# Preuves par induction dans le calcul de superposition

Abdelkader Kersani

UNIVERSITÉ DE GRENOBLE

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

**Abdelkader KERSANI**

Thèse dirigée par **Nicolas Peltier**

préparée au sein **Laboratoire d'informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

# Preuves par induction dans le calcul de superposition

Thèse soutenue publiquement le **30 octobre 2014**,
devant le jury composé de :

**M, Alexander Leitsch**
Professeur, Rapporteur
**M, Laurent Vigneron**
Professeur, Rapporteur
**M, Denis Trystram**
Professeur, Examinateur
**M, Sorin Stratulat**
Maître de conférence, Examinateur
**M, Nicolas Peltier**
Chargé de recherche au CNRS, Directeur de thèse

# Résumé

Nous nous intéressons à des formules de la logique du premier ordre où certaines constantes sont interprétées dans un domaine défini inductivement, comme les entiers. Le problème de la validité n'est pas semi-décidable pour ces formules. Le but de cette thèse est donc d'accroître les capacités des procédures de preuve les plus efficaces pour la logique du premier ordre (fondées sur le calcul de résolution et de superposition) afin de tenir compte de ces constantes particulières. Pour cela, nous adaptons le calcul de superposition en ajoutant notamment un mécanisme de détection de cycles qui simule une forme d'induction mathématique. Nous étudions dans un premier temps le cas particulier des entiers, puis nous généralisons certains des résultats obtenus au cas où les constantes inductives sont définies à l'aide de constructeurs monadiques (des mots). Nous présentons des classes syntaxiques pour lesquelles nous pouvons assurer la complétude et/ou la décidabilité. Nous décrivons un outil appelé SuperInd, fondé sur le démonstrateur Prover9, implémentant les résultats précédents. Enfin, nous décrivons certaines expérimentations et procédons à des comparaisons avec d'autres approches.

# Abstract

We consider first order formulas where some constant symbols are defined in an inductive domain. The validity problem is not semi-decidable for these formulas. This work aims to increase the capabilities of the usual first order proof procedures (usually based on superposition and resolution calculus) to handle these particular constant symbols. Thus, we adapt the superposition calculus using a loop detection mechanism encoding a form of mathematical induction. We first consider the particular case of natural numbers, then we generalize some of these results to the case where the inductive constant symbols are defined with monadic constructors (words). We present some syntactic classes for which we can ensure completeness and/or decidability. We describe a new tool named SuperInd, based on the theorem prover Prover9, implementing our previous results. Finally we describe some experimentations and some comparisons with other approaches.

# Remerciements

Je remercie mon directeur de thèse Nikolas pour tout : sa disponibilité, sa générosité, sa rigueur de travail, son soutien (surtout dans les moments difficiles), son amitié et surtout pour toute l'aide qui m'a apportée. Je suis heureux que Denis Trystram ait accepté d'être le président du jury, surtout qu'il a été un de mes professeurs, qu'il m'avait initié à la recherche en étant mon encadrant de stage de fin d'étude et qu'il a aussi été un collègue en dispensant ensemble un cours à l'Ensimag et pour tout cela je le remercie. Je suis plus qu'honoré que Alexander Leitsch et Laurent Vigneron soient les rapporteurs de cette thèse. Je remercie Alex avec qui j'ai beaucoup apprécié de travailler au sein du projet ASAP et Laurent Vigneron dont j'ai lu quelques travaux, j'espère que la tâche ne sera pas pénible. Je suis aussi content que Sorin Stratulat dont les travaux nous ont un peu inspirés soit dans le jury, et je le remercie.
Je remercie tous les membres de l'équipe CAPP du LIG et tous les membres du projet ASAP avec qui j'ai pris beaucoup de plaisir à travailler.
Enfin je remercie mes parents, mes frères et soeurs pour leurs soutiens et plus particulièrement ma soeur Karima et mon beau-frère Hakim, sans oublier mes amis, surtout ceux qui ont su compter les pages de ce manuscrit à savoir Sandrine, Boris, etc.

# Table des matières

# Première partie

# Introduction

# Chapitre 1

# Présentation du problème et motivations

## 1.1 Cadre général

Ce travail s'inscrit dans le cadre de la déduction automatique (voir par exemple [73] pour une synthèse récente), dont l'objet est de développer des algorithmes permettant de tester la validité de propriétés (e.g., théorèmes mathématiques) et d'en construire des preuves formelles. Ces procédures s'appuient sur la notion de preuve en logique mathématique qui est définie comme une succession d'inférences. Les propriétés sont exprimées dans des langages formels (dont la syntaxe et la sémantique sont définies mathématiquement à l'instar des langages de programmation), et des règles d'inférence permettent de déduire de nouvelles propriétés à partir des axiomes. Ces règles sont similaires aux syllogismes définis par Aristote dont la théorie est l'ancêtre de la logique moderne. Elles ont la forme d'un raisonnement avec plusieurs *propositions* appelées *prémisses* permettant d'affirmer la véracité de la *conclusion* :

$$\frac{\begin{array}{c}\text{Prémisse 1}\\ \text{Prémisse 2}\\ \dots\\ \text{Prémisse } n\end{array}}{\text{Conclusion}}$$

Des ensembles de règles d'inférence formant des systèmes de preuve ont été développés pour un grand nombre de logiques, les plus connues étant le calcul propositionnel (formules booléennes sans quantificateurs) et la logique du premier ordre (autorisant la quantification sur des éléments du domaine). Ces logiques possèdent des pouvoirs d'expression et donc des complexités très variées. Par exemple le problème de la satisfaisabilité est décidable en temps exponentiel pour les formules du

calcul propositionnel (ce problème est NP-complet, c'est-à-dire que tout problème résoluble par un algorithme non-déterministe polynomial lui est réductible et on conjecture habituellement qu'il n'est pas dans P, c'est-à-dire qu'il n'existe aucun algorithme polynomial pour le résoudre). Il est seulement semi-décidable pour la logique du premier ordre (c'est-à-dire qu'il existe un algorithme terminant uniquement si la formule est valide ou insatisfaisable) et indécidable pour la logique d'ordre supérieure, autorisant des quantifications sur les fonctions et les prédicats.

Cette thèse est dédiée à l'ajout de mécanismes de preuve par induction dans les procédures de preuve les plus communément utilisées en démonstration automatique afin de rendre ces procédures plus générales (au sens où elles peuvent démontrer plus de théorèmes, ou traiter une plus large classe de propriétés). Plus précisément, nous nous intéressons à la preuve de *schémas de formules du premier ordre*, c'est-à-dire des formules définies à partir des connectifs logiques usuels (conjonction, disjonction, négation, implication et équivalence), et autorisant des quantifications sur les éléments du domaine, mais dépendant également de paramètres définis sur un ensemble construit par induction (généralement des entiers). Ces formules utilisent donc deux types d'objets : des termes standards interprétés de manière arbitraire (de la manière usuelle en logique du premier ordre), et des termes inductifs, interprétés sur un domaine défini par induction à l'aide d'un ensemble de constructeurs (e.g., les entiers). Tester la validité de tels schémas de formules est plus difficile que prouver des formules standards, car les preuves doivent généralement être construites par induction sur le paramètre, or l'hypothèse d'induction n'est en général pas connue au départ. Il s'agit donc d'étendre les procédures de preuve usuelles afin de raisonner sur des formules contenant des paramètres inductifs et de générer de manière automatique des hypothèses d'induction permettant d'en établir la validité.

On distingue souvent trois types d'inférence [65] : la *déduction*, l'*induction* et l'*abduction*. La déduction est un raisonnement analytique basé sur la notion de tautologie c'est-à-dire que l"implication "Ensemble des prémisses $\Rightarrow$ Conclusion" est une formule qui est toujours vraie (tautologie). Par exemple, si nous considérons la règle du "modus ponens" qui s'écrit :

$$\frac{\begin{array}{c} P \\ P \Rightarrow Q \end{array}}{\mathrm{Q}}$$

nous remarquons que la formule $((P \Rightarrow Q) \wedge P) \Rightarrow Q$ est vraie pour toute valeur de $P$ et $Q$. Contrairement à la déduction, l'induction est un raisonnement synthétique consistant à déduire des lois générales à partir de cas particuliers, où l'on se base sur des propriétés vérifiées par certaines prémisses pour obtenir une propriété plus générale, englobant celles des prémisses. L'abduction vise à trouver la ou les hypo-

thèses susceptibles d'expliquer une certaine conclusion (correspondant en général à une observation).

La notion d'induction que nous considérons dans le présent document est très différente de celle esquissée ci-dessus. Il s'agit de l'induction mathématique (ou preuve *par récurrence*), qui correspond à une forme de déduction utilisée pour raisonner sur des ensembles définis comme des plus petits points fixes d'un certain opérateur monotone (ou plus grand point fixe – on parle alors de co-induction), pour plus de détails voir [29, 34, 83]. Cette technique de preuve consiste à prouver que l'ensemble des éléments vérifiant la propriété est stable par l'opérateur considéré, ce qui permet de conclure que la propriété est vraie pour tout élément du plus petit point fixe. L'avantage est évidemment qu'on peut utiliser la propriété à démontrer comme hypothèse, pour certaines valeurs des paramètres. Plus précisément, on distingue habituellement deux types d'induction.

— L'induction *structurelle*, qui s'applique à un ensemble défini en utilisant un certain ensemble $C$ de constructeurs. On peut montrer qu'une propriété $P(x)$ est vraie pour tout élément $x$ construit sur $C$ en prouvant que l'implication $P(x_1) \wedge \cdots \wedge P(x_n) \Rightarrow P(f(x_1, \ldots, x_n))$ est valide pour tout constructeur $f$ d'arité $n$ dans $C$.

— L'induction *noethérienne* (plus puissante) qui permet de raisonner sur un ensemble $E$ muni d'un bon ordre $<$ (c'est-à-dire d'ordre n'admettant pas de suite infinie décroissante). On démontre que $P(x)$ est vraie pour tout $x \in E$ en montrant que $P(x)$ est vraie si $P(y)$ est vérifiée pour tout $y < x$. De manière équivalente, on peut aussi démontrer que $\neg P(x) \Rightarrow \exists y(y < x \wedge \neg P(y))$ (cette technique est appelée "descente infinie", voir [82]).

Ces deux formes d'induction sont souvent utilisables indifféremment l'une à la place de l'autre. L'induction que nous cherchons à mécaniser dans cette thèse est une forme d'induction noethérienne. Nous définissons des procédures raisonnant sur des ensembles d'entiers ou de mots, en utilisant l'ordre naturel sur les entiers ou l'ordre lié à la taille du mot. L'une des particularités de notre travail est que les propriétés que nous considérons sont susceptibles de contenir des quantificateurs universels ou existentiels qui sont interprétés de manière standard (c'est à dire que les variables liées de $P$ sont interprétés dans un domaine arbitraire – non inductif). Notre objectif est donc de combiner la preuve par induction noethérienne avec les algorithmes les plus performants et les plus utilisés actuellement en démonstration automatique pour prouver des formules du premier ordre. Nous obtenons ainsi une logique strictement plus expressive que la logique du premier ordre – en particulier l'ensemble des formules valides n'est ni récursivement énumérable ni co-récursivement énumérable, comme nous le verrons par la suite. La plupart des travaux dans le domaine de la preuve par induction (notamment les approches par réécriture) s'intéressent à des logiques pour lesquels la satisfiabilité (ou l'existence

d'un contre-exemple) est semi-décidable.

## 1.2 La démonstration automatique

La démonstration automatique est un domaine situé à l'intersection de la logique mathématique, de l'informatique théorique et de l'intelligence artificielle. Il s'est développé à partir des années 60 notamment avec la découverte de la méthode de résolution par A.Robinson [75, 57] et l'avènement des ordinateurs. Son objectif est l'automatisation complète ou partielle de la recherche de preuve, grâce à des logiciels appelés *démonstrateurs automatiques* (*theorem prover*). À une extrémité du spectre, on trouve des démonstrateurs (tels que Vampire [72], E [79], Prover9 [60]) purement automatiques, où l'utilisateur n'intervient que pour formaliser le problème et les axiomes de la théorie et où le démonstrateur retourne la preuve, si elle existe. À l'autre extrémité, se situent des assistants de preuve tels que Coq [84] ou Isabelle [63], où la preuve est construite essentiellement par l'utilisateur, le rôle de l'ordinateur se limitant à vérifier que celle-ci est correcte. En pratique, les systèmes se positionnent entre ces deux extrêmes, avec des degrés d'interaction variables : par exemple la quasi-totalité des assistants de preuve autorisent la définition de tactiques permettant d'automatiser certaines tâches ou même de les déléguer à des systèmes dédiés (e.g., la construction de preuve pour des fragments décidables tels que la logique propositionnelle). De même, les démonstrateurs automatiques permettent à l'utilisateur de contrôler la stratégie de recherche de preuve : choix des prémisses à considérer en priorité, des règles d'inférence ou de détection de redondances etc.

La découverte de la règle de paramodulation [74] puis du calcul de superposition [14] a permis un traitement efficace de l'égalité et a ouvert la voie pour une large application en mathématique. Cependant, les théorèmes mathématiques sont généralement trop complexes et ne peuvent être exprimés en logique du premier ordre. Ces théorèmes sont donc démontrés soit avec des démonstrateurs d'ordre supérieur ou des assistants de preuve, soit avec des démonstrateurs du premier ordre étendus par de nouveaux formalismes plus expressifs qui ne sont pas semi-décidables.

La démonstration automatique connaît un essor considérable depuis les années 90, dû au développement des ordinateurs, à ses applications dans les différents domaines de l'informatique comme les bases de données, mais surtout à ses applications dans la vérification de programmes ou de circuits intégrés notamment après le bug FDIV du processeur Pentium (pour plus de détails voir `http://www.trnicely.net/pentbug/pentbug.html`) qui a conduit des grandes firmes comme Intel a faire appel à des démonstrateurs automatiques.

## 1.3 Point de départ et motivations

L'un des points de départ de nos travaux est l'approche décrite dans [2]. Ce travail a permis de définir une logique permettant de raisonner sur des schémas de formules propositionnelles, dépendant de paramètres entiers et définis par induction sur la valeur de ces paramètres (par exemple à l'aide de connectifs itérés : $\bigvee_{i=a}^{b} \phi$, où $i$ est une variable du langage et $a, b$ sont des expressions symboliques (non des entiers fixés). Cette logique permet par exemple de modéliser des circuits booléens construits inductivement, en composant $n$ circuits standards en séquence ($n$ étant un paramètre). L'objectif initial de cette thèse était d'étendre ces schémas de formules à la logique du premier ordre et d'étendre ensuite les procédures de preuve usuelles afin de prouver de manière automatique la validité de tels schémas. Les démonstrateurs les plus efficaces pour la logique du premier ordre sont généralement fondés sur la procédure de Résolution [75, 57] ou de superposition [14, 62] dans le cas des formules équationnelles, nous nous placerons donc dans ce cadre (à la différence de [2] qui considère des procédures fondées sur la méthode des tableaux sémantiques [45]), et nous proposons de nouvelles règles d'inférence permettant de mécaniser efficacement les preuves par induction.

Notre travail a été financé par le projet ANR-FWF ASAP (ANR-09-BLAN-0407-01), unissant les équipes CAPP du Laboratoire d'Informatique de Grenoble et TLG de l'Université Technique de Vienne. Ce projet avait un double point de départ : d'une part certains travaux avaient été conduits depuis les années 1990-2000 au sein de CAPP concernant l'étude de formalismes appelés *schématisation* visant à représenter de manière finie des séquences de termes ou formules structurellement similaires [66, 67]. D'autre part, l'équipe TLG avait défini un algorithme, nommé CERES : Cut Elimination by Resolution, permettant d'éliminer efficacement les coupures dans les preuves, i.e., de supprimer tous les lemmes intermédiaires pour se ramener à une preuve directe [12, 13]. Cet algorithme se révèle utile pour l'analyse des preuves, et notamment pour le calcul des séquents de Herbrand (qui peuvent être vus comme le "noyau" d'une preuve). Le projet visait à combiner ces deux axes de recherche de manière à étendre la méthode CERES à des preuves faisant appel à l'induction mathématique, qui peuvent être définies comme des schémas de preuves du premier ordre.

La méthode CERES s'appuie de manière essentielle sur la preuve par résolution. En effet, l'algorithme consiste à extraire à partir de la preuve initiale (avec coupure), un ensemble de clauses, appelé "ensemble caractéristique". La preuve de chacune des clauses de cet ensemble peut être construite à partir de la preuve initiale, en effaçant simplement toutes les coupures. Pour construire une preuve équivalente sans coupure, il suffit alors de construire une réfutation par résolution de l'ensemble caractéristique. Dans le cas des preuves standards, l'ensemble carac-

téristique est un ensemble de clauses du premier ordre, qui peut être réfuté par un démonstrateur tel que Prover9 ou E. Dans le cas des preuves par induction, l'ensemble caractéristique n'est pas un ensemble de clauses du premier ordre : il dépend de paramètres entiers, et peut être vu comme un schéma d'ensembles de clauses. Afin d'appliquer la méthode CERES à des preuves inductives, il s'est donc avéré nécessaire de développer des techniques de preuve automatisées permettant de construire des réfutations par résolution (ou plus précisément des schémas de réfutations) de tels ensembles. Ce travail constitue précisément une tentative de réponse à ce problème. Nos travaux possèdent toutefois bien d'autres applications potentielles, par exemple pour la vérification de systèmes dépendant de paramètres, tels que programme avec boucles (le paramètre dénotant alors le nombre d'itérations), preuve de propriété de fonctions récursives... Ils débordent largement du cadre ci-dessus, puisque nous considérons également des schémas définis sur des paramètres non-entiers.

## 1.4 Plan

Nous avons organisé ce manuscrit comme suit : une partie préliminaire rappelant les définitions et les résultats en logique et en déduction automatique dont nous avons besoin dans ce mémoire (cette partie n'a d'autre but que de rendre ce mémoire auto-suffisant, et le lecteur averti peut se passer de sa lecture), une partie contenant nos résultats théoriques et une partie contenant nos résultats pratiques, c'est-à-dire décrivant le travail d'implémentation que nous avons effectué. Nous présentons maintenant le contenu des différents chapitres.

- Dans le chapitre 3 nous introduisons le formalisme des *$n$-clauses* qui permet de coder des formules du premier ordre contenant des constantes interprétées comme des entiers. Nous proposons ensuite une procédure de preuve hybride pour les $n$-clauses, combinant la superposition pour générer des clauses, avec un mécanisme de détection de boucle permettant de générer et d'appliquer des hypothèses d'induction de manière purement automatisée.
- Au chapitre 4, nous montrons comment traduire le formalisme des schémas propositionnels de [2] vers les $n$-clauses.
- Au chapitre 5, nous considérons des formules contenant des constantes interprétées sur un domaine inductif. Nous introduisons un autre formalisme appelé les *$\alpha$-clauses* qui est une extension du langage précédent, autorisant des constantes interprétées sur un domaine défini par un ensemble de constructeurs monadiques (type mots). Nous proposons ensuite une procédure de preuve strictement plus générale que celle du chapitre 3.
- Au chapitre 6, nous présentons quelques critères sémantiques permettant

de garantir la complétude et/ou la terminaison.
- Au chapitre 7, nous utilisons les critères sémantiques précédemment identifiés pour définir deux classes syntaxiques décidables : la classe des clauses $\mu$-contrôlées (inspirée des travaux de [56] et paramétrée par une mesure de complexité $\mu$) et la classe des clauses quasi-fermées.
- Au chapitre 8, nous présentons les différents travaux connexes et nous les comparons brièvement avec notre approche.
- Au chapitre 9 nous présentons *SuperInd* un outil implémentant la procédure de preuve décrite dans le chapitre 3. SuperInd est un démonstrateur pour des ensembles de $n$-clauses, basé sur le démonstrateur pour la logique du premier ordre Prover9 [60], dont nous allons décrire les fonctionnalités et l'implémentation.
- Au chapitre 10 nous décrivons les différentes expérimentations que nous avons menées avec *SuperInd* en utilisant des benchemarks provenant de différentes sources. Nous procédons aussi à quelques comparaisons avec d'autres outils.

Nos travaux ont fait l'objet de deux publications : [53] correspondant au chapitre 3 et une partie du chapitre 10 et [54] correspondant aux chapitres 5, 6 et une partie du chapitre 7.

# Chapitre 2

# Préliminaires

Nous définissons dans ce chapitre la syntaxe et la sémantique de la logique du premier ordre. Puis nous présentons un certain nombre de résultats connus en démonstration automatique, notamment quelques outils théoriques que nous utilisons tout le long de ce mémoire.

## 2.1 Logique du premier ordre

Nous définissons dans cette section la logique du premier ordre (syntaxe et sémantique des termes, des formules et des clauses). Nous donnons tout d'abord les définitions dans un cadre mono-sorté, avant de les étendre au cas multi-sorté.

### 2.1.1 Syntaxe

Soit $\mathcal{X}$ un ensemble infini et dénombrable de variables, $\mathcal{F}$ un ensemble de symboles de fonctions et $\mathcal{P}$ un ensemble de symboles de prédicats. Une *signature* $\Sigma = (\mathcal{F}, \mathcal{P})$ correspond à l'ensemble des symboles du langage. L'*arité* est une fonction qui associe à chaque symbole de prédicat ou de fonction un unique entier positif. Si $f \in \mathcal{F}$ et $arité(f) = 0$ alors $f$ est appelé *constante*. Si $P \in \mathcal{P}$ et $arité(P) = 0$ alors $P$ est une *variable propositionnelle*.

**Définition 1 (*Termes*)** L'ensemble des termes $\mathcal{T}$ est le plus petit ensemble tel que :

— $\mathcal{T}$ contient les variables et les constantes.

— Si $f \in \mathcal{F}$ et $t_1, \ldots, t_n \in \mathcal{T}$ alors $f(t_1, \ldots, t_n) \in \mathcal{T}$ où $n = arité(f)$ et $n > 0$.

Un terme est dit *fermé* s'il ne contient pas de variables. $\Diamond$

**Définition 2 (*Atome, Littéral*)** Un *atome* est de la forme $P(t_1, \ldots, t_n)$, avec $n = arité(P)$ et $n \geq 0$, ou de la forme $s \simeq t$, où $t_1, \ldots, t_n, t, s \in \mathcal{T}$.

Un *littéral* est soit un atome (appelé : littéral *positif*) ou la négation d'un atome (appelé : littéral *négatif*). ◊

**Définition 3 (*Formule*)** Une *formule* du premier ordre est définie inductivement comme suit :

- Un atome est une formule.
- Si $\phi$ est une formule alors $\neg\phi$ est une formule.
- Si $\phi_1, \phi_2$ sont deux formules alors : $\phi_1 \Leftrightarrow \phi_2$, $\phi_1 \Rightarrow \phi_2$ , $\phi_1 \vee \phi_2$ et $\phi_1 \wedge \phi_2$ sont des formules.
- Si $\phi$ est une formule et $x \in \mathcal{X}$ alors $\forall x\, \phi$ et $\exists x\, \phi$ sont des formules.

Une variable $x$ est dite *libre* dans une formule $\phi$ si elle possède une occurrence dans cette formule en dehors de la portée de tout quantificateur $\forall x$ ou $\exists x$. Une formule est dite *fermée* si elle ne contient pas de variables libres. On note *Var($\psi$)* l'ensemble des variables libres qui apparaissent dans un terme ou une formule $\psi$. La *fermeture universelle* d'une formule $\phi$ est la formule $\forall x_1, \ldots, x_n\, \phi$ où pour tout $i$ $x_i \in \mathrm{Var}(\phi)$.
Nous allons définir une formule particulière, appelée une *clause*.

**Définition 4 (*Clause*)** Une *clause* du premier ordre est une disjonction (éventuellement vide) de littéraux (souvent considérée comme un multi-ensemble de littéraux) $L_1 \vee \cdots \vee L_n$. La clause vide est notée $\Box$. Une clause est dite *de Horn* si elle admet au plus un littéral positif. ◊

**Définition 5 (*Substitution*)** Une *substitution* est une fonction de l'ensemble des variables $\mathcal{X}$ dans l'ensemble des termes $\mathcal{T}$, qui associe un terme à une variable. On note $x\sigma$ le terme $\sigma(x)$ substitué à $x$.

Le *domaine* d'une substitution, noté *dom*($\sigma$), est l'ensemble des variables $x$ telles que $x\sigma \neq x$. Si pour tout $x \in \mathcal{X}$, $x\sigma \in \mathcal{X}$ et si $\sigma$ est injective alors $\sigma$ est appelée un *renommage*. On peut étendre une substitution aux termes et aux formules en remplaçant toutes les variables libres $x$ par le terme $x\sigma$. Plus formellement :

- $A\sigma \stackrel{\text{def}}{=} A$ si $A$ est une constante ou une variable propositionnelle.
- $F(t_1, \ldots, t_n)\sigma \stackrel{\text{def}}{=} F(t_1\sigma, \ldots, t_n\sigma)$ si $F \in \mathcal{P} \cup \mathcal{F}$.
- $(\phi \diamond \psi)\sigma \stackrel{\text{def}}{=} \phi\sigma \diamond \psi\sigma$ si $\phi, \psi$ sont deux formules et $\diamond \in \{\Leftrightarrow, \Rightarrow, \vee, \wedge\}$.
- $(\exists x\, \phi)\sigma \stackrel{\text{def}}{=} \exists x\, (\phi\theta)$ où $y\theta = y\sigma$ si $y \neq x$ et $x\theta = x$.
- $(\forall x\, \phi)\sigma \stackrel{\text{def}}{=} \forall x\, (\phi\theta)$ où $y\theta = y\sigma$ si $y \neq x$ et $x\theta = x$.

Soient $\sigma$, $\theta$ deux substitutions, la composition de $\sigma$ et $\theta$ est notée $\sigma\theta$. Une substitution $\sigma$ est une *instance* d'une substitution $\sigma'$ si il existe $\theta$ telle que $\sigma = \sigma'\theta$. On note $t \preceq s$ s'il existe une substitution $\sigma$ telle que $t = s\sigma$. Une substitution est un *unificateur* de deux termes $t$, $s$ si et seulement si $t\sigma = s\sigma$, $s$ et $t$ sont dits *unifiables*.

**Proposition 6** *Si $s$ et $t$ sont deux termes unifiables, il existe un unificateur unique $\sigma$ (modulo un renommage) tel que tout unificateur $\theta$ de $t$ et $s$ est une instance de $\sigma$. Dans ce cas $\sigma$ est appelé l'*unificateur le plus général *et on le note $mgu(t, s)$.*

### 2.1.2 Sémantique

**Définition 7 (*Interprétation*)** une *interprétation* est un couple $(\mathcal{D}, \mathcal{I})$ où :

- $\mathcal{D}$ est un ensemble non vide.
- $\mathcal{I}$ une fonction qui associe :
  - à chaque variable propositionnelle $P \in \mathcal{P}$ une valeur de vérité $\mathcal{I}_P \in \{\mathtt{V}, \mathtt{F}\}$.
  - à chaque symbole de prédicat $P \in \mathcal{P}$ d'*arité* $n > 0$ une fonction $\mathcal{I}_P$ de $\mathcal{D}^n$ dans $\{\mathtt{V}, \mathtt{F}\}$.
  - à chaque constante $c \in \mathcal{F}$ une valeur $\mathcal{I}_c \in \mathcal{D}$.
  - à chaque symbole de fonction $f \in \mathcal{F}$ d'*arité* $n > 0$ une fonction $\mathcal{I}_f$ de $\mathcal{D}^n$ dans $\mathcal{D}$.

  $\mathcal{I}$ est étendue aux variables en associant pour chaque variable une valeur unique de $\mathcal{D}$ notée $\mathcal{I}_x$. ◊

On note $\mathcal{I}_{\{x \leftarrow v\}}$ l'interprétation $\mathcal{J}$ qui coïncide avec $\mathcal{I}$ en tout point, sauf que l'interprétation de $x$ est à $v$ c'est-à-dire $\mathcal{J}(x) = v$.

L'interprétation d'un terme $t$ dans une interprétation $\mathcal{I}$ est définie de manière inductive comme suit :

- Si $t$ est une variable $x \in \mathcal{X}$ alors $\mathcal{I}(t) = \mathcal{I}_x$.
- Si $t$ est une constante $c \in \mathcal{F}$ alors $\mathcal{I}(t) = \mathcal{I}_c$.
- Si $t$ est de la forme $f(t_1, \ldots, t_n)$ alors $\mathcal{I}(t) = \mathcal{I}_f(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$.

La valeur de vérité d'une formule $\phi$ dans $\mathcal{I}$ est définie de manière inductive comme suit :

- Si $\phi$ est une variable propositionnelle $P \in \mathcal{P}$ alors $\mathcal{I}(\phi) = \mathcal{I}_P$.
- Si $\phi$ est de la forme $t \simeq s$ alors $\mathcal{I}(\phi) = \mathtt{V}$ si $\mathcal{I}(s) = \mathcal{I}(t)$ et $\mathcal{I}(\phi) = \mathtt{F}$ sinon.
- Si $\phi$ est un atome $P(t_1, \ldots, t_n)$ alors $\mathcal{I}(\phi) = \mathcal{I}_P(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$.
- Si $\phi = \phi_1 \vee \phi_2$ alors $\mathcal{I}(\phi) = \mathtt{V}$ ssi $\mathcal{I}(\phi_1) = \mathtt{V}$ ou $\mathcal{I}(\phi_2) = \mathtt{V}$.
- Si $\phi = \phi_1 \wedge \phi_2$ alors $\mathcal{I}(\phi) = \mathtt{V}$ ssi $\mathcal{I}(\phi_1) = \mathtt{V}$ et $\mathcal{I}(\phi_2) = \mathtt{V}$.
- Si $\phi = \neg\psi$ alors $\mathcal{I}(\phi) = \mathtt{V}$ ssi alors $\mathcal{I}(\psi) = \mathtt{F}$.
- Si $\phi = \forall x\, \psi$ alors $\mathcal{I}(\phi) = \mathtt{V}$ ssi pour tout $c \in \mathcal{D}$, $\mathcal{I}_{\{x \leftarrow c\}}(\psi) = \mathtt{V}$.
- Si $\phi = \exists x\, \psi$ alors $\mathcal{I}(\phi) = \mathtt{V}$ ssi il existe une valeur $c \in \mathcal{D}$, $\mathcal{I}_{\{x \leftarrow c\}}(\psi) = \mathtt{V}$.

Si $\mathcal{I}(\phi) = \mathtt{V}$ alors $\mathcal{I}$ est un *modèle* de $\phi$, on dit aussi que $\mathcal{I}$ *valide* $\phi$ et on note $\mathcal{I} \models \phi$. Si $\mathcal{I}(\phi) = \mathtt{F}$ alors $\mathcal{I}$ est un *contre-exemple* de $\phi$, et on note $\mathcal{I} \not\models \phi$. Nous étendons la notion de validité aux formules non fermées, en considérant la fermeture universelle $\forall x_1, \ldots, x_n \phi$ (où $\mathrm{var}(\phi) = \{x_1, \ldots, x_n\}$) de $\phi$ c'est-à-dire : $\mathcal{I}$

est un modèle d'une formule non fermée $\phi$ si $\mathcal{I} \models \forall x_1, \ldots, x_n \phi$.
Une formule est :

— *satisfaisable* si elle a au moins un modèle.
— *insatisfaisable* si elle possède aucun modèle.
— *valide* si elle possède aucun contre-exemple.

Une formule $\phi$ est une *conséquence logique* d'une formule $\psi$ (noté $\phi \models \psi$) ssi chaque modèle de $\phi$ est un modèle de $\psi$. Deux formules $\phi$ et $\psi$ sont dites *sat-équivalentes* ssi la condition suivante est vérifiée : $\phi$ est satisfaisable ssi $\psi$ est satisfaisable.

On note $\phi \equiv \psi$ is $\phi$ et $\psi$ ont même valeur de vérité dans toute interprétation.

On supposera généralement dans la suite que toutes les formules sont fermées, i.e., si une formule n'est pas fermée on considérera sa fermeture universelle.

**Définition 8** (*Interprétation de Herbrand*) Une interprétation $\mathcal{I}$ est *de Herbrand* si le domaine de $\mathcal{I}$ est une congruence sur l'ensemble des termes fermés. $\Diamond$

Une interprétation de Herbrand $\mathcal{I}$ qui valide une formule $\phi$ ($\mathcal{I} \models \phi$), est appelée *modèle de Herbrand*. Il est clair que si $\mathcal{I}$ est un modèle de Herbrand et $t$ un terme fermé alors $\mathcal{I}_t = t$.

**Définition 9** (*Modèle minimal de Herbrand*) Un modèle de Herbrand $\mathcal{I}$ d'un ensemble $S$ est *minimal* ssi pour tout modèle de Herbrand $\mathcal{J}$ de $S$ et pour tout atome fermé $A$ nous avons $\mathcal{I} \models A \Rightarrow \mathcal{J} \models A$. $\Diamond$

Tout ensemble de clause de Horn satisfaisable admet un modèle minimal.

## 2.2 Variantes et extensions

Nous définissons dans ce qui suit deux variantes de la logique du premier ordre que nous utilisons dans cette thèse.

