!\models c[x/0] \vee c[x/1]$$

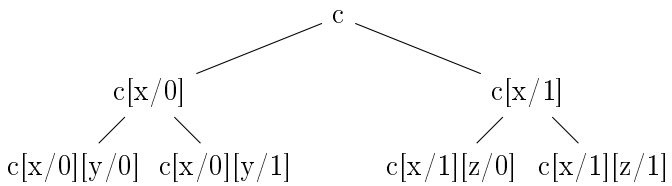
Donc, c est satisfaisable si $c[x/0]$ est satisfaisable ou si $c[x/1]$ est satisfaisable. Cela donne lieu à un arbre de recherche car il faut a priori exploiter les deux possibilités :



Notez que l'arbre de recherche est une représentation du *déroulement* de l'algorithme, il est normalement pas représenté directement en mémoire par le programme qui réalise cet algorithme. Une réalisation de l'algorithme dans un langage de programmation (disons, en Java) va plutôt utiliser des appels récursifs de fonctions, ou une pile (voir un cours d'algorithmique). L'intérêt de cet algorithme repose sur le fait qu'on peut donner des critères *simples* qui nous permettent de dire *dans certains cas* si une formule obtenue pendant ce calcul est satisfaisable ou pas :

1. Si la formule est la conjonction vide, notée `True`, alors la formule est une tautologie, et donc satisfaisable.
2. Si la formule contient une clause qui est la disjonction vide, notée `False`, alors la formule n'est pas satisfaisable.
3. Sinon on ne peut rien dire, il faut continuer à choisir des valeurs des variables.

On appelle la variable x pour laquelle on évalue la formule une fois pour la valeur $[x/1]$, une fois pour $[x/0]$, la *variable pivot*. Sur l'exemple (5.1), les deux choix possibles pour les valeurs de la variable pivot x nous donnent les deux alternatives qui sont toutes les deux non satisfaisables. Par conséquent, la formule c de (5.1) n'est pas satisfaisable. Dans ce cas on a un arbre de recherche très simple qui consiste seulement en trois nœuds c, $c[x/0]$ et $c[x/1]$. Bien sûr on ne va pas toujours trouver une réponse après une seule évaluation partielle, en général il faut continuer le parcours de l'arbre de recherche en choisissant une nouvelle variable pivot et d'évaluer partiellement la formule obtenue par rapport au choix fait pour la nouvelle variable pivot. Cela donne lieu à un arbre de recherche plus profond, comme par exemple



Remarquez qu'on n'est pas obligé de choisir la même variable pivot pour le nœud $c[x/0]$ que pour le nœud $c[x/1]$. L'efficacité d'une réalisation de l'algorithme DPLL dépend énormément de la stratégie pour trouver une variable pivot.

Une stratégie d'énumération, et une optimisation La discussion précédente sur le choix des variables et les évaluations des formules laisse une question ouverte : étant donnée une formule en forme conjonctive normale, comment choisir la variable pivot ? Un mauvais

choix répété de la variable pivot va résulter en un arbre de recherche très grand, et va donner lieu à un calcul qui est aussi long que la construction de la table de vérité. Un bon choix de la variable pivot par contre peut mener beaucoup plus rapidement à une réponse.

Si on prend par exemple la formule suivante :

$$\begin{array}{cccccc} & (x_1 \vee x_2 \vee x_3) & \wedge & (\neg x_1 \vee x_2 \vee x_3) & \wedge & x_3 \\ \wedge & (x_1 \vee x_2 \vee \neg x_3) & \wedge & (\neg x_1 \vee x_2 \vee \neg x_3) & \wedge & \neg x_3 \\ \wedge & (x_1 \vee \neg x_2 \vee x_3) & \wedge & (\neg x_1 \vee \neg x_2 \vee x_3) & & \\ \wedge & (x_1 \vee \neg x_2 \vee \neg x_3) & \wedge & (\neg x_1 \vee \neg x_2 \vee \neg x_3) & & \end{array}$$

Si on choisi les variables de pivot dans l'ordre $x_1 - x_2 - x_3$ alors on trouvera que la formule est non satisfaisable seulement quand on aura choisi les valeurs des trois variables. Dans ce cas on parcourt un arbre de recherche d'hauteur 3 avec $2^3 = 8$ feuilles, c'est-à-dire on engendre les mêmes affectations que dans la méthode des tables vérités, et on a très peu gagné par notre nouvelle méthode. En fait il y a un petit gain dû au fait qu'on simplifie la formule dès qu'on a fait un choix, par exemple $x_1 = 1$ De cette façon il y a plus de réutilisation de calculs entre les choix possibles des affectations que dans la méthode des tables de vérité, mais ce gain n'est pas très important.

Un meilleur choix comme variable de pivot est bien sûr la variable x_3, on trouve dans ce cas que la formule n'est pas satisfaisable pour les deux choix des valeurs de x_3 et on a trouvé la réponse avec un arbre de recherche d'hauteur 1.

Une bonne stratégie est donc : S'il y a une clause qui consiste en un seul littéral x ou $\neg x$ alors on choisit la variable x.

Définition 14 *Une clause (conjonctive ou disjonctive) qui consiste en un seul littéral est appelé* unitaire.

Pour ce cas on pourra même optimiser un peu le calcul : Si la formule c contient la clause unitaire x alors le choix $x = 0$ donne forcement une formule c qui contient la clause `False`. Puisqu'on sait déjà que $c[x/0]$ va être non satisfaisable il n'est même pas nécessaire de construire $c[x/0]$, et la formule c est satisfaisable si et seulement si $c[x/1]$ est satisfaisable. On peut donc passer de la formule c directement à la formule $c[x/1]$ qui est une formule plus petite, et cela sans avoir fait un choix. Le cas d'une formule c qui contient une clause unitaire $\neg x$ est analogue, dans ce cas c est satisfaisable si et seulement si $c[x/0]$ est satisfaisable.

Exemple :

$$\begin{array}{lllll} & c & = & (x \vee y) \wedge \neg y \wedge (\neg x \vee y \vee \neg z) & \text{est satisfaisable} \\ \text{ssi} & c[y/0] & = & x \wedge (\neg x \vee \neg z) & \text{est satisfaisable} \\ \text{ssi} & c[y/0][x/1] & = & \neg z & \text{est satisfaisable} \\ \text{ssi} & c[y/0][x/1][z/0] & = & \texttt{True} & \text{est satisfaisable} \end{array}$$

La formule `True` est satisfaisable (elle est même une tautologie), la formule de départ est alors également satisfaisable. Par contre on ne peut pas conclure que la formule de départ est une tautologie, notez que les étapes de raisonnement sont relatives à la *satisfaisabilité* et ne sont pas des équivalences logiques. Observez que l'évaluation partielle $c[y/0]$ fait paraître une nouvelle clause unitaire (la clause x) qui n'existait pas dans cette forme en c, et que l'évaluation partielle $c[y/0][x/1]$ fait paraître encore une nouvelle clause unitaire. Il est donc

possible que cette stratégie nous permette dans certains cas d'enchaîner des simplifications sans nécessiter de vrais choix des valeurs de variables pivot.

Cela laisse toujours la question ouverte quoi faire quand toutes les clauses contiennent au moins deux littéraux.

Variables qui apparaissent avec une seule polarité. Une deuxième optimisation concerne le cas des variables qui apparaissent avec une seule polarité dans la formule, comme par exemple

$$(x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (y \vee z) \wedge (x \vee \neg y) \tag{5.2}$$

Dans ce cas, x apparait seulement avec polarité positive.

Proposition 11 *Soit c en forme conjonctive normale.*
- *Si la variable x apparait seulement avec polarité positive en x alors c est satisfaisable si et seulement si $c[x/1]$ est satisfaisable.*
- *Si la variable x apparait seulement avec polarité négative en x alors c est satisfaisable si et seulement si $c[x/0]$ est satisfaisable.*

Démonstration: Nous montrerons ici seulement le premier énoncé, le deuxième se montre de façon analogue.

Si $c[x/1]$ est satisfaisable, disons $v \models c[x/1]$, alors on a que $1 = [\![c[x/1]]\!]v = [\![c]\!](v[x/1])$ et c est alors également satisfaisable.

Soit c satisfaisable, disons $v \models c$. On peut montrer facilement que pour toute clause d en c on a que $[\![d]\!](v[x/1]) \geq [\![d]\!]v$ car x ne peut paraître que positivement en d, et par conséquent que $[\![c[x/1]]\!]v = [\![c]\!](v[x/1]) \geq 1$. Par conséquent, $c[x/1]$ est satisfaisable. □

Par exemple, la formule (5.2) est satisfaisable si et seulement si la formule $(y \vee z)$ est satisfaisable.

L'optimisation consiste alors en le remplacement de c par $c[x/1]$ quand x n'apparait qu'avec polarité positive en c, et par $c[x/0]$ quand x n'apparait qu'avec polarité négative en c. Il se peut que, après une application de cette règle de simplification, une autre variable apparait maintenant avec une seule polarité. Une première simplification peut donc déclencher des autres simplifications successives. Exemple :

	c	$=$	$(x \vee \neg y) \wedge (y \vee \neg z_1) \wedge (\neg z_1 \vee \neg z_2)$	est satisfaisable
ssi	$c[x/1]$	$=$	$(y \vee \neg z_1) \wedge (\neg z_1 \vee \neg z_2)$	est satisfaisable
ssi	$c[x/1][y/1]$	$=$	$(\neg z_1 \vee \neg z_2)$	est satisfaisable
ssi	$c[x/1][y/1][z_1/0]$	$=$	`True`	est satisfaisable

La formule de départ est donc satisfaisable.

5.2 L'algorithme DPLL complet

Voici donc l'algorithme complet DPLL, écrit avec des appels récursifs . L'argument c de cette fonction doit être une formule en forme conjonctive normale où aucune variable parait deux

fois dans la même clause.

function $dp(c)$ =`case`

(1) `if` $c =$ `True` `then return` *true*
(2) `if` c contient une clause `False` `then return` $false$
(3) `if` c contient une clause unitaire x `then return` $dp(c[x/1])$
(4) `if` c contient une clause unitaire $\neg x$ `then return` $dp(c[x/0])$
(5) `if` x n'apparaît qu'avec polarité positive en c `then return` $dp(c[x/1])$
(6) `if` x n'apparaît qu'avec polarité négative en c `then return` $dp(c[x/0])$
(7) `else choisir` $x \in \mathcal{V}(c)$;`return` $dp(c[x/0])$ `or` $dp(c[x/1])$

Les tests différents dans l'instruction `case` peuvent être faits dans un ordre quelconque ; le dernier cas `else` s'applique seulement quand aucun des cas `if` s'applique. On a donc la liberté de choisir sa stratégie dans l'implémentation de cet algorithme. Cette liberté à un intérêt plutôt pour le choix d'une priorité entre les cas (3) à (6) car les deux premiers cas mènent toute de suite à la terminaison de l'algorithme et leur test associé est facile à réaliser (ces deux cas sont donc à considérer avec première priorité).

Les deux cas (5) et (6) sont parfois omis dans des implémentations car le coût supplémentaire de leur réalisation vaut, dans certaines applications, pas le gain. Ce coût supplémentaire peut être assez important car il faut pour toute variable maintenir un compteur pour son nombre d'occurrences avec polarité positive, et un autre compteur pour son nombre d'occurrences de polarité négative, et mettre à jour ces compteurs lors de chaque évaluation partielle.

Nous avons donné tous les arguments pour la correction de cet algorithme tout le long de la section 5.1. « Correction » veut ici dire que quand l'exécution d'un appel $dp(c)$ donne une réponse *true* alors la formule c est satisfaisable, et quand la réponse est $false$ alors c n'est pas satisfaisable. Il reste à montrer que l'exécution d'un appel $dp(c)$ termine toujours, cela est simplement assuré par le fait que dans chaque appel récursif l'argument (soit $c[x/0]$, soit $c[x/1]$) est plus petit que la formule c. Formellement il s'agit ici d'une preuve par induction *naturelle* sur la taille de la formule. Remarquer qu'on ne peut pas utiliser le principe d'induction *structurelle* pour la preuve de terminaison car les formules $c[x/0]$ et $c[x/1]$ qui peuvent apparaître dans les appels récursifs ne sont en général pas de sous-formules de la formule c.

Notez que l'application du cas (5) et du cas (6) de l'algorithme ont simplement pour effet de supprimer certaines des clauses : si une variable x par exemple n'apparait qu'avec polarité positive en c alors $c[x/1]$ est obtenue à partir de c en supprimant toutes les clauses qui contiennent x. Pour cette raison il n'est jamais possible que l'exécution du cas (5) ou du (6) produit une clause unitaire qui n'était pas déjà présente avant.

Un exemple complet :

$$(x_1 \vee \neg x_2 \vee y_1 \vee \neg y_2 \vee \neg z_2 \vee \neg z_4)$$
$$\wedge(x_2 \vee y_1) \wedge (x_2 \vee y_1 \vee y_2 \vee z_1 \vee z_4) \wedge (x_2 \vee \neg y_2 \vee z_1 \vee \neg z_2)$$
$$\wedge(x_2 \vee \neg y_1 \vee z_3 \vee \neg z_4) \wedge (x_2 \vee \neg y_2 \vee z_2 \vee \neg z_3)$$
$$\wedge(\neg x_2 \vee \neg y_1) \wedge (\neg x_2 \vee \neg y_1 \vee \neg y_2) \wedge (\neg x_2 \vee y_1 \vee y_2) \wedge (\neg x_2 \vee \neg y_2 \vee z_1) \wedge (\neg x_2 \vee \neg z_1 \vee z_2)$$
$$\wedge(\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Cette formule contient 8 variables, il y a donc 256 affectations à tester avec la méthode des tables de vérité.

Le cas (5) de l'algorithme DPLL s'applique avec la variable x_1 qui n'apparait qu'avec polarité positive dans la formule (dans la première clause). On obtient, en supprimant la première clause :

$$(x_2 \vee y_1) \wedge (x_2 \vee y_1 \vee y_2 \vee z_1 \vee z_4) \wedge (x_2 \vee \neg y_2 \vee z_1 \vee \neg z_2)$$
$$\wedge(x_2 \vee \neg y_1 \vee z_3 \vee \neg z_4) \wedge (x_2 \vee \neg y_2 \vee z_2 \vee \neg z_3)$$
$$\wedge(\neg x_2 \vee \neg y_1) \wedge (\neg x_2 \vee \neg y_1 \vee \neg y_2) \wedge (\neg x_2 \vee y_1 \vee y_2) \wedge (\neg x_2 \vee \neg y_2 \vee z_1) \wedge (\neg x_2 \vee \neg z_1 \vee z_2)$$
$$\wedge(\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Dans cette formule il n'y aucune clause unitaire et toute variable apparait avec les deux polarités. On est donc obligé d'appliquer (7). Nous choisissons ici la variable x_2, il y a maintenant deux cas :

Premier cas obtenu pour $x_2 = 0$

$$y_1 \wedge (y_1 \vee y_2 \vee z_1 \vee z_4) \wedge (\neg y_2 \vee z_1 \vee \neg z_2) \wedge (\neg y_1 \vee z_3 \vee \neg z_4) \wedge (\neg y_2 \vee z_2 \vee \neg z_3)$$
$$\wedge(\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Maintenant il y une clause unitaire y_1, on obtient donc avec (3) :

$$(\neg y_2 \vee z_1 \vee \neg z_2) \wedge (z_3 \vee \neg z_4) \wedge (\neg y_2 \vee z_2 \vee \neg z_3) \wedge (\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Dans cette formule, la variable y_2 n'apparait qu'avec polarité négative, on peut donc appliquer (6) et supprimer les deux clauses qui contiennent le littéral $\neg y_2$:

$$(z_3 \vee \neg z_4) \wedge (\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Encore une fois ce n'est que (7) qui s'applique, si on choisit la variable pivot z_3 on obtient encore deux sous-cas :

Premier sous-cas obtenu pour $z_3 = 0$

$$\neg z_4 \wedge z_4$$

qui donne *false* par application de (3) (ou de (4)). On doit donc considérer le second sous-cas :

Second sous-cas obtenu pour $z_3 = 1$. Il se trouve qu'on obtient la même formule que dans le premier sous-cas :

$$\neg z_4 \wedge z_4$$

qui donne *false* par application de (3) (ou de (4)).

Le premier cas n'a pas donné une réponse affirmative, il faut alors considérer le second cas :

Second cas obtenu pour $x_2 = 1$

$$\neg y_1 \wedge (\neg y_1 \vee \neg y_2) \wedge (y_1 \vee y_2) \wedge (\neg y_2 \vee z_1) \wedge (\neg z_1 \vee z_2)$$
$$\wedge(\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Il y a une clause unitaire $\neg y_1$, on peut donc appliquer (3) et obtient

$$\wedge y_2 \wedge (\neg y_2 \vee z_1) \wedge (\neg z_1 \vee z_2) \wedge (\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Maintenant il y une nouvelle clause unitaire y_2, et on obtient de la même façon

$$z_1 \wedge (\neg z_1 \vee z_2) \wedge (\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Encore une fois avec la clause unitaire z_1 :

$$\wedge z_2 \wedge (\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Application de (3) sur la clause unitaire z_2 (alternativement : application de (5) sur la variable x_2 qui n'apparait qu'avec polarité positive) :

$$(\neg z_3 \vee \neg z_4) \wedge (z_3 \vee z_4) \wedge (\neg z_3 \vee z_4)$$

Application de (7) avec la variable pivot z_3 donne

Premier sous-cas obtenu pour $z_3 = 0$:

$$z_4 \wedge z_4$$

Finalement, application de (3) (ou alternativement (5)) donne *true*. La formule est donc satisfaisable, et il n'est plus nécessaire de considérer le second sous-cas.

Il existe une forme duale de l'algorithme DPLL. Cet algorithme dual s'applique à des formules en forme disjonctive normale et sert à décider la *validité* des formules. On obtient l'algorithme DPLL dual à partir de l'algorithme DPLL de la même façon que nous avons obtenu les résultats de la section 4.5 à partir des résultats de la section 4.3.

La table suivante résume nos résultats sur les méthodes pour décider de la satisfaisabilité ou de la validité des formules en forme normale :

Forme de la formule :	forme disjonctive normale	forme conjonctive normale
Satisfaisabilité :	trivial (théorème 12)	algorithme DPLL
Validité :	algorithme DPLL dual	trivial (théorème 15)

5.3 Références et remarques

L'algorithme DPLL s'est révélé, malgré sa simplicité, très efficace dans un grand nombre d'applications. Il y a pourtant des cas dans lesquels on est obligé de traiter des formules propositionnelles d'une si grande complexité que DPLL n'y arrive plus.

Une extension importante de DPLL a été développée depuis la fin des années 80, donc presque 30 ans après la publication de l'algorithme original. L'idée à la base de cette extension est celle de l'*apprentissage* : on peut dans certains cas déduire de nouvelles clauses qui sont des conséquences des clauses de départ, et qui peuvent aider dans la suite pour éliminer des choix qui ne peuvent pas mener à un succès. Cette découverte était à la base des travaux de recherche qui ont donné lieu à toute une suite d'améliorations, et à des programmes comme `Chaff`, `GRASP`, ou MiniSat.

Chapitre 6

Modélisation en logique propositionnelle

Nous avons vu dans le chapitre précédent qu'il existe des outils pour décider de la satisfaisabilité de formules propositionnelles en forme conjonctive normale, et que ces outils sont souvent en pratique très efficaces. Dans ce chapitre nous allons montrer comment ces outils peuvent être utilisés pour construire des programmes qui résolvent efficacement des problèmes combinatoires (c'est-à-dire, des problèmes ou il faut trouver une solution parmi toutes les combinaisons possibles de certains choix élémentaires).