### 2.2.1 Logique clausale équationnelle du premier ordre

La logique clausale équationnelle est une logique du premier ordre dont les formules sont des clauses ne contenant pas d'autre prédicat que l'égalité. Cette logique est généralement utilisée dans le cadre de la preuve par saturation (calcul de superposition) car elle évite d'avoir à introduire des règles spécifiques pour traiter les littéraux non équationnels. Les clauses peuvent contenir des variables libres, implicitement quantifiées universellement, voir 2.1. Afin de coder les prédicats usuels (éléments de $\mathcal{P}$), nous utilisons une constante `V` c'est-à-dire que pour tous $q \in \mathcal{P}$ et $t \in \mathcal{T}$, la formule $q(t)$ est codée par $q(t) \simeq \mathtt{V}$ et $\neg q(t)$ correspond à $q(t) \not\simeq \mathtt{V}$.

Dans le cadre de la logique équationnelle, les interprétations peuvent être vues comme des congruences sur l'ensemble des termes fermés (en supposant que celui-ci n'est pas vide). En effet, nous avons vu qu'une interprétation $\mathcal{I}$ validait une formule de la forme $s \simeq t$ ssi $\mathcal{I}(s) = \mathcal{I}(t)$ (avec $s, t$ deux termes du premier ordre). Comme l'égalité satisfait les axiomes de réflexivité, de symétrie, de transitivité et de substitutivité, nous pouvons remarquer que la relation $\simeq_{\mathcal{I}} \stackrel{\text{def}}{=} \{(s,t) \in \mathcal{T} \times \mathcal{T} | \mathcal{I} \models s \simeq t\}$ est une *congruence* dans $\mathcal{T}$.

### 2.2.2 La logique du premier ordre multi-sortée

La logique multi-sortée est une généralisation de la logique du premier ordre. Elle est utilisée lorsque plusieurs types d'objets apparaissent dans le langage, par exemple, la formalisation des structures de données en informatique comme : les entiers, les listes ... Pour différentier ces objets, nous utilisons la notion de *sorte*, qui correspond à l'ensemble des objets qui sont de même type. Nous pouvons, par exemple, définir une sorte pour les entiers et une sorte pour les listes.

#### Signature

Une *signature* de la logique multi-sortée est de la forme : $(\mathcal{S}, \mathcal{F}, \tau)$ où :
- $\mathcal{S}$ est un ensemble de symboles de sortes qui contient obligatoirement la sorte `bool`.
- $\mathcal{F}$ est un ensemble de symboles de fonctions.
- $\tau$ est une fonction qui associe à chaque élément $f \in \mathcal{F}$ une séquence non vide d'éléments (sortes) de $\mathcal{S}$, et à chaque variable une sorte dans $S$. Si $\tau(f) = (\mathtt{s_1}, \ldots, \mathtt{s_n}, \mathtt{s})$ alors on note $f : \mathtt{s_1}, \ldots, \mathtt{s_n} \rightarrow \mathtt{s}$, où $n = \mathit{arité}(f)$, et $\mathtt{s_1}, \ldots, \mathtt{s_n}$ sont les co-domaines de $f$.

L'ensemble des prédicats $\mathcal{P}$ est défini par l'ensemble des fonctions $f$ de la forme $f : \mathtt{s_1}, \ldots, \mathtt{s_n} \rightarrow \mathtt{bool}$.

Les termes sont construits sur la signature multi-sortée en respectant les contraintes de sorte :

**Définition 10** Un *terme de sorte* $\mathtt{s}$ est soit une variable telle que $\tau(x) = \mathtt{s}$, ou bien de la forme $f(t_1, \ldots, t_n)$ où $\tau(f) = (\mathtt{s_1}, \ldots, \mathtt{s_n}, \mathtt{s})$ et $\forall i \in [\![1, n]\!]$, $t_i$ est un terme de sorte $\mathtt{s_i}$. $\Diamond$

Les atomes en logique équationnelle multisortée sont soit des termes de sorte `bool` soit de la forme $t \simeq s$ où $t$ et $s$ sont deux termes de même sorte. Les formules sont définies de manière usuelle.

**Sémantique**

Une interprétation est définie comme dans les sections 2.1 et 2.2, mais doit, en plus, associer à chaque sorte $\mathtt{s}$ un domaine $\mathcal{D}_{\mathtt{s}} \neq \emptyset$. L'interprétation des éléments de sorte $\mathtt{s}$ doit alors être choisi parmi $\mathcal{D}_{\mathtt{s}}$ et les fonctions $f : \mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s}$ sont interprétées comme des fonctions de $D_{\mathtt{s}_1} \times \cdots \times D_{\mathtt{s}_n}$ vers $\mathcal{D}_{\mathtt{s}}$.

## 2.3 Systèmes de preuve et déduction automatique

Un *système de preuve* est un ensemble d'*axiomes* et de *règles d'inférence* permettant de prouver un théorème. Une *règle d'inférence* ou de *déduction*, appliquée aux clauses $C_1 \ldots, C_n$ (appelées *prémisses*) afin de déduire logiquement une autre clause $C$ appelée *conclusion*, est notée :

$$\frac{C_1 \quad \ldots \quad C_n}{C}$$

Les variables libres partagées entre les formules $C_1, \ldots, C_n$ sont renommées pour éviter les conflits.

Il existe de nombreux systèmes de preuve, le premier étant l'*approche axiomatique* de *David Hilbert* qui est l'un des premiers à penser à la mécanisation de la preuve (voir [81]), mais il existe d'autre systèmes comme la *déduction naturelle*, ou le *calcul des séquents* [42]. L'objet de la déduction automatique est de proposer des systèmes de preuve efficaces, afin d'utiliser la puissance des ordinateurs dans la recherche de preuves. Nous n'utiliserons dans cette thèse que des systèmes de preuve opérant sur des ensembles de clauses :

**Définition 11** Une *procédure de preuve* est un ensemble de règles d'inférence qui, à partir d'un ensemble de clauses $S$, dérive de nouvelles clauses.

Les procédures de preuve pour la logique clausale utilisent généralement la preuve par *réfutation* (par l'absurde) c'est-à-dire si nous voulons montrer qu'une clause $C$ est une conséquence logique d'un ensemble de clauses $S$, il suffit de dériver de l'ensemble $S \cup \{\neg C\}$, la clause vide (contradiction). On juge une procédure de preuve selon deux propriétés qu'elle peut posséder à savoir la *correction* et la *complétude* définies par ce qui suit.

**Définition 12** Une procédure de preuve est dite *correcte* ssi toute clause déductible d'un ensemble de clauses $S$ est une conséquence logique de $S$. $\Diamond$

**Définition 13** Une procédure de preuve est dite *complète* ssi toute clause qui est une conséquence logique d'un ensemble de clauses $S$, est déductible de $S$. $\Diamond$

Cette propriété est rarement satisfaite en pratique. Comme les procédures de preuve utilisent la preuve par réfutation nous définissons une notion de complétude plus faible appelée *complétude réfutationnelle*.

**Définition 14** Une procédure de preuve est dite *complète pour la réfutation* ssi la clause vide (contradiction) est déductible de tout ensemble de clauses insatisfaisable. ◊

Il existe plusieurs procédures de preuve pour la logique du premier ordre nous présentons, dans ce qui suit, celles que nous utilisons dans le cadre de notre travail.

### 2.3.1 Le calcul de résolution et paramodulation

La règle de résolution (1), introduite par *Robinson* [75], est une généralisation du *modus ponens*.

$$\textit{Résolution} \quad \frac{P \vee L_1 \qquad \neg Q \vee L_2}{(L_1 \vee L_2)\sigma} \quad (1)$$

Si $\sigma = \text{mgu}(P, Q)$

La règle de résolution est accompagnée de la règle de *factorisation* (2) qui permet de supprimer les doublons et réduire la taille des clauses.

$$\textit{Factorisation} \quad \frac{L \vee R \vee C}{(L \vee C)\sigma} \quad (2)$$

Si $\sigma = \text{mgu}(L, R)$

Nous renvoyons à [57] pour plus de détails sur le calcul de résolution et ses propriétés.

#### Le traitement de l'égalité

Le prédicat d'égalité $\simeq$ est particulier, car il possède des propriétés qu'il faut prendre en compte comme la *réflexivité*, la *symétrie*, ... La règle de *paramodulation* [74] a été introduite pour éviter d'écrire ces propriétés sous forme axiomatique (ce qui se révèle en pratique peu efficace).

$$\textit{Paramodulation} \quad \frac{(t_1 \simeq t_2) \vee C_1 \qquad L[t'_1]_p \vee C_2}{(L[t_2]_p \vee C_1 \vee C_2)\sigma} \quad (3)$$

avec $\sigma = \text{mgu}(t_1, t'_1)$.

Afin de simuler la réflexivité de l'égalité, une autre règle appelée *réflexion* est

ajoutée à la paramodulation, elle est définie comme suit :

$$\textit{Réflexion} \quad \frac{(t \not\simeq s) \vee C}{C\sigma} \quad (4)$$

avec $\sigma = \mathrm{mgu}(t, s)$.

**Théorème 15** *([62]) Le calcul "Résolution + Paramodulation" (défini avec les règles (1),(2),(3) et (4)) est correct et complet pour la réfutation.*

### 2.3.2 Le calcul de superposition

L'application du calcul ainsi défini engendre parfois un grand nombre de clauses. Afin de le rendre plus efficace, certaines *stratégies* ont été proposées pour réduire l'espace de recherche sans affecter la complétude réfutationnelle. La forme la plus aboutie de ces stratégies a donné naissance au calcul de superposition, initialement introduit dans [14] (voir également [77]). Ces stratégies consistent à restreindre l'application des règles d'inférences afin d'éviter de générer des clauses qui ne jouent pas un rôle essentiel dans la preuve. Les restrictions sont généralement basée sur l'*ordre* des littéraux et sur une certaine fonction appliquée aux littéraux appelée *fonction de sélection*. Nous allons définir dans ce qui suit ces deux notions :

**Définition 16** Une relation $>$ est une *relation d'ordre* sur un ensemble $S$ si :
- $\forall x \in S$, $x \not> x$ ($>$ est irréflexive).
- $\forall x, y, z \in S$ si $x > y$ et $y > z$ alors $x > z$ ($>$ est transitive). $\Diamond$

Une relation d'ordre est *totale* sur un ensemble $S$ si $\forall x, y \in S\ x > y \vee y > x \vee x = y$. Et elle est *bien fondée* si il n'existe aucune suite infinie $s_1, \ldots, s_n, \ldots$ telle que $\forall i, s_i > s_{i+1}$.

**Définition 17** Nous supposons donné un ordre $>$ sur les termes. $>$ est un *ordre de réduction* si :
- il est bien fondé ;
- il est *stable par substitution* c-à-d pour tout terme $s$ et $t$ si $s > t$ alors pour toute substitution $\sigma$, $s\sigma > t\sigma$ ;
- il est *monotone* c-à-d que pour tout symbole de fonction $f$ et pour tous termes $s_1, \ldots, s_n$, $t$ si $s_i > t$ (avec $i < n$) alors $f(s_1, \ldots, s_i, \ldots, s_n) > f(s_1, \ldots, t, \ldots, s_n)$. $\Diamond$

L'ordre $>$ sur les termes est étendu aux littéraux équationnels (les littéraux non-équationnels pouvant être vus comme des littéraux équationnels de la forme $l \simeq \mathtt{V}$ ou $l \not\simeq \mathtt{V}$) en utilisant la notion de *multi-ensemble* qui est défini comme suit :

**Définition 18** Un *multi-ensemble* $\mathcal{M}$ sur un ensemble $S$ est une fonction de $S$ dans $\mathbb{N}$. On dit qu'un élément $x$ *appartient* à $\mathcal{M}$ ssi $\mathcal{M}(x) \neq 0$. Si $\mathcal{M}$, $\mathcal{M}_1$ et $\mathcal{M}_2$ sont des multi-ensembles alors :

- $\mathcal{M}$ est *vide* ssi $\forall x, \mathcal{M}(x) = 0$.
- $\mathcal{M}$ est *fini* ssi $\{x | \mathcal{M}(x) \neq 0\}$ est fini.
- $\mathcal{M}_1 \cup \mathcal{M}_2$ est un multi-ensemble $\mathcal{M}$ défini par : $\forall x\, \mathcal{M}(x) = \mathcal{M}_1(x) + \mathcal{M}_2(x)$.
- $\mathcal{M}_1 \cap \mathcal{M}_2$ est un multi-ensemble $\mathcal{M}$ défini par : $\forall x\, \mathcal{M}(x) = min(\mathcal{M}_1(x), \mathcal{M}_2(x))$.
- $\mathcal{M}_1 \setminus \mathcal{M}_2$ est un multi-ensemble $\mathcal{M}$ défini par : $\forall x\, \mathcal{M}(x) = min(0, \mathcal{M}_1(x) - \mathcal{M}_2(x))$.
- $\mathcal{M}_1 = \mathcal{M}_2$ ssi $\forall x\, \mathcal{M}_1(x) = \mathcal{M}_2(x)$. ◊

**Exemple 19** Un multi-ensemble $\mathcal{M}$ défini par : $\mathcal{M}(t) = 2$ et $\mathcal{M}(s) = 1$ correspond à (avec l'écriture ensembliste) $\{t, t, s\}$. ♣

**Définition 20** Soit $>$ une relation d'ordre sur $S_1 \cup S_2$. L'*extension multi-ensemble* $\geq_{mul}$ est la plus petite relation transitive telle que : pour tous multi-ensembles $\mathcal{M}_1$, $\mathcal{M}_2$, définis sur $S_1$, $S_2$ (respectivement), $\mathcal{M}_1 \geq_{mul} \mathcal{M}_2$ ssi $\mathcal{M}_2$ est de la forme $(\mathcal{M}_1 \setminus X) \cup Y$ avec $\forall y \in Y$, $\exists x \in X$ tel que $x > y$. ◊

**Définition 21** Soit $L$ le littéral de la forme $t \bowtie s$ (avec $\bowtie \in \{\simeq, \not\simeq\}$). $L$ est représenté par un multi-ensemble $\mathcal{M}_L$ défini comme suit :

- $\mathcal{M}_L = \{\{t\}, \{s\}\}$ si $L$ est positif ($\bowtie \equiv \simeq$).
- $\mathcal{M}_L = \{\{t, s\}\}$ si $L$ est négatif ($\bowtie \equiv \not\simeq$). ◊

**Définition 22** Soit $\geq$ un ordre de réduction sur les termes. il est étendu aux littéraux équationnels de la manière suivante :

$$L_1 \geq L_2 \Leftrightarrow \mathcal{M}_{L_1} \geq_{mul} \mathcal{M}_{L_2}$$

Nous allons maintenant, à l'aide de l'ordre de réduction $>$, définir la notion de *littéral maximal* dans une clause et la notion de *fonction de sélection.*

**Définition 23** Un littéral $l_i$ est *maximal* dans une clause de la forme $l_1 \vee \cdots \vee l_n$ ssi $\forall j\ l_j \not> l_i$. ◊

**Définition 24** Une *fonction de sélection sel* est une fonction qui associe à chaque clause $C$ un sous-ensemble de ses littéraux (appelé *littéraux sélectionnés*) telle que : soit $sel(C)$ contient un littéral négatif ou $sel(C)$ contient tous les littéraux maximaux (par rapport à un ordre de réduction $>$ fixé).

La règle de *superposition* [14] [62] est une variante de la paramodulation. Cette règle applique des remplacements comme la paramodulation mais en imposant d'autres conditions comme par exemple : appliquer la règle sur les littéraux sélectionnées. Formellement, la règle est définie comme suit :

$$\textit{Superposition} \quad \frac{C_1 \vee (l \simeq r) \qquad C_2 \vee w[t]_p \bowtie u}{((w[r]_p \bowtie u) \vee C_1 \vee C_2)\sigma} \quad (5)$$

avec $\bowtie \in \{\simeq, \not\simeq\}$, $\sigma = \text{mgu}(l, t)$, $r\sigma \not> l\sigma$, $u\sigma \not> w\sigma$ et $(l \simeq r)\sigma$ $(w \bowtie u)\sigma$ sont sélectionnés dans $(C_1 \vee (l \simeq r))\sigma$ et $(C_2 \vee w \bowtie u)\sigma$, respectivement et $t$ n'est pas une variable.
Afin de préserver les résultats de complétude, une autre règle est ajoutée, appelée *factorisation équationnelle*, elle permet de modifier la structure des clauses de telle sorte que le membre gauche d'un littéral équationnel n'apparaisse plus dans d'autres littéraux équationnels. Formellement :

$$\textit{Factorisation équationnelle} \quad \frac{(t \simeq s) \vee (l \simeq r) \vee C}{((s \not\simeq r) \vee (l \simeq r) \vee C)\sigma} \quad (6)$$

avec $\sigma = \text{mgu}(t, l)$, $l\sigma \not\leq r\sigma$, $t\sigma \not\leq s\sigma$ et $(t \simeq s)\sigma$ est sélectionné dans $((t \simeq s) \vee (l \simeq r) \vee C)\sigma$.

**Théorème 25** *([14],[62]) Le calcul de* superposition*, défini avec les règles (4), (5) et (6), est complet pour la réfutation.*

### 2.3.3 Règles de simplification et de suppression

Les procédures de preuve engendrent un très grand nombre de clauses même avec l'utilisation des stratégies ordonnées. Nous introduisons certaines règles qui permettent de filtrer certaines clauses générées qui n'apportent aucune nouvelle information et qui sont donc inutiles à la preuve, ces règles sont appelées *règles de suppression*. Ces règles obéissent au critère de redondance suivant, initialement défini dans [14].

**Définition 26** Une clause $C$ est *redondante* par rapport à un ensemble de clauses $S$ si pour toute instance fermée $C\sigma$, il existe $n$ clauses $D_1, \ldots, D_n$ et $n$ substitutions fermées $\theta_1, \ldots, \theta_n$ telles que $D_i\theta_i \leq C\sigma$ et $\{D_i\theta_i \mid i \in \{1..n\}\} \models C\sigma$. ◊

En utilisant la redondance nous définissons la notion de *saturation* que nous utiliserons comme un critère de décision et d'arrêt.

**Définition 27** Soit $E_{inf}$ un système d'inférence, un ensemble de clauses $S$ est *saturé* (par rapport à $E_{inf}$) si toute clause générée par $S$ (avec $E_{inf}$) est redondante par rapport à $S$. ◊

Nous allons maintenant définir la notion de *subsomption* :

**Définition 28** On dit qu'une clause $C$ *subsume* une clause $D$ s'il existe une substitution $\sigma$ telle que $C\sigma \subseteq D$. $\Diamond$

Notez que toute clause subsumée par une clause appartenant à un ensemble $S$ est une clause redondante par rapport à $S$. La règle de subsomption consiste à éliminer toutes les clauses qui sont subsumées par des clauses existantes, elle est notée comme suit :

$$\textit{Subsomption} \quad \frac{S \cup \{C\}}{S} \quad (a)$$

Avec $\exists D \in S$, telle que $C$ est subsumée par $D$.
Les tautologies sont aussi des clauses redondantes et pour cela nous définissons la règle d'*élimination de tautologies* notée comme suit :

$$\textit{Elimination de tautologies} \quad \frac{S \cup \{C \vee A \vee \neg A\}}{S} \quad (b_1)$$

La même règle dans le cas équationnel est définie comme suit :

$$\textit{Elimination de tautologies équationnelles} \quad \frac{S \cup \{C \vee t \simeq t\}}{S} \quad (b_2)$$

Nous allons maintenant définir les *règles de simplification*, qui sont des règles permettant de simplifier les clauses et de les écrire de manière à ce que les traitements qui leur sont appliqués soient minimaux, nous pouvons par exemple considérer la règle de factorisation comme une règle de simplification. La *démodulation* est l'une des règles de simplification, elle est notée comme suit :

$$\textit{Démodulation} \quad \frac{S \cup \{L[r]_p \vee C, (t \simeq s) \vee C'\}}{S \cup \{L[s\sigma]_p \vee C, (t \simeq s) \vee C')\}} \quad (c)$$

avec $t\sigma = r$ , $s\sigma < t\sigma$, $C'\sigma \subseteq C$, $(t \simeq s)\sigma$ est sélectionné dans $((t \simeq s) \vee C')\sigma$, $(t \simeq s \vee C')\sigma < L[r]_p \vee C$.

L'ajout des règles de simplification et de suppression n'affecte pas la complétude réfutationnelle des calculs de superposition et de résolution, comme le montre le théorème qui suit.

**Théorème 29** *([19], [62], [14]) Si un ensemble de clauses $S$ est insatisfaisable et saturé par rapport au calcul de superposition alors $\square \in S$.*

# Deuxième partie

# Superposition et Induction

# Chapitre 3

# Formules du $1^{er}$ ordre avec des termes de type entier

Nous considérons, dans ce chapitre, les formules de la logique du premier ordre contenant des constantes interprétées comme des entiers naturels. Nous illustrons cela dans l'exemple qui suit. Soit $\phi$ la conjonction des formules suivantes :

$$p(0, a)$$
$$\forall x, y \, \neg p(x, y) \vee p(x + 1, f(y))$$
$$\exists n \forall x \, \neg p(n, x)$$

La formule $\phi$ est satisfaisable dans le sens usuel (où les termes sont interprétés comme des termes du premier ordre). Un modèle donné, par exemple, par l'interprétation $p(x, y) = \mathtt{V}$ ssi $x \geq 0$, sur les entiers relatifs. Cependant la formule est insatisfaisable si nous considérons que le premier argument de $p$ est interprété comme un entier naturel (avec les interprétations standards de 0, 1 et +). La variable quantifiée $n$ doit être interprétée comme un entier naturel $k$, nous pouvons alors vérifier, par induction sur $k$, que $\forall k \in \mathbb{N} \, \phi \models p(k, f^k(a))$, ce qui implique que la formule est insatisfaisable. Les démonstrateurs usuels ne peuvent établir l'insatisfaisabilité de ce genre de formules dont le pouvoir d'expression dépasse le cadre de la logique du premier ordre (il est connu qu'il n'est pas possible d'exprimer par un ensemble d'axiomes le fait que $n$ appartient à $\mathbb{N}$). Dans ce chapitre, nous allons introduire une procédure de preuve permettant de raisonner sur ces formules hybrides.

L'idée est de représenter les clauses ci-dessus par des clauses contraintes, obtenues en remplaçant le paramètre $n$ par une nouvelle variable $y$ et en ajoutant la contrainte $n \simeq y$. Cela permet ensuite d'appliquer les règles d'inférence de manière usuelle, l'unification calculant automatiquement dans la partie contrainte les conditions sur le paramètre qui permettent l'application des règles. Par exemple la formule $\exists n \forall x \, \neg p(n, x)$ sera remplacée par la clause contrainte $[p(y, x) \not\simeq \mathtt{V} \mid n \simeq y]$

(le quantificateur existentiel est remplacé par une constante de type entier, de manière similaire à l'opération usuelle de Skolemisation, à ceci près que la constante doit être interprétée comme un entier). Par résolution avec la première clause 1, on obtient alors la clause vide avec la contrainte $n \simeq 0$, indiquant que l'ensemble est insatisfaisable si $n = 0$. L'insatisfaisabilité de la formule de départ est démontrée si on est capable de générer une réfutation pour chaque valeur de $n \in \mathbb{N}$ (ce qui est en général impossible sans induction).

Nous allons tout d'abord définir formellement le langage des clauses contraintes, ou $n$-clauses et étendre le calcul de superposition à ce nouveau formalisme. Dans un second temps, nous allons introduire une nouvelle règle permettant d'obtenir des lemmes inductifs de manière complètement automatique en analysant l'espace de recherche. Enfin, nous allons proposer deux algorithmes pour implémenter cette nouvelle règle inductive.

## 3.1 Logique des $n$-clauses

Dans ce chapitre, nous considérons la logique équationnelle multi-sortée où l'ensemble de sortes est $\mathcal{S} = \{\texttt{term}, \texttt{nat}, \texttt{bool}\}$. La sorte `term` contient les symboles des termes standards et `nat` représente les entiers naturels. La signature contient les symboles 0 et succ et ne contient aucun autre symbole de sorte `nat`. Nous considérons un symbole particulier $n$ que nous appelons *paramètre* qui n'appartient pas à la signature.

**Définition 30** Nous appelons *$n$-clause* le couple $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ où $C$ est une clause ne contenant aucun terme non variable de type `nat` et aucune équation entre variables de type `nat` et les $t_i$ (avec $1 \leq i \leq k$) sont des termes de sorte `nat`. Elle est *normalisée* si $k \in \{0, 1\}$. Si $k = 0$, $\bigwedge_{i=1}^{k} n \simeq t_i$ est equivalente à `V`, par convention $[C \mid \bigwedge_{i=1}^{k} n \simeq x]$ est simplement écrite $C$.

$C$ est la *partie clausale* de la $n$-clause, et $\bigwedge_{i=1}^{k} n \simeq t_i$ est la *contrainte*. $\Diamond$

Par définition, $n$ n'apparaît que dans la contrainte d'une $n$-clause. Une expression de la forme $f(n) \simeq a$ par exemple doit être écrite $[f(x) \simeq a \mid n \simeq x]$, où $x$ est une variable de sorte `nat`.

Nous avons vu que dans le cadre de la logique équationnelle, les interprétations pouvaient être définies comme des congruences sur l'ensemble des termes fermés. Dans notre formalisme, nous devons, en plus, spécifier la valeur du paramètre $n$. Celui-ci est interprété comme un entier, i.e., un terme construit sur les constructeurs libres 0 et succ. Nous définissons la notion d'interprétation comme suit :

**Définition 31** Une *interprétation* $\mathcal{I}$ est définie par un couple $(\mathcal{I}(n), \simeq_{\mathcal{I}})$ où $\mathcal{I}(n)$ est un entier naturel (identifié à un terme de la forme $\text{succ}^k(0)$) et $\simeq_{\mathcal{I}}$ est une congruence sur l'ensemble des termes fermés de sorte `term`. $\Diamond$

Nous étendons alors la notion de validité aux $n$-clauses :

**Définition 32** Une interprétation $\mathcal{I}$ *valide* une expression $E$ (notée $\mathcal{I} \models E$) ssi une des propositions suivantes est vraie.

- $E$ est un littéral fermé $t \simeq s$ (resp. $t \not\simeq s$) et $t \simeq_{\mathcal{I}} s$ (resp. $t \not\simeq_{\mathcal{I}} s$).
- $E$ est une clause fermée $\bigvee_{i=1}^{k} l_i$ et il existe $i \in [1, k]$ tel que $\mathcal{I} \models l_i$.
- $E$ est une $n$-clause $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ et pour toute substitution fermée $\sigma$ de domaine $\text{var}(E)$ telle que $\forall i \in [1, k], \mathcal{I}(n) = t_i\sigma$, nous avons $\mathcal{I} \models C\sigma$.
- $E$ est un ensemble de $n$-clauses et $\forall \mathcal{C} \in E$, $\mathcal{I} \models \mathcal{C}$. $\Diamond$

Par définition, $\mathcal{I} \models [\Box \mid n \simeq \text{succ}^k(0)]$ ssi $\mathcal{I}(n) \neq k$. De la même manière, $\mathcal{I} \models [\Box \mid n \simeq \text{succ}^k(x)]$ ssi $\mathcal{I}(n) < k$ (où $x$ est une variable). Par conséquent, une $n$-clause de la forme $[\Box \mid n \simeq \text{succ}^k(0)]$ (resp. $[\Box \mid n \simeq \text{succ}^k(x)]$) sera écrite $n \not\simeq k$ (resp. $n < k$).

La proposition suivante montre que toute $n$-clause qui n'est pas une tautologie est équivalente à une $n$-clause normalisée.

**Proposition 33** *Soit $\mathcal{C} = [C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ une $n$-clause. Si $t_1, \ldots, t_n$ sont unifiable, alors $\mathcal{C}$ est equivalent à $[C\sigma \mid n \simeq t_1\sigma]$, où $\sigma$ est le mgu$(t_1, \ldots, t_n)$. Sinon $\mathcal{C}$ est une tautologie.*

Preuve. Soit $\mathcal{I}$ une interprétation. Par définition $\mathcal{I} \models \mathcal{C}$, ssi $\mathcal{I} \models \mathcal{C}\theta$ pour toute substitution fermée $\theta$. Si $\theta$ n'est pas un unificateur pour $t_1, \ldots, t_k$ alors $t_1\theta, \ldots, t_k\theta$ sont distincts. Donc $\mathcal{I}(n)$ doit être différent de l'un des $t_i\theta$ (avec $1 \leq i \leq k$) et $\mathcal{I} \models \mathcal{C}\theta$. Ce raisonnement prouve que $\mathcal{C}$ est une tautologie si $t_1, \ldots, t_k$ ne sont pas unifiables. De plus, si $t_1, \ldots, t_k$ ont un unificateur $\sigma$, alors $\mathcal{I} \models \mathcal{C}$ ssi $\mathcal{I} \models \mathcal{C}\theta$ pour toute substitution $\theta$ qui est une instance de $\sigma$ c'est-à-dire ssi $\mathcal{I} \models \mathcal{C}\sigma\theta'$, pour toute substitution fermée $\theta'$. Nous avons alors $\mathcal{C}\sigma \equiv [C\sigma \mid \bigwedge_{i=1}^{k} n \simeq t_i\sigma] \equiv [C\sigma \mid n \simeq t_1\sigma]$. Par conséquent $\mathcal{C} \equiv [C\sigma \mid n \simeq t_1\sigma]$.

## 3.2 Indécidabilité

Il est clair que la logique n'est pas décidable car elle contient la logique du premier ordre (donc les ensembles de $n$-clauses satisfaisables ne sont pas récursivement énumérables). Dans le théorème qui suit nous montrons que la logique n'est pas non plus semi-décidable (contrairement à la logique du premier ordre, les ensembles de $n$-clauses insatisfaisables ne sont pas récursivement énumérables).

**Théorème 34** *Le problème d'insatisfaisabilité n'est pas semi-décidable pour les n-clauses contenant une seule variable de sorte* `nat`.

PREUVE. La preuve est par réduction au problème de Post. Soit $u^1, \ldots, u^n$ et $v^1, \ldots, v^n$ deux suites de mots sur un alphabet fini. Le problème de Post consiste à trouver une suite d'indices $i_1, \ldots, i_m$ telle que $u^{i_1} \cdot u^{i_2} \cdot \cdots \cdot u^{i_m} = v^{i_1} \cdot v^{i_2} \cdot \cdots \cdot v^{i_m}$. Nous allons construire un ensemble de $n$-clauses $S$ tel que $S$ est satisfaisable ssi il existe une suite d'indice $i_1, \ldots, i_m$ tel que $u^{i_1} \cdot u^{i_2} \cdot \cdots \cdot u^{i_m} = v^{i_1} \cdot v^{i_2} \cdot \cdots \cdot v^{i_m}$. Nous utilisons une constante $l$ qui va coder la liste d'indices avec les symboles de fonction *head* et *tail* qui retournent respectivement la tête et la queue de la liste. Un mot est codé comme une liste de caractères. Le mot vide est noté $\epsilon$ et *concat* est une fonction de concaténation définie de manière usuelle. Soient $x \in \{u, v\}$ et $y$ la séquence d'indices $i_1, \ldots, i_m$ alors $sol(y, x)$ est le mot $x^{i_1}.\ldots.x^{i_m}$. Les termes $word(u, i)$ et $word(v, i)$ représentent les mots $u^i$ et $v^i$, respectivement. Nous admettons les axiomes suivants

- $\neg empty(x) \vee head(x) \simeq 1 \vee \ldots \vee head(x) \simeq n$ (si une séquence n'est pas vide alors la tête $head(x)$ est dans $[\![1, n]\!]$)
- $word(u, i) \simeq u^i$, pour tout $i \in [\![1, n]\!]$ (définition de $u$).
- $word(v, i) \simeq v^i$, pour tout $i \in [\![1, n[\![$ (définition de $v$).
- $\neg empty(y) \vee sol(y, x) \simeq \epsilon$. Si une séquence $i_1, \ldots, i_m$ est vide alors la solution $x^{i_1}.\ldots.x^{i_m}$ est aussi vide.
- $empty(y) \vee sol(y, x) \simeq concat(word(x, head(y)), sol(tail(y), x))$. Si $y$ n'est pas vide alors $sol(y, x)$ correspond à la concaténation du mot $x^{i_1}$ et la solution correspondant à la séquence $i_2, \ldots, i_m$.
- $word(u, l) \simeq word(v, l)$. Les deux séquences sont égales.

Nous définissons maintenant un prédicat $length(i, l)$ qui va coder le fait que $l$ est de longueur $i$.

- $length(0, x) \vee \neg empty(x)$
- $\neg length(i + 1, x) \vee \neg empty(x)$
- $\neg length(0, x) \vee empty(x)$
- $length(i + 1, x) \vee \neg length(i, tail(x))$
- $\neg length(i + 1, x) \vee length(i, tail(x))$ ∎

Pour finir, les clauses suivantes codent le fait que la constante $l$ est une liste finie de longueur $n \neq 0$. Cette propriété est la seule qui ne peut être exprimée en logique du premier ordre. En son absence, $l$ pourrait correspondre à une liste infinie ou cyclique.

$$[length(x, l) \mid n \simeq x]$$

$$\neg empty(l)$$

## 3.3 Notion de rang

Intuitivement, une $n$-clause $[C \mid \mathcal{X}]$ peut dénoter :
- Soit une clause ordinaire si $\mathcal{X} = \mathtt{V}$.
- Soit une formule de la forme $n = k \Rightarrow C\{x \mapsto k\}$, où $k$ est un entier et $x$ est une variable entière de $C$, si $\mathcal{X}$ est de la forme $n \simeq \mathrm{succ}^k(0)$.
- Soit une formule de la forme $n \geq k \Rightarrow C\{x \mapsto n - k\}$, où $k$ est un entier et $x$ est une variable entière de $C$, si $\mathcal{X}$ est de la forme $n \simeq \mathrm{succ}^k(x)$.

Nous introduisons la notion de *rang* qui est une fonction associant un entier (ou $\bot$) à toute clause dans l'espace de recherche. Cette mesure est croissante avec les inférences (c'est-à-dire le *rang* d'une clause générée est toujours supérieure ou égale au *rang* des clauses parentes). Elle correspond à l'entier $k$ dans les items ci-dessus, modulé par la profondeur des termes entiers apparaissant dans la clause (de manière à ce que deux $n$-clauses $[p(x) \mid n \simeq x]$ et $[p(\mathrm{succ}(x)) \mid n \simeq \mathrm{succ}(x))]$ par exemple aient le même rang, i.e., que le rang soit stable par instanciation).

**Définition 35** Le *rang* d'une $n$-clause $[C \mid n \simeq \mathrm{succ}^i(x)]$ (avec $x \in \mathcal{X} \cup \{0\}$) est $i - p$, où $p$ est la profondeur maximale des termes de type `nat` de la clause $C$. Le rang d'une $n$-clause de la forme $[C \mid \mathtt{V}]$ est $\bot$. ◊

Pour tout ensemble de $n$-clauses $S$ et pour tout entier naturel $i$, on note $S[i]$ l'ensemble de $n$-clauses de $S$ de rang $i$. On note $S[\bot]$ l'ensemble des $n$-clauses où la contrainte est à `V`.

**Exemple 36** Nous présentons dans le tableau suivant le rang pour chaque clause.

| Numéro | la clause | le rang |
|---|---|---|
| 1 | $[q(x, y) \simeq \mathtt{V} \mid n \simeq x]$ | 0 |
| 2 | $[f(\mathrm{succ}(x), y) \simeq a \mid n \simeq \mathrm{succ}^2(x)]$ | 1 |
| 3 | $[p(x) \simeq \mathtt{V} \mid n \simeq x]$ | 0 |
| 4 | $[r(x, y) \simeq \mathtt{V} \mid \mathtt{V}]$ | $\bot$ |

Si $\mathcal{S} = \{1, 2, 3, 4\}$ alors $S[0] = \{1, 3\}$, $S[1] = \{2\}$ et $S[\bot] = \{4\}$. ♣

Le rang nous permet de construire une sorte de hiérarchie dans l'espace de recherche $S$, où les ensembles $S[i]$ représentent chaque ensemble ou niveau de cette hiérarchie.

## 3.4 La superposition pour les $n$-clauses

Le calcul de superposition standard s'adapte de manière très naturelle aux $n$-clauses. Les règles d'inférences sont étendues aux $n$-clauses en raisonnant sur la

partie clausale des clauses et en propageant simplement la partie contrainte des prémisses vers la conclusion.

Soit $<$ un ordre de réduction et *sel* une fonction de sélection.

**Superposition**

$$\frac{[C \vee t \bowtie s \mid \mathcal{X}], [D \vee u \simeq v \mid \mathcal{Y}]}{[C \vee D \vee t[v]_p \bowtie s \mid \mathcal{X} \wedge \mathcal{Y}]\sigma}$$

Si $\bowtie \in \{\simeq, \not\simeq\}$, $\sigma = \text{mgu}(u, t|_p)$, $u\sigma \not< v\sigma, t\sigma \not< s\sigma$, $t|_p$ n'est pas une variable, $(t \simeq s)\sigma \in sel((C \vee t \simeq s)\sigma)$, $(u \simeq v)\sigma \in sel((D \vee u \simeq v)\sigma)$.

**Réflection**

$$\frac{[C \vee t \not\simeq s \mid \mathcal{X}]}{[C \mid \mathcal{X}]\sigma} \qquad \text{Si } \sigma = \text{mgu}(t, s),\ (t \not\simeq s)\sigma \in sel((C \vee t \not\simeq s)\sigma)$$

**Factorisation équationnelle**

$$\frac{[C \vee t \simeq s \vee u \simeq v \mid \mathcal{X}]}{[C \vee s \not\simeq v \vee t \simeq s \mid \mathcal{X}]\sigma}$$

Si $\sigma = \text{mgu}(t, u)$, $t\sigma \not< s\sigma$, $u\sigma \not< v\sigma$, $(t \simeq s)\sigma \in sel((C \vee t \simeq s \vee u \simeq v)\sigma)$.

**Détection de redondance**

Grâce à la proposition 33, nous pouvons considérer dans la suite que toutes les $n$-clauses sont normalisées. Dans la définition qui suit, la relation de subsomption est étendue aux $n$-clauses :

**Définition 37** Soit $\mathcal{C} = [C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ et $\mathcal{C}' = [C' \mid \bigwedge_{i=1}^{l} n \simeq t'_i]$ deux $n$-clauses. $\mathcal{C}$ *subsume* $\mathcal{C}'$ (noté $\mathcal{C} \leq_{sub} \mathcal{C}'$) si il existe une substitution $\sigma$ telle que :
— $C\sigma \subseteq C'$
— $\{t_1, \ldots, t_k\}\sigma \subseteq \{t'_1, \ldots, t'_l\}\sigma$. ◊

**Proposition 38** *Si* $\mathcal{C} \leq_{sub} \mathcal{C}'$ *alors* $\mathcal{C} \models \mathcal{C}'$.

PREUVE. Soit $\mathcal{I}$ une interprétation telle que $\mathcal{I} \models \mathcal{C}$, avec $\mathcal{I}(n) = n_C$. Si $\mathcal{C} \leq_{sub} \mathcal{C}'$ alors il existe une substitution $\sigma$ telle que :

1. $C\sigma \subseteq C'$.

2. $\{t_1, \ldots, t_k\}\sigma \subseteq \{t'_1, \ldots, t'_l\}\sigma$. ∎

Soit $\theta$ une substitution fermée. S'il existe $i \in [\![1, l]\!]$ tel que $t'_i\theta \neq n_C$, alors on a $\mathcal{I} \models C'\theta$, par définition. Sinon, par 2, on doit avoir $t_i\sigma\theta \neq n_C$, pour tout $i \in [\![1, k]\!]$. Puisque $\mathcal{I} \models \mathcal{C}$, on en déduit que $\mathcal{I} \models C\sigma\theta$, donc par 1 que $I \models C'\theta$. D'où $I \models \mathcal{C}'$.

La relation de subsomption $\leq_{sub}$ est étendue aux ensembles de $n$-clauses comme suit :

**Définition 39** $S \leq_{sub} S'$ si pour toute clause $C' \in S'$, il existe une clause $C \in S$ telle que : $C \leq_{sub} C'$. ◊

Nous présentons maintenant un exemple d'application du calcul de superposition défini avec les règles ci-dessus.

**Exemple 40** Soit l'ensemble de $n$-clauses suivant :

$$\begin{array}{ll} 1 & p(0, a) \simeq \mathtt{V} \\ 2 & p(x, y) \not\simeq \mathtt{V} \vee p(\mathrm{succ}(x), f(y)) \simeq \mathtt{V} \\ 3 & [q(x, y) \not\simeq \mathtt{V} \mid n \simeq x] \\ 4 & q(x, y) \simeq \mathtt{V} \vee p(x, y) \not\simeq \mathtt{V} \end{array}$$

Cet ensemble de $n$-clauses est obtenu à partir de l'exemple présenté. En appliquant le calcul de superposition, nous obtenons les $n$-clauses suivantes (pour plus de clarté les littéraux de la forme $\mathtt{V} \not\simeq \mathtt{V}$ sont supprimés des clauses) :

$$\begin{array}{lll} 5 & [p(x, y) \not\simeq \mathtt{V} \mid n \simeq x] & \text{(superposition, 4, 3)} \\ 6 & [\Box \mid n \simeq 0] & \text{(superposition, 1, 5)} \\ 7 & [p(x, y) \not\simeq \mathtt{V} \mid n \simeq \mathrm{succ}(x)] & \text{(superposition, 2, 5)} \\ 8 & [\Box \mid n \simeq \mathrm{succ}(0)] & \text{(superposition, 1, 7)} \\ 9 & [p(x, y) \not\simeq \mathtt{V} \mid n \simeq \mathrm{succ}(\mathrm{succ}(x))] & \text{(superposition, 2, 7)} \\ 10 & \ldots & \end{array}$$

Une infinité de $n$-clauses de la forme $[\Box \mid n \simeq \mathrm{succ}^k(0)]$ (i.e. $n \not\simeq k$) sont générées, mais aucune contradiction ne peut être obtenue en un temps fini. ♣

La procédure ainsi définie n'est pas complète en général (sauf dans certains cas où par exemple le paramètre $n$ ne joue aucun rôle dans la preuve).

Afin d'accroître les capacités de la procédure, nous allons introduire une règle permettant de détecter des "cycles" dans l'espace de recherche et d'utiliser ces cycles pour générer une borne sur la valeur du paramètre entier. Dans l'exemple ci-dessus, nous pouvons remarquer sur l'exemple que la $n$-clause 9 est "presque" égale à la $n$-clause 7, au sens où les deux clauses expriment des propriétés identiques pour différentes valeurs du paramètre $n$ (les $n$-clauses ont la même partie clausale). Par conséquent, toutes les inférences réalisables sur la clause 7 sont aussi réalisables à partir de 9, ce qui entraîne nécessairement la divergence.

## 3.5 La détection de boucles

Nous définissons formellement la notion de boucle qui permet de simuler une forme d'induction noethérienne dans le calcul de superposition (par descente infinie). Nous allons, en premier lieu, définir une notion très importante pour la suite qui est le *décalage* (ou *shift*) :

**Définition 41** Pour tout entier naturel $i$ et pour toute $n$-clause $\mathcal{C} = [C \mid \bigwedge_{j=1}^{k} n \simeq t_j]$, le *décalage* de degré $i$ de la clause $\mathcal{C}$, noté $\mathcal{C}\downarrow_i$, est la $n$-clause $[C \mid \bigwedge_{j=1}^{k} n \simeq \mathrm{succ}^i(t_j)]$. Si $S$ est un ensemble de $n$-clauses alors $S\downarrow_i \stackrel{\text{def}}{=} \{\mathcal{C}\downarrow_i \mid \mathcal{C} \in S\}$. ◊

**Exemple 42** Soit $\mathcal{C} = [C \mid n \simeq x]$, $\mathcal{C}' = [C' \mid n \simeq \mathrm{succ}^2(x)]$ et $\mathcal{C}'' = [C'' \mid \mathtt{V}]$.
- $\mathcal{C}\downarrow_2 = [C \mid n \simeq \mathrm{succ}^2(x)]$
- $\mathcal{C}'\downarrow_1 = [C' \mid n \simeq \mathrm{succ}^3(x)]$
- $\mathcal{C}''\downarrow_3 = C''$.

Notez que si la partie contrainte est à `V` alors le décalage de la $n$-clause est la $n$-clause elle même (identité). ♣

Intuitivement $S\downarrow_i$ correspond à $S$ où $n$ est remplacé par $n - i$, d'où la proposition suivante :

**Proposition 43** *Soit $i,j \in \mathbb{N}$, soit $S$ un ensemble de $n$-clauses et soit $\mathcal{I}$ une interprétation. Si $\mathcal{I} \models S\downarrow_j$ et $\mathcal{I}(n) = i + j$ alors $\mathcal{I}_{\{n \leftarrow i\}} \models S$.* [1]

Preuve. Soit $\mathcal{C} = [C \mid \bigwedge_{l=1}^{k} n \simeq t_l] \in S$. Soit $\sigma$ une substitution telle que $\forall l \in [\![1, k]\!]$ $\mathcal{I}_{\{n \leftarrow i\}}(n) = t_l\sigma$. Cela implique que $\forall l \in [\![1, k]\!], t_l\sigma = i$ car par définition $\mathcal{I}_{\{n \leftarrow i\}}(n) = i$. Nous avons $\mathcal{C}\downarrow_j = [C \mid \bigwedge_{l=1}^{k} n \simeq \mathrm{succ}^j(t_l)]$. Comme $t_l\sigma = i$ et $\mathcal{I}(n) = i + j$, nous avons $\mathcal{I}(n) = t_l\sigma$, $\forall l \in [\![1, k]\!]$. Puisque $\mathcal{I} \models S\downarrow_j$, nous déduisons que $\mathcal{I} \models C\sigma$ et donc $\mathcal{I}_{\{n \leftarrow i\}} \models C\sigma$(car $C\sigma$ ne contient aucune occurrence de $n$). ■

**Définition 44** Soit $S$ un ensemble de $n$-clauses. Un couple d'entiers naturels $(i, j)$ (où $j \neq 0$) est une *boucle inductive* pour $S$ s'il existe un ensemble $S' \subseteq S$ tel que $S' \models n \not\simeq l$, pour tout $l \in [\![i, i + j[\![$ et $S' \models S'\downarrow_j$. ◊

**Théorème 45** *Si $(i, j)$ est une boucle inductive pour $S$ alors $S \models n < i$.*

Preuve. Soit $S' \subseteq S$, nous avons $\mathcal{I} \models S'$. soit $k$ l'entier naturel minimal supérieur ou égal à $i$ tel que $S'$ a un modèle $\mathcal{I}$ avec $\mathcal{I}(n) = k$. Si $k < i + j$ alors $S' \models n \not\simeq k$ ce qui n'est pas possible car, par définition, $\mathcal{I} \not\models n \not\simeq k$. Sinon ($k \geq i + k$), nous avons $\mathcal{I} \models S'\downarrow_j$, d'après la proposition 3.5, nous pouvons déduire que $\mathcal{I}_{\{n \leftarrow (k-j)\}} \models S'$, ce qui est impossible aussi car $i \leq k - j < k$ et $k$ est supposé minimal. ■

1. La notation $\mathcal{I}_{\{n \leftarrow i\}}$ est définie dans la section 2.1 des préliminaires.

## 3.6 La détection de boucles comme une règle d'inférence

Le théorème 45 nous permet de générer une nouvelle clause qui réduit l'espace de recherche, nous allons donc l'utiliser comme une règle d'inférence. Pour cela nous devons vérifier les conditions définies dans la définition 44, or il est clair que ces conditions sont indécidables. Nous devons donc transformer ces conditions sémantiques en conditions syntaxiques. Nous introduisons en premier lieu quelques notations qui nous permettent de définir ces conditions d'une manière très abstraite.

**Définition 46** Soit $S$ un ensemble de $n$-clauses. Une *relation d'inférence* $\delta$ pour $S$ est une fonction partielle associant à chaque $n$-clause $\mathcal{C} \in S$ un ensemble de $n$-clauses $\delta(\mathcal{C}) \in S$ tel que $\mathcal{C}$ est déductible de $\delta(S)$ par application d'une des règles d'inférence du calcul (en une étape exactement). $\Diamond$

Cette relation ainsi définie, représente les inférences obtenues avec le calcul de superposition défini dans la section 3.4, c'est-à-dire $\mathcal{D} \in \delta(\mathcal{C})$ ssi $\mathcal{D}$ est un parent de $\mathcal{C}$. Les $n$-clauses $\mathcal{C}$ telles que $\delta(\mathcal{C})$ n'est pas définie sont des hypothèses ($\mathcal{C} \in S_0$ où $S_0$ est l'ensemble initial qu'on veut réfuter). Cette relation d'inférence va induire une relation d'implication $\vdash_\delta$ entre sous-ensembles de $S$. On note $S' \vdash_\delta S''$ si toutes les $n$-clauses de $S''$ peuvent être dérivées de $S' \cup S[\bot]$. Formellement, la relation $\vdash_\delta$ est définie comme suit :

**Définition 47** Soit $S$ un ensemble de $n$-clauses et soit $\delta$ une relation d'inférence pour $S$. La relation $\vdash_\delta$ est la plus petite relation entre sous ensembles de $S$ tel que $S' \vdash_\delta S''$ si pour tout $n$-clause $\mathcal{C} \in S''$, une de ces conditions est satisfaite :

1. $\mathcal{C} \in S'$
2. $\delta(\mathcal{C})$ est définie et $S' \vdash_\delta \delta(\mathcal{C})$.
3. La partie contrainte de $\mathcal{C}$ est à `V`. $\Diamond$

Les deux premières conditions sont naturelles. L'idée de la dernière condition est de considérer les $n$-clauses dont la contrainte est à `V`, comme des propriétés universelles, qui sont donc valide indépendamment du paramètre $n$. Pour cette raison il est possible de supposer que ces $n$-clauses, si elles sont générées par le calcul, peuvent être utilisées comme hypothèse dans n'importe quelle dérivation.

**Proposition 48** *Soit $S$ un ensemble de $n$-clauses et soit $\delta$ une relation d'inférence pour $S$. Si $S' \vdash_\delta S''$ alors $S' \cup S[\bot] \models S''$.*

PREUVE. Pour toute clause $C \in S''$ nous avons $S' \vdash_\delta C$. Nous raisonnons par induction sur la longueur de la dérivation, et distinguons 3 cas :

1. $C \in S'$ et dans ce cas $S' \models C$.
2. La partie contrainte de $C$ est à $\mathtt{V}$ et dans ce cas $C \in S[\bot]$ donc $S[\bot] \models C$.
3. $\delta(\mathcal{C})$ est définie et $S' \vdash_\delta \delta(\mathcal{C})$. Par induction nous avons $S' \cup S[\bot] \models \delta(\mathcal{C})$, donc par transitivité $S' \cup S[\bot] \models C$. ∎

Pour tester la conséquence logique entre deux sous ensembles nous allons introduire la notion de relation d'implication immédiate.

**Définition 49** Une *relation d'implication immédiate* $\sqsupseteq$ est une relation décidable entre $n$-clauses telle que $\mathcal{C} \sqsupseteq \mathcal{D} \Rightarrow \mathcal{C} \models \mathcal{D}$. La relation $\sqsupseteq$ est étendue aux ensembles de $n$-clauses comme suit : $S \sqsupseteq S'$ ssi $\forall \mathcal{C}' \in S' \, \exists C \in S, \mathcal{C} \sqsupseteq \mathcal{C}'$. ◊

La relation $\sqsupseteq$ peut être, par exemple, l'identité (c'est-à-dire $\mathcal{C} = \mathcal{D}$ à un renommage près), l'inclusion ($\mathcal{C} \subset \mathcal{D}$) où encore la subsomption $\leq_{sub}$. Nous allons maintenant définir la notion de *cycle* qui correspond au pendant syntaxique de la notion de boucle inductive introduite dans la définition 44 (les conditions sont cette fois-ci décidables). Un cycle est paramétré par les relations $\sqsupseteq$ et $\delta$ ($\sqsupseteq$ étant choisie a priori, par exemple par un utilisateur et $\delta$ étant donné par le graphe d'inférence au moment où la règle est appliquée).

**Définition 50** Soit $S$ un ensemble de $n$-clauses et soit $\delta$ une relation d'inférence pour $S$. Soit $\sqsupseteq$ une relation d'implication immédiate. Un couple d'entiers naturels $(i, j)$ (avec $j \neq 0$) est un *cycle* pour $S$ par rapport à $\sqsupseteq$ et $\delta$ si il existe $S_{init}, S_{loop} \subseteq S$ tel que

— $S_{init} \subseteq S[i]$,
— $S_{loop} \subseteq S[i + j]$,
— $S_{init} \vdash_\delta \{n \not\simeq k \mid k \in [\![i, i + j - 1]\!]\}$,
— $S_{init} \vdash_\delta S_{loop}$
— et $S_{loop} \sqsupseteq S_{init}\downarrow_j$. ◊

**Exemple 51** On considère l'exemple 40. Supposons que $\sqsupseteq$ est la relation d'identité. Le couple $(0, 1)$ est un cycle, avec $S_{init} = \{5\}$ et $S_{loop} = \{7\}$. En effet, la clause $n \not\simeq 0$ est dérivable de la clause 5 et les clauses ne contenant pas $n$, par conséquent $S_{init} \vdash_\delta \{n \not\simeq 0\}$. De la même manière, la clause 7 peut être dérivée de la clause 5, avec les clauses ne contenant pas $n$.

Pour finir, nous avons $[p(x, y) \not\simeq \mathtt{V} \mid n \simeq x]\downarrow_1 = [p(x, y) \not\simeq \mathtt{V} \mid n \simeq \mathrm{succ}(x)]$, donc $S_{loop} \sqsupseteq S_{init}\downarrow_1$ (si on suppose que $\sqsupseteq$ contient la relation d'identité). De la même manière, $(0, 2)$ et $(1, 2)$ sont des cycles, correspondant aux ensembles $\{5\}$, $\{9\}$ et $\{7\}$, $\{9\}$ respectivement. ♣

Nous présentons un autre exemple un peu plus complexe qui vient de la théorie des tableaux :

**Exemple 52** Soit $T$ un tableau et soit $a, b_1, \ldots, b_n$ ses indices, tel que $\forall i\ a \neq b_i$. Nous considérons un tableau $T'$ obtenu de $T$ en remplaçant successivement la valeur des cellules $b_i$ par $c_i$. Nous voulons montrer que $T[a] = T'[a]$. Nous formalisons ce problème par l'ensemble des $n$-clauses suivant. Nous définissons tout d'abord une séquence de tableaux $T_i$ avec les clauses :

$$(1) \quad T(0) \simeq T$$

$$(2) \quad T(i+1) \simeq \mathit{store}(T(i), b(i), c(i))$$

Nous avons l'axiome

$$(3) \quad b(i) \not\simeq a$$

Nous considérons aussi les deux axiomes de la théorie des tableaux (voir par exemple [10])

$$(4) \quad \mathit{select}(\mathit{store}(t, x, v), x) \simeq v$$

$$(5) \quad x \simeq y \vee \mathit{select}(\mathit{store}(t, x, v), y) \simeq \mathit{select}(t, y)$$

L'inéquation $T(n)[a] \neq T[a]$ est définie comme suit :

$$(6) \quad \mathit{select}(T, a) \simeq d$$

$$(7) \quad [\mathit{select}(T(i), a) \not\simeq d \mid n \simeq i]$$

Nous appliquons le calcul de superposition et nous obtenons les $n$-clauses suivantes :

| | | |
|---|---|---|
| (8) | $[\mathit{select}(T, a) \not\simeq d \mid n \simeq 0]$ | (1,7) |
| (9) | $[\Box \mid \alpha \simeq 0]$ | (6,8) |
| (10) | $y \simeq b(i) \vee \mathit{select}(T(i+1), y) \simeq \mathit{select}(T(i), y)$ | (2,5) |
| (11) | $[a \simeq b(i) \vee \mathit{select}(T(i), a) \not\simeq d \mid n \simeq i+1]$ | (10,7) |
| (12) | $[a \simeq b(i) \vee \mathit{select}(T, a) \not\simeq d \mid n \simeq i+1]$ | (1,11) |
| (13) | $[a \simeq b(0) \mid n \simeq 1]$ | (6,12) |
| (14) | $[\Box \mid n \simeq 1]$ | (13,3) |
| (15) | $[a \simeq b(i) \vee \mathit{select}(T(i), a) \not\simeq d \mid n \simeq i+2]$ | (10,11) |

♣

Nous considérons les ensembles $S_{init} = \{11\}$ et $S_{loop} = \{15\}$. On vérifie aisément que $S_{init} \vdash_\delta S_{loop}$, $S_{loop} = S_{init}{\downarrow_1}$ et $S_{init} \vdash_\delta \alpha \not\simeq \mathrm{succ}(0)$, la règle de détection de boucles s'applique et nous pouvons générer la clause (16) $n \not\simeq i+1$. Les clauses 9 et 16 ensemble impliquent que l'ensemble de clause initial est insatisfaisable.

**Théorème 53** *Tous les cycles sont des boucles inductives.*

PREUVE. Soit $(i, j)$ un cycle pour $S$, par rapport aux relations $\sqsupseteq$ et $\delta$. Soit $S_{init}$ et $S_{loop}$ les deux sous ensembles de $S$ correspondant. Soit $S' = S_{init} \cup S[\bot]$. Nous avons $S_{init} \vdash_\delta \{n \not\simeq k\}$, pour tout $k \in [i, i + j - 1]$, donc par la proposition 48, $S' \models n \not\simeq k$ (pour tout $k \in [i, i + j - 1]$). De la même manière, nous avons $S' \models S_{loop}$, donc $S' \models S_{init}\downarrow_j$ (comme $\sqsupseteq$ est incluse dans $\models$). De plus on a $S'\downarrow_j = S_{init}\downarrow_j \cup S[\bot]\downarrow_j = S_{init}\downarrow_j \cup S[\bot]$ (puisque les $n$-clauses dont la contrainte est à $\mathtt{V}$ ne sont pas affectées par l'opération $S\downarrow_j$). Nous pouvons conclure que $S' \models S'\downarrow_j$ et que $(i, j)$ est bien une boucle inductive. ∎

D'après le théorème 45 nous pouvons ajouter une règle d'inférence que nous appelons *règle d'élimination de cycles*, elle est définie comme suit :

$$\text{Règle d'élimination des cycles :} \qquad \frac{S}{n < i}$$

Où $(i, j)$ est un cycle pour $S$ par rapport à une relation d'implication immédiate $\sqsupseteq$, et une relation d'inférence $\delta$ induite par les inférences appliquées à $S$. Le théorème 53 assure la correction de la règle d'élimination des cycles.

## 3.7 Détection de cycles et algorithme du point fixe

Nous présentons maintenant des algorithmes permettant de trouver un cycle et les deux sous-ensembles $S_{init}$ et $S_{loop}$ correspondant. Nous présentons deux algorithmes fondés sur un calcul de point fixe : $\text{CYCLE}_1$ et $\text{CYCLE}_2$. L'algorithme $\text{CYCLE}_1$ est le plus intuitif et il est basé sur la recherche d'un plus petit point fixe : il commence par considérer le plus petit ensemble possible $S_{init}$ c'est-à-dire l'ensemble de $n$-clauses dans $S[i]$ qui sont nécessaires pour dériver toutes les clauses $n \not\simeq k$ pour $k \in [\![i, i + j[\![$. D'après la définition 50, la condition $S_{loop} \sqsupseteq S_{init}\downarrow_j$ doit être vérifiée. Donc pour chaque clause $\mathcal{C}$ dans $S_{init}$, l'algorithme vérifie s'il existe une $n$-clause $\mathcal{D}$ dans $S[i+j]$ telle que $\mathcal{D} \sqsupseteq \mathcal{C}\downarrow_j$. Si ce n'est pas le cas, alors $(i, j)$ ne peut être un cycle et l'algorithme s'arrête. Sinon, tous les ancêtres de $\mathcal{D}$ apparaissant dans $S[i]$ sont ajoutés à $S_{init}$, de manière à ce que la condition $S_{init} \vdash_\delta S_{loop}$ (dans la définition 50) soit vérifiée. L'algorithme continue jusqu'à obtenir un point fixe, correspondant à un couple d'ensembles de $n$-clauses $(S_{init}, S_{loop})$ vérifiant les conditions de la définition 50.

Cet algorithme a l'inconvénient d'être non-déterministe. En effet, pour une $n$-clause $\mathcal{C}$ donnée, il peut y avoir plusieurs $n$-clauses $\mathcal{D}$ qui vérifient la condition désirée (sauf si $\sqsupseteq$ est l'identité). De la même manière, les ancêtres de $\mathcal{D}$ dans $S[i]$ ne sont pas uniques en général. Pour assurer la complétude toutes ces branches doivent être explorées, ce qui nécessite un nombre exponentiel de tests d'implication immédiate (même si le nombre d'étapes d'itération est polynomial par rapport à la taille de $S$).

**Algorithm 1** $\textsc{Cycle}_1(S, i, j)$

$S_0 \leftarrow \{n \not\simeq k, k \in [i, i+j[\}$
**if** $S[i] \not\vdash_\delta S_0$ **then**
  **return** F
**end if**
Choose a minimal set $S_{init} \subseteq S[i]$ s.t. $S_{init} \vdash_\delta S_0$
$S_{loop} \leftarrow \emptyset$
**while** $\exists \mathcal{C} \in S_{init} \mid S_{loop} \not\sqsupseteq \{\mathcal{C}\downarrow_j\}$ **do**
  **if** $\exists \mathcal{D} \in S[i+j] \mid \mathcal{D} \sqsupseteq \mathcal{C}\downarrow_j$ **then**
    Choose $\mathcal{D} \in S[i+j]$ such that $\mathcal{D} \sqsupseteq \mathcal{C}\downarrow_j$
    $S_{loop} \leftarrow S_{loop} \cup \{\mathcal{D}\}$
    **if** $S[i] \not\vdash_\delta \mathcal{D}$ **then**
      **return** F
    **else**
      Choose a set $S' \in S[i]$ such that $S' \vdash_\delta \{\mathcal{D}\}$
      $S_{init} \leftarrow S_{init} \cup S'$
    **end if**
  **else**
    **return** F
  **end if**
**end while**
**return** V

Pour pallier ce problème, nous proposons un second algorithme, $\textsc{Cycle}_2$, un peu plus complexe, fondé sur une recherche d'un plus grand point fixe. L'idée est de commencer par ajouter à $S_{init}$ toutes les $n$-clauses dans $S[i]$. Ensuite $S_{loop}$ est obtenu en considérant toutes les $n$-clauses dans $S[i+j]$ qui sont générées à partir de $S_{init}$. Afin d'assurer que la condition $S_{loop} \sqsupseteq S_{init}\downarrow_j$ de la définition 50 est vérifiée, nous supprimons dans $S_{init}$ les $n$-clauses $\mathcal{C}$ telles qu'il n'y a aucune $n$-clause $\mathcal{D}$ dans $S[i+j]$ avec $\mathcal{D} \sqsupseteq \mathcal{C}\downarrow_j$. Si la $n$-clause supprimée $\mathcal{C}$ est un ancêtre d'une clause $n \not\simeq k$ pour tout $i \in [\![i, i+j[\![$ alors $(i, j)$ n'est pas un cycle et l'algorithme s'arrête. Sinon, comme $\mathcal{C}$ est supprimée de $S_{init}$, nous supprimons toutes les $n$-clauses de $S_{loop}$ qui sont générées à partir de cette clause (afin de préserver l'invariant $S_{init} \vdash_\delta S_{loop}$). Cet algorithme est déterministe et donc utilise un nombre polynomial de tests d'implications immédiate (car le nombre d'itérations est borné de façon polynomiale par la taille de l'ensemble initial $S$). La complexité réelle de l'algorithme dépend de la relation $\sqsupseteq$ : elle est polynomiale si $\sqsupseteq$ peut être testée en un temps polynomial (par exemple si $\sqsupseteq$ est l'identité ou l'inclusion). Si $\sqsupseteq$ est la relation de subsomption alors elle est exponentielle (en effet le test de subsomption est NP-complet en logique du premier ordre [40]). L'inconvénient principal de $\textsc{Cycle}_2$ par

rapport à l'algorithme précédent CYCLE$_1$ est qu'il gère des ensembles de clauses bien plus grand car il commence par traiter l'ensemble entier des $n$-clauses de rang $i$. L'algorithme CYCLE$_1$ peut donc s'avérer préférable si les ensembles de clauses sont très grands ou si la relation d'implication immédiate est l'identité.