6.1 Exemple : Colorer une carte

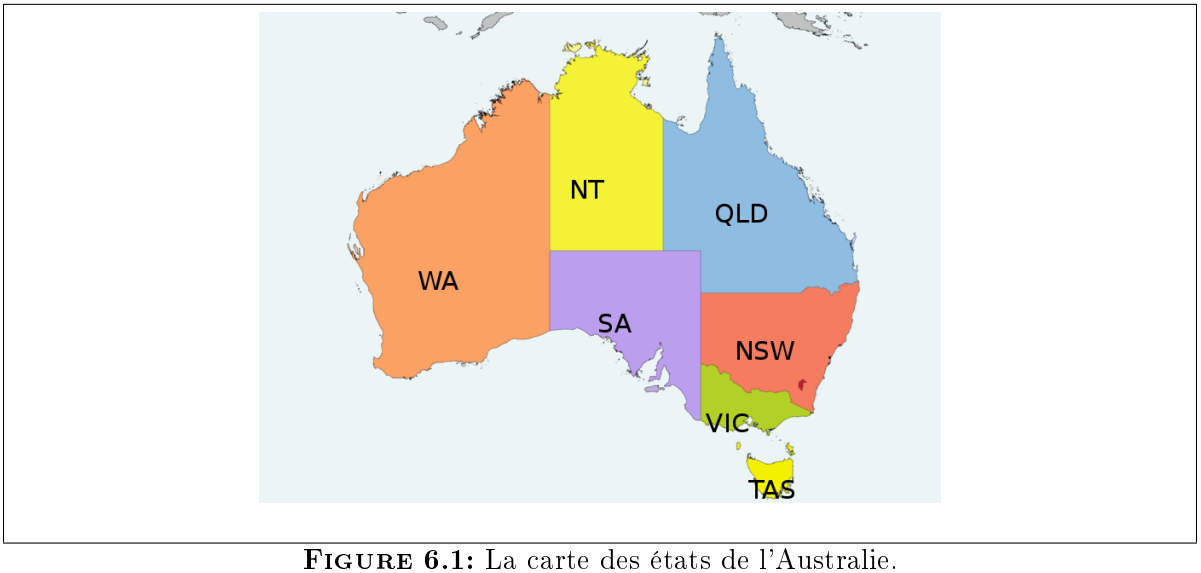


Figure 6.1: La carte des états de l'Australie.

Nous illustrons d'abord cette approche à l'aide d'un problème concret relativement simple : est-il possible de colorer une carte de l'Australie (voir figure 6.1) avec seulement trois couleurs (que nous appelons r, b, et v pour rouge, bleu, et vert) tel que deux états adjacents (qui ont

une frontière commune ; une île n'est adjacente à aucun autre état) n'ont jamais la même couleur ?

Toute coloration de la carte de l'Australie par les trois couleurs peut être vue comme une affectation qui associe à chaque état (parmi WA, NT, SA, QLD, NSW, VIC, et TAS) une couleur parmi r, b, et v. Or, les variables propositionnelles permettent seulement un choix binaire pour leur valeur : 0 ou 1. La première étape de la modélisation de notre problème en logique propositionnelle consiste à exprimer les colorations possibles comme des affectations à des variable propositionnelles. Dans cette première étape on ne s'intéresse pas encore à la modélisation de la *contrainte* qui consiste à interdire la même couleur pour deux états adjacent ; cette contrainte sera mise en place dans la deuxième étape de la modélisation.

Nous créons une variable propositionnelle pour *chaque* état et pour *chaque* couleur, cela fait donc en total $7 * 3 = 21$ variables. Nous notons une telle variable par exemple comme `[WA,r]`, l'idée est que cette variable vaut 1 quand l'état WA est coloré en rouge, et 0 sinon (s'il est coloré dans une autre couleur, ou pas coloré du tout). Puis, nous allons construire une formule propositionnelle en forme conjonctive normale sur ces 21 variables qui est vraie par rapport à une affectation si et seulement si l'affectation correspond à une coloration de la carte telle que deux états adjacent soient colorés différemment.

Première étape : caractériser les colorations. A priori, il est possible qu'une affectation des variables propositionnelles ne corresponde pas à une coloration de la carte, par exemple quand toutes les trois variables `[WA,r]`, `[WA,b]` et `[WA,v]` ont la valeur 0 (c'est-à-dire, l'état WA aurait aucune couleur), ou quand à la fois `[WA,r]` et `[WA,b]` ont la valeur 1 (l'état WA aurait à la fois la couleur rouge et la couleur bleue). La première partie de la formule sert donc à exclure toutes les affectations qui ne correspondent pas à une coloration. En fait, il faut exprimer que tout état a une couleur, et exactement une couleur. Cette formule consiste elle-même en deux parties. La première partie exprime que tout état a au moins une couleur :

$$\begin{aligned} P_1 := \quad & (\texttt{[WA,r]} \vee \texttt{[WA,b]} \vee \texttt{[WA,v]}) \\ \wedge \ & (\texttt{[NT,r]} \vee \texttt{[NT,b]} \vee \texttt{[NT,v]}) \\ \wedge \ & (\texttt{[SA,r]} \vee \texttt{[SA,b]} \vee \texttt{[SA,v]}) \\ \wedge \ & (\texttt{[QLD,r]} \vee \texttt{[QLD,b]} \vee \texttt{[QLD,v]}) \\ \wedge \ & (\texttt{[NSW,r]} \vee \texttt{[NSW,b]} \vee \texttt{[NSW,v]}) \\ \wedge \ & (\texttt{[VIC,r]} \vee \texttt{[VIC,b]} \vee \texttt{[VIC,v]}) \\ \wedge \ & (\texttt{[TAS,r]} \vee \texttt{[TAS,b]} \vee \texttt{[TAS,v]}) \end{aligned}$$

La deuxième partie de la première formule exprime le fait que tout état ne peut pas avoir deux couleurs différentes à la fois :

$$\begin{aligned} P_2 := \quad & (\neg\texttt{[WA,r]} \vee \neg\texttt{[WA,b]}) \wedge (\neg\texttt{[WA,r]} \vee \neg\texttt{[WA,v]}) \wedge (\neg\texttt{[WA,b]} \vee \neg\texttt{[WA,v]}) \\ \wedge \ & (\neg\texttt{[NT,r]} \vee \neg\texttt{[NT,b]}) \wedge (\neg\texttt{[NT,r]} \vee \neg\texttt{[NT,v]}) \wedge (\neg\texttt{[NT,b]} \vee \neg\texttt{[NT,v]}) \\ \wedge \ & (\neg\texttt{[SA,r]} \vee \neg\texttt{[SA,b]}) \wedge (\neg\texttt{[SA,r]} \vee \neg\texttt{[SA,v]}) \wedge (\neg\texttt{[SA,b]} \vee \neg\texttt{[SA,v]}) \\ \wedge \ & (\neg\texttt{[QLD,r]} \vee \neg\texttt{[QLD,b]}) \wedge (\neg\texttt{[QLD,r]} \vee \neg\texttt{[QLD,v]}) \wedge (\neg\texttt{[QLD,b]} \vee \neg\texttt{[QLD,v]}) \\ \wedge \ & (\neg\texttt{[NSW,r]} \vee \neg\texttt{[NSW,b]}) \wedge (\neg\texttt{[NSW,r]} \vee \neg\texttt{[NSW,v]}) \wedge (\neg\texttt{[NSW,b]} \vee \neg\texttt{[NSW,v]}) \\ \wedge \ & (\neg\texttt{[VIC,r]} \vee \neg\texttt{[VIC,b]}) \wedge (\neg\texttt{[VIC,r]} \vee \neg\texttt{[VIC,v]}) \wedge (\neg\texttt{[VIC,b]} \vee \neg\texttt{[VIC,v]}) \\ \wedge \ & (\neg\texttt{[TAS,r]} \vee \neg\texttt{[TAS,b]}) \wedge (\neg\texttt{[TAS,r]} \vee \neg\texttt{[TAS,v]}) \wedge (\neg\texttt{[TAS,b]} \vee \neg\texttt{[TAS,v]}) \end{aligned}$$

La formule construite jusqu'à maintenant, $P_1 \wedge P_2$, a la propriété que toute solution de cette formule correspond à une coloration *unique* de la carte, *et inversement*. Exprimé plus formelle-

ment, il existe une fonction *bijective* entre l'ensemble de toutes les solutions, et l'ensemble de toutes les colorations possibles de la carte. Soit

$$rep\colon \Big(\{\texttt{WA},\texttt{NT},\texttt{SA},\texttt{QLD},\texttt{NSW},\texttt{VIC},\texttt{TAS}\} \to \{r,b,v\}\Big) \to \{v \in X \to \{0,1\} \mid v \models P_1 \wedge P_2\}$$

où X est l'ensemble des 21 variables donné au-dessus, définie par

$$rep(f)([state, color]) = \begin{cases} 1 & \text{si } f(state) = color \\ 0 & \text{si } f(state) \neq color \end{cases}$$

On peut montrer que *rep* est bien une fonction bijective :
- Pour toute coloration f, l'affectation $rep(f)$ est bien une solution de la formule $P_1 \wedge P_2$, donc il s'agit d'une fonction avec le domaine et co-domaine indiqué.
- Si f_1 et f_2 sont des colorations différentes, alors $rep(f_1)$ et $rep(f_1)$ sont des affectations différentes. La fonction est donc injective.
- Pour toute solution v de $P_1 \wedge P_2$ il existe une coloration f telle que $rep(f) = v$. La fonction est donc surjective.

Deuxième étape : exprimer la contrainte spécifique. Ensuite, nous pouvons construire la formule qui exprime le fait que deux états adjacents ne peuvent pas avoir la même couleur : nous interdisons simplement tous les cas dans lesquels deux pays adjacents sont colorés pareil (soit rouge, soit bleue, soit vert) :

$$\begin{aligned} P_3 = \quad & (\neg\texttt{[WA,r]} \vee \neg\texttt{[NT,r]}) \wedge (\neg\texttt{[WA,b]} \vee \neg\texttt{[NT,b]}) \wedge (\neg\texttt{[WA,v]} \vee \neg\texttt{[NT,v]}) \\ \wedge\ & (\neg\texttt{[WA,r]} \vee \neg\texttt{[SA,r]}) \wedge (\neg\texttt{[WA,b]} \vee \neg\texttt{[SA,b]}) \wedge (\neg\texttt{[WA,v]} \vee \neg\texttt{[SA,v]}) \\ \wedge\ & (\neg\texttt{[NT,r]} \vee \neg\texttt{[SA,r]}) \wedge (\neg\texttt{[NT,b]} \vee \neg\texttt{[SA,b]}) \wedge (\neg\texttt{[NT,v]} \vee \neg\texttt{[SA,v]}) \\ \wedge\ & (\neg\texttt{[NT,r]} \vee \neg\texttt{[QLD,r]}) \wedge (\neg\texttt{[NT,b]} \vee \neg\texttt{[QLD,b]}) \wedge (\neg\texttt{[NT,v]} \vee \neg\texttt{[QLD,v]}) \\ \wedge\ & (\neg\texttt{[SA,r]} \vee \neg\texttt{[QLD,r]}) \wedge (\neg\texttt{[SA,b]} \vee \neg\texttt{[QLD,b]}) \wedge (\neg\texttt{[SA,v]} \vee \neg\texttt{[QLD,v]}) \\ \wedge\ & (\neg\texttt{[SA,r]} \vee \neg\texttt{[NSW,r]}) \wedge (\neg\texttt{[SA,b]} \vee \neg\texttt{[NSW,b]}) \wedge (\neg\texttt{[SA,v]} \vee \neg\texttt{[NSW,v]}) \\ \wedge\ & (\neg\texttt{[SA,r]} \vee \neg\texttt{[VIC,r]}) \wedge (\neg\texttt{[SA,b]} \vee \neg\texttt{[VIC,b]}) \wedge (\neg\texttt{[SA,v]} \vee \neg\texttt{[VIC,v]}) \end{aligned}$$

On constate que la dernière formule ne contient pas de variable pour l'état de Tasmanie, ce qui est normal puisque la Tasmanie est une île et donc adjacente à aucun autre état.

Finalement, la formule complète est $P_1 \wedge P_2 \wedge P_3$. Cette formule est satisfaisable si et seulement s'il existe une solution au problème de départ. Si $v \models P_1 \wedge P_2 \wedge P_3$, alors la coloration cherchée est la fonction f telle que $rep(f) = v$. Une telle fonction f existe car la fonction *rep* est surjective, et elle est uniquement déterminée pour une affection v donnée car la fonction *rep* est injective.

On voit que la première étape était plus au moins indépendante du fait qu'il s'agit ici de l'Australie : Cette étape aurait été la même pour l'Afrique ou l'Europe, sauf que pour ces continents il nous aurait fallu plus de variables (car ils ont plus de pays que l'Australie a d'états). La deuxième étape, par contre, était très spécifique à l'Australie, bien que le *principe* de construction sera le même pour des autres cartes. Dans la deuxième étape, la forme de la formule obtenue dépend beaucoup de la carte considérée. Par exemple, une carte des régions du Chili donnerait une formule complètement différente (consultez un atlas pour savoir pourquoi).

Généralisation. Regardons maintenant le problème de coloration d'une carte en général. Soient donnés :

- un ensemble de pays $1, \ldots, n$
- un ensemble de couleurs $1, \ldots, m$
- une fonction $A\colon \{1, \ldots, n\} \times \{1, \ldots, n\} \to \{true, false\}$ qui indique si deux pays sont adjacents.

Donc on simplifie la notation on numérotant les pays et les couleurs. L'ensemble des variables propositionnelles est

$$\{[i, j] \mid 1 \leq i \leq n, 1 \leq j \leq m\}$$

Nous pouvons écrire la formule maintenant très concisement en utilisant la notation $\bigwedge_i$ et $\bigvee_i$:

$$\begin{aligned} & \bigwedge_{1 \leq i \leq n} \Big(\bigvee_{1 \leq j \leq m} [i, j] \Big) \\ \wedge \quad & \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} \bigwedge_{j < k \leq m} (\neg[i, j] \vee \neg[i, k]) \\ \wedge \quad & \bigwedge_{1 \leq i \leq n} \bigwedge_{\substack{i < j \leq n \\ A(i,j) = true}} \bigwedge_{1 \leq k \leq m} (\neg[i, k] \vee \neg[j, k]) \end{aligned}$$

Ces trois formules correspondent respectivement aux formules P_1, P_2, et P_3 dans la construction exemple. On voit que la formule construite est déjà en forme conjonctive normale, ce qui découle assez naturellement de notre problème et de la modélisation que nous avons choisi. D'autres problèmes combinatoires peuvent nécessiter plus d'effort pour obtenir une forme conjonctive normale.

On peut maintenant facilement écrire un programme qui

1. lit la carte (dans une représentation appropriée) et le nombre de couleurs
2. construit la formule propositionnelle comme décrit au-dessus
3. lance un SAT-solveur externe (par exemple MiniSat) sur cette formule
4. utilise l'affectation trouvée par le SAT-solveur (quand une solution existe) pour afficher une coloration de la carte.

Comment est-ce qu'un SAT-solveur va se débrouiller pour trouver une solution à une telle formule ? Imaginons que le SAT-solveur réalise l'algorithme DPLL que nous avons vu au chapitre 5 (en vérité, les SAT-solveurs modernes sont beaucoup plus sophistiqués), et prenons la formule construite pour la carte de l'Australie. Initialement il n'y a ni de clause unitaire, ni de variable en une seule polarité. L'algorithme choisi donc une variable, disons `[WA,r]`, et essaye de la mettre à la valeur 1, et puis de la mettre à la valeur 0 dans le cas où le premier choix ne donne pas de solution. Quand la variable `[WA,r]` est mise à 1, toutes les clauses binaires qui contiennent le littéral négatif $\neg$`[WA,r]` vont se transformer en clause unaire, et donc déclencher un grand nombre de résolutions unaires. En particulier, le choix de la couleur rouge va être enlevé pour tous les pays qui sont adjacents avec WA. L'effet sera encore plus grand quand l'implémentation du SAT-solveur va choisir une variable avec le plus grand nombre d'occurrences dans la formule, c'est dans notre cas une des trois variables liées à l'état de Southern Australia car c'est cet état qui a le plus de voisins.

6.2 Exemple : Des mariages heureux

Nous essayons maintenant d'appliquer le même principe sur un problème différent :

Dans le cours de Master 2 « Outils Logiques avancés » il y a trois étudiants - Adam, Bruno, Chen - et trois étudiantes - Xanthia, Ylenia, Zoé. On sait que Adam aime Xanthia et Zoé, Bruno aime Ylenia, Chen aime Ylenia et Xanthia, tandis que Xanthia aime Bruno et Chen, Ylenia aime Adam et Bruno, Zoé aime Adam et Chen. Est-il possible de faire trois mariages parmi eux, tel que chacun(e) se marie avec une personne qu'il (qu'elle) aime ?

La première décision à prendre est : quelles sont les variables propositionnelles, et à quoi doivent elles correspondre dans notre problème de départ ? On pourrait être tenté de choisir des variables $\langle X, Y\rangle$, où X et Y sont des personnes, qui dénotent le fait que X aime Y. Ça serait un mauvais choix car il ne nous permettra pas de modéliser ce qui nous intéresse à la fin : les mariages entre les personnes. Une meilleure approche sera de choisir des variables $[X, Y]$, où X et Y sont des personnes, exprimant le fait que X se marie avec Y ($6 * 6 = 36$ variables). Cela nous permettra de modéliser les mariages, mais nous obligera à exprimer par une formule le fait que chaque mariage est entre un garçon et une fille, et en plus qu'un mariage entre X et Y est la même chose qu'un mariage entre Y et X. On peut se faciliter la tâche en choisissant mieux les variables : on choisit des variables $[G, F]$ où G est un garçon, et F une fille (seulement $3 * 3 = 9$ variables) : $[A, X]$, $[A, Y]$, $[A, Z]$, $[B, X]$, $[B, Y]$, $[B, Z]$, $[C, X]$, $[C, Y]$, $[C, Z$ (à partir de maintenant nous abrégerons les noms par leur lettre initiale).