**Algorithm 2** CYCLE$_2(S, i, j)$

$S_0 \leftarrow \{n \not\simeq k, k \in [i, i+j[\}$
$S_{init} \leftarrow S[i]$
**if** $S_{init} \not\vdash_\delta S_0$ **then**
  **return** F
**end if**
$S_{loop} \leftarrow \{\mathcal{D} \in S[i+j] \mid S_{init} \vdash_\delta \{\mathcal{D}\}\}$
**while** $\exists \mathcal{C} \in S_{init} \mid S_{loop} \not\sqsupseteq \{\mathcal{C}{\downarrow_j}\}$ **do**
  $S_{init} \leftarrow S_{init} \setminus \{\mathcal{C}\}$
  **if** $S_{init} \not\vdash_\delta S_0$ **then**
    **return** F
  **end if**
  Remove from $S_{loop}$ all the $n$-clauses $\mathcal{D}$ s.t. $S_{init} \not\vdash_\delta \{\mathcal{D}\}$
**end while**
**return** V

**Théorème 54** *Les algorithmes* CYCLE$_1$ *et* CYCLE$_2$ *terminent, sont corrects et complets c'est-à-dire* CYCLE$_1(S, i, j)$ = V *ssi* CYCLE$_2(S, i, j)$ = V *ssi* $(i, j)$ *est un cycle pour* $S$ *(par rapport aux relation* $\sqsupseteq$ *et* $\delta$*)*

PREUVE. Nous traitons les deux algorithmes séparément :

**L'algorithme** CYCLE$_1$ **:** Montrer la terminaison est trivial car à chaque itération de l'algorithme la taille de $S_{init}$ décroit strictement (la taille est bornée par la taille de l'ensemble total de $n$-clauses). Si $S[i] \not\models \{n \not\simeq k, k \in [\![i, i+j[\![\}$ alors, d'après la définition 50, $(i, j)$ ne peut être un cycle. Sinon, d'après la même définition 50, l'ensemble $S_{init}$ doit contenir l'ensemble de $n$-clauses impliquant $\{n \not\simeq k, k \in [\![i, i+j[\![\}$ (par rapport à $\vdash_\delta$). La dernière condition de la boucle assure que : $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$. De plus, l'invariant $S_{init} \vdash_\delta S_{loop}$ est nécessairement vérifié, car chaque fois qu'une clause $\mathcal{D}$ est ajoutée à $S_{loop}$, un ensemble $S'$ tel que $S' \vdash_\delta \{\mathcal{D}\}$ est ajouté à $S_{init}$. Finalement, toutes les $n$-clauses qui sont ajoutées dans $S_{init}$ dans la boucle sont dans $S[i]$, par conséquent l'invariant $S_{init} \subseteq S[i]$ est vérifié et après la boucle, toutes les conditions de la définition 50 sont vérifiées, et donc $(i, j)$ est un cycle.
Réciproquement, si $(i, j)$ est un cycle, alors il est facile à vérifier qu'il existe une exécution de l'algorithme CYCLE$_1$ où la valeur F n'est jamais retournée. Il suffit de

choisir les clauses $\mathcal{D}$ et $S'$ de façon à ce que $\mathcal{D} \in S_{loop}$ et $S' \subseteq S_{init}$ (où $S_{loop}$ et $S_{init}$ correspondent bien aux ensembles de la définition 50). Il est toujours possible de trouver $\mathcal{D}$ et $S'$ car par définition $S_{init} \vdash_\delta S_{loop}$ et $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$, avec $S_{init} \subseteq S[i]$ et $S_{loop} \subseteq S[i+j]$.

**L'algorithme** CYCLE$_2$ **:** La terminaison est aussi immédiate car à chaque itération la taille de $S_{init}$ décroît strictement. Cette fois encore, si $S[i] \not\models \{n \not\simeq k, k \in [\![i, i+j[\![\}$ alors d'après la définition 50, $(i, j)$ ne peut être un cycle. Sinon, nous devons avoir $S_{init} \subseteq S[i]$ et $S_{loop} \subseteq S[i+j]$. Comme dans le cas de l'algorithme précédent, la dernière condition de la boucle assure que $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$. De plus, l'invariant $S_{init} \vdash_\delta S_{loop}$ est encore vérifié, d'après la dernière instruction de la boucle "while". Par conséquent, après la boucle, toutes les conditions de la définition 50 sont satisfaites, et donc $(i, j)$ est un cycle.
Réciproquement, si $(i, j)$ est un cycle alors l'algorithme retourne `V`. En effet, les ensembles $S'_{init}$ et $S'_{loop}$ correspondant à la définition 50 apparaissent, nécessairement, à chaque itération dans les ensembles $S_{init}$ et $S_{loop}$ obtenus avec l'algorithme. Par définition, aucune $n$-clause $\mathcal{C} \in S'_{init}$ ne peut être supprimée de $S_{init}$, car nous avons $S_{loop} \sqsupseteq \{\mathcal{C}{\downarrow_j}\}$. De la même manière, aucune $n$-clause $\mathcal{D} \in S'_{loop}$ ne peut être supprimée de $S_{init}$, car nous avons aussi $S_{init} \vdash_\delta \{\mathcal{D}\}$. De même, la condition $S_{init} \not\vdash_\delta S_0$ n'est jamais vraie (car $S'_{init} \vdash_\delta S_0$). ∎

Dans l'exemple suivant, nous allons dérouler les deux algorithmes CYCLE$_1$ et CYCLE$_2$, pour illustrer leur fonctionnement :

**Exemple 55** Nous considérons l'ensemble de clauses suivant :

$$
\begin{array}{ll}
1 & p(x) \not\simeq \mathtt{V} \vee p(\mathrm{succ}(x)) \simeq \mathtt{V} \\
2 & q(x) \simeq \mathtt{V} \vee p(\mathrm{succ}(x)) \simeq \mathtt{V} \\
3 & f(\mathrm{succ}(x)) \simeq f(x) \\
4 & p(0) \simeq \mathtt{V} \\
5 & [p(x) \not\simeq \mathtt{V} \mid n \simeq x] \\
6 & [f(x) \simeq a \mid n \simeq x] \\
7 & [g(x) \simeq a \mid n \simeq x]
\end{array}
$$

Les clauses suivantes sont dérivées :

$$
\begin{array}{lll}
8 & [\Box \mid n \simeq 0] & \text{(superposition, 4, 5)} \\
9 & [p(x) \not\simeq \mathtt{V} \mid n \simeq \mathrm{succ}(x)] & \text{(superposition, 1,5)} \\
10 & [f(x) \simeq a \mid n \simeq \mathrm{succ}(x)] & \text{(superposition, 3,6)} \\
11 & [q(x) \simeq \mathtt{V} \mid n \simeq \mathrm{succ}(x)] & \text{(superposition, 2,5)}
\end{array}
$$
♣

Nous illustrons comment les deux algorithmes s'exécutent sur cet exemple. Notez que les clauses $1, 4, 5$ sont suffisantes pour montrer l'insatisfaisabilité (la clause 5 établit que $p(n)$ est faux pour un certain entier naturel $n$, et cela est impossible

car les clauses 1 et 4 impliquent $p(\text{succ}^k(0))$ pour tout $k \in \mathbb{N}$), les autres clauses sont uniquement ajoutées pour montrer le fonctionnement de l'algorithme. Nous considérons $i = 0$, $j = 1$ et l'identité comme conséquence immédiate $\sqsupseteq$. Nous avons $S_0 = \{8\}$, $S[i] = \{5, 6, 7\}$, $S[i + j] = \{9, 10, 11\}$, $S[\bot] = \{1, 2, 3, 4\}$. Pour simplifier les notations, le décalage de la clause numéro $K$, où $K \in [\![1, 11]\!]$ est noté $K{\downarrow_j}$ $(\forall j \in \mathbb{N})$.

**L'algorithme** CYCLE$_1$ **:** L'algorithme construit, en premier lieu, un ensemble de clauses $S_{init} \subseteq S[i]$ tel que $S_{init} \vdash_\delta S_0$. Les parents de la clause 8 sont $4 \in S[\bot]$ et $5 \in S[i]$, alors d'après la définition 47 nous avons $\{5\} \vdash_\delta \{8\}$, et donc $S_{init}$ prend pour valeur $\{5\}$. $S_{loop}$ est initialisé à $\emptyset$. Comme la clause 5 apparaît dans $S_{init}$ et $5{\downarrow_1} \notin S_{loop}$ (car $S_{loop} = \emptyset$), l'algorithme choisit une clause $\mathcal{D} \in S[i + j]$ telle que $\mathcal{D} \sqsupseteq 5{\downarrow_1}$. Nous avons $5{\downarrow_1} \stackrel{\text{def}}{=} [p(x) \not\simeq \mathtt{V} \mid n \simeq \text{succ}(x)]$, donc $5{\downarrow_1} = 9$. Pour cela la seule solution est $\mathcal{D} = 9$. Cette clause est alors ajoutée à $S_{loop}$. L'algorithme vérifie que $S[i] \vdash_\delta \mathcal{D}$ et ajoute à $S_{init}$ l'ensemble de clauses minimal $S'$ tel que $S' \vdash_\delta \{\mathcal{D}\}$. Comme la clause 9 est déduite des clauses 1 (qui apparaît dans $S[\bot]$) et 5 (qui apparaît dans $S[i]$), la seule solution est de prendre $S' = \{5\}$. Donc $S_{init}$ n'est pas affecté (car il contient déjà 5). La boucle "while" s'arrête car la seule clause dans $S_{init}{\downarrow_1}$ apparaît dans $S_{loop}$. L'algorithme termine en retournant $\mathtt{V}$, et donc le couple $(0, 1)$ est bien un cycle correspondant aux ensembles $S_{init} = \{5\}$ et $S_{loop} = \{9\}$.

**Algorithm** CYCLE$_2$ **:** $S_{init}$ est initialisé à $S[i]$, c'est-à-dire $\{5, 6, 7\}$. Ensuite, l'algorithme vérifie que $S_{init} \vdash_\delta S_0$, et initialise $S_{loop}$ avec l'ensemble des clauses $\mathcal{D} \in S[i + j]$ telles que $S_{init} \vdash_\delta \mathcal{D}$. Toutes les clauses dans $S[i + j]$ sont obtenues à partir de clauses dans $S_{init}$ et $S[\bot]$, par conséquent nous avons $S_{loop} = \{9, 10, 11\}$. Puis, l'algorithme s'assure que $S_{init}$ contient une clause $\mathcal{C}$ telle que $\mathcal{C}{\downarrow_j} \notin S_{loop}$, et la seule clause vérifiant cette condition est la clause 7. Donc cette clause est supprimée de $S_{init}$, qui est donc réduit à $\{5, 6\}$, et les clauses $\mathcal{D}$ telles que $\{5, 6\} \not\vdash_\delta \mathcal{D}$ sont supprimées de $S_{loop}$. Ici toutes les clauses dans $S_{loop}$ peuvent être déduites de $\{5, 6\} \cup S[\bot]$. Donc $S_{loop}$ n'est pas affecté et l'algorithme s'arrête et retourne $\mathtt{V}$. Les ensembles de clauses correspondant au cycle sont $S_{init} = \{5, 6\}$ et $S_{loop} = \{9, 10\}$. Notez que, comparé au cas précédent, les ensembles du cycle contiennent chacun une clause additionnelle (clauses 6 et 10), qui apparaît dans l'invariant inductif, mais ne joue aucun rôle dans la preuve (ultérieurement, ces clauses peuvent être identifiées et supprimées, en analysant le graphe des inférences).

# Chapitre 4

# Codage des schémas de formules propositionnelles

Nous allons présenter, dans ce chapitre, les schémas de formules en logique propositionnelle. Ce langage, défini dans [2], permet de décrire des séquences de formules dépendant de paramètres entiers définis à l'aide de conjonctions et de disjonctions itérées. Nous allons, dans un premier temps, rappeler la définition formelle des schémas, ensuite, nous allons montrer comment les traduire en ensembles de $n$-clauses. Cette traduction préserve la satisfiabilité (mais non l'équivalence).

## 4.1 Les schémas de formules propositionnelles

Les schémas de formules en logique propositionnelle sont des formules construites à l'aide des connecteurs $\bigwedge$, $\bigvee$ et de propositions indexées par des termes de l'arithmétique linéaire. Nous voulons, par exemple, exprimer des formules de la forme $p_1 \vee \cdots \vee p_n$ (où $p_1, \ldots, p_n$ sont des propositions et $n$ un entier naturel). Cette formule sera notée $\bigvee_{i=1}^{n} p_i$, où $1, i, n$ sont des constantes et variables du langage ($n$ n'est pas un entier fixé). Avant de donner la définition formelle d'un schéma nous rappelons quelques notions, tirées de [2] et [3].

Soit $n$ une constante de type `nat` (dénotant un paramètre). Toutes les expressions arithmétiques considérées dans ce chapitre sont supposées construites sur la signature $\mathbb{N} \cup \{+, n\}$ et sur un ensemble de variables. Une formule arithmétique est construite sur l'ensemble des expressions arithmétiques en utilisant les prédicats usuels $<, \leq, \simeq$ et les connectifs $\vee, \wedge$ etc. On dit qu'une formule arithmétique $\phi$ *encadre* une variable entière $i$ ssi il existe des expressions arithmétiques linéaires fermées $e$ et $f$ telles que $\phi \models e \leq i \wedge i \leq f$. Pour simplifier, nous allons considérer, dans ce chapitre, et à la différence de [2], qu'une expression qui encadre une variable $i$ est toujours de la forme $e \leq i \wedge i \leq f$, où $e$ et $f$ sont des termes fermés.

**Définition 56** (*Schéma*) Soit un ensemble de symboles propositionnels (de type `nat → bool`). On définit l'ensemble des *schémas* définis sur $n$ inductivement comme suit :

- Si $p$ est un symbole propositionnel et $e$ un entier naturel ou une expression arithmétique linéaire de la forme $x + k$ (où $x$ est une variable entière et $k$ un entier naturel) alors $p_e$ et $\neg p_e$ sont des schémas (appelés *propositions indexées*).
- Si $a$ et $b$ sont deux expressions arithmétiques, alors $a \leq b$ et $a \simeq b$ sont des schémas.
- Si $m_1$ et $m_2$ sont deux schémas alors $m_1 \circ m_2$, où $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.
- Si $m$ est un schéma ne contenant qu'une variable libre $i$ de type entier et $\phi$ une formule arithmétique qui encadre $i$ alors $\bigwedge_{i|\phi} m$ et $\bigvee_{i|\phi} m$ sont des schémas. ◊

**Exemple 57**

$$\bigwedge_{i|1\leq i\wedge i\leq n} p_i$$

et

$$\bigvee_{i|1\leq i\wedge i\leq n} p_i$$

sont des schémas. ♣

Comme $\phi$ est de la forme $e \leq i \wedge i \leq f$, le schéma :

$$\bigvee_{i|\phi} \dots$$

sera écrit

$$\bigvee_{i=e}^{f} \dots$$

Les conjonctions itérées sont écrites de la même manière.

**Exemple 58** La formule qui suit est un exemple de schéma :

$$p_0 \wedge (\bigwedge_{i=1}^{n} p_i \Rightarrow p_{i+1}) \ \wedge \neg p_n$$

♣

Nous omettons la définition formelle de la sémantique des schémas qui est très naturelle : le paramètre $n$ est interprété comme un entier naturel, les propositions indexées de la forme $p_k$ où $k$ est un entier sont interprétées comme `V` ou `F`, les symboles de l'arithmétique possèdent leur sémantique habituelle et les connectifs itérés $\bigvee_{i=a}^{b} \phi_i$ (resp. $\bigwedge_{i=a}^{b} \phi_i$) sont interprétés comme $\exists i \in [a,b]\, \phi_i$ (resp. $\forall i \in [a,b]\, \phi_i$). Nous renvoyons à [2, 6] pour plus de détails.

Nous allons maintenant présenter la traduction des schémas propositionnels en ensembles de $n$-clauses.

## 4.2 Des schémas vers les $n$-clauses

### 4.2.1 Codage des propositions indexées

D'après la définition 56, chaque proposition indexée est de la forme $p_{\epsilon.x+k}$, où $x$ est une variable entière, $k$ est un entier naturel et $\epsilon \in \{0,1\}$. Nous allons coder $p_{\epsilon.x+k}$ par un atome $p(\mathrm{succ}^k(\epsilon.x))$ où $x$ est une variable de sorte `nat`. Nous allons voir au chapitre 6, qu'afin d'assurer la complétude nous devons restreindre la profondeur des termes de sorte `nat` à 2. Pour cette raison, nous allons éliminer les termes de la forme $p(\mathrm{succ}^k(\epsilon.x))$, où $k \geq 2$, en introduisant de nouveaux prédicats $p^k$ vérifiant la formule $p^k(\epsilon.x) \Leftrightarrow p(\mathrm{succ}^k(\epsilon.x))$. Nous ajoutons les axiomes suivants qui assurent que $p^k(\epsilon.x)$ conserve la sémantique de $p(\mathrm{succ}^k(\epsilon.x))$ :

$$\mathcal{A}_k \stackrel{\text{def}}{=} \{p^l(x) \simeq p^{l-1}(\mathrm{succ}(x)), p^0(x) \simeq \mathtt{V}p(x) \mid 0 < l \leq k\}$$

**Proposition 59** *Si $I \models \mathcal{A}_k$ alors pour tout $l \in [\![0,k]\!]$, $I \models (p^l(x) \simeq p(\mathrm{succ}^l(x)))$.*

PREUVE. La preuve est triviale par induction sur $l$. ∎

### 4.2.2 Codage des équations et des inéquations

Nous allons montrer, dans cette sous-section, comment coder les inégalités et les équations. La formule $x + l > \epsilon.n + k$ (avec $l, k \in \mathbb{N}$, $\epsilon \in \{0,1\}$) est codée avec l'atome $q^{l,\epsilon,k}(x)$ défini par $\mathcal{A}'_{l',k'}$ l'ensemble d'axiomes suivant :

| | | | |
|---|---|---|---|
| 1. | $q^{l,\epsilon,k}(x) \simeq q^{0,\epsilon,k}(\mathrm{succ}^l(x))$ | 2. | $q^{0,\epsilon,k+1}(\mathrm{succ}(x)) \simeq q^{0,\epsilon,k}(x)$ |
| 3. | $q^{0,\epsilon,k}(0) \not\simeq \mathtt{V}$ | 4. | $q^{0,0,0}(\mathrm{succ}(x)) \simeq \mathtt{V}$ |
| 5. | $q^{0,1,0}(\mathrm{succ}(n)) \simeq \mathtt{V}$ | 6. | $q^{0,1,0}(n) \not\simeq \mathtt{V}$ |
| 7. | $q^{0,1,0}(\mathrm{succ}(x)) \simeq \mathtt{V} \vee q^{0,1,0}(x) \not\simeq \mathtt{V}$ | | |

où $l \in [0,l']$, $k \in [0,k']$ et $\epsilon \in \{0,1\}$.

Sémantiquement, l'atome $q^{l,\epsilon,k}(x)$ est équivalent à l'inéquation $x + l > \epsilon.n + k$, comme le montre la proposition suivante :

**Proposition 60** *Soit $I \models \mathcal{A}'_{l',k'}$. Pour tout $(l,k,\epsilon) \in [\![0,l']\!] \times [\![0,k']\!] \times \{0,1\}$, nous avons $I \models q^{l,\epsilon,k}(x) \simeq \mathtt{V} \Leftrightarrow x + l > \epsilon.n + k$.*

PREUVE. Supposons que $l = k = 0$. Si $\varepsilon = 0$ alors l'interprétation de $q^{l,\varepsilon,k}$ est fixée par les axiomes 3 et 4 et on a $q^{0,0,0}(\mathrm{succ}(x)) \simeq \mathtt{V}$ (i.e. $\mathrm{succ}(x) > 0$) et $q^{0,0,0}(0) \not\simeq \mathtt{V}$ (i.e. $0 \not> 0$). Si $\varepsilon = 1$, alors l'interprétation de $q^{l,\varepsilon,k}$ est fixée par les axiomes 5 et 6 et on a $q^{0,1,0}(\mathrm{succ}(x)) \simeq \mathtt{V}$ si $x = n$ (i.e. $\mathrm{succ}(n) > n$) et $q^{0,1,0}(x) \not\simeq \mathtt{V}$ si $x = n$ (i.e. $n \not> n$). De plus, le dernier axiome assure que $q^{0,1,0}(\mathrm{succ}(x))$ est $\mathtt{V}$ si $q^{0,1,0}(x)$ l'est. Puisque $q^{0,1,0}(\mathrm{succ}(n)) \simeq \mathtt{V}$, cela implique que $q^{0,1,0}(\mathrm{succ}(\mathrm{succ}(n)))$,$q^{0,1,0}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(n))))$, ... sont $\mathtt{V}$, et puisque $q^{0,1,0}(n) \not\simeq \mathtt{V}$, $q^{0,1,0}(n-1)$,$q^{0,1,0}(n-2)$, ... sont différents de $\mathtt{V}$. Donc $q^{0,1,0}(x) \simeq \mathtt{V}$ ssi $x \geq s(n)$.

En utilisant les axiomes 2 et 3 on montre aisément par induction sur $x$ que $q^{0,\varepsilon,k}(x)$ est équivalent à $q^{0,\varepsilon,k-x}(0)$ si $x < k$ et différent de $\mathtt{V}$ sinon.

Ensuite l'axiome 1 assure que $q^{l,\varepsilon,k}(x) \simeq q^{0,\varepsilon,k}(x+l)$. ∎

Nous pouvons également coder l'inéquation $x + l \leq \epsilon.n + k$, en la transformant comme suit :

$$x + l \leq \epsilon.n + k \equiv x + l + 1 \not> \epsilon.n + k$$

De la même manière, nous pouvons coder une équation de la forme $x + l \simeq \epsilon.n + k$ avec la formule suivante :

$$x + l \leq \epsilon.n + k \wedge x + l > \epsilon.n + k$$

### 4.2.3 Codage des connectifs itérés

Nous montrons maintenant comment coder les connectifs itérés par des ensembles de $n$-clauses. Il suffit pour cela de coder la définition inductive de ces connectifs par des axiomes. Nous supposons dans un premier temps que la borne inférieure est 0. Une formule de la forme $\bigvee_{i=0}^{b} \phi_i$ est codée par un atome du premier ordre $\psi_\vee(b+1)$ associé à l'ensemble des axiomes $E^\vee_\phi$ suivants :

$$\psi_\vee(0) \not\simeq \mathtt{V}$$

$$\psi_\vee(\mathrm{succ}(x)) \simeq \mathtt{V} \Leftrightarrow \phi'(x) \vee \psi_\vee(x) \simeq \mathtt{V}$$

où $\phi'(x)$ dénote un codage de $\phi_x$ obtenu en appliquant récursivement la procédure de codage. De façon similaire, une formule de la forme $\bigwedge_{i=0}^{b} \phi_i$ est codée par un atome $\psi_\wedge(b+1)$ défini à l'aide de l'ensemble $E^\wedge_\phi$ des axiomes suivants :

$$\psi_\wedge(0) \simeq \mathtt{V}$$

$$\psi_\wedge(\mathrm{succ}(x)) \simeq \mathtt{V} \Leftrightarrow \phi'(x) \wedge \psi_\wedge(x) \simeq \mathtt{V}$$

où $\phi'(x)$ dénote un codage de $\phi_x$. On note $E$ la conjonction de tous les axiomes ci-dessus (cet ensemble est infini, en pratique il suffit de considérer l'ensemble des axiomes correspondant aux connectifs itérés $\phi$ apparaissant dans la formule).

**Proposition 61** *Si $\mathcal{I} \models E$ alors pour tout $n \in \mathbb{N}$ et pour toute formule $\psi$ de la forme $\bigvee_{i=0}^{b} \phi_i$ (respectivement $\bigwedge_{i=0}^{b} \phi_i$) nous avons $\mathcal{I} \models \psi_\wedge(n) \simeq \mathtt{V} \Leftrightarrow \mathcal{I} \models \bigvee_{i=0}^{n} \phi_i$ (respectivement $\mathcal{I} \models \psi_\vee(n) \simeq \mathtt{V} \Leftrightarrow \mathcal{I} \models \bigvee_{i=0}^{n} \phi_i$).*

PREUVE. Par induction sur $n$. ∎

Les formules de la forme $\bigvee_{i=a}^{b} \phi_i$ où $a > 0$ sont éliminées en les transformant comme suit :

$$\bigvee_{i=a}^{b} \phi \;\equiv\; \bigvee_{i=0}^{b} (i \geq a \wedge \phi)$$

Puis nous procédons comme dans la sous-section 4.2.2 pour coder l'inégalité $i \geq a$. Cette transformation est aussi valable pour les conjonctions itérées et nous permet d'étendre le codage défini par la proposition 61 au cas où la borne inférieure est supérieure à 0.

### 4.2.4 Mise sous forme $n$-clausale

Nous avons présenté dans les sections précédentes la traduction d'un schéma en une formule du premier ordre contenant des constantes de type entier. Une fois que tous les schémas sont traduits, nous mettons les formules obtenues sous forme clausale en utilisant les algorithmes usuels (voir [64, 69]), ensuite nous remplaçons le paramètre $n$ par une variable $x$ en introduisant une contrainte $n \simeq x$. Nous obtenons ainsi un ensemble de $n$-clauses sat-équivalent. Nous allons maintenant appliquer cet algorithme sur un exemple.

**Exemple 62** Soit la formule :

$$p_0 \wedge (\bigwedge_{i=0}^{n} p_i \Rightarrow p_{i+2}) \;\wedge \neg p_n$$

Nous allons transformer cette formule en un ensemble de $n$-clauses.

1. Nous transformons dans un premier temps les propositions indexées en prédicat ordinaire, la formule devient alors :

$$p(0) \wedge (\bigwedge_{i=0}^{n} p(i) \simeq \mathtt{V} \Rightarrow p(\mathrm{succ}(\mathrm{succ}(i))) \simeq \mathtt{V}) \;\wedge \neg p(n) \simeq \mathtt{V}$$

2. Nous transformons la conjonction itérée en introduisant un prédicat $\psi$ défini par :

$$\psi_\wedge(0) \simeq \mathtt{V}$$

$$\psi_\wedge(\mathrm{succ}(x)) \simeq \mathtt{V} \Leftrightarrow (p(x) \Rightarrow p(\mathrm{succ}^2(x))) \vee \psi_\wedge(x) \simeq \mathtt{V}$$

et la formule de départ devient :

$$p(0) \simeq \mathtt{V} \wedge \psi_\wedge(n+1) \simeq \mathtt{V} \wedge p(n) \not\simeq \mathtt{V}$$

3. Nous éliminons les propositions indexées de profondeur strictement supérieure à 2, ce qui correspond dans la formule à $p(\mathrm{succ}^2(x))$. Pour cela on introduit un prédicat $p^1$ défini par la formule $F : p^1(x) \simeq p(\mathrm{succ}(x))$ et on remplace le terme $p(\mathrm{succ}^2(x))$ par $p^1(\mathrm{succ}(x))$.
4. Nous procédons à la clausification de toutes les formules introduites à savoir la formule $F$ et l'ensemble des axioms précédemment introduits. Nous réalisons l'abstraction du paramètre en introduisant les contraintes :
   — $p(n) \not\simeq \mathtt{V}$ est remplacé par la $n$-clause $[p(x) \not\simeq \mathtt{V} \mid n \simeq x]$
   — $\psi_\wedge(\mathrm{succ}(n))$ est remplacé par la $n$-clause $[\psi_\wedge(\mathrm{succ}(x)) \simeq \mathtt{V} \mid n \simeq x]$ ♣

**Proposition 63** *Si Sc est un schéma et que S est l'ensemble de n-clauses obtenues en traduisant le schéma Sc alors Sc et S sont sat-équivalents.*

PREUVE. Découle des propositions 59, 60 et 61, par induction structurelle sur la formule considérée. ■

# Chapitre 5

# Formules du $1^{er}$ ordre avec des termes de type mot

Dans ce chapitre, nous généralisons les notions introduites dans le chapitre précédent. Les $n$-clauses, définies dans le chapitre 3, représentent des formules hybrides contenant des termes interprétés comme des entiers naturels. Notre objectif est de traiter un cas un peu plus général, où ces formules hybrides contiennent des termes inductifs définis à l'aide de constructeurs monadiques c'est-à-dire des mots. Nous procédons comme précédemment : nous introduisons un nouveau formalisme (à la manière des $n$-clauses), nous étendons le calcul de superposition à ce nouveau formalisme, puis nous proposons une nouvelle version de la détection de boucles, plus générale et englobant la version présentée dans le chapitre 3. L'extension à des termes quelconques (constructeurs d'arités $\leq 2$) n'a pas été considérée dans le cadre de ce travail car la définition de la règle de détection de boucles s'avère beaucoup plus complexe ce qui limite sa portée et son intérêt pratique. La présence de constructeurs binaires oblige en effet à prendre en compte un ensemble de positions lors de l'identification du cycle. Nous discutons dans le chapitre 11 de cette extension.

Nous introduisons maintenant le nouveau formalisme appelé les $\alpha$-clauses et qui est une simple extension des $n$-clauses.

## 5.1 Logique des $\alpha$-clauses

Nous restons dans le cadre de la logique équationnelle multi-sortée. L'ensemble de sorte $\mathcal{S}$ est partionné en sous-ensembles $\mathcal{S} = \mathcal{S}_T \cup \mathcal{S}_I \cup \{\texttt{bool}\}$ disjoints. L'ensemble $\mathcal{S}_I$ regroupe les sortes des termes définis dans un domaine inductif et $\mathcal{S}_T$ les sortes des termes standards.

**Remarque 64** *Dans le chapitre 3, nous avons considéré le cas particulier où $\mathcal{S}_I$ ne contient qu'une seule sorte* `nat` *avec deux constructeurs* $0$ : `nat` *et* succ : $\texttt{nat} \rightarrow \texttt{nat}$.

Par commodité, nous supposons que seul le premier argument des fonctions peut être de type dans $\mathcal{S}_I$ (ceci n'est pas réellement restrictif puisque nous supposons que les formules ne contiennent qu'un seul paramètre, avec des axiomes contenant une unique variable de type inductif). D'autre part, les éléments de type $\mathcal{S}_I$ sont des mots, donc tous les constructeurs de domaine $\mathcal{S}_I$ sont monadiques et de co-domaine dans $\mathcal{S}_I$. Nous supposons donc que le profil d'un symbole $f$ non constant est sous l'une des formes suivantes :

1. $\mathtt{s_1} \times \ldots \times \mathtt{s_n} \rightarrow \mathtt{s}$ où $\mathtt{s_1}, \ldots, \mathtt{s_n} \in \mathcal{S}_\mathtt{I} \cup \mathcal{S}_\mathtt{T}$, et $\mathtt{s} \in \mathcal{S}_\mathtt{T} \cup \mathtt{bool}$.
2. $\mathtt{s} \rightarrow \mathtt{s'}$, où $\mathtt{s}, \mathtt{s'} \in \mathcal{S}_\mathtt{I}$.

Un symbole de fonction de profile $\mathtt{s} \rightarrow \mathtt{bool}$ est un *symbole de prédicat*. Une variable d'une sorte incluse dans $\mathcal{S}_I$ est appelée *$\mathcal{S}_I$-variable*. Un *$\mathcal{S}_I$-terme* est un terme d'une sorte incluse dans $\mathcal{S}_I$ et un terme est *standard* s'il est d'une sorte incluse dans $\mathcal{S}_T \cup \{\mathtt{bool}\}$.

**Définition 65** (*Termes hybrides*) Un *terme hybride* est un terme standard contenant un $\mathcal{S}_I$-terme. ◊

Nous avons aussi utilisé, dans le chapitre 3, un paramètre $n$ qui représente un entier naturel, comme nous considérons maintenant des termes quelconques, nous notons à présent le paramètre $\alpha$, et supposons que ce paramètre possède une sorte unique dans $\mathcal{S}_I$. Nous supposons comme précédemment que le paramètre $\alpha$ n'apparaît pas dans la signature. Nous pouvons, maintenant, définir les $\alpha$-clauses :

**Définition 66** (*Les $\alpha$-clauses) Une* $\alpha$-clause *est un couple* $[C \mid \bigwedge_{i=1}^{k} \alpha \simeq t_i]$ *où $C$ est une clause et les $t_i$ (avec $1 \leq i \leq k$) sont des $\mathcal{S}_I$-termes de même sorte que $\alpha$.* ◊

Comme dans le cas des $n$-clauses, nous utilisons quelques abréviations. Une $\alpha$-clause de la forme $[C \mid \mathtt{V}]$ est simplement notée $C$ et une $\alpha$-clause de la forme $[\Box \mid \bigwedge_{i=1}^{k} \alpha \simeq t_i]$ est notée $\bigvee_{i=1}^{k} \alpha \not\simeq t_i$.
La sémantique des $\alpha$-clauses est semblable à celle des $n$-clauses, une interprétation est aussi une paire $(\mathcal{I}(\alpha), \simeq_\mathcal{I})$ où $\simeq_\mathcal{I}$ est une congruence sur les termes standards fermés et $\mathcal{I}(\alpha)$ est un $\mathcal{S}_I$-terme fermé de la même sorte que $\alpha$ (voir section 3.1 du chapitre 3).

## 5.2 Procédure de preuve pour les $\alpha$-clauses

Nous étendons maintenant la procédure de preuve définie dans le chapitre 3 aux $\alpha$-clauses. Les règles d'inférence sont appliquées de la même manière que pour les $n$-clauses, l'unique différence concerne la détection de boucles, qui devient beaucoup plus complexe car nous passons d'un espace de recherche unidimensionnel (dans le cadre des entiers), à un espace sous forme d'arbre (les mots correspondant à un chemin dans un arbre étiqueté par les constructeurs de type dans $\mathcal{S}_I$). Dans la section suivante, nous introduisons quelques définitions qui nous permettent de formaliser la détection de boucles et de donner un théorème affirmant la correction de la règle correspondante. Nous démontrons aussi quelques lemmes intermédiaires que nous utilisons dans la preuve de ce théorème.