Première étape : caractériser les mariages Évidemment tout mariage entre ces personnes correspond à une affectation des variables. Le contraire n'est pas vrai : D'un côté, une affectation peut par exemple affecter 1 aux deux variables `[A,X]` et `[A,Y]`, ce qui fait vivre Adam en bigamie, ou affecter 0 aux trois variables `[C,X]`, `[C,Y]` et `[C,Z]`, ce qui laisse Chen célibataire. On construit donc des formules qui expriment l'absence du célibat : chaque garçon trouve une fille (formule P_1), et chaque fille trouve un garçon (formule P_1').

$$\begin{aligned}
P_1 := \quad & (\texttt{[A,X]} \vee \texttt{[A,Y]} \vee \texttt{[A,Z]}) \\
\wedge \quad & (\texttt{[B,X]} \vee \texttt{[B,Y]} \vee \texttt{[B,Z]}) \\
\wedge \quad & (\texttt{[C,X]} \vee \texttt{[C,Y]} \vee \texttt{[C,Z]}) \\
P_1' := \quad & (\texttt{[A,X]} \vee \texttt{[B,X]} \vee \texttt{[C,X]}) \\
\wedge \quad & (\texttt{[A,Y]} \vee \texttt{[B,Y]} \vee \texttt{[C,Y]}) \\
\wedge \quad & (\texttt{[A,Z]} \vee \texttt{[B,Z]} \vee \texttt{[C,Z]})
\end{aligned}$$

Puis, on construit des formules qui expriment l'absence de bigamie : Aucun garçon ne se marie avec deux filles (formule P_2), et aucune fille ne se marie avec deux garçons (formule P_2').

$$\begin{aligned}
P_2 := \quad & (\neg\texttt{[A,X]} \vee \neg\texttt{[A,Y]}) \wedge (\neg\texttt{[A,X]} \vee \neg\texttt{[A,Z]}) \wedge (\neg\texttt{[A,Y]} \vee \neg\texttt{[A,Z]}) \\
\wedge \quad & (\neg\texttt{[B,X]} \vee \neg\texttt{[B,Y]}) \wedge (\neg\texttt{[B,X]} \vee \neg\texttt{[B,Z]}) \wedge (\neg\texttt{[B,Y]} \vee \neg\texttt{[B,Z]}) \\
\wedge \quad & (\neg\texttt{[C,X]} \vee \neg\texttt{[C,Y]}) \wedge (\neg\texttt{[C,X]} \vee \neg\texttt{[C,Z]}) \wedge (\neg\texttt{[C,Y]} \vee \neg\texttt{[C,Z]}) \\
P_2' := \quad & (\neg\texttt{[A,X]} \vee \neg\texttt{[B,X]}) \wedge (\neg\texttt{[A,X]} \vee \neg\texttt{[C,X]}) \wedge (\neg\texttt{[B,X]} \vee \neg\texttt{[C,X]}) \\
\wedge \quad & (\neg\texttt{[A,Y]} \vee \neg\texttt{[B,Y]}) \wedge (\neg\texttt{[A,Y]} \vee \neg\texttt{[C,Y]}) \wedge (\neg\texttt{[B,Y]} \vee \neg\texttt{[C,Y]}) \\
\wedge \quad & (\neg\texttt{[A,Z]} \vee \neg\texttt{[B,Z]}) \wedge (\neg\texttt{[A,Z]} \vee \neg\texttt{[C,Z]}) \wedge (\neg\texttt{[B,Z]} \vee \neg\texttt{[C,Z]})
\end{aligned}$$

La formule $P_1 \wedge P_1' \wedge P_2 \wedge P_2'$ est satisfaite par tout les mariages légaux (monogames) entre toutes les personnes, et exclut toutes les affectations qui ne sont pas des mariages légaux entre

toutes les personnes du groupe. On pourrait remarquer qu'il aurait en principe suffit de choisir la formule $P_1 \wedge P_2'$ (ou $P_1' \wedge P_2$).

Exercice 13 *Montrer que* $P_1 \wedge P_2' \models P_1' \wedge P_2$.

Est-ce que ça veut dire que $P_1 \wedge P_2'$ est un meilleur choix que $P_1 \wedge P_1' \wedge P_2 \wedge P_2'$? La réponse est non, car un solveur peut bien profiter des informations supplémentaires même si elles sont en principes redondantes. Par exemple, si l'algorithme DPLL avait choisi en un moment une affectation qui marie un garçon à deux filles alors l'algorithme aurait trouvé tout de suite la contradiction s'il disposait de P_2. En présence de $P_1 \wedge P_2'$ seulement il trouve aussi la contradiction, mais possiblement plus tard. Donc, des informations redondantes peuvent mener à une amélioration de la performance de l'algorithme.

Deuxième étape : exprimer la contrainte spécifique Il reste maintenant à exprimer que chaque garçon se marie avec une fille qu'il aime, et que chaque fille se marie avec un garçon qu'elle aime.

$$\begin{array}{rcl} P_3 & = & ([A,X] \vee [A,Z]) \\ & \wedge & [B,Y] \\ & \wedge & ([C,X] \vee [C,Y]) \\ & \wedge & ([B,X] \vee [C,X]) \\ & \wedge & ([A,Y] \vee [B,Y]) \\ & \wedge & ([A,Z] \vee [C,Z]) \end{array}$$

Donc, on arrive à la formule

$$P_1 \wedge P_1' \wedge P_2 \wedge P_2' \wedge P_3$$

Maintenant, on constate que chacune des clauses de P_1 et de P_1' sont subsumées par des clauses en P_3. Il y a donc encore une fois une redondance, mais maintenant le cas est différent de ce qu'on avait observé à la première étape : la subsomption est un cas particulier de la redondance où la clause subsumée ne peut pas donner un avantage à l'algorithme DPLL. Les clauses subsumées ne servent donc vraiment à rien, et on peut les supprimer. On arrive finalement à la formule

$$P_2 \wedge P_2' \wedge P_3$$

qui est la meilleure (mais pas la seule) solution.

Exercice 14 *Est-ce que* $P_2 \wedge P_3$ *serait une solution correcte ? En général, dire exactement pour quels choix des formules parmi* $P_1, P_1', P_2, P_2', P_3$ *leur conjonction est une solution correcte du problème.*

6.3 Extension : contraintes de comptage

Fréquemment on doit modéliser une contrainte d'une des formes suivantes, pour un sous-ensemble $Y \subseteq X$ de variables et un entier n :

1. au moins n des variables dans Y sont vraies ;
2. au plus n des variables dans Y sont vraies ;
3. exactement n des variables dans Y sont vraies.

On parle alors d'une *contrainte de comptage*. Dans la coloration de la carte d'Australie étudiée dans la section 6.1, par exemple, on pourrait ajouter la contrainte que au moins 2 (ou au plus 2, ou précisément 2) des régions sont colorées en rouge. Dans ce cas, $n = 2$ et l'ensemble Y est l'ensemble des variables qui correspondent à une coloration rouge d'une des régions :

$$Y = \{\texttt{[WA,r]}, \texttt{[NT,r]}, \texttt{[SA,r]}, \texttt{[QLD,r]}, \texttt{[NSW,r]}, \texttt{[VIC,r]}, \texttt{[TAS,r]}\}$$

Comment exprimer une contrainte de comptage ? On voit d'abord que la troisième forme, qui demande que exactement n parmi les variables de Y sont vraies, peut être exprimée à l'aide des deux premières : il faut que au moins n des variables dans Y, et aussi au plus n des variables dans Y sont vraies. Regardons d'abord la contrainte : au moins n des variables dans Y sont vraies. Supposons, pour simplifier la notation, que $Y = \{y_1, \ldots, y_m\}$. L'ensemble Y contient alors m variables.

Si $n = 1$ la tãche est facile :

$$y_1 \vee y_2 \vee \ldots \vee y_m$$

Si $n = 2$ on peut engendrer toutes les paires de deux variables différentes de Y, et exprimer le fait que au moins une de ces paires est entièrement colorée en rouge :

$$(y_1 \wedge y_2) \vee \ldots \vee (y_1 \wedge y_m) \vee (y_2 \wedge y_3) \vee \ldots \vee (y_2 \wedge y_m) \vee \ldots \vee (y_{m-1} \wedge y_m)$$

Plus précisément :

$$\bigvee_{\substack{i=1,\ldots,m-1 \\ j=i+1,\ldots,m}} (y_i \wedge y_j)$$

Cette construction se généralise à un nombre n quelconque :

$$\bigvee_{\substack{i_1,\ldots,i_n \in \{1,\ldots,m\} \\ i_1 < \ldots < i_n}} \bigwedge_{j=1,\ldots,n} y_{i_j}$$

Dans cette forme générale, on construit une grande disjonction sur tous les n-uplets d'indices entre 1 et m, où en plus on impose que les indices dans chacun de ces n-uplets sont arrangés dans un ordre croissant. Pour chacun de ces n-uplets on exprime par la grande conjonction que les variables indiquées par ces indices sont vraies.

Malheureusement, la formule ainsi obtenue est on forme disjonctive normale et pas en forme conjonctive normale. On pourrait utiliser l'algorithme présenté en section 4.5 pour transformer cette formule en forme conjonctive normale. Une solution alternative est de construire la formule directement en forme conjonctive normale, en utilisant l'idée suivante : Dire que au moins n parmi les variables de Y sont vraies est équivalent à dire que pour n'importe quelle sélection de $m - n + 1$ variables de Y, au moins une dans cette sélection est vraie.

Exercice 15 *Soit $Y = \{y_1, \ldots, y_m\}$ un ensemble de variables propositionnelles, $n \leq m$, et v une affectation. Montrer que les deux énoncés suivants sont équivalentes :*

1. *il y a au moins n variables dans Y qui sont envoyées vers 1 par v ;*
2. *tout sous-ensemble $Z \subseteq Y$ de cardinalité $m - n + 1$ contient au moins une variable qui est envoyée vers 1 par v.*

L'avantage de cette formulation est qu'elle s'exprime naturellement en forme conjonctive normale :

$$\bigwedge_{\substack{i_1,\ldots,i_{m-n+1} \\ i_1<\ldots<i_{m-n+1}}} \bigvee_{j=1,\ldots,m-n+1} y_{i_j}$$

Dans notre exemple de la coloration de la carte d'Australie : Il y a au moins $n = 2$ régions colorées en rouge si dans tout choix de $7-2+1 = 6$ régions il y a au moins une qui est rouge :

$$\begin{aligned}
& (\texttt{[WA,r]} \vee \texttt{[NT,r]} \vee \texttt{[SA,r]} \vee \texttt{[QLD,r]} \vee \texttt{[NSW,r]} \vee \texttt{[VIC,r]}) \\
\wedge\ & (\texttt{[WA,r]} \vee \texttt{[NT,r]} \vee \texttt{[SA,r]} \vee \texttt{[QLD,r]} \vee \texttt{[NSW,r]} \vee \texttt{[TAS,r]}) \\
\wedge\ & (\texttt{[WA,r]} \vee \texttt{[NT,r]} \vee \texttt{[SA,r]} \vee \texttt{[QLD,r]} \vee \texttt{[VIC,r]} \vee \texttt{[TAS,r]}) \\
\wedge\ & (\texttt{[WA,r]} \vee \texttt{[NT,r]} \vee \texttt{[SA,r]} \vee \texttt{[NSW,r]} \vee \texttt{[VIC,r]} \vee \texttt{[TAS,r]}) \\
\wedge\ & (\texttt{[WA,r]} \vee \texttt{[NT,r]} \vee \texttt{[QLD,r]} \vee \texttt{[NSW,r]} \vee \texttt{[VIC,r]} \vee \texttt{[TAS,r]}) \\
\wedge\ & (\texttt{[WA,r]} \vee \texttt{[SA,r]} \vee \texttt{[QLD,r]} \vee \texttt{[NSW,r]} \vee \texttt{[VIC,r]} \vee \texttt{[TAS,r]}) \\
\wedge\ & (\texttt{[NT,r]} \vee \texttt{[SA,r]} \vee \texttt{[QLD,r]} \vee \texttt{[NSW,r]} \vee \texttt{[VIC,r]} \vee \texttt{[TAS,r]})
\end{aligned}$$

Considérons maintenant la deuxième forme d'une contrainte de comptage : au plus n des variables dans Y sont vraies. Cela est équivalent à dire que au moins $m - n$ des variables de Y sont fausses. Cet énoncé a donc presque la même forme que celle que nous venons d'étudier, sauf qu'on parle maintenant de variables fausses à la place de variables vraies. Il suffit donc de prendre la même construction, et d'appliquer une négation à toutes les variables. Les sélections des variables, pour lesquelles il faut exprimer qu'elles contiennent une variable fausse, ont maintenant $m - (m - n) + 1 = n + 1$ éléments :

$$\bigwedge_{\substack{i_1,\ldots,i_{n+1} \\ i_1<\ldots<i_{n+1}}} \bigvee_{j=1,\ldots,n+1} \neg y_{i_j}$$

Sur notre exemple Australien : on a au plus deux régions rouge si dans toute sélection de trois régions il y a au moins une qui n'est pas rouge :

$$\begin{aligned}
& (\neg\texttt{[WA,r]} \vee \neg\texttt{[NT,r]} \vee \neg\texttt{[SA,r]}) \\
\wedge\ & (\neg\texttt{[WA,r]} \vee \neg\texttt{[NT,r]} \vee \neg\texttt{[QLD,r]}) \\
\wedge\ & (\neg\texttt{[WA,r]} \vee \neg\texttt{[NT,r]} \vee \neg\texttt{[NSW,r]}) \\
& \vdots \qquad\qquad\qquad\qquad \vdots \\
\wedge\ & (\neg\texttt{[NSW,r]} \vee \neg\texttt{[VIC,r]} \vee \neg\texttt{[TAS,r]})
\end{aligned}$$

6.4 Le format DIMACS

Il s'agit d'un format standard pour les formules propositionnelles en forme normale conjonctive. Le nom de ce format est dû au *Center for Discrete Mathematics and Theoretical Computer Science*, centre de recherche américain qui avait organisé les premières compétitions internationales de SAT solveurs.

Pour écrire une formule en format DIMACS on suppose d'abord que les variables propositionnelles soient numérotées à partir de 1. La première ligne d'un fichier rédigé dans ce format contient l'entête qui consiste en le mot `p`, puis le mot `cnf`, puis le numéro maximal des variables utilisées dans la formule, puis le nombre de clauses disjonctives de la formules. Chacune des lignes suivantes décrit une clause disjonctive de la formule : pour décrire une clause on écrit simplement la liste des numéros de toutes les variables contenues dans la clause, avec un signe positif quand il s'agit d'un littéral positif, et avec un signe négatif quand il s'agit d'un littéral négatif. Toute ligne décrivant une clause (mais pas l'entête) se termine sur 0.

Par exemple, la formule

$$(x_1 \vee \neg x_5) \wedge (x_2 \vee x_5 \vee \neg x_1)$$

s'écrit en format DIMACS comme suit :

```
p cnf 5 2
1 -5 0
2 5 -1 0
```

6.5 Références et remarques

Les états (et, pour être exact, les territoires) de l'Australie sont Western Australia (WA), Northern Territory (NT), South Australia (SA), Queensland (QLD), New South Wales (NSW), Victoria (VIC), et Tasmania (TAS) si on ignore le tout petit territoire de la capitale (ACT).

Une partie du travail fait dans les codages de ce chapitre est due au fait que les variables propositionnelles ne peuvent prendre que deux valeurs différentes, tandis que souvent le problème d'origine parle de choix entre plus que deux valeurs possibles. La *programmation logique par contraintes*, qui va être présentées dans des cours du M1 et du M2, permet de raisonner directement avec des choix multiples entre valeurs (appelés des *domaines finis*).

Deuxième partie

Logique pour la programmation

Chapitre 7

Les expressions

Nous allons au chapitre 8 définir un petit langage de programmation. Les expressions arithmétiques sont un composant important de ce langage, et méritent que nous leur consacrons un chapitre à part.

7.1 Syntaxe des expressions arithmétiques et booléennes

Il y a un seul type de données dans le le langage IMP, c'est le type des entiers. Les variables des programmes IMP ne peuvent donc que contenir des valeurs entières. Les expressions booléennes existent mais ne servent que pour écrire les conditions dans les instructions conditionnelles et dans les boucles ; il n'y a pas de variables propositionnelles. Ce choix simplifie le formalisme car avec un seul type de données il n'y a pas de typage des variables, et par conséquent on peut complètement écarter les déclarations de variables de la définition du langage IMP.

Nous fixons d'abord un ensemble de variables arithmétiques :

$$X := \{x, x_1, x_2, x_3, \ldots, y, y_1, y_2, \ldots, z, z_1, z_2, z_3, \ldots\}$$

Nous admettons également les décorations des variables comme par exemple x', y''.

Nous définissons l'ensemble des expressions arithmétiques et l'ensemble des expressions booléennes par induction, suivant le même principe que dans la définition de la syntaxe des formules propositionnelles (définition 1). Cette fois nous donnons des définitions avec un nombre minimal de cas, et nous définissons toutes les autres notions que nous souhaitons en pratique comme des abréviations.

Définition 15 *L'ensemble des expressions arithmétiques AExpr est le plus petit ensemble de chaînes de caractères tel que :*

1. $X \subseteq AExpr$
2. $\mathbb{N} \subseteq AExpr$
3. *Si* $e \in AExpr$ *alors* $-e \in AExpr$
4. *Si* $e_1, e_2 \in AExpr$ *alors* $(e_1 + e_2) \in AExpr$
5. *Si* $e_1, e_2 \in AExpr$ *alors* $(e_1 * e_2) \in AExpr$

On définit $e_1 - e_2$ comme abréviation de $e_1 + (-e_1)$.

Définition 16 *L'ensemble des expressions booléennes BExpr est le plus petit ensemble de chaînes de caractères tel que :*

1. *Si* $e_1, e_2 \in AExpr$ *alors* $(e_1 \leq e_2) \in BExpr$
2. *Si* $f \in BExpr$ *alors* $\neg f \in BExpr$
3. *Si* $f_1, f_2 \in BExpr$ *alors* $(f_1 \vee f_2) \in BExpr$

On appelle une expression booléenne atomique *si elle est de la forme* $e_1 \leq e_2$.

Remarquez qu'on utilise pour un cas de la définition inductive des expressions booléennes les expressions arithmétiques, un ensemble qui est lui-même défini par induction.

Nous permettons aussi les abréviations habituelles suivantes :

Abbréviation	Définition
`True`	$0 \leq 0$
`False`	$1 \leq 0$
$f_1 \wedge f_2$	$\neg(\neg f_1 \vee \neg f_2)$
$e_1 = e_2$	$e_1 \leq e_2 \wedge e_2 \leq e_1$
$e_1 \geq e_2$	$e_2 \leq e_1$
$e_1 < e_2$	$e_1 + 1 \leq e_2$
$e_1 > e_2$	$e_2 + 1 \leq e_1$
$e_1 \neq e_2$	$e_1 < e_2 \vee e_1 > e_2$

ainsi que les abréviations de la logique propositionnelle vues à la section 3.5. La définition de l'abréviation $e_1 < e_2$ est justifiée par le fait que les expressions arithmétiques sont interprétées comme des entiers, et pas comme des réels.

7.2 Sémantique des expressions arithmétiques et booléennes

L'évaluation des expressions dépend, comme l'interprétation des formules propositionnelles, du contexte. Dans le cas des expressions, comme aussi un peu plus tard des programmes (section 8.2), un contexte est une fonction qui associe des valeurs à des variables. Cette notion de contexte semble naturelle pour un langage aussi simple que le langage que nous considérons dans ce cours. On peut voir une telle fonction, appelée *affectation* comme dans le cas de la logique propositionnelle, comme l'état de la mémoire d'un ordinateur qui exécute un programme. Il y a derrière cette modélisation un choix qui n'est pas complètement innocent, nous reviendrons à cette problématique à la fin de la section 8.2.

Définition 17 *Une* affectation *est une fonction*

$$\sigma : X \to \mathbb{Z}$$

L'ensemble des affectations est noté Aff. Le support *d'une affectation v est définie comme*

$$supp(\sigma) = \{x \in X \mid \sigma(x) \neq 0\}$$

Une affectation est donc toujours une fonction totale. Nous utilisons une notation pour les affectations :

$$x_1 \mapsto n_1, x_2 \mapsto n_2, \ldots, x_m \mapsto n_m$$

est l'affectation qui associe à toute variable x_i (avec $1 \leq i \leq m$) la valeur n_i, et qui associe 0 à toute autre variable.