### 5.2.1 Définitions et lemmes intermédiaires

Nous avons défini, dans le chapitre 3, la notion de décalage d'une $n$-clause comme étant une nouvelle $n$-clause qui garde la même partie clausale, en faisant une translation de la valeur du paramètre. À la différence de la notion introduite pour les entiers pour laquelle le décalage était appliqué uniformément pour toute valeur du paramètre, il est ici nécessaire de restreindre l'application de la fonction à des instances particulières de $\alpha$. L'idée est que, dans la mesure où l'espace de recherche possède à présent une structure arborescente et non linéaire, le décalage à appliquer peut dépendre de chaque instance du paramètre (chaque branche de l'espace de recherche correspond à un décalage différent). La fonction de décalage sera donc paramétrée par un terme $t$ dénotant la forme générale des instances sur lesquelles la translation doit être appliquée. Nous présentons maintenant une définition formelle (plus générale) de cette notion de décalage pour les $\alpha$-clauses :

**Définition 67** Une *substitution de décalage* pour une variable $x$ est une substitution de la forme $\{x \mapsto s\}$ où $s \neq x$ et $\text{var}(s) = \{x\}$. Soit $\theta$ une substitution de décalage, $C$ une $\alpha$-clause et $t$ un terme tel que $\text{var}(t) = \{x\}$. Le *décalage* d'une $\alpha$-clause $C$ est une $\alpha$-clause notée $\mathit{shift}(C, t, \theta)$ et définie comme suit :

- Si $C$ est de la forme $[D \mid \alpha \simeq t\sigma]$ pour une certaine substitution $\sigma$, alors $\mathit{shift}(C, t, \theta) \stackrel{\text{def}}{=} [D \mid \alpha \simeq t\theta\sigma]$.
- Sinon $\mathit{shift}(C, t, \theta) \stackrel{\text{def}}{=} C$.

Si $S$ est un ensemble de $\alpha$-clauses alors $\mathit{shift}(S, t, \theta) \stackrel{\text{def}}{=} \bigcup_{C \in S} \{\mathit{shift}(C, t, \theta)\}$. $\Diamond$

**Remarque 68** *Le décalage d'une n-clause $C$ que nous avions notée $C{\downarrow_j}$ correspond, ici, à $\mathit{shift}(C, x, \{x \to \text{succ}^j(x)\})$.*

Nous montrons dans l'exemple suivant comment obtenir le décalage d'une $\alpha$-clause.

**Exemple 69** Soit $C$ une $\alpha$-clause, les deux tableaux suivants montrent les différentes valeurs que peut prendre le décalage : $shift(C, t, \theta)$ pour un terme $t$ et une substitution $\theta$ tous les deux fixés :

| $C$ | $[h(g(x), y) \simeq y \mid \alpha \simeq f(g(x))]$ | |
|---|---|---|
| $t$ | $f(x)$ | $f(g(x))$ |
| $\theta$ | $\{x \mapsto f'(x)\}$ | $\{x \mapsto f'(x)\}$ |
| $shift(C, t, \theta)$ | $[h(g(x), y) \simeq y \mid \alpha \simeq f(f'(g(x)))]$ | $[h(g(x), y) \simeq y \mid \alpha \simeq f(g(f'(x)))]$ |

| $C$ | $[h(x, y) \simeq a \mid \alpha \simeq f(x)]$ | |
|---|---|---|
| $t$ | $f(x)$ | $f(g(x))$ |
| $\theta$ | $\{x \mapsto f'(x)\}$ | $\{x \mapsto f'(x)\}$ |
| $shift(C, t, \theta)$ | $[h(x, y) \simeq y \mid \alpha \simeq f(f'(x))]$ | $C$ |

♣

Nous appelons *t-ensemble*, un ensemble de $\alpha$-clauses $S$ où pour chaque $\alpha$-clause de la forme $[D \mid \alpha \simeq s]$ appartenant à $S$, nous avons $s \preceq t$. Nous définissons la fonction $s \mapsto \theta_t^{-1}(s)$ qui formalise, en quelque sorte l'inverse du décalage introduit dans la définition 67 :

**Définition 70** Soit $t, s$ deux termes d'une sorte incluse dans $\mathcal{S}_I$ et soit $\theta$ une substitution de décalage. $\theta_t^{-1}(s)$ est le terme défini comme suit :

$$\theta_t^{-1}(s) \stackrel{\text{def}}{=} \begin{cases} t\sigma & \text{si } s = t\theta\sigma, \text{ pour une certaine substitution } \sigma \\ s & \text{sinon} \end{cases}$$

**Remarque 71** *Cette fonction particulière est assez simple dans le cas des n-clauses, car les décalages sont définis comme des translations. La fonction inverse est juste la translation négative : pour obtenir la clause $C$ à partir de son décalage $D = C{\downarrow_j}$ il suffit d'appliquer la transformation $D{\downarrow_{(-j)}} = C$. Cette opération appliquée aux termes apparaissant dans la contrainte correspond à $\theta_x^{-1}(\text{succ}^{i+j}(x))$, où $\theta = \{x \mapsto \text{succ}^j(x)\}$.*

Voici un exemple des différentes valeurs que peut prendre la fonction $s \mapsto \theta_t^{-1}(s)$ :

**Exemple 72** Si $\theta = \{x \mapsto f'(x)\}$ alors
- $\theta^{-1}_{f(x)}(f(f'(g(x)))) = f(g(x))$,
- $\theta^{-1}_{f(g(x))}(f(g(f'(x)))) = f(g(x))$
- $\theta^{-1}_{f(g(x))}(f(f'(g(x)))) = f(f'(g(x)))$.

♣

Nous montrons, maintenant, comment la fonction $\theta_t^{-1}(s)$ affecte la taille d'un terme $s$ sur lequel elle est appliquée. On note *size(t)* la taille d'un terme $t$ c'est-à-dire le nombre de symboles qui apparaissent dans $t$.

**Proposition 73** *Soit $t, s$ deux termes d'une sorte incluse dans $\mathcal{S}_I$ et soit $\theta$ une substitution de décalage. Si $s \preceq t\theta$ et $dom(\theta) = var(t)$ alors $size(\theta_t^{-1}(s)) < size(s)$.*

PREUVE. Par définition $s$ est de la forme $t\theta\sigma$, pour une certaine substitution $\sigma$, et donc $\theta_t^{-1}(s) = t\sigma$. Soit $\{x\} = dom(\theta)$. Pour tous les termes $u$ et pour toutes les substitutions $\eta$, nous avons : $\text{size}(u\eta) = \text{size}(u) + \Sigma_{y \in V}(\text{size}(y\eta) - 1)$, où $V$ est le multi-ensemble des occurrences de variables dans $u$. En particulier, Si $u$ n'est pas fermé et d'une sorte incluse dans $\mathcal{S}_I$, nous avons : $\text{size}(u\eta) = \text{size}(u) - 1 + \text{size}(y\eta)$, où $y$ est l'unique variable de $u$. Nous avons donc : $\text{size}(t\sigma) = \text{size}(t) - 1 + \text{size}(x\sigma)$ (car $\{x\} = dom(\theta) = \text{var}(t)$) et comme $x\theta$ contient $x$, $\text{size}(t\theta\sigma) = \text{size}(t) - 1 + \text{size}(x\theta) - 1 + \text{size}(x\sigma)$. Puisque $\theta$ est une substitution de décalage, $x\theta$ n'est ni une variable (sinon $\theta$ serait un renommage) ni une constante (sinon $\theta$ serait fermée). Par conséquent $\text{size}(x\theta) > 1$ et $\text{size}(t\theta\sigma) > \text{size}(t\sigma)$.

La proposition suivante montre que la fonction $\theta_t^{-1}$ est injective sur les $\mathcal{S}_I$-termes instances de $t\theta$ :

**Proposition 74** *Soit $t$ un terme et $\theta$ une substitution. Soit $t_1$, $t_2$ deux termes tels que $t\theta \succeq t_1, t_2$. Si $t_1 \neq t_2$ alors $\theta_t^{-1}(t_1) \neq \theta_t^{-1}(t_2)$.*

PREUVE. Comme $t\theta \succeq t_i$, $t_i$ est de la forme $t\theta\sigma_i$, alors $\theta_t^{-1}(t_i) = t\sigma_i$. Si $t_1 \neq t_2$ alors $\sigma_1 \neq \sigma_2$, donc $\theta_t^{-1}(t_2) \neq \theta_t^{-1}(t_2)$. ∎

Le lemme suivant est important car il fait le lien entre les opérations $shift(S, t, \theta)$ and $\theta_t^{-1}$, que nous allons utiliser pour démontrer le théorème principal de la détection de boucles :

**Lemme 75** *Soit $t$ un terme, $S$ un $t$-ensemble, $\theta$ une substitution et $\mathcal{I}$ un modèle pour $shift(S, t, \theta)$ tel que $t\theta \succeq \mathcal{I}(\alpha)$. L'interprétation $\mathcal{J}$ telle que :*

- *$=_{\mathcal{J}}$ est identique à $=_{\mathcal{I}}$,*
- *et $\mathcal{J}(\alpha) \stackrel{def}{=} \theta_t^{-1}(\mathcal{I}(\alpha))$,*

*valide $S$.*

PREUVE. Soit $C$ une $\alpha$-clause dans $S$. D'après la définition 67, nous avons $shift(S, t, \theta) = C$, donc $\mathcal{I} \models C$. Nous déduisons que $\mathcal{J} \models C$ car $C$ ne contient pas $\alpha$. Supposons que $C$ est de la forme $[D \mid \alpha \simeq s]$. Soit $\sigma$ une substitution fermée appliquée aux variables de $C$. Nous devons montrer que $\mathcal{J} \models C\sigma$. Par définition, comme $S$ est un $t$-ensemble, nous avons $s \preceq t$ c'est-à-dire $s$ est de la forme $t\eta$ pour une certaine substitution $\eta$. Nous avons donc $shift(C, t, \theta) = [D \mid t\theta\eta]$. Comme $\mathcal{I} \models shift(C, t, \theta)$, nous avons $\mathcal{I} \models shift(C, t, \theta)\sigma$, donc $\mathcal{I} \models [D\sigma \mid t\theta\eta\sigma]$. Si $\mathcal{I} \models D\sigma$ alors $\mathcal{J} \models D\sigma$ et donc $\mathcal{J} \models C\sigma$ (car $\mathcal{I}$ et $\mathcal{J}$ coïncident sur les clauses ne contenant pas $\alpha$). Si $\mathcal{I} \models \alpha \not\approx t\theta\eta\sigma$ alors $\mathcal{I}(\alpha) \neq t\theta\eta\sigma$, et d'après la proposition 74, $\theta_t^{-1}(\mathcal{I}(\alpha)) \neq \theta_t^{-1}(t\theta\eta\sigma)$ et $\mathcal{J}(\alpha) \neq t\eta\sigma$, par conséquent $\mathcal{J} \models C\sigma$. ∎

Nous définissons maintenant la notion de *couverture* dans le cadre de la logique multi-sortée :

**Définition 76** Un ensemble de termes $T$ d'une même sorte `s` est *couvrant* pour un terme $t$ de sorte `s` ssi pour toute substitution fermée $\theta$ des variables de $t$, il existe un terme $s \in T$ tel que $t\theta \preceq s$. $\Diamond$

Par exemple $\{0, \text{succ}(x)\}$ est couvrant pour toute variable $y$ de type `nat`, $nil, cons(x, y)$ est couvrant pour les listes etc.

### 5.2.2 La détection de boucles pour les $\alpha$-clauses

Comme nous l'avons mentionné précédemment, l'espace de recherche a une structure d'arbre. La détection de boucles s'applique lorsque pour toutes les instances $s$ d'un terme donné $t$, la branche dans l'espace de recherche qui correspond au cas $\alpha = s$ est soit fermée (c'est-à-dire une clause de la forme $\alpha \not\simeq s'$, où $s' \succeq s$, est générée), soit elle peut être réductible, par décalage, à une branche correspondant à un terme strictement plus petit. Nous pouvons alors fermer la branche $\alpha = t$, par "descente infinie".

La règle de détection de boucles est formalisée par le théorème suivant :

**Théorème 77** *Soit $S$ un ensemble de $\alpha$-clauses. Supposons qu'il existe un ensemble $\{t_1, \ldots, t_n\}$ couvrant pour $t$ tel que :*

1. *Pour tout $i \in [\![1, n]\!]$, il existe un $t_i$-ensemble $S_i \subseteq S$.*
2. *Pour tout $i \in [\![1, n]\!]$ et pour tout terme fermé $s \preceq t_i$, une des conditions suivantes est vérifiée :*
   (a) *$S_i \models \alpha \not\simeq s$.*
   (b) *Il existe un entier $j \in [\![1, n]\!]$ et un décalage $\theta_s$ tel que $S_i \models shift(S_j, t_j, \theta_s)$ et $s \preceq t_j\theta_s$.*

*Nous avons alors $S \models \{\alpha \not\simeq t\}$.*

PREUVE. Soit $\mathcal{I}$ un modèle de $S$. Nous procédons comme suit : nous supposons que $\mathcal{I}(\alpha) = t\gamma$, pour une certaine substitution $\gamma$, et nous dérivons une contradiction. Puisque $T$ est couvrant pour $t$, il existe $i \in [\![1, n]\!]$ tel que $t\gamma \preceq t_i$. Comme $\mathcal{I} \models S$ et $S_i \subseteq S$, nous avons $\mathcal{I} \models S_i$. Nous supposons que $\text{size}(\mathcal{I}(\alpha))$ est minimal c'est-à-dire que pour toute interprétation $\mathcal{J}$, si $\text{size}(\mathcal{J}(\alpha)) < \text{size}(\mathcal{I}(\alpha))$ et s'il existe $j \in [\![1, n]\!]$ tel que $\mathcal{J}(\alpha) \preceq t_j$ alors $\mathcal{J} \not\models S_j$. Comme $\mathcal{I} \models S_i$, nous avons $S_i \not\models \alpha \not\simeq t\gamma$ (car $\mathcal{I}(\alpha) = t\gamma$). Ce qui implique, d'après la propriété 2, qu'il existe $j \in [\![1, n]\!]$ tel que $S_i \models shift(S_j, t_j, \theta_{t\gamma})$ et $t\gamma \preceq t_j\theta_{t\gamma}$. Nous avons alors $\mathcal{I} \models shift(S_j, t_j, \theta_{t\gamma})$. Soit $\mathcal{J}$ l'interprétation définie comme suit : $=_{\mathcal{J}}$ est identique à $=_{\mathcal{I}}$, et $\mathcal{J}(\alpha) \stackrel{\text{def}}{=} \theta_{t\gamma}^{-1}(\mathcal{I}(\alpha))$.

D'après le lemme 75, comme $\mathcal{I} \models \mathit{shift}(S_j, t_j, \theta_{t\gamma})$ et $S_j$ est un $t_j$-ensemble, nous avons $\mathcal{J} \models S_j$. Nous avons $\mathcal{I}(\alpha) = t\gamma \preceq t_j\theta_{t\gamma}$, donc $\mathcal{J}(\alpha) = \theta_{t\gamma}^{-1}(t_j) \preceq t_j$, et d'après la proposition 73, $\text{size}(\mathcal{J}(\alpha)) < \text{size}(\mathcal{I}(\alpha))$. Comme $\mathcal{I}$ est minimal, nous déduisons $\mathcal{J} \not\models S_j$, et cela est une contradiction. ∎

En pratique, trouver les ensembles $S_i$ et les substitutions de décalage $\theta_s$ et vérifier les conditions

1. $S_i \models \alpha \not\simeq s$
2. $S_i \models \mathit{shift}(S_j, t_j, \theta_s)$

n'est pas faisable (car ces conditions sont de nature sémantique et non syntaxique). Nous devons imposer des conditions syntaxiques plus fortes. Donc nous vérifions plutôt que

1. une clause de la forme $[\Box \mid \alpha \not\simeq s']$ (avec $s' \succeq s$) est dérivée d'une clause de $S_i$
2. $\mathit{shift}(S_j, t_i, \theta_s)$ est dérivée de $S_i$.

**Remarque 78** *Le théorème 45 du chapitre 3 peut être vu comme un corollaire du théorème 77. Dans ce cas, l'ensemble de termes $\{t_1, \ldots, t_n\}$ est de la forme $\{s^i(0), \ldots, s^{i+j-1}(0), s^{i+j}(x)\}$ (un tel ensemble est couvrant pour $s^i(x)$). Pour $k = 1, \ldots, j-1$, $S_k$ est de la forme $\{n \neq i + k - 1\}$, et $S_n$ $S_1, ..., S_{n-1}$ sont les ensembles de clauses $\{n \neq i + 1\}$ et $S_n$ est l'ensemble $S'$ de la définition 44. On vérifie facilement que, si $(i, j)$ est une boucle inductive pour $S$, alors la propriété 2 du théorème 77 est satisfaite (les instances de $t_1, \ldots, t_{n-1}$ satisfont 2.a et celles de $t_n$ satisfont 2.b). Ainsi les résultats du chapitre 3 sont des conséquences de ceux du présent chapitre. Nous avons préféré les présenter indépendamment car les définitions sont plus simples et plus naturelles.*

Nous présentons maintenant un exemple d'application.

**Exemple 79** Soit l'ensemble de clauses suivant :

$$\begin{array}{ll} (1) & p(a) \simeq \mathtt{V} \\ (2) & p(x) \not\simeq \mathtt{V} \vee p(f(x)) \simeq \mathtt{V} \\ (3) & p(x) \not\simeq \mathtt{V} \vee p(g(x)) \simeq \mathtt{V} \\ (4) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq x] \end{array}$$

On suppose que $a, f$ et $g$ sont les seuls constructeurs de type dans $\mathcal{S}_I$. Nous déri-

vons, avec notre procédure de preuve, les clauses suivantes :

$$\begin{array}{lcl}
(5) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq f(x)] & (2,4) \\
(6) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq g(x)] & (3,4) \\
(7) & [\Box \mid \alpha \simeq a] & (1,4) \\
(8) & [\Box \mid \alpha \simeq f(a)] & (1,5) \\
(9) & [\Box \mid \alpha \simeq g(a)] & (1,6) \\
(10) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq f(g(x))] & (3,5) \\
(11) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq f(f(x))] & (2,5) \\
(12) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq g(g(x))] & (2,6) \\
(13) & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq g(f(x))] & (3,6)
\end{array}$$

À ce stade, nous considérons le terme $f(x)$ : nous remarquons que l'ensemble $\{f(g(x)), f(f(x)), f(a)\}$ est couvrant pour $f(x)$. la clause 8 correspond à la clause $\alpha \not\simeq f(a)$. La clause 10 correspond à $\mathit{shift}(C_5, f(x), x \mapsto g(x))$ (où $C_5$ est la clause 5 dérivée ci-dessus) et la clause 11 correspond à $\mathit{shift}(C_5, f(x), x \mapsto f(x))$. Nous pouvons appliquer la règle de détection de boucles pour générer la clause $\alpha \not\simeq f(x)$. De la même manière nous générons la clause $\alpha \not\simeq g(x)$. Comme l'ensemble $\{f(x), g(x), a\}$ est couvrant, l'ensemble considéré est insatisfaisable. ♣

**Exemple 80** On considère l'ensemble de $\alpha$-clauses suivant

$$S = \{[p(x) \mid \alpha \simeq x], p(a) \not\simeq \mathtt{V}, p(f(x)) \not\simeq \mathtt{V} \vee q(x) \simeq \mathtt{V}, q(f(x)) \not\simeq \mathtt{V} \vee q(x) \simeq \mathtt{V}, q(a) \not\simeq \mathtt{V}\}$$

où $f(x)$ et $a$ sont des $\mathcal{S}_I$-termes et $x$ une $\mathcal{S}_I$-variable. Le calcul de superposition génère les clauses suivantes :

$$\alpha \not\simeq a$$

$$\alpha \not\simeq f^n(f(x)) \vee q(x)\ \forall n \in \mathbb{N}$$

$$\alpha \not\simeq f^n(f(a))\ \forall n \in \mathbb{N}$$

Comme les clauses $[p(x) \not\simeq \mathtt{V} \mid \alpha \simeq f(f(x))]$ et $\alpha \not\simeq f(a)$ peuvent être dérivées de $[p(x) \simeq \mathtt{V} \mid \alpha \simeq f(x)]$ (avec les clauses de $S[\bot]$), et comme $[p(x) \simeq \mathtt{V} \mid \alpha \simeq f(f(x))] = \mathit{shift}([p(x) \simeq \mathtt{V} \mid \alpha \simeq f(x)], f(x), \{x \to f(x)\})$, et que l'ensemble $\{a, f(x)\}$ est couvrant (car $f$ et $a$ sont les seuls symboles de domaine dans $\mathcal{S}_I$) nous pouvons alors appliquer la détection de boucles, et dériver la clause $\alpha \not\simeq f(x)$. Nous avons généré les clauses $\alpha \not\simeq a$ et $\alpha \not\simeq f(x)$, donc $S$ est insatisfaisable.

# Chapitre 6

# Complétude

Nous avons montré dans le chapitre 3 que le problème de l'insatisfaisabilité des $\alpha$-clauses n'est pas semi-décidable (théorème 34, les $\alpha$-clauses étant strictement plus générales que les $n$-clauses). La procédure de preuve définie dans le chapitre 5 ne peut donc pas être complète dans le cas général. Nous montrons dans ce chapitre que nous pouvons garantir la complétude en imposant certaines conditions aux ensembles de clauses considérés. En premier lieu, nous allons introduire des conditions sémantiques très générales mais suffisantes pour assurer la complétude. Nous allons, dans le chapitre suivant, donner des exemples de classes syntaxiques concrètes qui vérifient ces conditions sémantiques.

## 6.1 Définitions préliminaires

Afin de simplifier les définitions et les preuves, nous supposons que les clauses ne contiennent aucune constante de sorte incluse dans $\mathcal{S}_I$ (nous pouvons pour cela remplacer toute constante $a$ par une une fonction $a(x)$, où $x$ est une variable d'une nouvelle sorte dans $\mathcal{S}_I$ (à noter que la signature doit contenir une constante de la même sorte que $x$ pour garantir qu'il existe au moins un terme de sorte dans $\mathcal{S}_I$). Nous supposons aussi que pour chaque $\alpha$-clause de la forme $[C \mid \alpha \simeq t]$, $t$ n'est pas une variable ; cette condition n'est pas restrictive, nous pouvons garantir qu'elle est toujours vérifiée en introduisant un nouveau symbole de fonction $f : \mathtt{s} \to \mathtt{s}'$, où $\mathtt{s}$ est la sorte de $\alpha$ et $\mathtt{s}'$ est un nouveau symbole de sorte, puis en remplaçant dans toutes les contraintes des $\alpha$-clauses le terme $t$ par $f(t)$ (comme $f$ est l'unique symbole de la sorte $\mathtt{s}'$, pour toute interprétation $\mathcal{I}$, $\mathcal{I}(\alpha)$ est de la forme $f(\dots)$). Ces restrictions nous permettent de garantir que tous les $\mathcal{S}_I$-termes apparaissant dans les contraintes sont de la forme $f_1(\dots(f_n(x))\dots)$, où $n > 0$ et $x$ une $\mathcal{S}_I$-variable, ce qui simplifie certaines définitions.

**Définition 81** Une $\alpha$-clause $C$ est dite $\mathcal{S}_I$-*plate* si tout $\mathcal{S}_I$-terme de $C$ est une variable. $\Diamond$

Nous introduisons dans la définition suivante une fonction $\text{succ}_\succ$ qui représente, d'une certaine manière, la fonction successeur par rapport à $\succ$ c'est-à-dire si $t$ est un $\mathcal{S}_I$-terme alors $\text{succ}_\succ(t)$ est le plus petit terme (modulo $\succ$) tel que $\text{succ}_\succ(t) \succ t$.

**Définition 82** On note $\text{succ}_\succ$ la fonction partielle telle que $\text{succ}_\succ(t) \stackrel{\text{def}}{=} f_1(\ldots(f_{n-1}(x)))$ si $t$ est de la forme $f_1(\ldots(f_n(x))\ldots))$ pour une certaine variable $x$ ($\text{succ}_\succ$ est indéfini sinon).

**Exemple 83**

$$\text{succ}_\succ(f(g(x))) = \text{succ}_\succ(f(h(x))) = f(x)$$
$$\text{succ}_\succ(f(x)) = \text{succ}_\succ(g(x)) = x$$

$\text{succ}_\succ(x)$ n'est pas défini. ♣

Nous avons vu dans le cas des $n$-clauses que l'espace de recherche $S$ est hiérarchisé en utilisant la notion de rang. Nous allons maintenant donner une définition plus générale de la notion de *rang* afin de l'appliquer aux $\alpha$-clauses.

**Définition 84** Le *rang* d'une $\alpha$-clause $C$ est définie comme suit :

- Si $C$ est de la forme $[D \mid \alpha \simeq i]$ et $D$ est $\mathcal{S}_I$-plate alors $\text{rang}(C) \stackrel{\text{def}}{=} i$.
- Si $C$ est de la forme $[D \mid \alpha \simeq i]$ et $D$ n'est pas $\mathcal{S}_I$-plate alors $\text{rang}(C) \stackrel{\text{def}}{=} \text{succ}_\succ(i)$. [1].

Si $C$ ne contient pas de paramètre $\alpha$ (la contrainte est à $\mathtt{V}$) alors $\text{rang}(C) \stackrel{\text{def}}{=} \bot$. $\Diamond$

Nous gardons la notation, introduite pour les $n$-clauses, pour les ensembles de $\alpha$-clauses de même rang c'est-à-dire si $S$ est un ensemble de $\alpha$-clauses et $i$ un $\mathcal{S}_I$-terme alors $S[i]$ (respectivement $S[\bot]$) est le sous-ensemble de $\alpha$-clauses de $S$ de rang $i$ (de rang $\bot$).

**Exemple 85** Nous présentons dans le tableau suivant la valeur du rang pour chaque $\alpha$-clause $C$.

| $C$ | $\text{rang}(C)$ |
|---|---|
| $p(x) \simeq \mathtt{V} \vee \ldots$ | $\bot$ |
| $[p(x) \simeq \mathtt{V} \mid \alpha \simeq f(x)]$ | $f(x)$ |
| $[p(g(x)) \simeq \mathtt{V} \mid \alpha \simeq f(g(x))]$ | $f(x)$ |
| $[q(f(x)) \mid \alpha \simeq f(x)]$ | $x$ |
| $[q(x) \mid \alpha \simeq x]$ | $x$ |
| $\alpha \not\simeq g(x)$ | $g(x)$ |

♣

1. Cette définition nous permet de préserver le rang lors d'une instansiation c'est-à-dire nous voulons que les clauses de la forme $[p(g(x)) \simeq \mathtt{V} \mid \alpha \simeq f(g(x))]$ et $[p(x) \simeq \mathtt{V} \mid \alpha \simeq f(x)]$ soient de même rang

L'incomplétude de la procédure de preuve est liée au fait qu'il n'est pas toujours possible d'appliquer le théorème 77 (chapitre 5). En effet, il existe une infinité de clauses possibles, même modulo un décalage, donc rien ne permet de garantir qu'un cycle pourra être éventuellement détecté. Afin de pallier ce problème, une première idée consiste à imposer des conditions supplémentaires permettant de garantir que l'ensemble de clauses correspondant à toute valeur donnée du paramètre est fini. Cette condition est toutefois assez restrictive (elle implique aussi la terminaison). Par conséquent, nous allons la raffiner en appliquant la détection de cycles uniquement sur un sous-ensemble de clauses, obtenu en restreignant l'espace de recherche à un ensemble fini $\mathfrak{F}$. Nous allons montrer qu'il est possible de définir cet ensemble $\mathfrak{F}$ de manière à ce que l'existence d'un cycle soit toujours garantie, si l'ensemble est insatisfaisable. Cela n'implique cependant pas la terminaison (ni la décidabilité), car, dans le cas où l'ensemble est satisfaisable pour une certaine valeur du paramètre, le calcul peut engendrer un ensemble infini de clauses correspondant à cette valeur du paramètre.

Nous supposerons dans ce chapitre et le suivant que les $\alpha$-clauses sont construites sur une certaine classe syntactique de clauses $\mathfrak{C}$ (c'est-à-dire pour toute clause $[C \mid \mathcal{X}]$, $C$ est dans $\mathfrak{C}$). Nous définissons la classe $\mathfrak{F}$ comme les $\alpha$-clauses $\mathcal{S}_I$-plates qui peuvent interagir avec les $\alpha$-clauses non plates. Nous montrerons par la suite que ces clauses sont suffisantes pour garantir l'existence d'un cycle.

**Définition 86** On note $\mathfrak{F}$ l'ensemble des clauses $\mathcal{S}_I$-plates $C \in \mathfrak{C}$ tel que il existe deux clauses $D, E \in \mathfrak{C}$ vérifiant $C, D \vdash E$ et $D$ n'est pas $\mathcal{S}_I$-plate. $\Diamond$

Si $S$ est un ensemble de $\alpha$-clauses, on note $\mathfrak{F}(S)$ l'ensemble des $\alpha$-clauses $[C \mid \mathcal{X}]$ où $C \in \mathfrak{F}$.

## 6.2 Classe admissible

Nous présentons maintenant les conditions sémantiques suffisantes pour assurer la complétude :

**Définition 87** La classe $\mathfrak{C}$ est *admissible* ssi elle vérifie les conditions suivantes :

- ($c_1$) $\mathfrak{C}$ est stable pour la superposition c'est-à-dire si $S \vdash C$ et $S \subseteq \mathfrak{C}$ alors $C \in \mathfrak{C}$.
- ($c_2$) Chaque clause de $\mathfrak{C}$ contient exactement une $\mathcal{S}_I$-variable. En plus, si $C$ et $D$ contiennent, respectivement, les $\mathcal{S}_I$-variables $x$ et $y$ et si une inférence est appliquée aux clauses $C$ et $D$ avec l'unificateur $\sigma$, alors $\sigma(x)$ est $x$, $y$ ou de la forme $f(y)$, pour une certaine fonction $f$. De plus, si $\sigma(x) = f(y)$ alors $C$ est $\mathcal{S}_I$-plate et $D$ n'est pas $\mathcal{S}_I$-plate. De la même manière, si $C$ contient une $\mathcal{S}_I$-variable $x$ et si une inférence unaire est appliquée à $C$ avec l'unificateur $\sigma$ alors $\sigma(x) = x$.

($c_3$) $\mathfrak{F}$ est finie (à un renommage de variables près). $\Diamond$

La condition ($c_1$) est naturelle, elle nous assure que toutes les $\alpha$-clauses générées à partir d'un ensemble de clauses de $\mathfrak{C}$, par superposition restent dans $\mathfrak{C}$. La condition ($c_2$) contrôle les inférences de manière à ce que la profondeur des $\mathcal{S}_I$-termes n'augmente pas arbitrairement. La condition ($c_3$) assure que le nombre de $\alpha$-clauses de $\mathfrak{F}(S)$ de même rang est fini. La condition ($c_2$) implique que l'espace de recherche a une structure hiérarchique par rapport au rang c'est-à-dire pour tout terme $t \succeq s$, chaque $\alpha$-clause de rang $s$ est nécessairement dérivée de $\alpha$-clauses de rang $t$ (avec les clauses de rang $\bot$). Ce résultat s'applique aussi pour $\mathfrak{F}$ c'est-à-dire nous allons montrer que $t \succeq s \Rightarrow S[\bot] \cup \mathfrak{F}(S[t]) \vdash_\delta S[\bot] \cup \mathfrak{F}(S[s])$. Enfin, grâce à la condition ($c_3$), il existe un nombre fini d'ensembles de clauses $\mathfrak{F}(S[t])$ (à un renommage près), ce qui garantit que la détection de boucles pourra éventuellement s'appliquer.

Avant de présenter le théorème qui prouve la complétude, nous allons montrer quelques résultats intermédiaires qui portent essentiellement sur l'évolution du rang lors des inférences.

**Proposition 88** *Si $P_1, P_2, C$ sont des $\alpha$-clauses telles que $P_1, P_2 \vdash C$ alors rang$(C) = \bot$ ssi rang$(P_1)$ = rang$(P_2) = \bot$.*

PREUVE. D'après le calcul de superposition pour les $\alpha$-clauses, la contrainte de $C$ est la conjonction des contraintes de $P_1$ et $P_2$.

— Si rang$(C) = \bot$ alors la contrainte de $C$ est à `V` et donc celles de $P_1$, $P_2$ sont aussi à `V` et donc rang$(P_1)$ = rang$(P_2) = \bot$.
— L'autre sens est trivial, si les contraintes de $P_1$ et $P_2$ sont à `V` alors leur conjonction (ce qui correspond à la contrainte de $C$) est à `V`, et donc rang$(C) = \bot$. ∎

**Lemme 89** *Si $P_1, P_2, C$ sont des $\alpha$-clauses telles que $P_1, P_2 \vdash C$ et $P_1, P_2$ sont $\mathcal{S}_I$-plate alors $C$ est $\mathcal{S}_I$-plate et rang$(P_i) \in \{\bot,$ rang$(C)\}$ (à un renommage près).*

PREUVE. Comme $P_1, P_2$ sont $\mathcal{S}_I$-plate, par la condition ($c_2$), l'unificateur le plus général est un renommage. Donc $C$ est $\mathcal{S}_I$-plate.