Définition 18 *L'évaluation $[\![e]\!]\sigma$ d'une expression arithmétique e par rapport à l'affectation σ est définie par récurrence sur la structure de e :*

$$\begin{array}{lcl} [\![x]\!]\sigma & = & \sigma(x) \\ [\![n]\!]\sigma & = & n \\ [\![-e]\!]\sigma & = & -[\![e]\!]\sigma \\ [\![(e_1 + e_2)]\!]\sigma & = & [\![e_1]\!]\sigma + [\![e_2]\!]\sigma \\ [\![(e_1 * e_2)]\!]\sigma & = & [\![e_1]\!]\sigma * [\![e_2]\!]\sigma \end{array}$$

Par exemple, avec $\sigma = [x \mapsto 1; y \mapsto 2; z \mapsto 3]$ on a que

$$\begin{array}{ll} & [\![((x + y) * (x * z))]\!]\sigma \\ = & [\![(x + y)]\!]\sigma * [\![(x * z)]\!]\sigma \\ = & ([\![x]\!]\sigma + [\![y]\!]\sigma) * ([\![x]\!]\sigma * [\![z]\!]\sigma) \\ = & (\sigma(x) + \sigma(y)) * (\sigma(x) * \sigma(z)) \\ = & (1 + 2) * (1 * 3) \\ = & 9 \end{array}$$

La sémantique des expressions booléennes est définie de façon similaire :

Définition 19 *L'évaluation $[\![f]\!]\sigma$ d'une expression booléenne f par rapport à l'affectation σ est définie par récurrence sur la structure de f :*

$$\begin{array}{lcl} [\![(e_1 \leq e_2)]\!]\sigma & = & \begin{cases} 1 & si\ [\![e_1]\!]\sigma \leq [\![e_2]\!]\sigma \\ 0 & si\ [\![e_1]\!]\sigma > [\![e_2]\!]\sigma \end{cases} \\ [\![\neg f]\!]\sigma & = & \begin{cases} 0 & si\ [\![f]\!]\sigma = 1 \\ 1 & si\ [\![f]\!]\sigma = 0 \end{cases} \\ [\![(f_1 \vee f_2)]\!]\sigma & = & \begin{cases} 0 & si\ [\![f_1]\!]\sigma = 0\ et\ [\![f_2]\!]\sigma = 0 \\ 1 & si\ [\![f_1]\!]\sigma = 1\ ou\ [\![f_2]\!]\sigma = 1 \end{cases} \end{array}$$

Définition 20 *Soit f une expression booléenne .*
- *On écrit $\sigma \models f$ si $[\![f]\!]\sigma = 1$.*
- *On dit que f est* satisfaisable *s'il existe une affectation σ telle que $\sigma \models f$.*
- *On dit que f est* valide *si $\sigma \models f$ pour toute affectation σ.*

Remarquez l'analogie avec la définition 4 de la logique propositionnelle.

7.3 Validité d'expressions booléennes

D'une certaine façon, les expressions booléennes sont construites à partir des formules booléennes en remplaçant les variables propositionnelles par des expressions booléennes *atomiques* de la forme $(e_1 \leq e_2)$. On peut donc naturellement transférer les définitions et résultats de la logique propositionnelle aux expressions booléennes :

Définition 21 *Une* tautologie *est une expression booléenne de la forme* $p[x_1/f_1, \ldots, x_n/f_n]$ *où* p *est une tautologie propositionnelle,* $\mathcal{V}(p) = \{x_1, \ldots, x_n\}$ *et* $f_1, \ldots, f_n$ *sont des expressions booléennes.*

En d'autres mots, une tautologie (dans le sens des expressions booléennes) est obtenue en remplaçant dans une tautologie propositionnelle (c'est-à-dire une formule propositionnelle valide) toutes les variables propositionnelles par des expressions booléennes. Par exemple :

$$(x \leq y) \vee \neg(x \leq y)$$

est une tautologie, car cette expression booléenne est obtenue en remplaçant dans la tautologie propositionnelle $z \vee \neg z$ la variable propositionnelle z par l'expression arithmétique $x \leq y$. Un autre exemple est

$$\neg((x \leq y * z) \vee (z + y \leq x * x)) \leftrightarrow (\neg(x \leq y * z) \wedge \neg(z + y \leq x * x))$$

car cette formule est obtenue en remplaçant dans la tautologie $\neg(x' \vee y') \leftrightarrow (\neg x' \wedge \neg y')$ la variable propositionnelle x' par $(x \leq y * z)$ et la variable propositionnelle y' par $(z + y \leq x * x)$.

Une remarque concernant notre notation : Nous avons choisi d'utiliser le même répertoire X de variables à la fois pour les variables propositionnelles et pour les variables arithmétiques. Ce choix a l'avantage d'éviter l'introduction d'une nouvelle notation. Normalement, quand on étudie soit la logique propositionnelle, soit la logique de Hoare il n'y a pas de risque de confusion, c'est seulement dans cette section qu'on a les deux au même temps et qu'il faut bien préciser quand on parle de variables propositionnelles et quand on parle d'expressions arithmétiques.

Théorème 17 *Toute tautologie (dans le sens de la définition 21) est valide.*

Ce théorème découle facilement de la définition d'une tautologie. On obtient également des algorithmes pour décider si une expression booléenne est une tautologie : il suffit de remplacer toutes les occurrences de la même expression atomique par une nouvelle variable propositionnelle, et d'utiliser une des méthodes vues dans les chapitres 2, 3 et 5 pour savoir si la formule propositionnelle ainsi obtenue est une tautologie (dans le sens de la logique propositionnelle).

Il y a une subtilité dans cette méthode qui est dûe au fait que nous permettons dans la définition d'une tautologie de remplacer une variable propositionnelle par une expression booléenne quelconque, et pas seulement par une expression atomique. Par exemple, quand nous remplaçons dans la tautologie (propositionnelle) $x \vee \neg x$ la variable x par l'expression booléenne $(y > z * z \vee z * z > y)$ nous obtiendrons la tautologie

$$(y > z * z \vee z * z > y) \vee \neg(y > z * z \vee z * z > y)$$

Or, quand nous appliquons la méthode décrite ci-dessus nous obtiendrons pas exactement la même structure : On commence par remplacer dans la formule toute expression booléenne *atomique* par une nouvelle variable (si on a plusieurs occurrences de la même expression atomique on les remplace toutes par la même variable). Ici nous avons deux expressions atomiques différentes, $(y > z * z)$ et $(z * z > y)$. Si on choisit pour la première expression la variable propositionnelle x_1 et pour la deuxième la variable propositionnelle x_2 on obtient donc comme formule propositionnelle

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

qui est aussi une tautologie (propositionnelle) ! La question est : Est-ce qu'on tombe avec notre méthode de décomposition toujours sur une tautologie quand l'expression de départ est une tautologie ? La réponse est « oui », et la raison est le théorème 5 ! Sur notre exemple ce théorème nous dit que puisque $x \vee \neg x$ est une tautologie propositionnelle, la formule $(x_1 \vee x_2) \vee \neg(x_1 \vee x_2)$ l'est aussi. On ne peut donc rien perdre par notre méthode de décomposition, et notre méthode est complète.

Tandis que toute tautologie est valide, l'inverse n'est pas vrai : il y a des expressions valides qui ne sont pas des tautologies. Voici quelques exemples :

$$\begin{aligned}
&((x > y) \wedge (y > z)) \rightarrow (x > z)\\
&((x > y) \rightarrow (x + z > y + z)\\
&(x * y = 0) \rightarrow (x = 0 \vee y = 0)\\
&((x_1 = y_1) \wedge (x_2 = y_2)) \rightarrow (x_1 + x_2 = y_1 + y_2)\\
&y \neq 0 \rightarrow x * x \neq 2 * (y * y)
\end{aligned}$$

La première expression exprime une propriété de la relation $>$ (la transitivité), la deuxième et la troisième expriment des propriétés des opérateurs arithmétiques. La quatrième expression est aussi valide mais pour des raisons qui sont complètement indépendantes de l'arithmétique, cette expression est simplement valide à cause des propriétés de l'égalité. La dernière expression exprime le fait que le nombre rationnel $\frac{x}{y}$ n'est pas la racine de 2, cette expression est donc valide car aucun nombre rationnel est la racine de 2.

On voit sur ces exemples qu'il faut en général une bonne connaissance de l'arithmétique pour savoir si une expression booléenne est valide ou pas. Une question intéressante est s'il y a un algorithme qui peut, analogue au cas de la logique propositionnelle, décider pour n'importe quelle expression booléenne si elle valide ou pas. Il est a priori concevable qu'un tel algorithme n'existe pas, nous verrons à la fin de cet ouvrage (au chapitre 12) quelques exemples de problèmes pour lesquels un algorithme *ne peut pas exister*.

Pour donner une idée de la difficulté du problème, l'expression booléenne

$$3 * x * x * x * y * y - 2 * x * x * z + 7 * z * z * z \neq 3$$

est valide si est seulement le polynôme $3x^3y^2 - 2x^2z + 7z^3 - 3$ n'a pas de racine *dans les entiers*. Cet exemple montre comment il est difficile en général de décider de la validité d'une expression booléenne.

Heureusement, on ne rencontre normalement pas d'expressions booléennes si difficiles que ça quand on s'intéresse à la correction de programmes. Les expressions booléennes pour lesquelles nous aurions à décider de la validité sont souvent des tautologies, ou elles sont valides pour des raisons assez simples.

Les résultats de la logique propositionnelle peuvent être généralisés pour obtenir des résultats analogues pour les expressions booléennes. Nous citons par exemple ici la proposition de substitution car nous en aurions besoin plus tard :

Proposition 12 *Pour toute expression booléenne f, variables différentes $x_1, \ldots, x_n$, expressions arithmétiques $e_1, \ldots, e_n$, et affectation v on a que*

$$[\![f[x_1/e_1, \ldots, x_n/e_n]]\!]v = [\![f]\!](v[x_1/[\![e_1]\!]v, \ldots, x_n/[\![e_n]\!]v])$$

La démonstration de cette proposition est analogue à la démonstration de la proposition 5.

7.4 Références et remarques

Le mathématicien David Hilbert a donné en 1900, lors d'un congrès international de mathématiciens, une liste de 23 problèmes mathématiques fondamentaux, et notre problème de validité des expressions booléennes est une reformulation du dixième problème, qui est aussi probablement le problème le plus célèbre de cette liste. La question d'un algorithme pour la validité des expressions booléennes a occupé les mathématicien pendant longtemps. C'était le mathématicien russe Yuri Matijacevič qui a finalement montré en 1970 qu'un tel algorithme ne peut pas exister [**?**].

Si on se restreint à des expressions booléennes sans le symbole de multiplication le problème devient décidable, c'est-à-dire il y a des algorithmes pour décider de la validité. Ces algorithmes sont normalement présentés dans un cours de *Démonstration automatique*, ou parfois dans un cours sur les *Automates*.

Les expressions booléennes présentées dans ce chapitre sont un cas particulier d'une logique plus expressive appelée *logique du premier ordre*. Cette logique contient des constructions comme par exemple l'existence d'une certaine valeur. Cela permet par exemple de dire « il existe un z tel que $x = z * y$ », en d'autres mots que y est un diviseur de x. Cette logique sera étudiée au cours de *logique* du L3.

Chapitre 8

Les programmes

Dans ce chapitre nous allons définir formellement la syntaxe et la sémantique d'un petit langage de programmation appelé IMP. Ce langage de programmation nous servira pour étudier la logique de Hoare au chapitre 9, il est simplifié au maximum et ne contient que les éléments les plus importants.

8.1 Syntaxe des programmes IMP

Pour définir la syntaxe des programmes IMP nous avons d'abord besoin de la notion d'une *liste*. Une liste est soit *vide*, et on la note alors ϵ, soit c'est une séquence de plusieurs éléments. On note une telle liste composée de n éléments e_1 à e_n par

$$[e_1; \ldots; e_n]$$

La longueur d'une liste vide est 0, et la longueur d'une liste composée de n éléments est n. Par exemple,

$$[(x + y); (17 + z_1); (42 * (y + 5))]$$

est une liste d'expression arithmétiques, et sa longueur est 3. Pour être exact il faut assurer que la notation n'est pas ambiguë, ce qui est assuré quand les éléments d'une liste ne peuvent pas se terminer sur le symbole « ; ». Une notion importante est l'opération de *concaténation* de deux listes : Si l est une liste $[e_1; \ldots; e_n]$ et l' une liste $[e'_1; \ldots; e'_m]$ alors leur *concaténation* est la liste

$$[e_1; \ldots, e_n; e'_1, \ldots; e'_m]$$

Nous écrivons $l; l'$ pour cette liste. Il s'agit en principe d'un abus de notation car de cette façon nous utilisons le symbole ; pour deux choses différentes, d'un part pour combiner deux *éléments* d'une liste, et d'autre part pour combiner deux *listes*. Puisque nous considérons ici pas des listes dont les éléments sont eux-mêmes des listes cela ne risque pas d'introduire une ambiguïté, et nous pouvons autoriser cette notation.

En particulier, la liste vide est l'élément neutre de l'opération de concaténation, c'est-à-dire $\epsilon; l = l$ et $l; \epsilon = l$.

Définition 22 *L'ensemble Inst des* instructions *est le plus petit ensemble de chaînes de caractères tel que :*

1. *si* $x \in X$ *et* $e \in AExpr$ *alors* $x := e \in Inst$ *;*
2. *si* $f \in BExpr$ *et* S_1, S_2 *sont des listes d'éléments de Inst alors* **if** f **then** S_1 **else** S_2 **fi** $\in Inst$ *;*
3. *si* $f \in BExpr$ *et* S *est une liste d'éléments de Inst alors* **while** f **do** S **od** $\in Inst$.

Un programme *est une liste d'instructions. L'ensemble des programmes est noté Imp.*

On définit **if** f **then** S **fi** comme abréviation de **if** f **then** S **else** ϵ **fi**. On remarque que notre définition de la syntaxe exige un « parenthésage » du corps d'une boucle (**do** ... **od**) et des deux branches d'une instructions conditionnelle (**then** ... **else** et **else** ... **fi**). La raison est que si on n'avait pas exigé le **fi** à la fin d'une instruction conditionnelle alors on aurait une ambiguïté dans le cas de programmes comme

$$\textbf{if } x \leq 0 \textbf{ then } y := y - x \textbf{ else } y := y + x; y := y + y$$

car on ne saurait pas si la dernière affectation fait partie de la deuxième branche de l'instruction conditionnelle, ou si elle suit après l'instruction conditionnelle. Avec notre définition de la syntaxe on a un théorème de lecture unique comme nous l'avons déjà vu pour la logique propositionnelle.

Un exemple est le programme dont nous avons déjà parlé dans l'introduction :

```
z := 0;
y1 := 0;
while y1 ≠ y do
      z := z + x;
      y1 := y1 + 1;
od
```

8.2 Sémantique des programmes IMP

La sémantique des programme est définie par des transitions entre configurations.

Définition 23 *L'ensemble des* configurations *est* $Conf = Imp \times Aff$.

En d'autres mots, une configuration est une paire d'un programme $S \in Imp$ et d'une affectation $\sigma \in Aff$. Intuitivement, S est le programme qui *reste à exécuter*, et l'affectation σ est l'état de la mémoire. On aurait aussi pu prendre le contenu d'un *compteur de programme* au lieu du programme qui reste à exécuter ; la raison pour notre choix de modélisation est simplement que la représentation par un reste de programme est un peu plus simple à gérer qu'un compteur de programme.

La définition centrale est la relation de transition d'une configuration vers une autre.

Définition 24 *La relation* $\Rightarrow$ *entre configurations est la plus petite relation telle que*

- $\langle x := e; S, \sigma\rangle \Rightarrow \langle S, \sigma[x/[\![e]\!]\sigma]\rangle$
- $\langle \textbf{if } f \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S, \sigma\rangle \Rightarrow \begin{cases} \langle S_2; S, \sigma\rangle & si\ [\![f]\!]\sigma = 0 \\ \langle S_1; S, \sigma\rangle & si\ [\![f]\!]\sigma = 1 \end{cases}$

– $\langle \textbf{while } f \textbf{ do } S' \textbf{ od}; S, \sigma\rangle \Rightarrow \begin{cases} \langle S, \sigma\rangle & si\ [\![f]\!]\sigma = 0 \\ \langle S'; \textbf{while } f \textbf{ do } S' \textbf{ od}; S, \sigma\rangle & si\ [\![f]\!]\sigma = 1 \end{cases}$

Dans cette définition nous permettons aussi que le bout de programme p soit la liste vide. Intuitivement, la relation $\Rightarrow$ exprime une étape dans l'avancement de l'exécution d'un programme. Remarquez qu'il n'y a aucune configuration c telle que $\langle \epsilon, \sigma\rangle \Rightarrow c$, l'idée est qu'un programme vide ϵ dans une configuration représente le fait que l'exécution du programme a terminé (il ne reste plus rien à faire).

La transition pour le cas d'une affectation se lit comme suit : on évalue l'expression e par rapport à l'affectation actuelle σ, puis on met à jour l'affectation (l'état de la mémoire) et on passe à la suite. Dans le cas d'une conditionnelle on évalue la condition f pour savoir laquelle des deux branches S_1 ou S_2 est à exécuter, et après avoir exécuté cette branche on continue avec ce qui suit après la conditionnelle, c'est-à-dire S. Dans le cas d'une boucle on évalue la condition, si la condition est fausse on passe à ce qui suit après la boucle, si la condition est vraie on exécute d'abord le corps S' de la boucle, et puis on exécute de nouveau la boucle.

Voici un exemple avec une instruction conditionnelle :

$$\begin{aligned}
& \langle x := 2; y := 5; x := x * y; \textbf{if } x > y + 1 \textbf{ then } z := 0 \textbf{ else } z := 1 \textbf{ fi}; z := z + 10, []\rangle \\
\Rightarrow\ & \langle y := 5; x := x * y; \textbf{if } x > y + 1 \textbf{ then } z := 0 \textbf{ else } z := 1 \textbf{ fi}; z := z + 10, [x \mapsto 2]\rangle \\
\Rightarrow\ & \langle x := x * y; \textbf{if } x > y + 1 \textbf{ then } z := 0 \textbf{ else } z := 1 \textbf{ fi}; z := z + 10, [x \mapsto 2; y \mapsto 5]\rangle \\
\Rightarrow\ & \langle \textbf{if } x > y + 1 \textbf{ then } z := 0 \textbf{ else } z := 1 \textbf{ fi}; z := z + 10, [x \mapsto 10; y \mapsto 5]\rangle \\
\Rightarrow\ & \langle z := 0; z := z + 10, [x \mapsto 10; y \mapsto 5]\rangle \\
\Rightarrow\ & \langle z := z + 10, [x \mapsto 10; y \mapsto 5; z \mapsto 0]\rangle \\
\Rightarrow\ & \langle \epsilon, [x \mapsto 10; y \mapsto 5; z \mapsto 10]\rangle
\end{aligned}$$

Un deuxième exemple avec une boucle :

$$\begin{aligned}
& \langle x := 3; y := 2; \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, []\rangle \\
\Rightarrow\ & \langle y := 2; \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 3]\rangle \\
\Rightarrow\ & \langle \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 3; y \mapsto 2]\rangle \\
\Rightarrow\ & \langle x := x - 1; y := y * y; \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 3; y \mapsto 2]\rangle \\
\Rightarrow\ & \langle y := y * y; \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 2; y \mapsto 2]\rangle \\
\Rightarrow\ & \langle \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 2; y \mapsto 4]\rangle \\
\Rightarrow\ & \langle x := x - 1; y := y * y; \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 2; y \mapsto 4]\rangle \\
\Rightarrow\ & \langle y := y * y; \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 1; y \mapsto 4]\rangle \\
\Rightarrow\ & \langle \textbf{while } x > 1 \textbf{ do } x := x - 1; y := y * y \textbf{ od}, [x \mapsto 1; y \mapsto 16]\rangle \\
\Rightarrow\ & \langle \epsilon, [x \mapsto 1; y \mapsto 16]\rangle
\end{aligned}$$

Finalement on peut définir la sémantique des programmes :

Définition 25 *Soit p un programme et σ et σ' des affections. La* sémantique *du programme S par rapport à σ est σ' quand il existe une séquence*

$$\langle S, \sigma\rangle \Rightarrow \langle S_1, \sigma_1\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma'\rangle$$

On écrit alors $[\![S]\!]\sigma = \sigma'$ Quand une telle séquence n'existe pas alors la sémantique n'est pas définie, on écrit alors $[\![S]\!]\sigma = \bot$.