— Si rang$(C) = \bot$ alors par la proposition 88, rang$(P_1)$ = rang$(P_2) = \bot$.
— Si rang$(C) \neq \bot$ alors $C$ est de la forme $[C' \mid \alpha \simeq t]$. Par définition, il existe $i \in \{1, 2\}$ tel que $P_i$ est de la forme $[C'' \mid \alpha \simeq t']$, où $t'$ doit être un renommage de $t$. Nous pouvons déduire que rang$(C)$ = rang$(P_i)$. ∎

**Lemme 90** *Si $P_1, P_2, C$ sont des $\alpha$-clauses telles que $P_1, P_2 \vdash C$ et $C$ n'est pas $\mathcal{S}_I$-plate alors rang$(P_i) \in \{\bot,$ rang$(C)\}$ (à un renommage près).*

PREUVE. Nous avons deux cas :

— Si $\text{rang}(C) = \bot$ alors, par la proposition 88, $\text{rang}(P_1) = \text{rang}(P_2) = \bot$.

— Si $\text{rang}(C) \neq \bot$ alors comme $C$ n'est pas $\mathcal{S}_I$-plate, $C$ est de la forme $[D \mid \alpha \simeq t]$, où $t$ est un terme tel que $D \in \mathfrak{C}$ contient un $\mathcal{S}_I$-terme de la forme $f(x)$. D'après le lemme 89, il existe $i \in \{1, 2\}$ tel que $P_i$ n'est pas $\mathcal{S}_I$-plate. $P_i$ est de la forme $[D_i \mid t_i]$ où $t = t_i\sigma$ et $\sigma$ l'unificateur considéré. Par $(c_3)$, nous avons $t = t_i$ (à un renommage près). Donc $\text{rang}(P_i) = \text{rang}(C) = \text{succ}_\succ(t)$. De plus, si nous considérons l'autre clause parente $P_j$ (avec $j \neq i$), nous aurons deux cas :

  — $P_j$ est $\mathcal{S}_I$-plate et dans ce cas soit $P_j$ est de rang $\bot$, soit par $(c_2)$, $P_j$ est de la forme $[D_j \mid \text{succ}_\succ(t)]$ (où $D_j \in \mathfrak{C}$ et est $\mathcal{S}_I$-plate) et donc $\text{rang}(P_j) = \text{rang}(C) = \text{succ}_\succ(t)$.

  — $P_j$ n'est pas $\mathcal{S}_I$-plate et dans ce cas $P_j$ est soit de rang $\bot$, soit par $(c_2)$, $P_j$ est de la forme $[D_j \mid t]$ (où $D_j \in \mathfrak{C}$ et n'est pas $\mathcal{S}_I$-plate), donc $\text{rang}(P_j) = \text{rang}(C) = \text{succ}_\succ(t)$. ∎

**Lemme 91** *Si $P_1, P_2, C$ sont des $\alpha$-clauses telles que $P_1, P_2 \vdash C$ alors rang$(P_i) \in \{\bot, \text{rang}(C), \text{succ}_\succ(\text{rang}(C))\}$ (à un renommage près).*

PREUVE. Si $C$ n'est pas $\mathcal{S}_I$-plate alors, d'après le lemme 90, $\text{rang}(P_i) \in \{\bot, \text{rang}(C)\}$. Si $C$ est $\mathcal{S}_I$-plate et $P_1, P_2$ sont $\mathcal{S}_I$-plate alors, d'après le lemme 89, $\text{rang}(P_i) \in \{\bot, \text{rang}(C)\}$. Il nous reste donc à traiter le cas où $C$ est $\mathcal{S}_I$-plate et $P_1, P_2$ ne le sont pas c'est-à-dire $\exists i \in \{1, 2\}$ tel que $P_i$ n'est pas $\mathcal{S}_I$-plate. Nous avons donc les cas suivants :

— Si $\text{rang}(C) = \bot$ alors, d'après la proposition 88, $\text{rang}(P_1) = \text{rang}(P_2) = \bot$.

— Si $\text{rang}(C) \neq \bot$ alors $C$ est de la forme $[D \mid \alpha \simeq t]$ et par la condition $(c_2)$, $P_i$ est soit de la forme $[D_i \mid \alpha \simeq t]$ ou de la forme $[D_i \mid \alpha \simeq \text{succ}_\succ(t)]$ (ou de rang $\bot$) :

  — Dans le cas où $P_i$ est de la forme $[D_i \mid \alpha \simeq t]$, si $P_i$ est $\mathcal{S}_I$-plate alors $\text{rang}(P_i) = t$. Si $P_i$ n'est pas $\mathcal{S}_I$-plate alors $\text{rang}(P_i) = \text{succ}_\succ(t)$, et donc pour résumer $\text{rang}(P_i) \in \{\text{rang}(C), \text{succ}_\succ(\text{rang}(C))\}$.

  — Dans le cas où $P_i$ est de la forme $[D_i \mid \alpha \simeq \text{succ}_\succ(t)]$, par $(c_2)$, $P_i$ est nécessairement $\mathcal{S}_I$-plate, donc $\text{rang}(P_i) = \text{succ}_\succ(t) = \text{rang}(C)$. ∎

Nous allons maintenant présenter le lemme clef, l'idée est que chaque clause de rang supérieur à un rang donné $r$, au sens de $\succ$, peut être dérivée par un ensemble de $\alpha$-clauses $\mathcal{S}_I$-plates de rang $r$.

**Lemme 92** *Soit $r$ un terme. Soit $S$ un ensemble de $\alpha$-clauses tel que $D \in S$ ssi rang$(D) = \bot$ ou rang$(D) \succ r$. Soit $C$ une $\alpha$-clause telle que $S \vdash_\delta C$ et $r \succ$ rang$(C)$. Il existe un ensemble de $\alpha$-clauses $\mathcal{S}_I$-plates $S'$ de rang $r$ tel que $S \vdash_\delta S'$ et $S[\bot] \cup S' \vdash_\delta C$.*

PREUVE. Soit $C$ une clause déductible de $S$ et qui satisfait les deux hypothèses d'induction suivantes :

1. $r \succ \text{rang}(C)$
2. $r = \text{rang}(C)$ et $C$ n'est pas $\mathcal{S}_I$-plate

Soit $S'$ un ensemble de $\alpha$-clauses qui satisfait les propriétés du lemme. Nous montrons que $S' \cup S[\bot] \vdash C$ par induction sur la dérivation qui génère $C$. Comme $\text{rang}(C) \preceq r$ nous avons $C \notin S$ donc $C$ est dérivée de deux $\alpha$-clauses $P_1, P_2$. On montre que $S' \cup S[\bot] \vdash P_1, P_2$ et on déduit que $S' \cup S[\bot] \vdash C$. Nous allons maintenant réaliser une étude de cas sur les différentes valeurs que peut prendre le rang des $P_i, i \in \{1, 2\}$ :

— Si $\text{rang}(P_i) = \bot$ alors par la proposition 88, $S[\bot] \vdash P_i$.
— Si $r \succ \text{rang}(P_i)$, pour un certain $i \in \{1, 2\}$ alors la condition 1 est satisfaite. Comme la dérivation qui génère $P_i$ est de taille inférieure à la dérivation qui génère $C$, nous avons donc $S' \cup S[\bot] \vdash P_i$.
— Si $\text{rang}(P_i) \succeq r$, pour un certain $i \in \{1, 2\}$ alors nous montrons que $\text{rang}(P_i) = r$.
  — Si $C$ satisfait la condition 1 c'est-à-dire $r \succ \text{rang}(C)$, alors, d'après le lemme 91, $\text{rang}(P_i) = \text{succ}_{\succ}(\text{rang}(C)) = r$ car $\text{rang}(P_i) \succ r$.
  — Si $C$ satisfait la condition 2 c'est-à-dire $\text{rang}(C) = r$ et $C$ n'est pas $\mathcal{S}_I$-plate alors, par le lemme 90, $\text{rang}(P_i) = \text{rang}(C) = r$.

  Comme $\text{rang}(P_i) = r$, si $P_i$ n'est pas $\mathcal{S}_I$-plate alors $P_i$ vérifie la condition 2, comme la dérivation qui génère $P_i$ est plus petite que la dérivation qui génère $C$, on déduit que $S' \cup S[\bot] \vdash P_i$. Si $P_i$ est $\mathcal{S}_I$-plate, on montre que $P_i \in S'$. Si $P_j$, où $j = 3 - i$, n'est pas $\mathcal{S}_I$-plate alors il existe une clause $D = P_j$ telle que $P_i, D \vdash C$ ce qui veut dire $P_i \in \mathfrak{F}$, donc $P_i \in S'$. Si $P_j$ est $\mathcal{S}_I$-plate alors par le lemme 90, $C$ est $\mathcal{S}_I$-plate, $\text{rang}(P_i) = \text{rang}(C)$ et $C$ vérifie la condition 1 et $P_i$, $\forall i \in \{1, 2\}$ aussi, donc $S' \cup S[\bot] \vdash P_i$. ∎

Nous allons par la suite utiliser ce lemme pour raffiner la détection de boucles. En effet, il nous permet de restreindre l'application de la détection de boucles aux $\alpha$-clauses $\mathcal{S}_I$-plates, ce qui réduit le nombre de clauses à considérer.

Nous allons maintenant présenter le théorème principal qui formalise le résultat de complétude.

**Théorème 93** *Soit $\mathfrak{C}$ une classe admissible et soit $S$ un ensemble de $\alpha$-clauses de $\mathfrak{C}$. Si $S$ est saturé et insatisfaisable alors ou bien $\square \in S$ ou alors $\alpha \not\approx x \in S$.*

PREUVE. Comme $\mathfrak{C}$ est admissible, $\mathfrak{F}$ est finie, donc le nombre de sous-ensembles de $\mathfrak{F}$ est aussi fini, à un renommage des variables près. Soit $k$ un entier quelconque strictement supérieur au nombre de sous-ensembles de $\mathfrak{F}$. Soit $S'$ l'ensemble des $\alpha$-clauses dans $S$ de rang $r$ tel que $\text{size}(r) \leq k$. Soit $t$ un terme fermé de la même

sorte que $\alpha$. D'après le lemme 97, $S$ contient une clause qui subsume $\alpha \not\simeq t$. Si $\text{size}(t) \leq k$ alors cette clause appartient à $S'$. Supposons que $\text{size}(t) > k$. Donc $t$ est de la forme $f_1(f_2(\ldots(f_n(a))\ldots))$, où $n \geq k$. Soit $t_i = f_1(f_2(\ldots(f_i(x))\ldots))$ (avec $i \in [1, n]$). Nous avons $t_1 \succ t_2 \succ \ldots \succ t_n \succ t$. Par définition, pour tout terme $r$ $\mathfrak{F}(S'[r])$ est l'ensemble de clauses de la forme $[C \mid \alpha \simeq r']$ (à un renommage de variables près), où $C \in \mathfrak{F}$ et soit $r' = r$ ou bien $r = \text{succ}^{\prec}(r')$.
D'après le principe des tiroirs[2], comme $n \geq k$, il existe deux indices $i < j$ tels que la partie clausale de $\mathfrak{F}(S'[t_i])$ est identique à celle de $\mathfrak{F}(S'[t_j])$. Dans ce cas, $\mathfrak{F}(S'[t_i])$ et $\mathfrak{F}(S'[t_j])$ sont identiques à un décalage près, de la forme $x \to f_{i+1}(\ldots(f_j(x))\ldots)$, et nous devons avoir $\mathfrak{F}(S'[t_j]) = \mathit{shift}(S'[\mathfrak{F}(t_i)], t_i, \theta)$, où $\theta = \{x \to f_{i+1}(\ldots(f_j(x))\ldots)\}$.
Soit $\mathcal{S}'$ l'ensemble contenant tous les ensembles de clauses $\mathfrak{F}(S'[t_j])$ (où $t$ est un terme tel que $\text{size}(t) > k$). De la même manière, soit $\mathcal{S}$ l'ensemble contenant tout les ensembles de clauses $\mathfrak{F}(S'[t_j])$ et tout les ensembles de clauses $\{\alpha \neq t\}$, pour tout $t$ tel que $\text{size}(t) \leq k$. Comme $\mathfrak{F}$ est finie, ces ensembles sont aussi finis, et doivent être de la forme $\mathcal{S} = \{\mathfrak{F}(S'[t_i^l]) \mid l \in [1, m]\} \cup \{\alpha \not\simeq t \mid \text{size}(t) \leq k\}$, et $\mathcal{S}' = \{\mathfrak{F}(S'[t_j^l]) \mid l \in [1, m]\} \cup \{\alpha \not\simeq t \mid \text{size}(t) \leq k\}$.. Par définition, $S'[t_i^l]$ est un $t_i^l$-ensemble et les termes $t_i^1, \ldots, t_m^l$, avec les termes fermés de taille inférieure ou égale à $k$, sont couvrant. En plus, par définition de $\mathcal{S}'$, pour tout les termes fermés $s$, il existe un ensemble $\mathfrak{F}(S'[t_j^l])$ tel que $t_j^l \succeq s$. D'après le lemme 92, comme $t_i^l \succ t_j^l$, nous avons $S[\bot] \cup \mathfrak{F}(S'[t_i^l]) \vdash S[\bot] \cup \mathfrak{F}(S'[t_j^l])$. Pour finir, comme $\mathfrak{F}(S'[t_i])$ et $\mathfrak{F}(S'[t_j])$ sont identiques à un décalage près de la forme $\theta_l : x \to f_{i+1}(\ldots(f_j(x))\ldots)$, nous avons donc $S[\bot] \cup \mathfrak{F}(S'[t_j^l]) = \mathit{shift}(S[\bot] \cup \mathfrak{F}(S'[t_i^l]), t_i, \theta_l)$, avec $t_j^l = t_i^l\theta_l$. Puisque $s \preceq t_j^l$ nous avons $s \preceq t_i^l\theta_l$. Toutes les conditions du théorème 77 sont vérifiées, on peut donc dériver de $S'$ une clause de la forme $\alpha \not\simeq x$. De plus $S$ est saturé, donc cette clause est redondante par rapport à $S$. ∎

## 6.3 Détection de satisfaisabilité

Dans la logique clausale standard, la satisfaisabilité d'un ensemble de clauses $S$ peut être établie par saturation, dans le cas où l'ensemble de clauses dérivé de $S$ est fini et ne contient pas la clause vide $\square$. Ce qui n'est pas faisable dans le cas des $\alpha$-clauses, sauf dans certains cas triviaux où par exemple $\alpha$ ne joue pas un rôle dans la preuve. Cela est dû au fait que le rang augmente indéfiniment durant les inférences. Néanmoins nous pouvons construire le test de satisfaisabilité suivant, qui utilise une sorte de *saturation partielle*.

2. Le principe des tiroirs de Dirichlet stipule que si $n$ chaussettes se trouvent dans $k$ tiroirs et que $n > k$ alors il y a des tiroirs qui contiennent plus d'une chaussette. En mathématique cela correspond à une fonction $f : E \to F$ où $|E| > |F|$ ($E$ et $F$ sont deux ensembles finis) qui ne peut donc pas être injective.

**Définition 94** Un ensemble de $\alpha$-clauses est *saturé* par rapport à un terme (fermé) $t$ si pour toute clause $C$ telle que $S \vdash C$, une des deux conditions suivantes est vraie :

1. $C$ est redondante dans $S$.
2. $C$ est de la forme $[D \mid \alpha \simeq s]$, où $s$ et $t$ ne sont pas unifiables. $\Diamond$

Si un ensemble de $\alpha$-clauses est saturé par rapport à un certain terme $t$ et ne contient pas $\alpha \not\simeq t$, alors nous pouvons montrer que $S$ a un modèle dans lequel la valeur de $\alpha$ est $t$. Si $\mathfrak{C}$ est finie (c'est-à-dire le calcul de superposition termine sur les ensembles de $\alpha$-clauses de $\mathfrak{C}$), alors le nombre de $\alpha$-clauses de chaque rang est fini. Ce qui implique que pour tout terme fermé $t$, il est possible d'obtenir, à partir d'un ensemble initial quelconque $S$, un ensemble de clauses contenant $S$ et saturé par rapport à $t$. Si, en plus, il existe un terme $t$ tel que cet ensemble partiellement saturé, ne contient pas la clause $\alpha \not\simeq t$, nous pouvons détecter la satifaisabilité. Si un tel terme n'existe pas alors $S$ est insatisfaisable, donc nous pouvons assurer la terminaison dans les deux cas même si l'ensemble de $\alpha$-clauses est infini.

**Théorème 95** *Le problème de satisfaisabilité est décidable pour les ensembles de $\alpha$-clauses dont les parties clausales appartiennent à une classe admissible $\mathfrak{C}$ finie.*

## Preuve

Nous présentons, en premier lieu, une définition et deux lemmes intermédiaires, ensuite nous allons montrer le théorème. La définition suivante transforme un ensemble de $\alpha$-clauses en un ensemble de clauses standards, en fixant la valeur du paramètre $\alpha$ :

**Définition 96** Soit $S$ un ensemble de $\alpha$-clauses et soit $t$ un terme. On note $S_t^*$ l'ensemble de clauses (standards) obtenues de $S$ par :

- la suppression de toutes les clauses de la forme $[D \mid \alpha \simeq s]$, où $s$ et $t$ ne sont pas unifiables.
- le remplacement de toutes les clauses restantes de la forme $[D \mid \alpha \simeq t]$ par $D\sigma$ où $\sigma = \mathrm{mgu}(t, s)$. $\Diamond$

Le lemme suivant fait le lien entre $S$ et $S_t^*$ :

**Lemme 97** *Soit $S$ un ensemble de $\alpha$-clauses et soit $t$ un terme fermé de la même sorte que $\alpha$. Si $S_t^*$ est saturé, alors $S$ a un modèle $I$ tel que $I(\alpha) = t$. Pour cela, si $S$ est saturé et ne contient ni la clause vide ni aucune clause subsumant $\alpha \not\simeq t$, alors $S$ a un modèle $\mathcal{I}$ tel que $\mathcal{I}(\alpha) = t$*

PREUVE. (esquisse) Par définition, si $I(\alpha) = t$, alors $I \models S$ ssi $I \models S_t^*$. Donc la première partie de la preuve découle de la complétude de la superposition. Pour finir, on peut vérifier, facilement, que si $S$ est saturé alors $S_t^*$ l'est aussi, d'où le second résultat. ∎

**Lemme 98** *Si $S$ est saturé par rapport à un terme fermé $t$ et ne contient pas une clause de la forme $\alpha \not\simeq s$, où $s \succeq t$, alors $S$ est satisfaisable.*

PREUVE. (esquisse) Par définition, si $S$ est saturé par rapport à $t$, alors $S_t^*$ doit être saturé. Donc $S$ est satisfaisable.

Nous allons maintenant récapituler les résultats et présenter la preuve du théorème 95 :

PREUVE. Soit $S$ un ensemble de $\alpha$-clauses. Si $S$ est insatisfaisable, alors, d'après le théorème 93, une clause de la forme $\alpha \not\simeq x$ peut, éventuellement, être dérivée, nous pouvons alors déduire que $S$ est insatisfaisable et arrêter la recherche. Si $S$ est satisfaisable, alors il existe un terme $t$ tel que $S_t^*$ est satisfaisable. Par définition, $S_t^*$ ne dépend que des clauses de rang $\perp$ ou de rang $s \succeq t$, si $\mathfrak{C}$ est finie alors le nombre des ces $\alpha$-clauses est aussi fini, donc l'ensemble de $\alpha$-clauses est éventuellement saturé par rapport à $t$. D'après le lemme 98, on peut arrêter la recherche. ∎

L'exemple suivant illustre le cas où la satisfaisabilité peut être détectée.

**Exemple 99** Soit l'ensemble de clauses suivant :

$$\begin{array}{ll} 1 & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq x] \\ 2 & [p(x) \not\simeq \mathtt{V} \vee p(f(f(x))) \simeq \mathtt{V} \mid \mathtt{V}] \\ 3 & [p(a) \mid \mathtt{V}] \end{array}$$

Nous dérivons les clauses suivantes :

$$\begin{array}{ll} 4 & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq f(f(x))] \\ 5 & [\square \mid \alpha \simeq a] \\ 6 & [\square \mid \alpha \simeq f(f(a))] \end{array}$$

Une valeur possible pour $\alpha$ serait $f(a)$. En effet, les clauses correspondant au cas $n = f(a)$ (i.e. l'ensemble $S^*_{f(a)}$, où $S$ est l'ensemble contenant toutes les clauses ci-dessus) sont :

$$\{p(f(a)) \not\simeq \mathtt{V}, p(x) \not\simeq \mathtt{V} \vee p(f(f(x))) \simeq \mathtt{V}, p(a) \simeq \mathtt{V}\}$$

et il est clair que cet ensemble est saturé. ♣

# Chapitre 7

# Exemples de classes complètes

La définition 87 et le théorème 93 fournissent une caractérisation d'une classe d'ensemble de $\alpha$-clauses pour laquelle la superposition avec détection de boucles est complète. Cependant cette caractérisation est sémantique, au sens où aucun algorithme n'existe pour tester si les conditions de la définition 87 sont vérifiées ou non pour une certaine classe de $\alpha$-clauses. Dans ce chapitre, nous montrons l'applicabilité de ces résultats en donnant deux exemples de classes syntaxiques de $\alpha$-clauses qui sont admissibles.

Notons tout d'abord qu'il est très facile d'assurer que les conditions $(c_1)$ et $(c_2)$ de la définition 87 sont satisfaites : il suffit de considérer n'importe quelle classe syntaxique stable par superposition, avec au plus une variable de sorte dans $\mathcal{S}_I$ (il est facile de voir que cette dernière condition est préservée par les inférences pourvu que les littéraux sélectionnés contiennent toujours cette variable). Le point clef est d'assurer que la condition $(c_3)$ est vraie, à savoir que $\mathfrak{F}$ (c'est-à-dire les clauses $\mathcal{S}_I$-plates susceptibles d'interagir avec les clauses non $S_I$-plates) est fini. Pour définir la première classe admissible, nous allons imposer des conditions supplémentaires sur les clauses considérées, afin de garantir que la valeur des clauses de $\mathfrak{F}$, estimée suivant une certaine mesure de complexité (telle que la profondeur ou la taille des clauses) reste inférieure à une certaine constante. Pour la deuxième classe admissible nous allons considérer des clauses dont les seules variables sont des $\mathcal{S}_I$-variables, nous allons définir un nouveau calcul de superposition pour cette classe et montrer sa complétude et sa terminaison (ce qui entraîne immédiatement l'admissibilité).

## 7.1 Les clauses $\mu$-contrôlées

### 7.1.1 Fonction de complexité

Afin de rester le plus général possible, la mesure de complexité n'est pas supposée fixée. Nous nous contentons de donner les propriétés devant être satisfaites par cette mesure, ainsi que quelques exemples de fonctions satisfaisant les conditions requises. À noter que la définition que nous utilisons est proche de celle utilisée dans [56] pour prouver la terminaison de l'hyper-résolution sur certaines classes. À la différence de [56] notre but n'est pas de prouver que le calcul termine, mais uniquement qu'il n'engendre qu'un nombre fini de clauses possédant certaines propriétés.

**Définition 100** On note $\mu$ une *fonction de complexité* qui associe à chaque terme fermé un entier naturel. Nous supposons que pour tout $k \in \mathbb{N}$, l'ensemble des termes fermés de complexité inférieur à $k$ ne contenant aucun $S_I$-terme de profondeur strictement supérieure à 2 est fini. $\Diamond$

Les fonctions de complexité usuelles sont, par exemple, la profondeur (la longueur maximale des positions des termes standards, sans compter les $\mathcal{S}_I$-termes), ou la taille (nombre de positions qui ne sont pas des $\mathcal{S}_I$-termes). À noter que dans tous les cas les termes de type dans $\mathcal{S}_I$ peuvent être ignorés dans le calcul de complexité. La fonction $\mu$ est étendue aux atomes, littéraux et clauses comme suit :

- $\mu(t \not\simeq s) \stackrel{\text{def}}{=} \mu(t \simeq s) \stackrel{\text{def}}{=} \max(\mu(t), \mu(s))$.
- $\mu(\bigvee_{i=1}^{n} l_i) \stackrel{\text{def}}{=} \max\{\mu(l_i) \mid i \in [1, n]\}$.

On note $t =_\mu s$ si pour toute substitution $\sigma$ nous avons $\mu(t\sigma) = \mu(s\sigma)$. Pour tout entier naturel $\nu$, on note $t \leq_\mu^\nu s$ si pour toute substitution $\sigma$ telle que $\mu(t\sigma) > \nu$, nous avons $\mu(t\sigma) \leq \mu(s\sigma)$. Les relations $=_\mu$ et $\leq_\mu^\nu$ ne sont pas faciles à tester en général car l'ensemble des substitutions $\sigma$ est infini. Cependant, plusieurs algorithmes définis dans [56, 68] permettent de tester si $t =_\mu s$ ou $t \geq_\mu^\nu s$, pour différentes fonctions $\mu$. Par exemple, si $\mu$ est la profondeur du terme, alors $t =_\mu s$ ssi les conditions suivantes sont vérifiées :

- $\mu(t) = \mu(s)$.
- $\text{var}(s) = \text{var}(t)$.
- la profondeur maximale de chaque variable est la même dans $t$ et $s$.

### 7.1.2 Définition de la classe

Avant de présenter ce que nous appelons les clauses *$\mu$-contrôlées*, nous introduisons quelques notations et définitions nécessaires pour la suite.

**Définition 101** On suppose donnés deux ensembles de symboles de prédicats :

— Un ensemble de *prédicats de contrôle* noté $\Omega_c$.
— Un ensemble de *prédicats de $\mathcal{S}_I$-propagation* noté $\Omega_i$.

Un littéral est dit un *littéral de contrôle* (resp. *littéral de $\mathcal{S}_I$-propagation*) si son atome est de la forme $p(t_1, \ldots, t_n) \simeq \mathtt{V}$, où $p \in \Omega_c$ (resp. $p \in \Omega_i$) et $t_1$ est un $\mathcal{S}_I$-terme. Nous appelons les littéraux restants *les littéraux standards.* ◊

Intuitivement, les littéraux de propagation sont les seuls littéraux pouvant contenir la fonction successeur succ. Ce sont par conséquent les seuls susceptibles d'exprimer des propriétés dépendantes de plusieurs valeurs du paramètre (telles que par exemple des définitions inductives), et ainsi de "propager" des informations entre des clauses correspondant à des valeurs distinctes de $\alpha$. Les littéraux de contrôle sont des littéraux contenant toutes les variables non arithmétiques apparaissant dans les littéraux de propagation : ils permettent d'assurer que la complexité des termes apparaissant dans les littéraux de propagation reste inférieure à une certaine constante.

Soit $\mathcal{S}_p$ un ensemble de noms de sortes, vérifiant les propriétés suivantes :
— Pour tout symbole de prédicat $p \in \Omega_c \cup \Omega_i$ de la forme $\mathtt{s_1} \times \ldots \times \mathtt{s_n} \rightarrow \mathtt{bool}$, et pour tout $i \in [\![1, n]\!]$, nous avons $\mathtt{s_i} \in \mathcal{S}_\mathtt{p}$.
— Pour tout symbole de fonction de la forme $\mathtt{s_1} \times \ldots \times \mathtt{s_n} \rightarrow \mathtt{s}$, si $\mathtt{s} \in \mathcal{S}_\mathtt{p}$ alors $\forall i \in [\![1, n]\!], \mathtt{s_i} \in \mathcal{S}_\mathtt{p}$.

Intuitivement, les sortes incluses dans $\mathcal{S}_p$ regroupent les sortes des termes qui apparaissent dans les littéraux de contrôle et les littéraux de $\mathcal{S}_I$-propagation. Nous avons besoin d'isoler ce type de termes afin de garantir que l'application de la superposition à l'intérieur des littéraux de contrôle ou de propagation ne peut en modifier la complexité.

**Définition 102** Un ensemble de clauses $S$ est *$\mu$-contrôlé* ssi les conditions suivantes sont vérifiées :

1. Pour tout atome $t \simeq s$ qui apparaît dans une clause de $S$, si $t$ et $s$ sont de sorte incluse dans $\mathcal{S}_p$, alors $t =_\mu s$.
2. Il existe un entier naturel $\nu$ tel que, pour toute clause $C$, et pour tout littéral de $\mathcal{S}_I$-propagation ou de contrôle positif $L$ dans $C$, nous avons $L \leq_\mu^\nu C'$ où $C'$ est l'ensemble des littéraux de contrôle négatifs dans $C$.
3. Tous les littéraux sont $\mathcal{S}_I$-plats, sauf les littéraux de $\mathcal{S}_I$-propagation.
4. Toute clause dans $S$ contient, au plus, une $\mathcal{S}_I$-variable.
5. Les atomes dans $S$ sont ou bien de la forme $f(i, \vec{t}) \simeq g(i, \vec{s})$, (atome équationnel) ou bien de la forme $p(i, \vec{t}) \simeq \mathtt{V}$ (atome non-équationnel), où $i$ est un $\mathcal{S}_I$-terme de profondeur au plus 2. ◊

En particulier, chaque ensemble de clauses $\mathcal{S}_I$-plates, ne contenant pas d'atomes équationnels, est $\mu$-contrôlée (avec $\mathcal{S}_p = \Omega_c = \Omega_i = \emptyset$).

— La condition 1 nous assure que les inférences du calcul n'affectent pas la complexité des termes qui apparaissent dans les littéraux de contrôle et de $\mathcal{S}_I$-propagation.
— La condition 2 stipule que la complexité de chaque littéral de contrôle positif et de chaque littéral de $\mathcal{S}_I$-propagation est dominée par la complexité des littéraux de contrôle négatifs appartenant à la clause.
— La condition 3 assure que toutes les dépendances entre $\mathcal{S}_I$-termes distincts sont codés par les littéraux de propagation. Les littéraux restant représentent les relations entre termes construits sur le même $\mathcal{S}_I$-terme.
— La condition 5 interdit d'avoir des équations entre deux termes $s$ et $t$, où $s$ contient un $\mathcal{S}_I$-terme et $t$ ne contient aucun $\mathcal{S}_I$-terme, à l'exception des atomes non-équationnels. En particulier les équations entre variables sont interdites.

**Exemple 103** Dans cet exemple la fonction de complexité $\mu$ choisie est la *profondeur* des termes. Soit $\Omega_c = \{q\}$, $\Omega_i = \{p\}$. Les clauses :

$$p(i, f(c)) \simeq \mathtt{V}$$
$$p(i, x) \not\simeq \mathtt{V} \vee p(s(i), x) \simeq \mathtt{V} \vee q(i, x) \not\simeq \mathtt{V}$$
$$p_a(b) \simeq \mathtt{V}$$
$$[p(i, f(b)) \simeq \mathtt{V} \mid \alpha \simeq i]$$
$$q(i, x) \simeq \mathtt{V} \vee q(i, f(x)) \simeq \mathtt{V}\ q(i, f(c)) \simeq \mathtt{V}$$
$$p(i, x) \simeq \mathtt{V} \vee r(i, y) \not\simeq \mathtt{V} \vee r(i, f(y)) \simeq \mathtt{V}$$
$$r(i, a) \simeq \mathtt{V}$$

sont $\mu$-contrôlées, avec $\nu = 2$.

Notez que les conditions correspondant aux littéraux de contrôle assurent que pour tous les littéraux de la forme $p(i, t)$ ou $q(i, t)$ générés par le calcul, $t$ est de profondeur inférieure ou égale à 2, alors que les littéraux de racine $r$ sont de profondeur arbitraire.
Les clauses :

$$f(x) \simeq g(f(x))$$
$$q(i, x) \not\simeq \mathtt{V} \vee q(i, f(x)) \simeq \mathtt{V}$$
$$p(i, f(x)) \simeq \mathtt{V} \vee q(i, x) \not\simeq \mathtt{V}$$
$$p(i, x) \simeq \mathtt{V}$$
$$a(s(i)) \simeq b(i)$$
$$p(i) \simeq \mathtt{V} \vee p(j) \simeq \mathtt{V}$$
$$f(x) \simeq h(i, x)$$

ne sont pas $\mu$-contrôlées, parce qu'elles ne respectent pas les conditions $1, 2, 2, 2, 3, 4$ et 5, respectivement. En particulier, la condition 1 n'est pas respectée par la clause

$f(x) \simeq g(f(x))$, car $f(x)$ et $g(f(x))$ sont de profondeurs différentes. La condition 2 n'est pas respectée par la clause $\neg q(i,x) \vee q(i,f(x))$ (par rapport à la valeur de $\nu$), parce que la profondeur de $q(i,x)$ n'est pas assymptotiquement supérieure à celle de $q(i,f(x))$. ♣

Nous présentons maintenant la stratégie que nous allons appliquer :

**La stratégie de recherche**

Nous allons appliquer le calcul de superposition défini pour les $\alpha$-clauses, avec une stratégie définie de la manière suivante.