Il faut bien sûr montrer que cette définition de la sémantique n'est pas ambiguë :

Proposition 13 *Pour tout programme S et affectation σ il existe au plus une affectation σ' telle que $\langle S, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma'\rangle$.*

Démonstration: On constate d'abord, par inspection de la définition 24 que pour tout programme S_1 et affectation σ_1 il existe au plus un programme S_2 et une affectation σ_2 tels que $\langle S_1, \sigma_1\rangle \Rightarrow \langle S_2, \sigma_2\rangle$.
Supposons maintenant par l'absurde que $\langle S, \sigma\rangle \Rightarrow \ldots \langle \epsilon, \sigma'\rangle$ et aussi $\langle S, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma''\rangle$ avec $\sigma' \neq \sigma''$. Puisqu'il n'y aucune transitions possible à partir d'une configuration avec un programme vide on a alors forcement une divergence :

$$\begin{array}{rcl} \langle S, \sigma\rangle & \Rightarrow \ldots \Rightarrow & \langle S_1, \sigma_1\rangle \\ \langle S_1, \sigma_1\rangle & \Rightarrow & \langle S_2, \sigma_2\rangle \\ \langle S_1, \sigma_1\rangle & \Rightarrow & \langle S_2', \sigma_2'\rangle \end{array}$$

avec $\langle S_2, \sigma_2\rangle \neq \langle S_2', \sigma_2'\rangle$.
Ce qui est absurde. □

Le cas de la boucle dans la définition de la sémantique est remarquable car on passe d'un programme à un programme qui est *plus grand*, on a même que le programme qui parait sur la gauche de la relation $\Rightarrow$ est contenu dans le programme qui parait sur la droite ! Cela explique pourquoi nous n'avons pas simplement défini la sémantique des programmes par récurrence, comme nous l'avons fait pour les formules propositionnelles et les expressions. En fait il est possible de définir la sémantique aussi des programmes IMP par récurrence structurelle, on parle alors d'une sémantique *dénotationnelle* tandis que nous avons défini ici une sémantique *opérationnelle*. Mais cela nécessite quelque résultats mathématiques de la théorie des ordres partiels, voir un cours de Sémantique.

Proposition 14 *Pour tous programmes $S_1, S_2, S_3 \in Imp$ et affectations $\sigma_1, \sigma_2 \in Aff$: Si*

$$\langle S_1, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle S_2, \sigma_2\rangle$$

on a alors aussi que

$$\langle S_1; S_3, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle S_2; S_3, \sigma_2\rangle$$

Exercice 16 *Montrer la proposition 14.*
L'énoncé inverse de la proposition 14 est : Si

$$\langle S_1; S_3, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle S_2; S_3, \sigma_2\rangle$$

on a alors aussi que

$$\langle S_1, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle S_2, \sigma_2\rangle$$

Est-ce que cet énonée est vrai ?

Proposition 15 *Si on a que*
- $[\![f]\!]\sigma = 1$
- $[\![S]\!]\sigma = \sigma_1$

– $[\![\textbf{while } f \textbf{ do } S \textbf{ od}]\!]\sigma_1 = \sigma_2$
alors on a aussi que

$$[\![\textbf{while } f \textbf{ do } S \textbf{ od}]\!]\sigma = \sigma_2$$

Démonstration: Nous notons $p_b = [\textbf{while } f \textbf{ do } S \textbf{ od}]$. Puisque $[\![S]\!]\sigma = \sigma_1$ il existe une séquence

$$\langle S, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma_1\rangle$$

Par proposition 14, on a aussi que

$$\langle S; S_b, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle S_b, \sigma_1\rangle$$

Puisque $[\![\textbf{while } f \textbf{ do } S \textbf{ od}]\!]\sigma_1 = \sigma_2$ il existe une séquence

$$\langle S_b, \sigma_1\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma_2\rangle$$

Finalement, on obtient en enchaînant ces deux séquences et on utilisant le fait que $[\![f]\!]\sigma = 1$:

$$\langle \textbf{while } f \textbf{ do } S \textbf{ od}, \sigma\rangle \Rightarrow \langle S; S_b, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle S_b, \sigma_1\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma_2\rangle$$

□

Proposition 16 *Soit $p_1, p_2 \in Imp$ et $\sigma \in Aff$. Alors :*

1. *Si $[\![p_1]\!]\sigma = \perp$ alors $[\![p_1; p_2]\!]\sigma = \perp$*
2. *Si $[\![p_1]\!]\sigma \neq \perp$ alors $[\![p_1; p_2]\!]\sigma = [\![p_2]\!]([\![p_1]\!]\sigma)$*

Exercice 17 *Montrer la proposition 16.*

8.3 Références et remarques

On aurait aussi pu essayer de définir la sémantique avec une récurrence du genre de la proposition 15. Une difficulté est qu'avec le principe de la définition par récurrence on dit comment calculer le résultat d'une fonction appliquée à un argument composé connaissant les résultats des applications de la fonction à ces composantes. Or, la « récurrence » dans la proposition 15 n'est pas de cette forme car on obtient la sémantique de p_b à partir de la sémantique de p_b même (mais par rapport à une affectation différente). Il est toutefois possible de faire une telle récurrence correctement si on utilise par exemple des règles d'inférence, voir un cours de *Sémantique*. Dans ce cas on obtient une sémantique à *grands pas* car on défini de cette façon la sémantique de programmes de plus en plus grands. La sémantique que nous avons définie dans ce chapitre, par contre, est une sémantique à *petits pas* car elle procède par exécution des instructions élémentaires.

Il faut être conscient que notre modélisation des états d'une machine exécutant un programme par des affectations est une abstraction, et qu'on peut être obligé de changer la modélisation quand on essaye de modéliser des constructions plus sophistiquées des langages de programmation. Un premier exemple est une extension possible du langage de programmation par une notion de *pointeurs*. Si on souhaite modéliser un langage de programmation avec pointeurs on sera obligé de changer la notion d'affectation, et d'utiliser une liaison en deux étapes d'abord

des noms de variables vers des cases mémoire, puis des cases mémoire vers des valeurs. Un autre exemple un peu moins évident et celui des langages de programmation avec des procédures et passage des paramètres par référence. Dans ce cas on observe des effets de partage (la même case mémoire peut être accessible par plusieurs variables différentes) qui sont le mieux modélisés par une liaison en deux étapes.

Ce chapitre suit largement le chapitre 2 du livre [**?**], dont nous avons aussi emprunté le nom « IMP »pour le langage de programmation, et du chapitre 3 du livre [**?**]. *Imp*, qui ici est un acronyme pour « Imperative Programming Language « (langage de programmation impérative) est aussi le nom anglais pour une espèce de petits démons mythologiques [**?**].

Chapitre 9

Les formules de Hoare

9.1 Syntaxe et sémantique des formules de Hoare

Définition 26 *Une* formule de Hoare *est de la forme*

$$\{p\}\, S\, \{q\}$$

où $p, q \in BExpr$ sont des des expressions booléennes et $S \in Imp$ est un programme. L'ensemble des formules de Hoare est noté Hoare.

On appelle l'expression p dans $\{p\}\, S\, \{q\}$ la *pre-condition*, et q la *post-condition*. Une telle formule exprime le fait que *si* l'exécution du programme S dans un état de mémoire initial qui satisfait la pre-condition p termine, alors l'état de mémoire résultant de cette exécution satisfait la post-condition q. Par exemple,

$$\{x > 0\}\, x := x + 1; y := x * x\, \{y > 1\}$$
$$\{\texttt{True}\}\ \textbf{while}\ x \neq 0\ \textbf{do}\ x := x - 1\ \textbf{od}\ \{x = 0\}$$

sont deux formules de Hoare.

Définition 27 *Soit $\sigma \in Aff$ une affectation et $\{p\}\, S\, \{q\} \in Hoare$ une formule de Hoare. L'affectation σ satisfait la formule de Hoare $\{p\}\, S\, \{q\}$, noté*

$$\sigma \models \{p\}\, S\, \{q\}$$

si
– Si $\sigma \models p$
– et si $[\![S]\!]\sigma = \sigma' \neq \bot$
– alors $\sigma' \models q$.
La formule $\{p\}\, S\, \{q\}$ est valide*, noté $\models \{p\}\, S\, \{q\}$, si on a $\sigma \models \{p\}\, S\, \{q\}$ pour toute affectation $\sigma \in Aff$.*

Il y a donc deux hypothèses dans la définition de $\sigma \models \{p\}\, S\, \{q\}$: il faut que $\sigma \models p$, et il faut aussi que $[\![S]\!]\sigma \neq \bot$. Si une de ces deux hypothèses n'est pas satisfaite alors on a trivialement que $\sigma \models \{p\}\, S\, \{q\}$ (voir figure 9.1). On dit aussi que les formules de Hoare expriment des

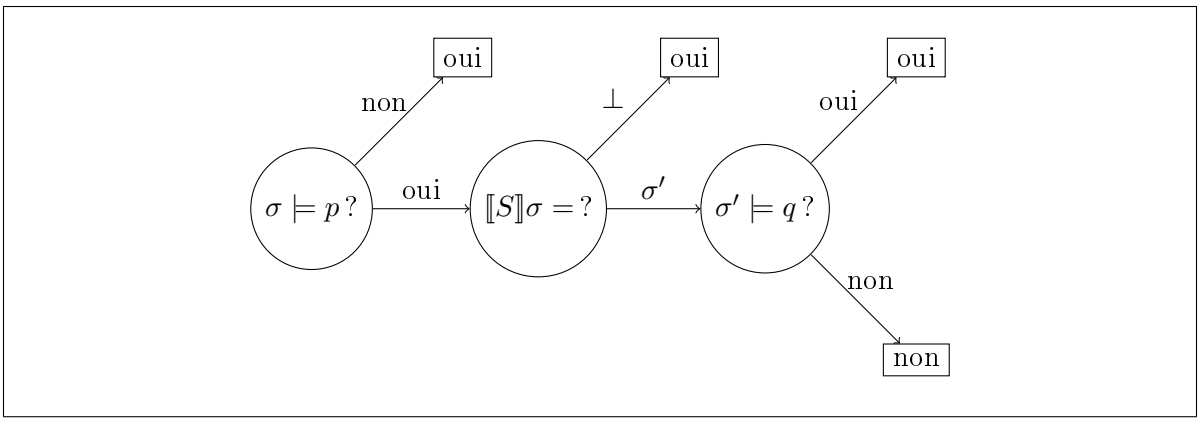


FIGURE 9.1: Évaluation d'une formule de Hoare $\{p\}\, S\, \{q\}$ par rapport à une affectation σ

énoncés de *correction partielle*. On dit « partiel » car une telle formule n'énonce pas la terminaison, contrairement à un énoncé de *correction totale* qui énonce aussi la terminaison du programme.

Pourquoi ces deux hypothèses dans les formules de Hoare ? La première hypothèse est assez naturelle car on ne s'intéresse souvent pas à l'exécution d'un morceaux de programme dans un état quelconque, mais seulement dans des états qui peuvent se produire à un certain point d'un programme. La raison pour la deuxième hypothèse, et la raison pourquoi on s'intéresse ici à des énoncés de correction partielle et pas de correction totale, est plus délicate. Nous reviendrons à cette question à la fin du chapitre (section 9.3).

9.2 Exemples

Exemple 1 On a

$$[x \mapsto 1] \models \{x = 1\}\, x := x + 1\, \{x = 2\}$$

car
- La pre-condition est satisfaite : $[x \mapsto 1] \models x = 1$
- L'exécution du programme dans l'état initial $[x \mapsto 1]$ termine et donne $[\![x := x + 1]\!][x \mapsto 1] = [x \mapsto 2]$
- Finalement, $[x \mapsto 2] \models x = 2$.

Exemple 2 On a aussi que

$$[x \mapsto 42] \models \{x = 1\}\, x := x + 1\, \{x = 2\}$$

car $[x \mapsto 42] \not\models x = 1$, la formule de Hoare est donc aussi satisfaite par l'affectation $[x \mapsto 42]$.

Exemple 3 On peut même dire que la formule $\{x = 1\}\, x := x + 1\, \{x = 2\}$ est valide. Pour le montrer soit $\sigma \in \mathit{Aff}$ une affectation quelconque.
- Soit on a que $\sigma \not\models x = 1$, et la formule de Hoare $\{x = 1\}\, x := x + 1\, \{x = 2\}$ est par conséquent satisfaite par σ.

– Soit on a que $\sigma \models x = 1$, et par conséquent que $\sigma(x) = 1$. Le programme $x := x + 1$ termine, c'est-à-dire il existe une affectation σ' avec $[\![x := 1]\!]\sigma = \sigma'$ D'après la définition de la sémantique des programmes on a que

$$\langle x := x + 1, \sigma\rangle \Rightarrow \langle \epsilon, \underbrace{\sigma[x/[\![x+1]\!]\sigma]}_{\sigma'}\rangle$$

Puisque $[\![x]\!]\sigma = 1$, on a que $[\![x + 1]\!]\sigma = 2$, donc $\sigma'(x) = 2$. Par conséquent, $\sigma' \models x = 2$.

Exemple 4 On a que

$$[x \mapsto 2, y \mapsto 3] \models \{True\}\ \textbf{if } y > x \textbf{ then } z := 1 \textbf{ else } z := 0 \textbf{ fi}\ \{z > 0\}$$

car on a d'abord que la pre-condition est satisfaite par l'affectation :

$$[x \mapsto 2, y \mapsto 3] \models True$$

Puis il suffit d'exécuter le programme :

$$\begin{array}{ll} & \langle \textbf{if } y > x \textbf{ then } z := 1 \textbf{ else } z := 0 \textbf{ fi}, [x \mapsto 2, y \mapsto 3]\rangle \\ \Rightarrow & \langle z := 1, [x \mapsto 2, y \mapsto 3]\rangle \\ \Rightarrow & \langle \epsilon, [x \mapsto 2, y \mapsto 3, z \mapsto 1]\rangle \end{array}$$

et évidemment $[x \mapsto 2, y \mapsto 3, z \mapsto 1] \models z > 0$. Par contre la formule de Hoare n'est pas valide car par exemple l'affectation $[x \mapsto 5, y \mapsto 3]$ ne la satisfait pas :

$$\begin{array}{ll} & \langle \textbf{if } y > x \textbf{ then } z := 1 \textbf{ else } z := 0 \textbf{ fi}, [x \mapsto 5, y \mapsto 3]\rangle \\ \Rightarrow & \langle z := 0, [x \mapsto 5, y \mapsto 3]\rangle \\ \Rightarrow & \langle \epsilon, [x \mapsto 5, y \mapsto 3, z \mapsto 0]\rangle \end{array}$$

et $[x \mapsto 5, y \mapsto 3, z \mapsto 0] \not\models z > 0$.

Exemple 5 Nous montrons que la formule

$$\{\texttt{True}\}\ \textbf{if } x > y \textbf{ then } z := x \textbf{ else } z := y \textbf{ fi}\ \{z \geq x \wedge z \geq y\}$$

est valide. Soit $\sigma \in \mathit{Aff}$ une affectation quelconque. On a évidemment que $\sigma \models \texttt{True}$, il faut donc vérifier que la post-condition est satisfaite par le résultat de l'exécution du programme avec état initial σ. Il y a deux cas, soit σ satisfait la condition $x > y$, soit σ ne la satisfait pas.

Cas $\sigma(x) > \sigma(y)$ On a alors

$$\begin{array}{ll} & \langle \textbf{if } x > y \textbf{ then } z := x \textbf{ else } z := y \textbf{ fi}, \sigma\rangle \\ \Rightarrow & \langle z := x, \sigma\rangle \\ \Rightarrow & \langle \epsilon, \underbrace{\sigma[z/\sigma(x)]}_{\sigma'}\rangle \end{array}$$

Maintenant on a que $\sigma'(z) = \sigma(x)$ et $\sigma'(x) = \sigma(x)$, donc $\sigma' \models z \geq x$. On a aussi que $\sigma'(y) = \sigma(y)$, et que $\sigma'(z) = \sigma(x) > \sigma(y)$ selon la distinction de cas. Donc, $\sigma' \models z \geq y$.

Cas $\sigma(x) \leq \sigma(y)$ Ce cas est analogue au premier cas :

$$\begin{aligned} & \langle \textbf{if } x > y \textbf{ then } z := x \textbf{ else } z := y \textbf{ fi}, \sigma \rangle \\ \Rightarrow \quad & \langle z := y, \sigma \rangle \\ \Rightarrow \quad & \langle \epsilon, \underbrace{\sigma[z/\sigma(y)]}_{\sigma'} \rangle \end{aligned}$$

Maintenant on a que $\sigma'(z) = \sigma(y)$ et $\sigma'(y) = \sigma(y)$, donc $\sigma' \models z \geq y$. On a aussi que $\sigma'(x) = \sigma(x)$, et que $\sigma'(z) = \sigma(y) \geq \sigma(x)$ selon la distinction de cas. Donc, $\sigma' \models z \geq x$.

Exemple 6 Maintenant des exemples avec une boucle. D'abord on a que

$$[x \mapsto 10] \models \{x > 0\} \ \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od} \ \{x = 52\}$$

La raison est simplement que $[x \mapsto 10]$ satisfait la pre-condition $x > 0$, mais l'exécution de la boucle ne termine pas :

$$\begin{aligned} & \langle \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od}, [x \mapsto 10] \rangle \\ \Rightarrow \quad & \langle x := x + 1; \textbf{while } x < 5 \textbf{ do } x := x + 1 \textbf{ od}, [x \mapsto 10] \rangle \\ \Rightarrow \quad & \langle \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od}, \sigma[x \mapsto 11] \rangle \\ \Rightarrow \quad & \langle x := x + 1; \textbf{while } x < 5 \textbf{ do } x := x + 1 \textbf{ od}, [x \mapsto 11] \rangle \\ \Rightarrow \quad & \langle \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od}, \sigma[x \mapsto 12] \rangle \\ \Rightarrow \quad & \langle x := x + 1; \textbf{while } x < 5 \textbf{ do } x := x + 1 \textbf{ od}, [x \mapsto 12] \rangle \\ \Rightarrow \quad & \langle \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od}, \sigma[x \mapsto 13] \rangle \\ \Rightarrow \quad & \ldots \end{aligned}$$

Pour la même raison, la formule

$$\{x > 5\} \ \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od} \ \{x = 52\}$$

est valide. Par contre, la formule

$$\{x > 0\} \ \textbf{while } x > 5 \textbf{ do } x := x + 1 \textbf{ od} \ \{x = 52\}$$

n'est pas valide (pourquoi ?)

Exemple 7 L'exemple suivant montre une autre raison pour laquelle une formule de Hoare avec une boucle peut être vraie :

$$[x \mapsto 20] \models \{\texttt{True}\} \ \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od} \ \{x \leq 0\}$$

On a bien sûr que $[x \mapsto 20] \models \texttt{True}$. L'exécution du programme donne

$$\begin{aligned} & \langle \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od}, x \mapsto 20 \rangle \\ \Rightarrow \quad & \langle x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od}, x \mapsto 20 \rangle \\ \Rightarrow \quad & \langle \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od}, x \mapsto 19 \rangle \\ \vdots \quad & \vdots \\ \Rightarrow \quad & \langle x := x - 1; \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od}, x \mapsto 1 \rangle \\ \Rightarrow \quad & \langle \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od}, x \mapsto 0 \rangle \\ \Rightarrow \quad & \langle \epsilon, x \mapsto 0 \rangle \end{aligned}$$

Puisque $[x \mapsto 0] \models x \geq 0$, la formule de Hoare est satisfaite par $[x \mapsto 20]$. Mais en fait on aurait pu donner un argument beaucoup plus simple que de faire tout ce calcul : On peut, à partir d'une configuration

$$\langle \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od}, \sigma \rangle$$

arriver sur une configuration $\langle \epsilon, \sigma' \rangle$ seulement quand $\sigma' \models \neg x > 0$, car la dernière transition était forcement pour le cas d'une boucle où la condition n'est pas vraie. Ce qui montre qu'on a même que

$$\models \{\texttt{True}\} \ \textbf{while } x > 0 \textbf{ do } x := x - 1 \textbf{ od } \ \{x \leq 0\}$$

Exemple 8 Finalement il y a une troisième raison pour laquelle une formule de Hoare avec une boucle dans le programme peut être valide, et cette raison est liée à la nature d'une boucle. Nous prétendons que

$$\models \{x > 0\} \ \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od } \ \{x > 0\}$$

Cette fois l'argument utilise une induction sur la longueur de la séquence d'exécution. Nous montrons d'abord que (*) si $\sigma \models x > 0$ et

$$\begin{array}{ll} & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma \rangle \\ \Rightarrow & \ldots \\ \Rightarrow & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma' \rangle \end{array}$$

alors on a aussi que $\sigma' \models x > 0$. Nous le montrons par induction sur le nombre n de configurations dans cette séquence où le programme est **while** $z > 0$ **do** $z := z - 1; x := x + 1$ **od**. Remarquez que n est simplement le nombre de fois que le corps de la boucle est exécutée dans cette séquence, plus 1.

Cas : $n = 1$. Dans ce cas on a une séquence d'exécution de longueur 1 qui consiste en une seule configuration :

$$\langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma \rangle$$

et on a donc que $\sigma = \sigma'$, et $\sigma' \models x > 0$.

Cas : n>1 . Dans ce cas on a une séquence d'exécution de la forme suivante :

$$\begin{array}{ll} & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma \rangle \\ \Rightarrow & \ldots \\ \Rightarrow & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma_1 \rangle \\ \Rightarrow & \langle z := z - 1; x := x + 1; \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma_1 \rangle \\ \Rightarrow & \langle x := x + 1; \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma_1[z/\sigma_1(z) - 1 \rangle \\ \Rightarrow & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma_1[z/\sigma_1(z) - 1, x/\sigma_1(x) + 1] \rangle \end{array}$$

Nous avons donc que $\sigma' = \sigma_1[z/\sigma_1(z) - 1, x/\sigma_1(x) + 1]$. Application de l'hypothèse d'induction nous donne que $\sigma_1(x) > 0$. On a que $\sigma'(x) = \sigma_1(x) + 1$, donc

$$\sigma'(x) = \sigma_1(x) + 1 > \sigma_1(x) > 0$$

Finalement supposons une séquence d'exécution

$$\begin{array}{ll} & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma \rangle \\ \Rightarrow & \ldots \\ \Rightarrow & \langle \epsilon, \sigma' \rangle \end{array}$$

Cette séquence doit être de la forme suivante

$$\begin{array}{ll} & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma \rangle \\ \Rightarrow & \ldots \\ \Rightarrow & \langle \textbf{while } z > 0 \textbf{ do } z := z - 1; x := x + 1 \textbf{ od}, \sigma' \rangle \\ \Rightarrow & \langle \epsilon, \sigma' \rangle \end{array}$$

avec $\sigma' \models \neg z > 0$. Nous concluons avec la propriété (*) montrée ci-dessus que $\sigma' \models x > 0$.
Notez que $\sigma' \models \neg z > 0$, le fait que la condition de la boucle est fausse à la fin de la séquence d'exécution, n'est pas du tout utilisé dans cette preuve. Le cœur de l'argument est le fait qu'il y a un *invariant*, ici $x > y$, qui est vrai chaque fois quand on est au début de la boucle. Nous avons déjà vu un argument de ce genre dans l'introduction.

Exemple 9 Notre dernier exemple montre une astuce souvent utilisée. Parfois on souhaite exprimer dans un énoncé de correction partielle que la valeur finale (après exécution du programme) d'une variable est dans une certaine relation avec la valeur initiale de cette variable, c'est-à-dire sa valeur avant l'exécution du programme. Pour avoir dans la post-condition un accès à la valeur initiale d'une variable on peut la « sauvegarder » dans une nouvelle variable qui n'est pas touchée par le programme. Par exemple,

$$\models \{x' = x\}\, x := x + 1\, \{x = x' + 1\}$$

exprime le fait trivial que, après l'exécution de l'instruction d'affectation, la valeur finale de la variable x est 1 plus sa valeur initiale. C'est une conséquence de la définition 27 : Soit σ une affectation quelconque. Si $\sigma(x) = \sigma(x')$, alors on a après exécution du programme que pour l'affectation résultante σ' : $\sigma'(x') = \sigma(x')$ car le programme ne touche pas la variable x', et par conséquent on a que $\sigma'(x') = \sigma(x') = \sigma(x)$.

9.3 Références et remarques

Nous revenons à la question : pourquoi se restreint-on dans la logique de Hoare à des énoncés de correction partielle et pas (au moins pas dans un premier temps) à des énoncés de correction totale ? La raison se voit dans l'exemple 8 de la section précédente : Pour montrer la correction partielle d'un morceaux de programme avec une boucle on a souvent besoin d'un *invariant* pour faire un argument inductif. Pour montrer la terminaison, par contre, on a besoin d'une valeur qui est décrémentée à chaque étape de l'exécution. En d'autres mots il faut trouver le bon *variant* pour la preuve de terminaison. Ceci est une première explication de pourquoi on essaye de séparer les preuves de correction partielle des preuves de terminaison. (Il y une autre raison plus fondamentale qui vient de la théorie de modèles, mais ça dépasse de loin le cadre de ce cours.)

La logique présentée dans ce chapitre est nommé après *Tony Hoare*, chercheur britannique qui a reçu en 1980 le très prestigieux prix *Turing Award* (une sorte de Nobel pour l'informatique) pour ses nombreuses contributions à l'informatique.

Chapitre 10

Un calcul pour les formules de Hoare

10.1 Le calcul de Hoare

Nous avons vu dans la section 9.2 des exemples de preuves de correction partielle « à la main », c'est-à-dire dans le style d'une preuve mathématique habituelle. Les exemples ont montré que c'est assez long, même fastidieux, d'écrire toutes ces preuves à la main. Il est très difficile de rédiger une preuve dans ce style pour un programme de plusieurs centaines de lignes de code (ce qui est toujours un programme *très* petit).

Le calcul de Hoare va nous faciliter la tâche de trouver une preuve de correction partielle, pour deux raisons :

- Dans un premier temps, le calcul nous permet de *rédiger une preuve* plus compacte, en utilisant des *règles de preuves* qui schématisent des raisonnement récurrents dans les preuves de correction partielle. Les règles sont présentées dans cette section, et dans la section 11.1 nous expliquerons comment construire des preuves à partir de ces règles.
- Dans un deuxième temps, le calcul nous permet aussi de mieux diriger la *recherche d'une preuve*, et même d'automatiser en partie la recherche d'une preuve. C'est le sujet de la section 11.2.

Définition 28 *Une* règle d'inférence *est écrite dans la forme*

$$R\ \frac{J_1 \quad \dots \quad J_n}{J}$$

où J et les J_i sont des schémas de formules de Hoare, et R est le nom *de la règle. On appelle les J_i les* hypothèses*, et J la* conclusion *de la règle. Il est permis que $n = 0$, dans ce cas on parle d'un* axiome*. Parfois la règle est accompagnée d'une condition de côté qui met des conditions supplémentaires sur ce que $J_1, \dots, J_n, J$ peuvent dénoter.*

Le calcul de Hoare *consiste en les règles d'inférences donne sur la figure 10.1.*

Le nom de la règle est parfois omis quand il n'a pas d'importance.

Ces règles d'inférences sont censées décrire des règles de raisonnement dans la logique de Hoare. C'est-à-dire, nous attendons d'une telle règle qu'elle soit correcte dans le sens suivant :

Définition tentative : *Une règle d'inférence* $\frac{J_1 \quad \dots \quad J_n}{J}$ *est* correcte *quand on a*

1. L'axiome d'affectation

$$\mathsf{A}\ \frac{}{\{p[x/t]\}\, x := t\, \{p\}} \qquad \text{où } p \in \mathit{BExpr},\ t \in \mathit{AExpr},\ x \in X$$

2. La règle de composition

$$\mathsf{C}\ \frac{\{p\}\, S_1\, \{q\} \qquad \{q\}\, S_2\, \{r\}}{\{p\}\, S_1; S_2\, \{r\}} \qquad \text{où } p, q, r \in \mathit{BExpr},\ S_1, S_2 \in \mathit{Imp}$$

3. La règle de la conditionnelle

$$\mathsf{I}\ \frac{\{p \wedge f\}\, S_1\, \{q\} \qquad \{p \wedge \neg f\}\, S_2\, \{q\}}{\{p\}\ \textbf{if}\ f\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}} \qquad \text{où } p, q, f \in \mathit{BExpr},\ S_1, S_2 \in \mathit{Imp}$$

4. La règle de la boucle

$$\mathsf{W}\ \frac{\{p \wedge f\}\, S\, \{p\}}{\{p\}\ \textbf{while}\ f\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg f\}} \qquad \text{où } p, f \in \mathit{BExpr} \text{ et } S \in \mathit{Imp}$$

5. La règle de la conséquence

$$\mathsf{P}\ \frac{\{p\}\, S\, \{q\}}{\{p'\}\, S\, \{q'\}} \qquad \text{où } p, q, p', q' \in \mathit{BExpr},\ S \in \mathit{Imp},\ p' \models p,\ q \models q'$$

FIGURE 10.1: Les règles d'inférence du calcul de Hoare.

– *Si toutes ses hypothèses sont valides,* $\models J_1, \ldots, \models J_n$,
– *alors sa conclusion est aussi valide,* $\models J$.

Cette définition n'est pas toute à fait exacte. La raison est que nous utilisons dans l'écriture des règles d'inférence des variables pour dénoter n'importe quel programme, expression booléenne, etc. Ces variables sont parfois appelées des *méta-variables* pour les distinguer des variables qui paraissent dans les programmes et dans les expressions booléennes. En fait, nous avons déjà rencontré des méta-variables dans les règles de réécriture du chapitre 4. Il faut donc parler de *toutes les instances* des règles d'inférences. Par exemple, il y a une infinité d'instances de la règle de composition. Deux exemples d'instances sont :

$$\mathsf{C}\ \frac{\{x = 1\}\, x := x + 1\, \{x = 2\} \qquad \{x = 2\}\, x := x * x\, \{x = 4\}}{\{x = 1\}\, x := x + 1; x := x * x\, \{x = 4\}}$$

$$\mathsf{C}\ \frac{\{x = 1\}\, x := x + 1; x := x + 2\, \{x = 17\} \qquad \{x = 17\}\, x := x - 1\, \{x = 16\}}{\{x = 1\}\, x := x + 1; x := x + 2; x := x - 1\, \{x = 16\}}$$

Si on dit « la règle de composition est correcte » on veut en fait dire que toutes ses instances (comme par exemple les deux données ci-dessus, mais aussi toutes les autres) sont correctes. La bonne définition est donc :

Définition 29 *Une règle d'inférence* $\dfrac{J_1 \quad \dots \quad J_n}{J}$ *est* correcte *quand on a pour toutes ses instances* $\mathsf{R}\ \dfrac{p_1 \quad \dots \quad p_n}{p}$ *:*
– *Si toutes ses hypothèses sont valides,* $\models p_1, \dots, \models p_n$,
– *alors sa conclusion est aussi valide,* $\models p$.

Sur l'exemple donné on s'aperçoit qu'au moins les deux instances données ci-dessus sont correctes : Pour la première c'est le cas parce que la conclusion est valide, pour la deuxième instance c'est le cas parce qu'il y a une hypothèse (la première) qui n'est pas valide.

Dans le cas particulier d'une règle d'inférence qui est un axiome, la définition 29 dit que

Un axiome $\dfrac{}{J}$ *est* correct *quand on a pour toutes ses instances* $\dfrac{}{p}$ *: p est valide, c'est-à-dire que* $\models p$.

Regardons maintenant les règles une par une.

L'axiome de l'affectation À première vue cet axiome peut étonner car on pourrait attendre (à tort) une substitution dans la post-condition au lieu d'une substitution dans la pre-condition. Intuitivement le raisonnement est comme suit : On veut que, après exécution de l'affectation, la post-condition q soit vraie pour la nouvelle valeur de x. Or, la valeur de x après l'affectation est la valeur de l'expression t avant l'affectation. Donc, toute propriété de x après l'affectation correspond à la même propriété de t avant l'affectation. Plus formellement :

Proposition 17 *La règle d'affectation est correcte.*

Démonstration: Soit $\sigma \models p[x/t]$ et $[\![x := t]\!]\sigma = \sigma'$. Il faut montrer que $\sigma' \models p$.
Selon la définition de la sémantique des programmes, $\sigma' = \sigma[x/[\![t]\!]\sigma]$. Il faut donc montrer que

$$\sigma[x/[\![t]\!]\sigma] \models p$$

Ceci est équivalent, à cause de la proposition 12, à

$$\sigma \models p[x/t]$$

ce qui est exactement notre première hypothèse. □

On a vu en TD qu'un axiome naïf $\dfrac{}{\{\mathtt{True}\}\, x := t\, \{x = t\}}$ est vrai seulement si $x \notin \mathcal{V}(t)$. Que dit notre axiome d'affectation si on part de la post-condition $x = t$? On obtient

$$\mathsf{A}\ \frac{}{\{(x = t)[x/t]\}\, x := t\, \{x = t\}}$$

Maintenant *si* $x \notin \mathcal{V}(t)$ alors on a que $(x = t)[x/t]$ donne la formule $t = t$ qui est trivialement vraie. Par contre, si la variable x parait en t alors $t[x/t]$ donne quelque chose différent de t, et la formule n'est en général pas vraie.

La règle de composition dit simplement que pour démontrer une assertion de correction partielle $\{p\}\, S\, \{r\}$, où S est une séquence d'instructions, il suffit de couper la séquence S en deux morceaux S_1 et S_2 et de montrer une assertion de correction partielle pour S_1 et une autre pour S_2. Dans ces deux assertions on utilise un « lemme » q qui exprime une propriété qui doit être vraie après avoir exécuté S_1, et que nous pouvons donc utiliser comme pre-condition dans l'assertion de correction partielle de S_2.

Proposition 18 *La règle de composition est correcte.*

Démonstration: Soient $\{p\}\, S_1\, \{q\}$ et $\{q\}\, S_2\, \{r\}$ valides. Il faut montrer que $\{p\}\, S_1; S_2\, \{r\}$ est valide.

Soit σ une affectation avec $\sigma \models p$ et $[\![S_1; S_2]\!]\sigma = \sigma'$. Il y a donc σ'' avec $[\![S_1]\!]\sigma = \sigma''$ et $[\![S_2]\!]\sigma'' = \sigma'$. Puisque $\{p\}\, S_1\, \{q\}$ est valide on a que $\sigma'' \models q$, et puisque $\{q\}\, S_2\, \{r\}$ est valide on a donc aussi que $\sigma' \models q$. □

La règle de la conditionnelle permet de démontrer la correction d'une instruction conditionnelle on considérant la branche **then** (où on a le droit d'utiliser le fait que la condition est vraie) et la branche **else** (où on a le droit d'utiliser le fait que la condition est fausse).

Exercice 18 *Montrer que la règle de la conditionnelle est correcte.*

La règle de la conséquence permet de faire deux choses à la fois :
- renforcer la pre-condition : Si on sait qu'un programme est correct sous une certaine pre-condition p, alors il est certainement aussi correct si on restreint encore plus la pre-condition.
- affaiblir la post-condition : Si on sait que tous les états qui peuvent résulter de l'exécution du programme satisfont une certaine propriété q alors il satisfont aussi toute propriété qui est plus faible.

Formellement :

Proposition 19 *La règle de conséquence est correcte.*

Démonstration: Soient $p' \to p$, $\{p\}\, S\, \{q\}$, et $q \to q'$ valides. Pour montrer que $\{p'\}\, S\, \{q'\}$ est valide soit σ une affectation avec $\sigma \models p'$, et soit $[\![S]\!]\sigma = \sigma'$.

Puisque $p' \to p$ est valide et $\sigma \models p'$ nous avons que $\sigma \models p$.

Puisque $\{p\}\, S\, \{q\}$ est valide, $\sigma \models p$, et $[\![S]\!]\sigma = \sigma'$, nous avons aussi que $\sigma' \models q$.

Puisque $q \to q'$ est valide et $\sigma' \models q$ nous avons aussi que $\sigma' \models q'$. □

La règle de la boucle est la règle la plus intéressante. L'expression p dans cette règle est l'invariant. La règle dit que si l'exécution du corps S de la boucle conserve la véracité de l'invariant p (où en plus on a le droit d'utiliser le fait que la garde f de la boucle est vraie quand on commence l'exécution du corps), alors on a aussi que la véracité de p est conservée par l'exécution de la boucle entière. De plus on sait qu'après l'exécution de la boucle la garde de la boucle est fausse.

Par exemple, une instance de cette règle est

$$\mathsf{W}\ \frac{\{x = y \wedge x > 42\}\, x := x - 1; y := y - 1\, \{x = y\}}{\{x = y\}\ \textbf{while}\ x > 42\ \textbf{do}\ x := x - 1; y := y - 1\ \textbf{od}\ \{x = y \wedge x \leq 42\}}$$

Afin de démontrer la correction de la règle de la boucle nous avons besoin d'une petite proposition :

Proposition 20 *Si* $\langle \textbf{while } f \textbf{ do } S \textbf{ od}, \sigma\rangle \Rightarrow \ldots \Rightarrow \langle \epsilon, \sigma'\rangle$ *en* n *itérations de la boucle, alors*

1. *Soit on a* $n = 0$, $\sigma = \sigma'$, *et* $\sigma' \models \neg f$
2. *Soit* $n \geq 1$, *et il existe* $\sigma_1, \ldots, \sigma_{n+1} \in \mathit{Aff}$ *telles que*
   (a) $\sigma = \sigma_1$
   (b) $\sigma' = \sigma_{n+1}$
   (c) $[\![S]\!]\sigma_i = \sigma_{i+1}$ *pour tout* i, $1 \leq i \leq n$
   (d) $\sigma_i \models f$ *pour tout* i, $1 \leq i \leq n$
   (e) $\sigma_{n+1} \models \neg f$

Dans cette proposition, σ_i est l'état de la mémoire au début de la i-ème itération de la boucle, et σ_{i+1} est l'état de la mémoire à la fin de la i-ème itération de la boucle.

Exercice 19 *Montrer la proposition 20 par induction sur le nombre d'itération dans l'exécution de la boucle.*

Proposition 21 *La règle de la boucle est correcte.*

Démonstration: Soit $\models \{p \wedge f\}\, S\, \{p\}$, et $\sigma \in \mathit{Aff}$ telle que $\sigma \models p$ et

$$[\![\textbf{while } f \textbf{ do } S \textbf{ od}]\!]\sigma = \sigma'$$

Soit n le nombre d'itérations de la boucle pendant l'exécution du programme S par rapport à l'état initial σ.