- Si une clause ne contient que des littéraux de contrôle et de $\mathcal{S}_I$-propagation alors les littéraux de contrôle négatifs sont sélectionnés (s'il en existe). Sinon les littéraux maximaux sont sélectionnés.
- L'ordre vérifie les conditions suivantes :
  - Si $i$ est un $\mathcal{S}_I$-terme alors l'inégalité $p(f(i),t_1,\ldots,t_n) > q(i,s_1,\ldots,s_m)$ est vraie pour tous les symboles $p,q,t_1,\ldots,t_n,s_1,\ldots,s_m$ et $f$.
  - Les littéraux de propagation de $\mathcal{S}_I$-terme $i$ sont supposés strictement inférieurs aux autres littéraux de même $\mathcal{S}_I$-terme $i$.

La deuxième condition sur l'ordre peut paraître un peu forte, car l'atome de $\mathcal{S}_I$-propagation peut contenir des variables qui n'apparaissent pas dans un atome standard. Cependant, nous pouvons facilement l'imposer en supposant que les termes qui sont à la racine dans les atomes standard sont d'une sorte particulière qui ne peut apparaître dans un atome de $\mathcal{S}_I$-propagation (nous pouvons à ce moment là supposer que ces termes sont strictement supérieurs à ceux contenus dans les atomes de $\mathcal{S}_I$-propagation).

**Théorème 104** *La classe des clauses $\mu$-contrôlées est une classe admissible.*

PREUVE. Nous montrons simultanément que les conditions $c_1, c_2$ et $c_3$ sont vérifiées. En particulier, nous démontrons que $c_3$ est vérifiée en montrant que les clauses de $\mathfrak{F}$ ne contiennent aucune variable exceptées les $\mathcal{S}_I$-variables et sont de complexité inférieure à $\nu$ (les deux conditions impliquent, clairement, que $\mathfrak{F}$ est fini à un renommage près). Nous considérons deux clauses $\mu$-contrôlées $C[t]_p$ et $u \simeq v \vee D$ et une clause $C[v]_p\sigma \vee D\sigma$ obtenue par superposition, à partir des deux premières clauses (la preuve pour les autres règles d'inférence est similaire).

- Supposons que $C[t]_p$ est $\mathcal{S}_I$-plate et que $u \simeq v \vee D$ n'est pas $\mathcal{S}_I$-plate. $u$ ne peut être $\mathcal{S}_I$-plate à cause de l'ordre utilisé pour restreindre les inférences (sinon $u$ ne serait pas maximal). Donc $u$ est de la forme $p(f(i),\vec{t})$, où $p \in \Omega_i$. Par conséquent, $t$ est de la forme $p(j,\vec{s})$ (comme $t$ et $s$ sont unifiables, ils doivent avoir la même racine). Cependant, comme $t$ est selectionné, $C[t]_p$ ne peut contenir ni des atomes standard (sinon $t$ ne serait pas maximal)

ni des littéraux de contrôle négatifs (sinon ce littéral serait selectionné). L'ensemble des littéraux de contrôle négatifs est donc vide, et d'après la condition 2 de la définition 102, nous avons $\mu(C[t]_p) \leq^{\nu}_{\mu} \square$. Cela implique que $C[t]_p$ est de complexité $\nu$ et ne contient aucune variable (à l'exception des $\mathcal{S}_I$-variable). Le même raisonnement s'applique si $u \simeq v \vee D$ est $\mathcal{S}_I$-plate et $C[t]_p$ ne l'est pas (dans ce cas $u \simeq v \vee D$ ne contient aucune variable à part les $\mathcal{S}_I$-variables et nous avons nécessairement $\mu(u \simeq v \vee D) \leq \nu$). Nous pouvons déduire que les clauses dans $\mathfrak{F}$ ne contiennent pas de variables (excepté les $\mathcal{S}_I$-variables) et sont de complexité inférieure ou égale à $\nu$.

— Comme la profondeur d'un $\mathcal{S}_I$-terme est au plus 2 et que, à cause des restrictions de l'ordre défini dans la stratégie, seuls les littéraux contenant les $\mathcal{S}_I$-termes les plus profonds sont éligibles à l'inférence, la condition $c_2$ est facile à vérifier. Notez que cela implique que le nombre de $\mathcal{S}_I$-variables n'augmente pas.
— Il reste à vérifier la première condition $c_1$ c'est-à-dire que $C[v]_p\sigma \vee D\sigma$ est $\mu$-contrôlé. Nous montrons que cette clause vérifie toutes les conditions de la définition 102.
  — Soit $t \simeq s$ un atome dans $C[v]_p\sigma \vee D\sigma$, où $t, s$ sont d'une sorte dans $\mathcal{S}_p$. Si $t \simeq s$ est de la forme $(t' \simeq s')\sigma$ où $t' \simeq s'$ apparaît dans l'une des clauses parentes, alors comme les clauses parentes sont $\mu$-contrôlée par hypothèse, nous avons nécessairement $t' =_{\mu} s'$ et donc $t =_{\mu} s$. Sinon, $t \simeq s$ est de la forme $(t'[v]_q \simeq s')\sigma$, où $t' \simeq s'$ apparaît dans l'une des clauses parentes et $t'|_q = u$. Cette fois encore, nous avons $t' =_{\mu} s'$. De plus, par définition de $\mathcal{S}_p$, $u$ et $v$ sont nécessairement d'une sorte incluse dans $\mathcal{S}_p$. Par conséquent, nous avons $u =_{\mu} v$ et donc $t' =_{\mu} t'[v]_q$. Pour cela nous déduisons $t'[v]_q\sigma =_{\mu} s'\sigma$.
  — Soit $L$ un littéral appartenant à $C[v]_p\sigma \vee D\sigma$ et qui est soit un littéral de $\mathcal{S}_I$-propagation ou un littéral de contrôle positif. Par définition, $L$ est obtenu à partir d'un littéral appartenant à l'une des clauses parentes, en appliquant la substitution $\sigma$ et en remplaçant (éventuellement) une occurrence du terme $u\sigma$ par $v\sigma$. Si $u$ n'apparaît pas dans $L$ alors $L = L'\sigma$ et donc $L =_{\mu} L'\sigma$. Si $u$ apparaît dans $L$ alors par définition de $\mathcal{S}_p$, $u$ doit être d'une sorte incluse dans $\mathcal{S}_p$, nous avons alors $u =_{\mu} v$ et donc la relation $L =_{\mu} L'\sigma$ est vérifiée dans les deux cas. Comme la clause parente contenant $L'$ est $\mu$-contrôlée, elle contient aussi une disjonction de littéraux de contrôle $M$ tel que $M \geq^{\nu}_{\mu} L'$. Donc $C[v]_p\sigma \vee D\sigma$ contient une disjonction de littéraux $M'$ obtenue à partir de $M\sigma$ en remplaçant $u\sigma$ par $v\sigma$. Si $u \simeq v$ est un littéral de contrôle, alors $D$ contient une disjonction de littéraux de contrôle négatifs $D'$ tels que $D' \geq^{\nu}_{\mu} u \simeq v$.

Nous avons alors $D'\sigma \geq^{\nu}_{\mu} M\sigma \geq^{\nu}_{\mu} L$. Sinon, si $u$ apparaît dans $M$ alors il est d'une sorte incluse dans $\mathcal{S}_p$. Pour cela nous avons $u =_{\mu} v$, d'où nous déduisons que $M' =_{\mu} M\sigma$ et donc $M' \geq^{\nu}_{\mu} L$.

— Supposons que $C[v]_p\sigma \vee D\sigma$ contient un littéral $L$ qui n'est pas de $\mathcal{S}_I$-propagation et qui n'est pas $\mathcal{S}_I$-plat. Ce littéral est obtenu à partir d'un littéral $L'$ appartenant à une des clauses parentes en appliquant la substitution $\sigma$ et en remplaçant le terme $u\sigma$ par $v\sigma$. $L'$ ne peut être un littéral de $\mathcal{S}_I$-propagation. Alors $L'$ est $\mathcal{S}_I$-plate, et donc $\sigma$ ne peut être plate et par la condition $c_2$, la clause parente ne contenant pas le littéral $L'$ ne peut être $\mathcal{S}_I$-plate. Cependant nous avons montré que la seule clause $\mathcal{S}_I$-plate qui peut interagir avec les clauses qui ne sont pas $\mathcal{S}_I$-plates ne contient que des littéraux de $\mathcal{S}_I$-propagation, cela contredit notre hypothèse de départ sur $L$.

— La condition 4 est une conséquence immédiate de $c_2$.

— La dernière condition est vérifiable de manière triviale, puisque la condition est vérifiée pour les deux parents il est évident qu'elle est préservée par remplacement et instantiation. ■

Intuitivement, les littéraux de $\mathcal{S}_I$-propagation codent les propriétés des termes de sorte dans $\mathcal{S}_I$ et les relations entre ces termes, tandis que les autres littéraux codent les propriétés des termes standards (correspondant à un $\mathcal{S}_I$-terme donné). L'utilisation des littéraux de contrôle nous assure que la taille des littéraux de $\mathcal{S}_I$-propagation est bornée alors que les littéraux standards peuvent être des littéraux arbitraires du premier ordre.

### 7.1.3 Exemples

Plusieurs classes concrètes peuvent être obtenues simplement en instanciant la mesure de complexité $\mu$. Ces classes vérifient les conditions 3, 4 et 5 qui ne dépendent pas de la mesure de complexité.

**Exemple 105** (La profondeur) Si la mesure de complexité est la profondeur des termes, la classe des clauses $\mu$-contrôlées correspond à une classe où :

1. Pour tout atome $t \simeq s$ qui apparaît dans une clause de $S$, si $t$ et $s$ sont de sorte incluse dans $\mathcal{S}_p$, alors toutes les instances de $t$ et $s$ sont de même profondeur.
2. La profondeur des littéraux de $\mathcal{S}_I$-propagation ou de contrôle positif $L$ dans une clause $C$, est assymptotiquement bornée par la profondeur des littéraux de contrôle négatifs dans $C$, c'est-à-dire que les variables apparaissant dans la première classe de littéraux doivent aussi apparaître à une profondeur plus grande ou égale dans au moins un littéral de la seconde classe.

**Exemple 106** (La taille) Si la mesure de complexité est la taille des termes (c'est-à-dire le nombre de symboles), la classe des clauses $\mu$-contrôlées correspond à une classe où :

1. Pour tout atome $t \simeq s$ qui apparaît dans une clause de $S$, si $t$ et $s$ sont de sorte incluse dans $\mathcal{S}_p$, alors toutes les instances de $t$ et $s$ sont de même taille.
2. La taille des littéraux de $\mathcal{S}_I$-propagation ou de contrôle positif $L$ dans une clause $C$, est assymptotiquement bornée par la taille des littéraux de contrôle négatifs dans $C$. On peut par exemple supposer que le nombre d'occurrences de chaque variable est strictement supérieur dans les littéraux du second type que dans ceux du premier type.

Toutes ces classes (où on a instancié la mesure de complexité) sont strictement plus expressives que la logique du premier ordre (qui correspond au cas où les ensembles $\Omega_c$ et $\Omega_i$ sont vides). Le théorème 93 assure que notre procédure de preuve est complète pour les ensembles de clauses $\mu$-contrôlées. Si, en plus, les clauses considérées appartiennent à une classe où la superposition termine (comme la classe monadique, le fragment gardé,. . . ) ou si tous les littéraux sont des littéraux de $\mathcal{S}_I$-propagation ou de contrôle alors le théorème 95 assure la terminaison et la décidabilité. Nous obtenons alors un résultat de complétude général pour les schémas de clauses du premier ordre et aussi des résultats de décidabilité pour les schémas construits sur des sous-classes décidables de la logique du premier ordre.

En particulier, il est facile de vérifier que chaque schéma régulier de la logique propositionnelle défini dans [6] et rappelé au chapitre 4 peut être exprimé comme un ensemble de clauses $\mu$-contrôlées, comme le montre la proposition suivante.

**Proposition 107** *Si Sc est un schéma de formules propositionnelles régulier alors nous pouvons construire un ensemble $\mathcal{A}$ de $\alpha$-clauses sat-équivalent à Sc et $\mu$-contrôlé.*

PREUVE. Nous supposons que $\mathcal{S}_I = \{\texttt{nat}\}$ (c'est-à-dire que les $\mathcal{S}_I$-termes sont des entiers naturels), que la mesure de complexité est la profondeur (avec $\nu = 2$), et enfin que tous les prédicats sont de $\mathcal{S}_I$-propagation c'est-à-dire que $\Omega_c = \emptyset$. Il est facile de vérifier que l'ensemble $\mathcal{A}$ est $\mu$-contrôlé après clausification car, puisque tous les atomes sont de la forme $p(t)$, où $t$ est de sorte `nat`, leur valeur suivant $\mu$ est toujours égale à 1. De plus, par construction, la profondeur des termes de type entier est au plus 2, et les axiomes ne contiennent qu'une seule variable de type `nat`.

D'après la proposition 107, les formules de la logique propositionnelle temporelle linéaire [70] peuvent aussi être codées avec des ensembles de clauses $\mu$-contrôlé

(voir [7] pour une traduction de LTL vers les schémas réguliers). Plusieurs propriétés des structures de données usuelles, qui sont définies inductivement, comme les listes et les arbres peuvent être écrites sous forme de clauses $\mu$-contrôlées. Par exemple, un arbre peut être noté comme une constante $\tau$ qui a un seul argument de type $\mathcal{S}_I$-terme [1], où $\tau(p)$ représente l'étiquette du nœud à la position $p$. Ensuite nous pouvons facilement coder le fait que certaines propriétés du premier ordre sont vérifiées pour toutes les étiquettes de l'arbre (ou par toutes les étiquettes de certaines positions, ou certains ensembles réguliers de positions). Cependant, nous ne pouvons pas exprimer le fait, par exemple, qu'un arbre est trié, car cela requiert l'utilisation d'atomes de la forme $\tau(p.0) \leq \tau(p) \leq \tau(p.1)$, qui utilisent plusieurs $\mathcal{S}_I$-termes dans un même terme hybride. Les relations entre des arbres distincts peuvent aussi être exprimées du moment qu'elles préservent la forme de l'arbre (par exemple nous pouvons établir qu'un arbre est obtenu de $\tau$ en appliquant une certaine fonction $f$ sur chaque nœud).

## 7.2 Les clauses équationnelles quasi-fermées

Un ensemble de clauses est dit *quasi-fermé* si toutes les variables qu'il contient sont des $\mathcal{S}_I$-variables. Le problème de satisfaisabilité est indécidable pour les clauses quasi-fermées car le test de satisfiabilité pour des schémas de clauses équationnelles fermées s'y ramène facilement, et ce problème est prouvé indécidable dans [9]. Dans cette section, on définit une classe de clauses quasi-fermées pour laquelle la satisfaisabilité est décidable.

**Définition 108** Un ensemble de clauses $S$ est *$\mathcal{S}_I$-fermé* si :
- Toute clause non vide de $S$ contient une seule variable $i$ et $i$ est une $\mathcal{S}_I$-variable.
- Tous les symboles de fonction ont un $\mathcal{S}_I$-terme comme un des arguments, ce $\mathcal{S}_I$-terme est de la forme $i$ ou $f(i)$ (où $i$ est une $\mathcal{S}_I$-variable).
- Tout terme $t$ contenu dans $S$ contient exactement un $\mathcal{S}_I$-terme. $\Diamond$

La troisième condition permet uniquement de simplifier les résultats, et n'entraine aucune perte de généralité (par exemple un terme $f(i, a(s(i)))$ qui contient deux $\mathcal{S}_I$-termes peut être écrit $f(i, a'(i))$ en ajoutant l'axiome $a(s(i)) \simeq a'(i))$ (voir Section 4.2.1 pour une généralisation de cette transformation). La différence principale entre les clauses $\mathcal{S}_I$-fermées et la classe plus générale des clauses quasi-fermées est que ces dernières peuvent contenir des symboles de fonction ne dépendant pas des $\mathcal{S}_I$-termes, par exemple $f(a(i)) \simeq b(i)$ est quasi-fermé mais non $\mathcal{S}_I$-fermé, tandis que $f(i, a(i)) \simeq b(i)$ est $\mathcal{S}_I$-fermé et quasi-fermé.

1. Il est clair que les positions peuvent être codées avec une signature monadique.

La classe des clauses $\mathcal{S}_I$-fermées n'est pas stable pour la superposition. Par exemple, à partir des équations $f(i, a'(i)) \simeq a(i)$ et $a(s(i)) \simeq a'(i)$ on génère $f(i, a(s(i))) \simeq a(i)$ qui n'est pas $\mathcal{S}_I$-fermée, si l'ordre est tel que $a(s(i))$ est inférieur à $a'(i)$. A noter que si $a(s(i))$ est supérieur à $a'(i)$ alors l'application de la superposition entre les équations $f(i, a(i)) \simeq a(i)$ et $a(s(i)) \simeq a'(i)$ produit $f(s(i), a'(i)) \simeq a(s(i))$ qui n'est pas $\mathcal{S}_I$-fermée. Nous allons donc utiliser une procédure de preuve différente, la *superposition à la racine*, qui présente les particularités suivantes. D'une part, les règles de superposition ne sont appliquées qu'à la racine des termes, et d'autre part, certaines des conditions permettant l'application de ces règles ne sont pas testées de manière statique lors de l'inférence, mais ajoutées aux clauses générées, sous forme de littéraux supplémentaires. Par exemple, la superposition à la racine est applicable entre les équations $f(i, a(i)) \simeq a(i)$ et $f(i, b(i)) \simeq a(i))$ et engendre la clause $a(i) \not\simeq b(i) \vee a(i) \simeq a(i)$. Le premier littéral correspond à une condition assurant l'égalité des sous-termes $a(i)$ et $b(i)$, permettant ainsi l'application de la règle de superposition standard. La correction de cette règle dérive de manière immédiate des axiomes de substitutivité.

Formellement, le calcul de superposition à la racine est défini par les règles d'inférence suivantes :

**La superposition à la racine :**

$$\frac{f(j, t_1, \ldots, t_n) \bowtie s \vee C,\ f(k, s_1, \ldots, s_m) \simeq u \vee D}{(u \bowtie s \vee \bigvee_{i=1}^{n}(t_i \not\simeq s_i) \vee C \vee D)\sigma}$$

où $\sigma = \mathrm{mgu}(i, k)$, $f(j, t_1, \ldots, t_n)$ est un terme maximal dans $f(j, t_1, \ldots, t_n) \bowtie s \vee C$ et $f(k, s_1, \ldots, s_m)$ est un terme maximal dans $f(k, s_1, \ldots, s_m) \simeq u \vee D$.

**La décomposition à la racine :**

$$\frac{f(j, t_1, \ldots, t_n) \not\simeq f(k, s_1, \ldots, s_n) \vee C}{\bigvee_{i=1}^{n}(t_i \not\simeq s_i) \vee C}$$

où $\sigma = \mathrm{mgu}(i, k)$, $f(j, t_1, \ldots, t_n)$ ou $f(k, s_1, \ldots, s_n)$ est maximal dans $f(j, t_1, \ldots, t_n) \not\simeq f(k, s_1, \ldots, s_n) \vee C$.

**La factorisation :**

$$\frac{f(j, t_1, \ldots, t_n) \simeq u \vee f(k, s_1, \ldots, s_n) \simeq v \vee C}{f(j, t_1, \ldots, t_n) \simeq u \vee u \not\simeq v \vee \bigvee_{i=1}^{n} t_i \not\simeq s_i \vee C}$$

où $\sigma = \text{mgu}(j,k)$, $f(j,t_1,\ldots,t_n)$ est maximal dans $f(j,t_1,\ldots,t_n) \simeq u \vee f(k,s_1,\ldots,s_n) \simeq v \vee C$.

À noter que dans les deux dernières règles, $\sigma$ est nécessairement l'identité, puisque les clauses ne contiennent qu'une seule variable et que tous les $S_I$-termes contiennent cette variable (donc $j$ et $k$ sont soit égaux, soit non-unifiables).

**Lemme 109** *Le calcul de superposition à la racine est complet pour la réfutation.*

PREUVE. Soit $S$ un ensemble de clauses saturé par rapport à la superposition à la racine. Supposons que $S$ ne contient pas la clause vide. On montre alors que $S$ est satisfaisable. Soit $S'$ l'ensemble des instances fermées des clauses de $S$. Il est facile de vérifier que $S'$ est aussi saturé par rapport à la superposition à la racine, et que $\square \notin S'$. Nous construisons une interprétation $\mathcal{I}$ comme suit : chaque terme $t = f(t_1,\ldots,t_n)$ est associé à une forme normale $\mathcal{I}(t)$, telle que $t =_{\mathcal{I}} s$ est vrai ssi $\mathcal{I}(t) = \mathcal{I}(s)$. La valeur de $\mathcal{I}(f(t_1,\ldots,t_n))$ est construite inductivement comme suit :

- Si $S'$ contient une clause de la forme $f(s_1,\ldots,s_n) \simeq u \vee \bigvee_{i=1}^{k} f(s_1,\ldots,s_n) \simeq v_i \vee C$ (avec éventuellement $k = 0$), où :
  - $f(s_1,\ldots,s_n) > u$, $\forall i \in [\![1,k]\!], u > v_i$ et tous les termes dans $C$ sont strictement inférieurs à $f(s_1,\ldots,s_n)$,
  - $\mathcal{I} \not\models C$, $\forall i \in [\![1,k]\!], \mathcal{I} \not\models u \simeq v_i$ et pour tout $i \in [\![1,n]\!]$, $t_i =_{\mathcal{I}} s_i$,

  alors $\mathcal{I}(t) \stackrel{\text{def}}{=} \mathcal{I}(u)$ (s'il existe plusieurs clauses de cette forme alors $u$ est le plus petit terme choisi parmi les termes correspondant aux clauses vérifiant les conditions ci-dessus).
- Sinon, $\mathcal{I}(f(t_1,\ldots,t_n)) \stackrel{\text{def}}{=} f(\mathcal{I}(t_1),\ldots,\mathcal{I}(t_n))$.

L'interprétation $\mathcal{I}$ est bien définie, puisque la valeur d'un terme $t$ ne dépend que des valeurs des termes qui sont strictement inférieurs à $t$. De plus il est clair que la substitutivité est satisfaite, puisque l'interprétation de $f(t_1,\ldots,t_n)$ ne dépend que de celle de $t_1,\ldots,t_n$. Supposons que $S'$ contient une clause $C$ telle que $\mathcal{I} \not\models C$. Supposons aussi, sans perte de généralité, que $C$ est la plus petite clause ayant cette propriété. Cela implique que toute clause contenue dans $S'$ et plus petite que $C$ est vraie dans $\mathcal{I}$. Par conséquent, toute clause redondante par rapport à $S'$ et plus petite que $C$ doit être vraie dans $\mathcal{I}$ (car, par définition, cette clause est une conséquence logique des clauses de $S'$ qui sont plus petites que $C$). $C$ est non vide, donc de la forme form $f(t_1,\ldots,t_n) \bowtie s \vee D$, où $f(t_1,\ldots,t_n) \bowtie s$ est le littéral maximal dans $C$ et $f(t_1,\ldots,t_n) > s$. Supposons que $f(t_1,\ldots,t_n) \bowtie s$ est négatif. Comme $\mathcal{I} \not\models f(t_1,\ldots,t_n) \bowtie s$, nous devons avoir $\mathcal{I}(f(t_1,\ldots,t_n)) = \mathcal{I}(s)$. D'après la définition de $\mathcal{I}$, nous avons deux possibilités :

- $s = f(s_1,\ldots,s_n)$ où $\forall i \in [\![1,n]\!], t_i =_{\mathcal{I}} s_i$. Alors en appliquant la décomposition à la racine, nous pouvons dériver : $\bigvee_{i=2}^{n} t_i \not\simeq s_i \vee D$, qui est strictement

inférieure à $C$ et fausse dans $\mathcal{I}$. Comme $S'$ est saturé, cette clause doit être redondante par rapport à $S'$, ce qui est impossible.

— Sinon, $f(t_1, \ldots, t_n)$ doit être réductible, donc $S'$ doit contenir une clause de la forme $f(s_1, \ldots, s_n) \simeq u \vee \bigvee_{i=1}^{k} f(s_1, \ldots, s_n) \simeq v_i \vee C$, vérifiant les conditions ci-dessus. En appliquant la superposition à la racine, nous dérivons : $u \bowtie s \vee \bigvee_{i=1}^{k} f(s_1, \ldots, s_n) \simeq v_i \vee \bigvee_{i=2}^{n} t_i \not\simeq s_i \vee C$. Comme $S'$ est saturé, cette clause doit forcément être redondante par rapport à $S'$. De plus, elle est strictement inférieure à $C$ et fausse dans $\mathcal{I}$. ∎

Supposons, maintenant, que $f(t_1, \ldots, t_n) \bowtie s$ est positif. Si $S'$ contient une clause différente de $C$ et vérifiant les conditions ci-dessus, alors la règle de superposition à la racine s'applique de la même manière que dans la deuxième possibilité (traitée ci-dessus). Si $D$ est de la forme $f(t_1, \ldots, t_n) \simeq s' \vee D'$, où $s =_{\mathcal{I}} s'$, alors par factorisation, nous dérivons la clause $f(t_1, \ldots, t_n) \simeq s \vee s \not\simeq s' \vee D'$, qui est strictement plus petite que $C$ et fausse dans $\mathcal{I}$, ce qui est impossible. Si $D$ n'est pas de cette forme, alors $C$ vérifie toutes les conditions ci-dessus, et d'après la définition de $\mathcal{I}$ nous avons $\mathcal{I}(f(t_1, \ldots, t_n)) = I(s)$ et donc $\mathcal{I} \models f(t_1, \ldots, t_n) \simeq s$, ce qui est impossible.

Les résultats du chapitre 6 sont énoncés spécifiquement pour le calcul de superposition, mais il est facile de vérifier qu'ils ne dépendent pas du calcul utilisé : n'importe quel calcul convient, pourvu qu'il soit correct et complet. Ils s'appliquent donc également à la superposition à la racine.

**Théorème 110** *La classe des clauses quasi-fermées est admissible. La superposition à la racine termine sur des ensembles de clauses quasi-fermées.*

PREUVE. Il est évident que les règles d'inférence ne dérivent que des clauses quasi-fermées (car aucune autre variable, mise à part les $\mathcal{S}_I$-variables, ne peut être introduite). D'après la définition de la classe, tous les symboles de fonction contiennent un argument de sorte $\mathcal{S}_I$, qui est de la forme $i$ ou $f(i)$, où la $\mathcal{S}_I$-variable $i$ est la seule variable dans la clause. Par conséquent, l'unificateur $\sigma$ est de la forme $\{i \mapsto i'\}$ ou $\{i \mapsto f(i')\}$, où $i$ et $i'$ sont deux variables contenues dans les clauses parentes. En particulier, le nombre de $\mathcal{S}_I$-variables ne peut augmenter.

Par définition de l'ordre, aucune inférence ne peut être appliquée à un terme d'argument $i$, excepté si la clause est $\mathcal{S}_I$-plate. Donc, la profondeur des $\mathcal{S}_I$-termes ne peut augmenter, et la condition $c_2$ est satisfaite.

Finalement, il est clair que les inférences ne peuvent pas augmenter la profondeur des termes, donc le nombre de clauses est fini (à un renommage de $\mathcal{S}_I$-variables près). En particulier, la condition $c_3$ est vérifiée.

Il est important de mentionner que la classe des clauses quasi-fermées n'est pas stable par superposition à la racine. Par exemple les clauses $f(a(i)) \not\simeq c$ et

$f(b(i)) \simeq c$ engendrent $a(i) \not\simeq b(i')$ (notons que la variable $i$ doit être renommée pour que les ensembles de variables de clauses parentes soient disjoints).

# Chapitre 8

# Travaux voisins

Nous présentons dans ce chapitre un bref survol des travaux voisins en démonstration automatique, plus précisément dans le domaine des preuves par induction, du calcul de superposition avec contrainte et des schématisations de termes ou de formules.

## 8.1 Mécanisation des preuves par induction

La mécanisation des preuves par induction a fait l'objet d'un nombre important de travaux. Ces travaux s'appliquent naturellement à la preuve de schéma, qui est un cas particulier de preuve par induction. Nous proposons dans ce chapitre une présentation synthétique des principales approches existantes ainsi qu'une comparaison avec nos travaux. On pourra consulter [34, 29, 83] pour une présentation plus détaillée et plus générale de ce domaine de recherche.

L'induction est une technique de preuve très puissante permettant de raisonner sur des ensembles infinis et dénombrables, tels que les entiers ou les listes. Dans le cadre de la logique du premier ordre, on distingue usuellement trois types d'approches : l'induction usuelle qui est basée sur un schéma d'induction explicite (i.e. la ou les hypothèses d'induction sont fixées a priori), codé soit sous forme d'axiomes soit comme une règle d'inférence, l'induction implicite basée sur des procédures de réduction et l'induction sans induction, qui se ramène à un test de satisfaisabilité classique (preuve par consistance).

### 8.1.1 Induction explicite

Les approches basées sur l'induction explicite (voir [29] [30] et [16]) sont le plus souvent utilisées par les assistants de preuve, et on y emploie des heuristiques efficaces pour dériver automatiquement les schémas d'induction appropriés. Par

exemple, si on considère l'ensemble de clauses suivant (présenté en introduction du chapitre 3) : $\{p(0,a), \forall x,y\, \neg p(x,y) \vee p(x+1, f(y)), \exists n \forall x\, \neg p(n,x)\}$. On procède en remplaçant la clause but $\exists n \forall x\, \neg p(n,x)$ par

$$\exists n\, \forall x\, \neg p(n,x) \wedge \forall m\ m+1 \not\simeq n \vee \exists x\ p(m,x)$$

qui code le fait que $\forall x\, \neg p(m,x)$ est vraie pour $m = n$ et fausse pour le prédécesseur de $n$. Un tel schéma peut être dérivé en supposant que $n$ est la valeur minimale qui satisfait $\forall x\, \neg p(n,x)$. Nous pouvons aussi procéder d'une autre manière en utilisant la clause $\exists x\, p(m,x)$ comme un invariant inductif et ajouter le schéma d'induction suivant :

$$(\exists x\, p(0,x) \wedge \forall m((\exists x\, p(m,x)) \Rightarrow (\exists x\, p(m+1,x)))) \Rightarrow \forall m \exists x\, p(m,x)$$

En ajoutant ces axiomes ainsi que l'axiome de domaine $\forall n\, n = 0 \vee \exists m\, n = m+1$ nous pouvons facilement établir l'insatisfaisabilité de l'ensemble de départ avec un démonstrateur du premier ordre.

Cette approche possède l'avantage d'être efficace et facile à mettre en oeuvre. Son inconvénient est que le schéma d'induction doit être connu à l'avance, ce qui n'est pas forcément toujours le cas. Par exemple l'insatisfaisabilité de la formule

$$p(0) \wedge \forall p(x) \Rightarrow (p(x+1) \wedge \neg q(x+1)) \wedge \exists n\, q(n+1)$$

ne peut être établie de cette manière. On vérifie en effet que la formule :

$$p(0) \wedge \forall x\ p(x) \Rightarrow (p(x+1) \wedge \neg q(x+1)) \wedge \exists n\ \ (q(n+1) \wedge \neg q(n))$$

est satisfaisable (au sens classique). La formule ne peut être démontrée qu'en établissant au préalable l'insatisfiabilité de : $p(0) \wedge \forall x\ p(x) \Rightarrow (p(x+1) \wedge \neg q(x+1)) \wedge \exists n\ \neg p(n)$, ce qui peut être réalisé en ajoutant la condition $\forall m\ m+1 \not\simeq n \vee p(m)$. Cela correspond à une application de la règle de coupure : $\psi$ est prouvée en démontrant successivement $\psi \wedge \phi$ et $\psi \wedge \neg\phi$ (où $\phi$ dénote un lemme). Il est connu que les coupures ne peuvent être éliminées en général des preuves par induction, ce qui indique que les preuves ne peuvent être obtenues sans recourir à des lemmes. Le lemme peut être difficile à identifier et à expliciter, par exemple si la preuve fait appel à une récursion croisée entre deux formules. Notre approche permet de générer automatiquement certains de ces lemmes en détectant des cycles dans l'espace de recherche. Dans l'exemple ci-dessous, le calcul de superposition dérive les clauses $\neg p(n)$ puis $\neg p(n-1)$ et on vérifie facilement qu'un cycle existe entre ces deux dernières clauses (car $p(n-1)$ est déductible uniquement à partir de $\neg p(n)$). Les récursions croisées correspondent à des cas où l'invariant contient plus d'une clause, chaque clause étant générée à partir de l'ensemble des clauses au rang précédent. Naturellement tous les lemmes ne peuvent être obtenus de cette manière (certains ne peuvent même pas être exprimés en logique du premier ordre).

### 8.1.2 Induction implicite

Tout comme notre approche, l'induction implicite vise à éviter d'avoir à spécifier explicitement le schéma d'induction. Cette approche est fondée sur la réécriture. Elle consiste à réécrire la conjecture à démontrer, à l'aide des axiomes, jusqu'à obtenir `V` ou `F`. Les variables sont instanciées de manière paresseuse, en fonction des axiomes à appliquer (par exemple en utilisant la sur-réduction - ou narrowing). Une telle procédure est complète au sens où tous les contre-exemples peuvent être obtenus, mais l'ensemble des instances à considérer étant évidemment infini, elle ne termine pas en général (contrairement à la logique du premier ordre, la validité est ici co-semi-decidable mais non-semi-décidable). L'induction est simulée en ajoutant un mécanisme permettant d'utiliser les formules précédemment considérées comme axiome (la correction est assurée par le fait que la preuve est implicitement réalisée par induction noethérienne en utilisant l'ordre de réduction induit par le système de réécriture considéré), voir par exemple [25].