- Si $n = 0$ on a, d'après la proposition 20, que $\sigma' = \sigma$ et $\sigma' \models \neg f$. Puisque $\sigma \models p$ on obtient que $\sigma' \models p \wedge \neg f$.
- Si $n \geq 1$ alors il y a une séquence $\sigma = \sigma_1, \ldots, \sigma_{n+1} = \sigma'$ comme décrite à la proposition 20. Par la même proposition on a que $\sigma' \models \neg f$. Puisque $\sigma_1 \models p \wedge f$, et puisque $\models \{p \wedge f\}\, S\, \{p\}$, on obtient par induction que $\sigma_{n+1} \models p$. Donc, $\sigma' \models p \wedge f$. □

Exercice 20 *Dire pour chacune des règles suivantes si elle est correcte ou pas.*

1.
$$\frac{\{p_1\}\, S\, \{q_1\} \qquad \{p_2\}\, S\, \{q_2\}}{\{p_1 \wedge p_2\}\, S\, \{q_1 \wedge q_2\}}$$

2.
$$\frac{\{p\}\, S\, \{q\}}{\{\mathit{True}\}\ \textbf{if } p \textbf{ then } S \textbf{ else } x := 0 \textbf{ fi }\ \{q\}}$$

10.2 Références et remarques

Le calcul donné à la figure 10.1 à été proposé par Tony Hoare en 1969. Notre présentation est simplifiée dans le mesure où nous utilisons des expressions booléennes pour les pre-conditions et les post-conditions. Souvent on utilise pour les pre-conditions et les post-conditions les formules d'une logique plus expressive, et dont nos expressions booléennes sont un cas particulier : la *logique du premier ordre*.

Nous avons ici présenté la notion des règles d'inférences et des arbres de preuves dans le contexte de la logique de Hoare, mais en vérité il s'agit des notions très générales qui peuvent s'appliquer à n'importe quelle logique. Par exemple on pourrait aussi donner des règles d'inférences pour la logique propositionnelle, ce que nous n'avons pas fait car il existe des algorithmes de décision très efficace pour le cas restreint de la logique propositionnelle, comme par exemple l'algorithme DPLL présentée au chapitre 5. Par contre les règles d'inférence deviennent un outil incontournable dès que l'on s'intéresse à des logiques plus expressives que la logique propositionnelle, comme par exemple la logique du premier ordre. Dans ce cas il convient d'étudier d'abord des règles d'inférence pour la logique propositionnelle, et puis de les étendre vers les logiques plus expressives (voir un cours de *Logique*).

Chapitre 11

Preuves de correction partielle

11.1 Les preuves dans le calcul de Hoare

Le calcul de Hoare que nous avons étudié au chapitre précédent consiste en des règles de raisonnement qui nous permettent de conclure la validité d'une formule, dénotée par la conclusion d'une règle d'inférence, en supposant que les formules dénotées par les hypothèses de la règle d'inférence soient valides. Dans cette section nous nous intéressons à des preuves entières de correction partielle de programmes en utilisant le calcul du chapitre 10. Une preuve est obtenue en itérant, à partir des axiomes du calcul, les règles d'inférence. Puisque les règles ont en général plusieurs hypothèses, la structure ainsi obtenue est un *arbre de preuve*, ou aussi simplement appelé une *preuve*.

Intuitivement, une preuve est un arbre dont tous les nœuds sont étiquetés par des formules de Hoare ou des expressions booléennes, et pour tous les nœuds :
– si le nœud est étiqueté par p et si $p_1, \ldots, p_n$ sont les étiquettes des prédécesseurs de ce nœud,
– alors $\dfrac{p_1 \quad \cdots \quad p_n}{p}$ est une instance d'une des règles d'inférence du calcul de Hoare.

Dans le cas particulier où un nœud étiqueté avec une formule p est une feuille, et n'a alors pas de prédécesseurs, cette condition dit que $\dfrac{-}{p}$ doit être une règle d'inférence, c'est-à-dire un axiome (car il n'y a pas d'hypothèses).

Les arbres de preuve sont dessinés dans un style particulier : les nœuds sont connectés avec leurs prédécesseurs par une barre horizontale, comme dans les règles d'inférences. On appelle *racine* le nœud d'un arbre qui n'a pas de successeur. Par exemple

$$\dfrac{\dfrac{\dfrac{}{p_1} \quad \dfrac{}{p_2} \quad \dfrac{\dfrac{}{p_3}}{p_4}}{p_5} \qquad \dfrac{}{p_6}}{p_7}$$

est un arbre dont la racine est étiquetée par p_7, et pour qu'il soit une preuve il faut que $\dfrac{-}{p_1}$,

$\frac{}{p_2}$, $\frac{}{p_3}$ et $\frac{}{p_6}$ soient des instances d'axiomes, et que $\frac{p_3}{p_4}$, $\frac{p_1 \qquad p_2 \qquad p_4}{p_5}$ et $\frac{p_5 \qquad p_6}{p_7}$ soient des instances de règles d'inférence.

La définition formelle des preuves utilise le principe de la définition inductive :

Définition 30 *L'ensemble des* preuves dans le calcul de Hoare *est le plus petit ensemble tel que :*

- *Si* $\mathsf{R}\ \frac{}{p}$ *est instance d'un axiome alors* $\mathsf{R}\ \frac{}{p}$ *est une preuve (de la formule* p*).*
- *Si* $\Pi_1, \ldots, \Pi_n$ *sont des preuves des formules* $p_1, \ldots, p_n$ *respectivement, et si* $\mathsf{R}\ \frac{p_1 \qquad \ldots \qquad p_n}{p}$ *est instance d'une règle d'inférence, alors* $\mathsf{R}\ \frac{\Pi_1 \qquad \ldots \qquad \Pi_n}{p}$ *est une preuve (de la formule* p*).*

Théorème 18 *S'il y a une preuve de* p *alors* p *est valide.*

Démonstration: Preuve par récurrence :

Cas de base Nous avons une preuve qui consiste en une seule instance d'un axiome p. Il s'agit nécessairement de l'axiome de l'affectation (puisque c'est le seul axiome de notre système de règles).Nous concluons avec la proposition 17 que p est valide.

Cas de récurrence Nous avons une preuve composée $\mathsf{R}\ \frac{\Pi_1 \qquad \ldots \qquad \Pi_n}{p}$ où chaque Π_i est une preuve de p_i, et $\mathsf{R}\ \frac{p_1 \qquad \ldots \qquad p_n}{p}$ est une instance de la règle d'inférence R. Par hypothèse de récurrence, les p_i sont valides, donc p est aussi valide car toutes les règles d'inférence du calcul de Hoare sont correctes (propositions 18 à 21). □

Des exemples de preuves sont donnés en figure 11.1 et en figure 11.2. Dans ces preuves nous permettons des simplifications évidentes dans les expressions booléennes, comme par exemple de remplacer une formule $\texttt{True} \wedge p$ par p, etc. Remarquez qu'il existe en général plusieurs preuves différentes pour la même formule (quand elle est valide). Un premier exemple pour cela est donné page 99 avec une programme consistant en deux affectations.

11.2 Vers une automatisation des preuves ?

Est-ce qu'il y a une méthode pour trouver la preuve pour une formule de Hoare donnée dans le cas où la formule est valide, et de détecter quand la formule de Hoare donnée n'est pas valide ? Un premier problème est déjà qu'il n'y a pas d'algorithme qui peut décider pour une expression booléenne quelconque si elle est valide ou pas, comme nous l'avons expliquée à la section 7.3. Est-ce qu'une telle méthode existe au moins pour les cas « simples » dans lesquelles ont n'est pas affronté à des expressions booléennes trop difficiles ? Un obstacle est certainement le fait qu'il est possible que plusieurs preuves existent pour la même formule de Hoare, comme nous l'avons vu à la section précédente.

Exemple d'une preuve avec deux affectations :

$$\mathsf{C}\ \dfrac{\mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{x+1\geq 1\}\,x:=x+1\,\{x\geq 1\}}}{\{x\geq 0\}\,x:=x+1\,\{x*x\geq 1\}} \qquad \mathsf{A}\ \dfrac{}{\{x*x\geq 1\}\,x:=x*x\,\{x\geq 1\}}}{\{x\geq 0\}\,x:=x+1;x:=x*x\,\{x\geq 1\}}$$

Une preuve différente pour la même formule :

$$\mathsf{C}\ \dfrac{\mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{x+1\geq 1\}\,x:=x+1\,\{x\geq 1\}}}{\{x\geq 0\}\,x:=x+1\,\{x\geq 1\}} \qquad \mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{x*x\geq 1\}\,x:=x*x\,\{x\geq 1\}}}{\{x\geq 1\}\,x:=x*x\,\{x\geq 1\}}}{\{x\geq 0\}\,x:=x+1;x:=x*x\,\{x\geq 1\}}$$

Exemple d'une preuve avec une conditionnelle :

$$\mathsf{I}\ \dfrac{\mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{x\geq y\}\,z:=x\,\{z\geq y\}}}{\{x>y\}\,z:=x\,\{z\geq y\}} \qquad \mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{y\geq y\}\,z:=y\,\{z\geq y\}}}{\{x\leq y\}\,z:=y\,\{z\geq y\}}}{\{\mathtt{True}\}\ \textbf{if}\ x>y\ \textbf{then}\ z:=x\ \textbf{else}\ z:=y\ \textbf{fi}\ \{z\geq y\}}$$

FIGURE 11.1: Preuves de correction partielle.

Une bonne approche est de commencer par construire la preuve à partir de la post-condition, et puis de se frayer un chemin de la fin du programme au début. La règle de composition nous donne le moyen d'avancer sur ce chemin : Si on veut montrer la validité d'une formule de Hoare comme $\{p\}\,S;I\,\{q\}$ où I est la dernière instruction du programme entier, alors il suffit de montrer la validité de $\{p\}\,S\,\{r\}$ et de $\{r\}\,I\,\{q\}$ pour un « lemme » r bien choisi. La question évidente est : comment choisir ce lemme r ?

Si on a en général plusieurs possibilités de choisir le lemme r pour qu'on puisse démontrer la validité de $\{r\}\,I\,\{q\}$, un mauvais choix de r peut avoir la conséquence qu'on ne réussira pas dans la suite de démontrer la validité de $\{p\}\,S\,\{r\}$. La meilleure stratégie est de choisir un r aussi faible que possible (possible veut dire : tel qu'on puisse montrer que $\{r\}\,I\,\{q\}$ est valide). Plus la formule r est faible, plus on a de chances de démontrer $\{p\}\,S\,\{r\}$. Cette idée nous mène à la définition 31 au-dessous. Nous allons montrer dans la proposition 23 que cette définition nous donne effectivement la formule qui est la plus faible parmis toutes les preconditions.

Définition 31 *Soient S un programme et q une expression booléenne. Nous disons qu'une expression booléenne p est une* plus faible pre-condition *de S par rapport à q si pour toute affectation $\sigma \in Aff$:*

$$\sigma \models p \qquad ssi \qquad \Big([\![S]\!]\sigma = \bot \quad ou \quad [\![S]\!]\sigma \models q\Big)$$

Dans ce cas nous écrivons $wp(S,q) = p$.

Attention, nous ne prétendons pas que pour tout programme S et expression booléenne q une telle plus faible pre-condition de S par rapport à q existe. En fait nous verrons dans la suite

L'exemple 6 de la section 9.2 (une boucle qui ne termine pas)

$$\mathsf{P}\ \dfrac{\mathsf{W}\ \dfrac{\mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{x+1>5\}\,x:=x+1\,\{x>5\}}}{\{x>5\wedge x>5\}\,x:=x+1\,\{x>5\}}}{\{x>5\}\ \textbf{while}\ x>5\ \textbf{do}\ x:=x+1\ \textbf{od}\ \{x>5\wedge\neg x>5\}}}{\{x>5\}\ \textbf{while}\ x>5\ \textbf{do}\ x:=x+1\ \textbf{od}\ \{x=52\}}$$

L'exemple 7 de la section 9.2 (la garde d'une boucle n'est pas vraie directement après la boucle)

$$\mathsf{P}\ \dfrac{\mathsf{W}\ \dfrac{\mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{\mathtt{True}\}\,x:=x-1\,\{\mathtt{True}\}}}{\{\mathtt{True}\wedge x>0\}\,x:=x-1\,\{\mathtt{True}\}}}{\{\mathtt{True}\}\ \textbf{while}\ x>0\ \textbf{do}\ x:=x-1\ \textbf{od}\ \{\mathtt{True}\wedge\neg x>0\}}}{\{\mathtt{True}\}\ \textbf{while}\ x>0\ \textbf{do}\ x:=x-1\ \textbf{od}\ \{x\leq 0\}}$$

Exemple d'une preuve avec une boucle utilisant un raisonnement avec un invariant :

$$\mathsf{P}\ \dfrac{\mathsf{W}\ \dfrac{\mathsf{C}\ \dfrac{\mathsf{P}\ \dfrac{\mathsf{A}\ \dfrac{}{\{x-1=y-1\}\,x:=x-1\,\{x=y-1\}}}{\{x=y\wedge x>5\}\,x:=x-1\,\{x=y-1\}}\qquad \mathsf{A}\ \dfrac{}{\{x=y-1\}\,y:=y-1\,\{x=y\}}}{\{x=y\wedge x>5\}\,x:=x-1;y:=y-1\,\{x=y\}}}{\{x=y\}\ \textbf{while}\ x>5\ \textbf{do}\ x:=x-1;y:=y-1\,\{x=y\wedge\neg x>5\}}}{\{x=42\wedge x=y\}\ \textbf{while}\ x>5\ \textbf{do}\ x:=x-1;y:=y-1\ \textbf{od}\ \{y\leq 5\}}$$

Figure 11.2: Exemples de preuves pour des programmes avec boucle.

qu'il y a des cas dans lesquels une plus faible pre-condition n'existe pas. Si une plus faible pre-condition existe, est-elle unique ? Évidemment elle ne peut pas être unique dans le sens stricte car elle est définie seulement par sa sémantique (c'est-à-dire, par caractérisation des affectations pour lesquelles elle est vraie). Donc, si p_1 est une plus faible pre-condition de S par rapport à q, et si p_2 est logiquement équivalent à p_1, alors p_2 est aussi une plus faible pre-condition de S par rapport à q. La proposition suivante en fait énonce que la notion de pre-condition la plus faible est *unique à équivalence logique près* :

Proposition 22 *Soit p_1 et p_2 sont des plus faibles pre-conditions de S par rapport à q alors $p_1 \Subset p_2$.*

Démonstration: Soit $\sigma \in \mathit{Aff}$ une affectation. Alors, nous avons selon la définition 31 que

$\sigma \models p_1$
ssi $[\![S]\!]\sigma = \bot$ ou $[\![S]\!]\sigma \models q$
ssi $\sigma \models p_2$

Donc, $p_1 \Subset p_2$. □

Pour cette raison, nous parlons aussi souvent de *la* pre-condition la plus faible.

La proposition suivante donne quelques propriétés importantes de cette notion, et justifie son nom.

Proposition 23 *Soit p la plus faible pre-condition de S par rapport à q. Alors*

1. $\models \{p\}\, S\, \{q\}$
2. *Si* $\models \{p'\}\, S\, \{q\}$ *alors* $p' \models p$.

Démonstration:

1. Soit $\sigma \in \mathit{Aff}$. Si $\sigma \not\models p$ alors $\sigma \models \{p\}\, S\, \{q\}$, et si $\sigma \models p$ alors nous avons après la définition 31 que $[\![S]\!]\sigma = \bot$ ou $[\![S]\!]\sigma \models q$, donc $\sigma \models \{p\}\, S\, \{q\}$.
2. Soit $\sigma \in \mathit{Aff}$ avec $\sigma \models p'$, nous devrons montrer que $\sigma \models p$. Puisque $\sigma \models \{p'\}\, S\, \{q\}$ et $\sigma' \models p$, nous avons que $[\![S]\!]\sigma = \bot$ ou $[\![S]\!]\sigma \models q$, et donc après définition 31 que $\sigma \models p$. □

Une fois la notion de la plus faible pre-condition établie, il nous faut maintenant étudier comment on peut effectivement la calculer pour un programme donné et une post-condition donnée. La façon utilisée pour construire cette pre-condition, si elle existe, va certainement dépendre de la structure du programme. Puisque l'ensemble des programmes est défini inductivement, il faudra répondre aux questions suivantes :

- Comment calculer une plus faible pre-condition pour une seule affectation ?
- Comment calculer une plus faible pre-condition pour une séquence d'instructions, sous l'hypothèse qu'on sait calculer des plus faibles pre-conditions pour chaque instructions dans la liste ?
- Comment calculer une plus faible pre-condition pour une instruction conditionnelle, sous l'hypothèse qu'on sait calculer de plus faibles pre-conditions pour les deux branches ?
- Comment calculer une plus faible pre-condition pour une boucle, sous l'hypothèse qu'on sait calculer une plus faible pre-condition pour son corps ?

Ici, « calculer une plus faible pre-condition pour un programme S » veut dire avoir une méthode pour calculer $wp(S, q)$ pour *n'importe quelle* post-condition q. Commençons avec une affectation :

Proposition 24 $wp(x := t, q) = q[x/t]$.

Démonstration: Soit $\sigma \in \mathit{Aff}$. Nous avons la chaîne d'équivalence suivante :

$[\![x := t]\!]\sigma = \bot$ ou $[\![x := t]\!]\sigma \models q$
ssi $[\![x := t]\!]\sigma \models q$ (car le programme termine toujours)
ssi $\sigma[x/[\![t]\!]\sigma] \models q$ (selon la définition de la sémantique des programmes)
ssi $\sigma \models q[x/t]$ (par proposition 12)

□

Puis, pour les séquences d'instructions, il suffit évidemment d'étudier la question pour une séquence consistant en deux instructions puisqu'on peut répéter l'opération tant que nécessaire :

Proposition 25 *Si* $wp(S_2, q) = p$ *et* $wp(S_1, p) = r$ *alors* $wp(S_1; S_2, q) = r$

Démonstration: Soit $\sigma \in \mathit{Aff}$. Nous avons la chaîne d'équivalence suivante :

$\sigma \models r$
ssi $[\![S_1]\!]\sigma = \bot$ ou $[\![S_1]\!]\sigma \models p$ (car $wp(S_1, p) = r$)
ssi $[\![S_1]\!]\sigma = \bot$ ou $[\![S_2]\!]([\![S_1]\!]\sigma) = \bot$ ou $[\![S_2]\!]([\![S_1]\!]\sigma) \models q$ (car $wp(S_2, q) = p$)
ssi $[\![S_1; S_2]\!] = \bot$ ou $[\![S_1; S_2]\!]\sigma \models q$ (par proposition 16)

□

Regardons maintenant la question pour une instruction conditionnelle :

Proposition 26 *Si* $wp(S_1, q) = p_1$ *et* $wp(S_2, q) = p_2$ *alors*

$$wp(\textbf{if } f \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, q) = ((f \wedge p_1) \vee (\neg f \wedge p_2))$$

Exercice 21 *Montrer la proposition 26.*

Une question reste : comment construire la plus faible pre-condition dans le cas d'une boucle ? C'est ici que nous tombons sur un problème. Voici un exemple d'une boucle où une plus faible pre-condition n'existe pas. Soit S le programme

$$\textbf{while } x > 0 \textbf{ do } x := x - y \textbf{ od}$$

et q la formule $x = 0$. Quelle pourrait être la plus faible pre-condition de S par rapport à q ? Il faudrait quelle exprime d'abord le fait que $x \geq 0$ (car, si $x < 0$), le programme termine et la post-condition n'est pas satisfaite), et en plus il faudrait que si $y \geq 0$ alors que x soit un multiple de y. Mais comment exprimer le fait que x soit un multiple de y ? Nous ne pouvons pas l'exprimer dans notre logique car pour cela il nous faudrait dire « il *existe* une valeur z telle que $x = y * z$ », sans connaître cette valeur ! Cela est seulement possible dans une extension de notre logique qui dépasse le cadre de ce cours.

En fait il se trouve que la question de la construction d'une plus faible pre-condition dans le cas d'une boucle est liée à la question de trouver le bon invariant pour une boucle (voir l'exercice 22. Malheureusement il n'y a pas de bonne méthode pour trouver un bon invariant. Il est *en principe* possible de construire un tel invariant à partir de la post-condition de la boucle mais cela nécessite une logique plus expressive (la logique du premier ordre que nous

avons déjà plusieurs fois évoquée). La justification de la construction repose sur un théorème de l'arithmétique modulaire, et l'invariant obtenu n'est pas un invariant « naturel ».

La seule méthode praticable est — de connaître le bon invariant. La meilleure personne à connaître l'invariant est le programmeur qui a écrit le programme. Il est donc crucial que le programmeur donne les invariants des boucles avec le programme. Ça peut être simplement en forme d'un commentaire dans le programme, mais certains langages de programmation permettent d'écrire un invariant directement dans le programme, comme par exemple avec l'instruction `assert` de Objective CAML. L'utilisation de cette instruction n'est pas restreinte à des invariants de boucles, le programmeur peut mettre une telle *assertion* à n'importe quel point de contrôle dans le programme. Les assertions servent d'une part à la documentation du programme (et en particulier à la vérification de sa correction partielle, comme nous avons vu), mais elles sont aussi utiles pour mieux tester le programme. Dans le cas du langage Objective CAML, le compilateur permet de générer du code qui *évalue* les assertions chaque fois qu'une instruction `assert` est rencontrée, et qui signale quand une assertion n'est pas satisfaite. Il est aussi possible de générer du code qui ignore les assertion, et dans ce cas on a aucune perte de performance dans le code engendré. Une instruction similaire est disponible dans le langage `C++` où elle est définie par le pre-processeur (au lieu d'être définie dans le noyeau du langage, ou dans une bibliothèque).

Exercice 22 *Soit* $S =$**while** e **do** S_1 **od**, *et supposons que* $r = wp(S, q)$. *Montrer que*

1. $\models \{r \wedge e\}\, S_1\, \{r\}$
2. $r \wedge \neg e \models q$

Indication : *Pour la première question, utiliser le fait que le programme* S *est équivalent au programme* **if** e **then** $S_1; S$ **fi**.

Un exemple de la construction d'une preuve par la méthode des plus faibles pre-conditions est donné page 104.

Malheureusement la page n'est pas suffisamment large pour dessiner l'arbre en un seul morceaux :

$$\mathsf{P}\ \frac{\mathsf{A}\ \dfrac{}{\{(x+y)*(x+y)-(x+y+x+y)=y*y-1)\}\,x:=x+y\,\{x*x-(x+x)=y*y-1\}}}{\{x=1\}\,x:=x+y\,\{x*x-(x+x)=y*y-1\}}$$

$$\mathsf{C}\ \frac{\{x=1\}\,x:=x+y\,\{x*x-(x+x)=y*y-1\}\qquad \mathsf{A}\ \dfrac{}{\{x*x-(x+x)=y*y-1\}\,z:=x+x\,\{x*x-z=y*y-1\}}}{\{x=1\}\,x:=x+y;z:=x+x\,\{x*x-z=y*y-1\}}$$

$$\mathsf{C}\ \frac{\mathsf{C}\ \dfrac{\{x=1\}\,x:=x+y;z:=x+x\,\{x*x-z=y*y-1\}\qquad \mathsf{A}\ \dfrac{}{\{x*x-z=y*y-1\}\,x:=x*x\,\{x-z=y*y-1\}}}{\{x=1\}\,x:=x+y;z:=x+x;x:=x*x\,\{x-z=y*y-1\}}\qquad \mathsf{A}\ \dfrac{}{\{x-z=y*y-1\}\,x:=x-z\,\{x=y*y-1\}}}{\{x=1\}\,x:=x+y;z:=x+x;x:=x*x;x:=x-z\,\{x=y*y-1\}}$$

La règle de calcul pour la plus faible pre-condition est la raison pourquoi nous essayons toujours de construire la preuve dans le sens qui commence avec la fin du programme, c'est-à-dire avec la post-condition. Une approche duale qui commence au début du programme avec la pre-condition, et qui avance par construction successive d'une *plus forte post-condition* est également possible, le problème est simplement que la règle de construction d'une plus forte post-condition dans le cas d'une affectation est plus compliquée.

11.3 Références et remarques

On trouve parfois dans la littérature des définitions de la plus faible pre-condition qui sont différentes à notre définition 31. Selon la terminologie parfois utilisée dans la littérature nous aurions du parler d'une *plus faible pre-condition libérale*.

L'extension de notre logique par un opérateur « il existe une valeur telle que ... » amène à la *logique du premier ordre* qui va être étudiée dans le cours de logique du L3. La logique du premier ordre permet par exemple d'exprimer que x est un multiple de y par $\exists z(x = y * z)$, ou encore que x est un nombre premier. Par ailleurs, cette extension est loin d'être anodine.

La preuve qu'il est théoriquement possible de construire automatiquement le bon invariant d'une boucle si on dispose d'une logique suffisamment expressive, par exemple la logique du premier ordre, a été donnée par Stephen A. Cook en 1978 [**?**]. On trouve un bon exposé de la preuve par exemple dans [**?**].

Chapitre 12

Problèmes non décidables

Nous avons déjà à la section 7.3 parlé d'un problème pour lequel il existe aucun algorithme qui résout ce problème : il s'agit de la validité des expressions booléennes. Dans ce chapitre nous allons expliquer comment on peut conclure qu'un certain problème ne peut par principe pas être résolu par un algorithme. Bien entendu nous parlons ici des problèmes mathématiques ou informatiques avec une spécification très précise. Comme préparation nous regardons d'abord un paradoxe logique qui nous servira plus tard pour la preuve.

12.1 Le paradoxe du barbier

Est-ce qu'il existe un endroit où le barbier du village rase tous les habitants du village qui ne se rasent pas eux-mêmes et seulement ceux-ci ? Il est sous-entendu que le barbier du village habite au village même.

Nous pouvons montrer avec un argument par absurde qu'un tel endroit ne peut pas exister. Supposons par l'absurde qu'il existe un village dans lequel le barbier rase tous les habitants du village qui ne se rasent pas eux-mêmes et seulement ceux-ci. Il y a deux cas possibles :

1. Le barbier ne se rase pas lui-même. Or, le barbier doit raser tous ceux qui ne se rasent pas eux-mêmes, donc il doit se raser lui-même. Contradiction.
2. Le barbier se rase lui-même. Or, le barbier ne rase pas les gens qui se rasent eux-mêmes, donc il ne doit pas se raser lui-même. Contradiction.

Puisqu'on a une contradiction dans les deux cas, notre hypothèse était fausse. Un tel barbier ne peut pas exister.

La formalisation de cet argument va nous permettre dans la section suivante de démontrer qu'un programme avec un certain comportement ne peut pas exister. Soit V l'ensemble de tous les habitants masculins de notre village. Nous interprétons tout habitant $m \in V$ aussi comme une fonction $V \to \{$ « oui », « non » $\}$, dans le sens que

$$\text{pour tout } m' \in V : m(m') = \begin{cases} \text{« oui »} & \text{si } m \text{ rase } m' \\ \text{« non »} & \text{si } m \text{ ne rase pas } m' \end{cases}$$

Notre barbier b est un habitant de notre village. Nous reformulons la spécification du com-

portement de b avec la nouvelle notation :

$$\text{pour tout } m \in V : b(m) = \begin{cases} \text{« oui »} & \text{si m(m)=« non »} \quad (b1) \\ \text{« non »} & \text{si m(m)=« oui »} \quad (b2) \end{cases}$$

La preuve d'impossibilité s'écrit maintenant comme suit. Il y a deux cas possibles pour la valeur de $b(b)$:

1. $b(b) =$ « oui ». Donc, selon (b2) avec $m = b$ nous avons que $b(b) =$ « non ». Contradiction.
2. $b(b) =$ « non ». Donc, selon (b1) avec $m = b$ nous avons que $b(b) =$ « oui ». Contradiction.

Par conséquent, un tel barbier b n'existe pas. Dans la preuve ci-dessus on a le droit de mettre $b = m$ car la spécification de b en (b1) et (b2) est pour *toutes* les valeurs de m, donc en particulier pour $m = b$.

Le fait qu'on interprète tout élément $m \in V$ comme une fonction sur V n'a a priori rien d'inquiétant. Par exemple on peut très bien interpréter tout nombre naturel $n \in \mathbb{N}$ comme une fonction $\mathbb{N} \to \mathbb{N}$ (par exemple la fonction qui associe à tout nombre x la valeur $n + x$), ou comme une fonction $\mathbb{N} \to \{\text{« oui »}, \text{« non »}\}$ (par exemple la fonction qui associe à tout nombre x la valeur « oui » quand $x \leq n$, et la valeur « non » quand $x > n$). L'argument ci-dessus nous dit simplement que pour *certaines spécification* il n'y a aucune fonction qui répond à la spécification.

12.2 Le problème d'arrêt

L'argument de la section précédente repose sur le fait que nous avons pu interpréter un habitant du village, par exemple le barbier, comme une fonction qui est appliquée à un autre habitant. Si on veut transférer ce genre de raisonnement vers les programmes il faut donc trouver un moyen qui nous permet d'interpréter un programme comme une fonction des programmes vers un ensemble de valeurs. La solution naturelle est de considérer des programmes qui peuvent prendre des programmes en entrées (par exemple lire un fichier qui contient le programme d'entrée, puis le traiter).

Le langage de programmation *Imp* du chapitre 8 n'est pas suffisant pour deux raisons :

1. Les programmes *Imp* ne contiennent pas d'instructions d'entrée et de sortie.
2. Le seul type de données des programmes *Imp* est `int`, il n'y a pas de type de données pour représenter la syntaxe d'un programme.

Il nous faut donc un langage de programmation plus riche. Nous n'allons pas définir formellement ce langage de programmation, pour pouvez dans la suite simplement supposer que le langage de programmation est un langage de programmation qui contient les constructions habituelles comme traiter des arguments, appeler des sous-programmes (selon le langage en tant que méthode, procédure, fonctions, etc.) , des conditionnelles, etc. Par exemple le langage peut être Java, ou Objective CAML.

Des programmes qui traitent des programmes sont absolument normaux en informatique. Il suffit de penser à un éditeur de texte qui peut traiter le texte d'un autre programme (ou, pourquoi pas, son propre texte de programme), ou encore à un compilateur qui prend en

entrée un programme source, par exemple en Java, et sort un autre programme, par exemple en langage machine.

On considère des programmes qui prennent en argument un programme ou plusieurs programmes. On peut supposer qu'avec chaque programme le nombre des arguments est indiqué. Si P est un programme qui attend n arguments alors nous notons $P(P_1, \ldots, P_n)$ le résultat de l'exécution du programme P avec les n arguments $P_1, \ldots, P_n$. Nous supposons que tout programme envoie un résultat en forme d'une chaîne de caractères *quand il a terminé*. Il y a bien sûr la possibilité que l'exécution d'un programme sur des arguments donnés ne termine pas. Nous écrivons $P(P_1, \ldots, P_n) = s$ quand l'exécution termine et envoie le résultat s, et $P(P_1, \ldots, P_n) = \perp$ quand l'exécution ne termine pas. Le fait qu'un programme peut ne pas terminer pose une petite difficulté supplémentaire par rapport à la section 12.1.

Nous pouvons maintenant donner une spécification de la fonction pour laquelle nous montrons dans la suite qu'il n'y a a pas de programme :

$$\text{Pour tout programme } P_1,\, P_2 : H(P_1, P_2) = \begin{cases} \text{« oui »} & \text{si } P_1(P_2) \neq \perp \\ \text{« non »} & \text{si } P_1(P_2) = \perp \end{cases}$$

On appelle H le *problème d'arrêt*. Il s'agit évidemment d'un problème fondamentale de l'informatique, et on serait bien content d'avoir un programme qui répond à cette spécification. Cela permettrait par exemple d'écrire un compilateur qui rejette les programmes qui ne terminent pas. Une solution naïve qui consiste simplement en exécutant P_1 avec entrée P_2 et de « voir si ça termine » ne marchera pas : Si l'exécution termine alors on peut certainement répondre « oui », mais qu'est-ce qu'on peut conclure si le programme tourne toujours après une heure, ou après un jour, ou après deux mois ?

Nous allons dans la suite montrer qu'un tel programme H ne peut pas exister. Supposons par l'absurde qu'un tel programme H existe. On ne peut pas appliquer tout de suite l'argument de la section 12.1 à cette fonction H, nous allons plutôt construire d'abord un autre programme B en utilisant le programme hypothétique H. On construit le programme B suivant :

```
a := ⟨ premier argument du programme ⟩;
h := H(a,a);
if h = « non »
then return(« oui »)
else
  while True do x := x+1 od
fi
```

Ce programme B utilise le programme H, dont nous avons supposé l'existence, en tant que sous-programme. Le programme B fait le calcul suivant : il détermine (en utilisant le sous-programme H) si le programme donné en argument, quand exécuté sur lui même, termine ou pas. Si le programme ne termine pas alors il répond « oui », si le programme termine alors il entre dans une boucle infinie. On peut résumer ce fonctionnement comme suit :

$$\text{Pour tout programme } P : B(P) = \begin{cases} \text{« oui »} & \text{si } H(P,P) = \text{« non »} \quad \text{si } P(P) = \perp & (B1) \\ \perp & \text{si } H(P,P) = \text{« oui »} \quad \text{si } P(P) \neq \perp & (B2) \end{cases}$$

Ce programme correspond maintenant au barbier fictif de la section 12.1, et l'argument d'impossibilité est analogue à l'argument à la fin de la section 12.1. La différence essentielle est qu'il y a pour un programme aussi la possibilité de ne pas terminer.

Il y a deux cas possibles pour la valeur de $B(B)$:

1. $B(B) =$ « oui ». Donc, $B(B) \neq \perp$, et selon (B2) nous avons que $B(B) = \perp$. Contradiction.
2. $B(B) = \perp$. Donc, selon (B1) nous avons que $B(B) =$ « oui ». Contradiction.

Par conséquent, le programme B ne peut pas exister. Or nous avons construit le programme B à partir du (sous-)programme hypothétique H. Donc, le programme H ne peut pas exister.

On dit : « Le problème d'arrêt n'est pas décidable ».

12.3 Références et remarques

Le paradoxe du barbier est une version vulgarisée d'un paradoxe célèbre que le logicien britannique *Bertrand Russell* a publié au début du 20ème siècle. Des paradoxes de ce genre ont été beaucoup utilisé pour montrer des incohérences dans certaines constructions mathématiques, par exemple dans la théorie des ensembles.

Alan Turing, un des pères de l'informatique, a montré en 1936 que le problème d'arrêt est indécidable. Le problème d'arrêt n'est pas le seul problème informatique qui n'est pas décidable, en fait on peut montrer que toutes les propriétés sémantiques et non triviales de programmes ne sont pas décidable (voir un cours de *Calculabilité*).

[**?**] est un joli ouvrage de vulgarisation qui explique de façon très amusante l'idée du paradoxe par « auto-référence » qui est derrière le paradoxe du barbier, et aussi derrière l'indécidabilité du problème d'arrêt.

Dans ce chapitre nous avons abordé les problèmes de calculabilité avec des arguments intuitifs et pas toujours trop formels. Une étude approfondie du phénomène de non-décidabilité nécessite un formalisme de calcul dédié comme par exemples les *machines de Turing*. Ces formalismes présentent l'avantage qu'ils sont d'une part suffisamment puissants pour admettre le raisonnement que nous avons vu à la section 12.2, mais d'autre part suffisamment simples pour être simulés par des autres formalismes. Cela permet par exemple de démontrer que certains puzzles mathématiques ne sont pas décidables (voir un cours de *Calculabilité*).

Annexe A

Différences importantes aux années précédentes

A.1 De l'année 07/08 à l'année 08/09

- Les affectations (à la fois pour la logique propositionnelle et pour la logique de Hoare) sont maintenant des fonctions totales, ce qui enlêve un bon nombre de complications techniques dans les lemmes.
- Proposition 6 ajoutée.
- Preuve formelle de la terminaison de la mise en forme normale de négation (propositions 7 et 8).
- Parle des arbres syntaxiques (page 17).
- Un nouveau chapitre sur les expressions (seulement regroupement des sections, le conténu de chapitre était déjà dans le poly de l'année précédente mais à des endroits différents).
- Nous utilisons maintenat le symbole « ; » aussi pour la concaténation de listes, le symbole @ n'est plus utilisée.

A.2 De l'année 08/09 à l'année 09/10

- Ajouté exercice 1 (donc décalage des numéros des exercices).
- Ajouté la définition de la conséquence d'un ensemble ($T \models p$, définition 6).
- Ajouté le théorème 6.
- Ajouté proposition 14.
- Calcul de Hoare : supprimé l'axiome des expressions booléennes (qui introduisait toute expression booléenne qui est valide). Simplifié la règle de conséquences : Les implications entre les expression booléennes sont maintenant des conditions de côté. Donc, l'ancienne règle

$$\frac{p' \to p \qquad \{p\}\, S\, \{q\} \qquad q \to q'}{\{p'\}\, S\, \{q'\}}$$

  est devenu

$$\frac{\{p\}\, S\, \{q\}}{\{p'\}\, S\, \{q'\}} \qquad \text{si } p' \models p \text{ et } q \models q'$$

Du coup, les exemples des preuves dans les figures 11.1 et 11.2 deviennent beaucoup plus digestes.
- Les règles d'inférences portent un nom, le nom doit être utilisé dans l'écriture d'une preuve. Dans les preuves, on demande de mettre un trait aussi au-dessus d'une feuille.
- La définition de la plus faible pre-condition a changé : maintenant elle est définie sémantiquement, avant elle était définie comme la plus faible (par rapport à l'implication logique) parmi toutes les pre-conditions valides. Cela a fait possible d'énoncer (et de montrer) la proposition 26. Le fait que la plus faible pre-condition est en fait la plus faible parmi toutes les pre-conditions valides est maintenant simplement une observation (proposition 23).
- Ajouté le nouveau chapitre 6 sur la modélisation en logique propositionnelle.

A.3 De l'année 09/10 à l'année 10/11

- La preuve de correction de la règle While du calcul de Hoare (proposition 21) a été considérablement simplifiée par l'introduction de la proposition 20.
- Ajout d'un deuxième exemple pour la modélisation en logique propositionnelle (mariages heureux, Section 6.2).
- Ajout de définition et propriétés de la subsomption aussi entre clauses disjonctives (dans Section 4.5).
- Le sens de la définition de subsomption pour les clauses conjonctives a été inversé : c subsume d si c est inclus en d (pareil pour clauses conjonctives et disjonctives). Par conséquence, on a dans les deux cas que les clauses subsumées sont redondantes.
- Chapitre 2 : ajouter des « repères de notation ».

A.4 De l'année 10/11 à l'année 11/12

- Fautes de frappe dans définition 11 et définition 22.
- Chapitre 6 : ajout d'une section sur le format DIMACS.

A.5 De l'année 11/12 à l'année 12/13

- Chapitre 11 : erreur dans le programme dont la weakest precondition n'est pas exprimable.
- Chapitre 6 : ajout d'une section sur les contraintes de comptage.