Il existe un grand nombre de travaux sur l'induction implicite. Des procédures de preuve par induction utilisant la réécriture sont proposées dans [55, 80, 26, 27] (voir également [71] pour des résultats préliminaires plus anciens). Le principe est de tester la validité de la conjecture sur un ensemble fini d'instances non fermées (appelé "cover-set" dans [71] et "test-set" dans [55, 26, 27]) Les instances de la conjecture qui sont plus petites (selon l'ordre de réduction considéré) peuvent être utilisées lors du calcul. Les test-sets doivent vérifier certaines conditions techniques pour assurer la correction de la procédure de preuve (assurant par exemple que l'instanciation permette l'application d'au moins une règle de réécriture) et garantir la détection des conjectures non valides (génération de contre-exemples). Cette approche est applicable à des systèmes de réécriture conditionnels, non complets, et non nécessairement convergents (bien que dans ce cas la complétude réfutationnelle n'est pas assurée). Kapur et Subramaniam ont proposé dans [51], une méthode pour mixer l'induction avec les procédures destinées aux problèmes de décision. Ils ont donc défini une classe syntaxique d'équations où la validité inductive est décidable. Le démonstrateur inductif *RRL* ([52],[50],[85]) utilise des méthodes d'approximation du problème de couverture par ensemble pour garantir la terminaison avec une réponse positive ou négative sur des équations de la classe. La validité peut donc être vérifiée sans l'intervention de l'utilisateur, ce qui permet d'intégrer l'induction dans des raisonnements complètement automatiques. Dans [43], les classes de conjectures équationnelles définies dans [51] (où on est restreint à des équations) sont étendus à des formules arbitraires non-quantifiées et dans [44] une classe décidable plus large et moins restrictive que [51] est proposée. Des travaux plus récents [39] mixent des systèmes de réécriture avec l'arithmétique linéaire. Ces résultats sont implémentés dans le logiciel Sail2. Des méthodes similaires ont été développées par la suite, et utilisées par les différents démons-

trateurs inductifs comme ACL-2 ([58, 28]), CLAM ([31, 29]), SPIKE ([25]). Dans [83], Stratulat passe en revue les différentes méthodes de preuve par induction en faisant une comparaison entre les différents principes d'induction. Ensuite il analyse les principes d'induction qui peuvent être basés soit sur les termes soit sur les formules. Il montre que chaque principe d'induction basé sur les termes peut être représenté par un principe basé sur les formules, cependant les contraintes d'instanciation empêchent alors de définir des récursions croisées. Il propose une nouvelle technique de preuve par induction basée sur les formules combinant les avantages des deux méthodes.

L'approche par induction implicite est applicable sur des fonctions définies par un système de réécriture. Elle peut être utilisée (au moins en théorie) pour prouver des schémas de formules propositionnelles. Pour cela il suffit de définir, à l'aide d'un système de réécriture satisfaisant les propriétés ci-dessus, une fonction $f$ retournant la valeur de vérité de la formule considérée dans une interprétation (représentée par une liste de variables propositionnelles, dont la longueur dépend de l'interprétation du paramètre $n$), puis de prouver que la formule $f(x) = \mathtt{V}$ n'admet pas de solution (ce qui prouve que la formule est insatisfaisable). Cette fonction est facile à définir, la vérification de modèle étant évidemment décidable et les axiomes correspondants peuvent être construits automatiquement à partir de la formule. Cette approche ne peut être utilisée avec des schémas de formules du premier ordre, car dans ce cas les modèles ne peuvent être énumérés (l'existence d'un modèle n'est pas semi-décidable, le test de satisfiabilité ne peut donc être codé par un système de réécriture - a fortiori convergent).

### 8.1.3 Induction sans induction

L'induction sans induction (voir [36, 35]) est une méthode qui consiste à réduire la validité inductive à une preuve par consistance (ou un test de satisfaisabilité), pour pouvoir ensuite utiliser un démonstrateur du premier ordre. Le but de cette méthode est d'obtenir une automatisation complète, c'est-à-dire éviter à l'utilisateur d'introduire des schémas d'induction explicites. Etant donné un ensemble de clauses $\varepsilon$, $C$ un ensemble de clauses ou conjectures, le but de cette procédure est de vérifier si $C$ est vraie dans le modèle de Herbrand particulier $\mathcal{I}$ de $\varepsilon$. Généralement, $\varepsilon$ est un ensemble de clauses de Horn et $\mathcal{I}$ est le plus petit modèle de Herbrand de $\varepsilon$. H. Comon et R. Nieuwenhuis ont introduit dans [35] un ensemble $\mathcal{A}$, appelé *I–axiomatization* de $\mathcal{I}$, et ont montré que tester si $\mathcal{I} \models C$ revient à vérifier la consistance de l'ensemble $\mathcal{A} \cup \varepsilon \cup C$. Ils utilisent le calcul de superposition pour vérifier la consistance de l'ensemble $\mathcal{A} \cup \varepsilon \cup C$ et inférer la validité inductive de $C$. La méthode reste, à ce point, semi-automatique, car même si la consistance est vérifiée automatiquement, il reste à dériver l'ensemble $\mathcal{A}$. Il est expliqué dans [35] comment obtenir une I-axiomatization automatiquement, et comment assurer

la terminaison en utilisant des stratégies basées sur des ordres de réduction bien choisis. Dans cette approche, l'induction est implicitement simulée par les règles d'inférence et de détection de redondance (e.g., la subsumption), et elle se fonde sur l'ordre de réduction utilisé par le calcul.

L'un des inconvénients de cette méthode est qu'elle est appliquée le plus souvent dans le cas où l'ensemble $\varepsilon$ est de Horn, permettant ainsi d'assurer l'existence d'un unique modèle minimal de Herbrand. Néanmoins, il est montré dans [35] comment étendre cette approche vers des ensembles de clauses qui ne sont pas de Horn. Un autre inconvénient concerne la correction par rapport à la notion de la validité inductive, car cette procédure permet en fait de montrer une propriété légèrement différente de la validité inductive, comme le montre l'exemple suivant :

**Exemple 111** Si on considère les axiomes de théorie de l'addition suivant :

$$S = \left\{ \begin{array}{c} 0 + x \simeq x \\ s(x) + y \simeq s(x + y) \end{array} \right.$$

Le modèle minimal de Herbrand est donné par l'interprétation usuelle de $+$. La clause $C : x \not\simeq s(x)$ n'est pas une conséquence inductive de $S$, car le modèle de Herbrand ne contenant que la valeur 0 n'est pas un modèle de $C$, même si $\mathcal{I} \models C$.♣

Cependant le test de consistance est équivalent à la validité inductive dans le cas où $C$ est positive. Il existe quelques classes syntaxiques d'axiomatisations $\varepsilon$, où la validité inductive est décidable.

Cette approche est destinée aux formules de la forme $\forall \overrightarrow{x} \phi$ et ne peut traiter le cas des $\alpha$-clauses où alternent les quantificateurs existentiels et universels. Elle est complète pour la réfutation mais la terminaison n'est pas assurée car l'ensemble de clauses $\mathcal{A}$ ne possède pas en général les propriétés de l'ensemble $\varepsilon$ (par exemple le fait que $\varepsilon$ soit de Horn n'implique pas forcément que $\mathcal{A}$ le soit aussi). De la même manière, la saturation de $\mathcal{A} \cup \varepsilon \cup C$ ne termine pas nécessairement même si $\varepsilon \cup C$ appartient à un fragment où la saturation est assurée.

## 8.2 Superposition avec contrainte

### 8.2.1 Procédures de preuve hybrides

Des procédures de preuve sont proposées dans [15, 1, 17, 41], pour traiter des problèmes mélangeant des formules du premier ordre avec des théories non axiomatiques comme l'arithmétique. L'objectif est de traiter des formules dépendantes à la fois de théories qui ne peuvent être axiomatisées efficacement en logique du premier ordre (mais pour lesquelles des procédures de décisions existent) et d'axiomes

du premier ordre standard (définissant par exemple des fonctions définies sur des termes contenant des entiers ou des réels). Ces procédures sont dites hybrides et fonctionnent en réalisant une abstraction des termes apparaissant dans les formules pour les séparer en deux parties : une partie du premier ordre sur laquelle nous allons pouvoir appliquer les calculs usuels comme la superposition ou la résolution, et une partie contrainte liée à la théorie qui, d'une certaine manière, contraindra l'application des règles d'inférence. Ces contraintes permettent de restreindre le domaine de validité des clauses. Leur satisfiabilité est testée par une procédure de décision externe. Par exemple la formule $f(1+1) \not\simeq a \wedge f(2) \simeq a$ est transformée en l'ensemble de clauses contraintes : $\{[f(x) \not\simeq a \mid x = 1+1], [f(y) \simeq a \mid y = 2]\}$. Les variables $x$ et $y$ sont introduites pour dénoter les termes $1+1$ et 2 respectivement, et les égalités $1+1 = x$ et $2 = y$ sont ajoutées aux contraintes. Grâce à cette transformation, le calcul de superposition engendre la clause contrainte : $[a \not\simeq a \mid 1+1 = x \wedge 2 = x]$, et par réflection : $[\Box \mid 1+1 = x \wedge 2 = x]$. L'équation $1+1 = 2$ étant valide (modulo l'arithmétique), cette dernière clause est insatisfaisable. Notons que le mécanisme utilisé pour séparer le raisonnement dans la théorie et le raisonnement en logique du premier ordre est strictement identique à celui que nous utilisons pour "abstraire" le paramètre entier dans les $n$-clauses. [18] présente une technique visant à réduire le nombre de variables introduites durant le processus d'abstraction. Elle est fondée sur une analyse plus fine de la preuve de complétude permettant de restreindre les termes à "abstractiser". En réduisant le nombre de variables introduites, elle permet de renforcer l'effet des restrictions d'ordre sur les règles d'inférence, rendant ainsi le calcul plus efficace.

En général, la complétude ne peut être garantie, sauf si la formule considérée vérifie des conditions assez fortes : la théorie doit être compacte (au sens où tout ensemble insatisfaisable contient un sous-ensemble fini insatisfaisable) et tous les termes dont le type est dans la théorie doivent être égaux à des termes de cette théorie (par exemple si la formule contient une constante n, cette constante doit être prouvable égale à un entier).

Ces conditions ne sont pas satisfaites en général dans notre contexte : ainsi l'arithmétique est non compacte en présence d'un paramètre $n$ de type entier (par exemple $\{n \neq k \mid k \in \mathbb{N}\}$ est insatisfaisable mais ne contient aucun ensemble fini insatisfaisable). D'autre part l'égalité du paramètre à un entier n'est pas prouvable à partir des axiomes. De manière générale, la superposition avec contrainte ne permet pas de simuler le raisonnement inductif, ce qui est précisément l'objet des règles de détection de cycles que nous avons introduites. En l'absence de raisonnement inductif, la superposition avec contrainte engendre à partir de toute formule insatisfaisable $\phi[n]$ une infinité de clauses de la forme $n \neq k$ ($k \in \mathbb{N}$), mais aucune contradiction ne peut être générée en un temps fini. D'autre part, le langage des $n$-clauses pourrait aisément être étendu en considérant des contraintes interprétées

dans des théories gérées par des démonstrateurs externes, ce qui permettraient de traiter des schémas de formules définies sur ces théories (par exemple des schémas de formules arithmétiques).

Plusieurs approches se sont inspirées des techniques décrites ci-dessus pour traiter le cas des formules contenant des termes définis dans un domaine inductif. Nous présentons dans cette section ces différentes approches.

### 8.2.2 La superposition dans des domaines fixes

Les travaux décrits dans [49] et [48] sont proches de notre approche. Ils traitent des clauses contenant des quantificateurs existentiels et des termes interprétés sur des domaines inductifs. Leur méthode consiste à utiliser la procédure de superposition hybride définie ci-dessus. Ils définissent un calcul de superposition qui permet de raisonner sur des domaines fixes appelé SFD (Superposition for Fixed Domains). Cette procédure est ensuite enrichie avec des raisonnements basés sur le modèle minimal de Herbrand, et permet entre autres de montrer des conjectures où alternent les quantificateurs existentiels et universels (de la forme $\forall\exists*$). Ils utilisent le fait qu'une théorie satisfaisable saturée par superposition admet implicitement un modèle minimal (fondé sur l'ordre de réduction considéré).

Afin d'éviter la skolémisation, le calcul SFD utilise une représentation explicite des variables existentielles. Les clauses sont donc définies avec des contraintes qui restreignent l'instanciation des variables existentielles pour une clause. Par exemple, la formule $\exists u\ \forall y\ \ P(x,y) \wedge \neg P(a,a)$ correspond aux clauses avec contraintes : $u \simeq x \mid\mid \rightarrow P(x,y)$ et $u \simeq x \mid\mid P(a,a) \rightarrow$. Si on applique la résolution sur les deux clauses en unifiant leurs contraintes, on obtient une clause vide avec la contrainte $u \simeq a$, cela veut dire que $u \mapsto a$ ne satisfait pas la clause initiale. Notez que cette approche est identique à la façon dont nous représentons la valeur du paramètre. Ce calcul SFD est enrichi avec une règle inductive définie dans [49]. Cette règle code le fait que lorsqu'on prouve par contradiction une formule $\forall\alpha\,\phi(\alpha)$, où $\alpha$ est défini dans un domaine inductif, on suppose ordinairement que $\alpha$ est le terme minimal tel que $\phi(\alpha)$ n'est pas vrai, c'est-à-dire que la formule $\neg\phi(\alpha)$ est vraie pour tout $\beta$ strictement inférieur à $\alpha$ (par rapport à un certain ordre bien fondé). Le but de la règle en question est de dériver la formule $\neg\phi(\beta)$, avec des contraintes qui assurent que $\beta < \alpha$. Si nous considérons, par exemple, l'ensemble de clauses présenté en introduction du chapitre 3. Nous codons l'ensemble comme suit :

$$\begin{array}{ll} 1 & n \simeq x \mid\mid P(x,y) \rightarrow \\ 2 & \Box \mid\mid P(x,y) \rightarrow P(s(x),y) \\ 3 & \Box \mid\mid \rightarrow P(0,a) \end{array}$$

En appliquant les règles d'inférence de SFD et la règle d'induction, on dérive les

clauses suivantes :

| | | |
|---|---|---|
| 4 | $n \simeq 0 \mid\mid \square$ | superposition$(1, 3)$ |
| 5 | $n \simeq s(x) \mid\mid P(x, y) \rightarrow$ | superposition$(1, 2)$ |
| 6 | $n \simeq s(x) \mid\mid \rightarrow P(x, y)$ | induction$(1)$ |
| 7 | $n \simeq s(x) \mid\mid \square$ | superposition$(5, 6)$ |

Puisque $\{0, s(x)\}$ est couvrant dans $\mathbb{N}$ ((c'est-à-dire tout terme est soit 0, soit de la forme $s(x)$), l'ensemble de clauses est insatisfaisable.

Cette approche est étroitement liée à la notre, d'ailleurs, le calcul de superposition est exactement le même. La différence réside dans la façon de générer les lemmes inductifs, la règle d'induction dans [49] suppose que ces lemmes sont définis explicitement par l'utilisateur, tandis que la détection de boucle les génère de façon dynamique en analysant l'espace de recherche, ce qui permet de prouver des formules qui ne sont pas démontrables par induction sur la formule de départ. La contrepartie est que notre approche est plus restrictive en ce qui concerne les types inductifs (une seule variable existentielle de type mot).

Quelques résultats de complétude sont présentés dans [48], mais les critères d'obtention de ces résultats sont très restrictifs, Ils supposent par exemple que toutes les clauses sont de Horn et que les fonctions sont toutes monadiques. Ces résultats ne sont pas comparables avec ceux décrits au chapitre 6.

### 8.2.3 La méthode de résolution pour les schémas de formules du premier ordre

Les résultats présentés dans [8] constituent un des points de départ de nos travaux. Les auteurs traitent dans un premier temps le cas des schémas de formules propositionnelles en proposant un nouveau formalisme pour coder les schémas, qui est très similaire aux $n$-clauses. Ils définissent un calcul de résolution dans ce nouveau formalisme et montrent sa correction et sa complétude réfutationnelle. Ils étendent ensuite leur procédure au premier ordre et ils obtiennent une procédure correcte mais non complète. La différence essentielle avec l'approche que nous proposons est que la règle de détection de boucle s'applique sur les ensembles de clauses engendrés par saturation à chaque niveau (c'est-à-dire pour une valeur donnée du paramètre). Elle n'est donc applicable que dans le cas où la saturation est possible, ce qui n'est pas le cas en général lorsque l'ensemble considéré n'est pas propositionnel.

Une autre différence concerne les clauses dites *d'élagage* c'est-à-dire des clauses de la forme $n \not\simeq \text{succ}^i(x)$ (avec $i \in \mathbb{N}$), qui sont des conséquences logiques de l'ensemble des clauses générées dans notre cas, alors qu'elle préserve seulement la satisfaisabilité dans [8] c'est-à-dire si $S$ est l'ensemble de clauses et si les conditions

de la détection de boucles sont vérifiées, alors $S$ est satisfaisable ssi $S \cup \{C\}$, où $C$ est la clause d'élagage.

## 8.3 Schématisations

### 8.3.1 Méthode des tableaux pour les schémas propositionnels réguliers

Les travaux décrits dans [2] ont introduit une formalisation de la notion de *schéma itéré de formules* en logique propositionnelle (voir chapitre 4). La logique de ces schémas n'est pas complète pour la réfutation mais possède la propriété de complétude pour la satisfaisabilité (la logique est donc *semi-décidable*). Un système de preuve appelé *STAB* a été construit. Ce système étend la méthode des tableaux sémantiques pour tenir compte des schémas. Cette procédure est ensuite enrichie avec une règle détection de cycles, qui raisonne inductivement sur le paramètre $n$. Une classe de schémas, appelée les *schémas réguliers*, a été identifiée, pour laquelle STAB termine, et pour laquelle le problème de la satisfaisabilité est donc décidable. La complexité s'avère être doublement exponentielle [4], et la procédure *STAB* munie de la détection de cycles est implémentée dans l'outil *RegStab* [5]. Une autre procédure de preuve, basée sur une extension de la procédure DPLL [61] est décrite dans [2, 3].

La méthodologie est similaire à celle que nous avons adoptée, à savoir ajout d'une technique de détection de cycles à une méthode de preuve existante. Cependant, nous nous plaçons dans un cadre strictement plus général en autorisant des variables du premier ordre et des indices de type mot et nous basons sur des procédures de preuve de nature très différentes. Nous rappelons que la classe des schémas réguliers peut être réduite (en utilisant la transformation décrite au chapitre 4) en une classe admissible pour laquelle la superposition termine, l'inverse n'étant pas vrai. Les résultats de décidabilité de [2] sont donc (strictement) subsumés par ceux décrits dans ce mémoire.

### 8.3.2 Les schématisations de termes

La schématisation de termes est une façon de représenter, de manière finie, une infinité d'objets (comme des clauses) similaires, en décrivant formellement la structure des termes contenus dans ces objets. Considérons, par exemple, l'ensemble suivant :

$$\{P(0), \forall x\ P(x) \Rightarrow P(s(s(x)))\}$$

En appliquant la résolution, on dérive une liste infinie de clauses :

$$\{P(0), P(s(s(0))), \ldots, P(s^{2n}(0)), \ldots\}$$

La schématisation de termes consiste à remplacer cette liste infinie par une seule clause compacte de la forme : $\forall N P(s^{2N}(0))$, où $N$ est une nouvelle variable. Il s'agit pour cela de définir des langages permettant de décrire formellement ces séquences de termes (ici $s^{2N}(0)$) et des algorithmes permettant de manipuler ces séquences de manière automatique (par exemple pour résoudre la problème d'unification). Différents formalismes ont été proposés dans ce but comme les *ρ-termes*, les *I-termes* [33], les *R-termes* [78] et les *grammaires primales* [46]. Ces approches se focalisent essentiellement sur la résolution du problème d'unification (qui s'avère très complexe pour les formalismes les plus expressifs). A noter que les variables entières sont quantifiées uniquement universellement, ce qui implique que le pouvoir d'expression reste équivalent à la logique du premier ordre, contrairement aux $n$-clauses.

L'outil décrit dans [21, 20] permet d'analyser l'espace de recherche afin de détecter les listes infinies de clauses mentionnées ci-dessus, et d'engendrer les schémas de termes correspondants. Cela nécessite pour cela de développer des méthodes de preuve opérant sur des axiomes contenant des schémas de termes. Deux calculs sont proposés pour cela dans [22, 20]. Le premier est une extension de la résolution ordonnée aux $I$-termes. Ce calcul est correct et complet pour la réfutation dans le cas non équationnel. Le deuxième est une extension de la superposition aux $I$-termes. Ce calcul est correct mais perd la complétude réfutationnelle car les règles d'inférence de la superposition ne conservent pas la structure des $I$-termes, nous illustrons cela dans l'exemple suivant. Soit l'ensemble de clauses :

1. $P(f(a, \diamond)^N \cdot b)$ (correspondant à l'ensemble : $\{P(f(a,b)), P(f(a, f(a,b))), \ldots\}$)
2. $\neg P(c)$
3. $f(a,b) \simeq d$
4. $f(a,d) \simeq c$

Il est clair que l'ensemble est insatisfaisable, car en instanciant $N$ par 2 nous obtenons la clause $P(f(a, f(a,b)))$ et en appliquant la superposition successivement : de 3 dans 1 et de 4 dans 1 nous obtenons la clause $P(c)$ qui engendre la clause vide avec la clause 2. Or en appliquant la superposition usuelle (étendu aux $I$-termes) une seule unification est possible entre $P(f(a, \diamond)^N \cdot b)$ et $f(a,b)$ avec la substitution $\sigma = \{N \mapsto 1\}$. La clause $P(d)$ est générée et aucune autre unification n'est possible entre $d$ et $f(a,b)$ et entre $d$ et $f(a,d)$, l'ensemble est donc saturé. Le problème dans cet exemple est que la superposition appliquée sur les $I$-termes fait perdre leur structure. Pour y remédier une nouvelle règle de superposition est proposée, appelée $H$-superposition. Cette règle permet de déplier un nombre arbitraire de fois (ce

nombre étant codé par une variable) les contextes inductifs avant d'appliquer le remplacement. Dans l'exemple précédent, l'application de la $H$-superposition entre les clauses 3 et 1 permet d'obtenir la clause $P(f(a, \diamond)^N \cdot f(a, d))$ qui, en appliquant la superposition avec 4, engendre la clause $P(c)$ nécessaire pour dériver la clause vide.

# Troisième partie

# Implémentation et expérimentations

# Chapitre 9

# Description de SuperInd

Nous avons présenté, dans le chapitre 3, une procédure de preuve hybride basée d'une part sur le calcul de superposition et d'autre part sur la détection de boucles. Nous avons implanté cette procédure sous forme d'un prototype de recherche que nous appelons *SuperInd* (**Super**position & **Ind**uction). Comme le calcul n'est presque pas modifié, nous utilisons un démonstrateur par résolution et paramodulation existant (Prover9) pour appliquer les règles d'inférence. Cependant, le démonstrateur n'est pas utilisé comme une boite noire, car la structure des $n$-clauses est différente de celle des clauses usuelles du premier ordre ce qui nous a contraint à adapter les règles d'inférence (en modifiant le code du démonstrateur) pour tenir compte des informations additionnelles associées aux clauses.

## 9.1 Prover9

Prover9 est un démonstrateur automatique pour la logique du premier ordre avec égalité. Il a été développé par McCune (voir [60]) et correspond à une nouvelle version du démonstrateur Otter [59]. Il implémente une palette de règles d'inférences telles que : la résolution binaire et unaire, l'hyperrésolution et la paramodulation binaire. Ces règles d'inférence permettent de définir une procédure de preuve efficace soit manuellement en utilisant les différentes options pour activer les règles à appliquer, soit en utilisant le mode automatique où Prover9 analyse la forme et la structure des clauses en entrée pour, ensuite, choisir les règles qui semblent les plus adaptées (d'un point de vue heuristique) pour obtenir la preuve [1]. Afin de faciliter la lecture de ce chapitre et la compréhension des modifications apportées au logiciel, nous présentons brièvement les principales fonctionnalités de

1. Afin d'éviter la redondance nous rappelons les principales fonctionnalités de Prover9 dans la présentation de notre outil SuperInd (voir la section 9.2).

Prover9 ainsi que quelques détails d'implémentation. Nous renvoyons à [60] pour une description complète.

### 9.1.1 Format d'entrée

Si nous voulons montrer qu'un ensemble de clauses est contradictoire, nous devons le mettre entre les deux commandes $formulas(sos)$ et $end\_of\_list$.

**Exemple 112** Pour un ensemble de clauses

$$S = \{\neg man(x) \vee mortal(x), man(george), \neg mortal(george)\}$$

le fichier d'entrée est noté (voir [60] pour la syntaxe complète) :

```
formulas(sos).
  -man(x) | mortal(x).
  man(george).
  -mortal(george).
end_of_list.
```

Le même exemple peut s'écrire de manière plus naturelle en séparant les axiomes de la formule à prouver : si nous voulons montrer que $mortal(george)$ est une conséquence logique de la formule $(man(x) \Rightarrow mortal(x)) \wedge man(george)$, le fichier d'entrée peut être de la forme :

```
formulas(assumptions).
  man(x) -> mortal(x).
  man(george).
end_of_list.

formulas(goals).
  mortal(george).
end_of_list
```

Prover9 utilise la preuve par réfutation, donc la clause qui se trouve dans $formulas(goals)$ sera ajoutée négativement (c'est-à-dire $\neg mortal(george)$), à l'ensemble des clauses hypothèses pour chercher la clause vide (cela correspond au premier format). Nous allons maintenant décrire l'algorithme principal de Prover9.

### 9.1.2 Algorithme

Prover9 est basé sur une variante de l'algorithme de saturation appelée l'algorithme de la clause active ("given clause algorithm"). Cet algorithme utilise deux listes de clauses principales : *sos* et *usable*. Au début de l'exécution, la liste *usable* est vide et la liste *sos* contient toutes les clauses du fichier d'entrée après un pré-processing (ce processing correspond entre autres à la clausification et l'application de diverses règles de simplification, préservant la satisfaisabilité, que nous ne détaillerons pas ici). Puis l'algorithme entre dans la boucle principale que nous décrivons ici :

**Tant que** la liste *sos* n'est pas vide :

1. Sélectionner une clause **active** dans la liste *sos* et la mettre dans la liste *usable*.
2. Dériver une nouvelle clause avec les différentes règles d'inférence actives. Chaque nouvelle clause générée doit avoir comme parents la clause active et éventuellement une autre clause dans la liste *usable*.
3. Appliquer les règles de simplification et de suppression sur chaque nouvelle clause.
4. Ajouter dans liste *sos* toutes les nouvelles clauses retenues.

**Fin** de la boucle Tant que.

### 9.1.3 Implémentation

Prover9 est écrit en langage $C$, et utilise la bibliothèque LADR qui est aussi utilisée par Mace4 (voir [60]). Afin de pouvoir présenter les différentes modifications que nous avons introduites dans le code existant, nous allons décrire les outils contenus dans la bibliothèque LADR, et nous allons voir les différentes représentations des objets usuels de la logique du premier ordre comme les termes, les clauses ou les ensembles de clauses. Puis nous allons décrire le module principal qui importe cette bibliothèque et implémente l'algorithme principal.

#### La bibliothèque LADR

Les différents objets du langage du premier ordre sont représentés par des *structures* (*struct*) et des *listes chainées* :

### Termes

Un terme est représenté par une liste chainée *Term* d'éléments de type *term* définie comme suit :

```
typedef struct term * Term;     /* Term is a pointer to a term struct */

struct term {
  int            private_symbol; /* const/func/pred/var symbol ID */
  unsigned char  arity;          /* number of auguments */
  Term           *args;          /* array  of pointers to args */
   ...
};
```

*private_symbol* est un identifiant entier qui correspond à un symbole de terme (une table de hashage est construite à partir du fichier d'entrée qui fait correspondre à chaque symbole de terme un entier). *arity* correspond à l'arité du terme. *args* est un tableau des arguments du terme. Nous renvoyons le lecteur à la documentation de Prover9 pour les autres champs. Nous avons trois types de termes : les variables, les constantes et les termes complexes (une fonction appliquée à des termes), ces trois types sont définis grâce à l'identifiant et à l'arité comme suit :

1. Si $private\_symbol \geq 0$ alors le terme est une variable. Une fonction booléenne $VARIABLE(t)$ est implémentée et retourne *vrai* dans ce cas.
2. Si $private\_symbol < 0$ et $arity = 0$ alors le terme est une constante. Une fonction booléenne $CONSTANT(t)$ est implémentée et retourne *vrai* dans ce cas.
3. Si $private\_symbol < 0$ et $arity > 0$ alors le terme est complexe. Une fonction booléenne $COMPLEX(t)$ est implémentée et retourne *vrai* dans ce cas.

### Clauses

Une clause est représentée par une structure *topform* définie comme suit :

```
struct topform {

  int              id;
  struct clist_pos *containers;     /* Clists that contain the Topform */
  struct just      *justification;
  double           weight;
  Literals         literals;        /* NULL can mean the empty clause */
  ...
```

```
};
```

*id* est un identifiant entier de la clause. *containers* correspond à l'ensemble des listes de clauses (utilisées par le système) qui contiennent la clause. *justification* correspond aux clauses parentes de la clause et *weight* est le poids de la clause. *literals* est l'ensemble des littéraux de la clause, il est de type *Literals* qui est une liste chainée définie comme suit :

```
typedef struct literals * Literals;

struct literals {
  BOOL      sign;
  Term      atom;
  Literals  next;
};
```

*sign* est un booléen qui prend la valeur `V`si le littéral est positif et la valeur `F`sinon. *atom* est l'atome du littéral et enfin *next* est la liste des littéraux restant.

### Les règles d'inférence

Les règles d'inférence sont implémentées sous forme de fonctions génériques, elles utilisent un pointeur de fonctions pour passer en argument le "processing" défini par l'utilisateur et toutes les règles d'inférence utilisent l'indexation, à l'exception de la paramodulation. Voici, par exemple, la signature d'une fonction qui implémente la résolution binaire :

```
void binary_resolution(Topform c,
       int res_type,  /* POS_RES, NEG_RES, ANY_RES */
       Lindex idx,
       void (*proc_proc) (Topform))
```

*res_type* correspond au type de résolution que nous voulons appliquer [2] et *idx* représente l'ensemble des littéraux de la liste *usable* sous forme d'index. On teste l'applicabilité de la résolution avec les littéraux de la clause *c*. Si la résolution est appliquée et qu'une nouvelle clause *p* est générée, alors nous appliquons la fonction, passée en argument, sur *p* c'est-à-dire *proc_proc*(*p*).

2. Il y a trois types de résolution : la résolution positive, la résolution négative et la résolution sans restrictions.

**Les listes dans LADR**

Il existe trois types de listes :

1. Le type *Ilist* qui représente une liste d'entiers, où on peut stocker, par exemple, la liste des identifiants des clauses parentes d'une clause.
2. Le type *Plist* qui est une liste de pointeurs (donc éventuellement des clauses).
3. Le type *Clist* qui est une liste de clauses utilisée par le module principal de Prover9. Par exemple, les liste *sos* et *usable* sont de type *Clist*.

**Le module prouveur**

Ce module implémente l'algorithme principal et importe la bibliothèque LADR pour utiliser ses différents outils. Le module *search.c* réunit les outils principaux pour la recherche de preuve. Il contient une *structure* statique *Glob* qui englobe l'ensemble des informations utilisées dans la recherche de preuve, par exemple les listes *sos* et *usable*[3]. Ce module contient la fonction principale qui implémente la grande boucle définie dans la section 9.1.2, dont la signature est comme suit :

```
Prover_results search(Prover_input p)
```

*Prover_input* est une structure contenant l'ensemble de clauses initial avec toutes les informations correspondant comme la stratégie employée, les options activées... *Prover_results* est une structure contenant entre autres, la preuve (sous forme de liste de clauses), les statistiques, le temps d'exécution...

## 9.2 Extensions

SuperInd est une extension de Prover9 destiné à réfuter des ensembles de $n$-clauses comme définis dans le chapitre 3. Nous avons vu, dans le chapitre 3, que nous avions considéré que toutes les $n$-clauses étaient normalisées, donc les $n$-clauses sont de la forme $[C \mid n \simeq t]$ (où $t$ est de sorte `nat`) ou de la forme $[C \mid \mathtt{V}]$. Les clauses de la forme $[C \mid \mathtt{V}]$ (de rang $\bot$) sont notées directement avec la syntaxe de Prover9 (voir [60]), quant aux clauses de la forme $[C \mid n \simeq t]$, la partie clausale est notée avec la syntaxe de Prover9 à laquelle est ajoutée un littéral $N(t)$. Ce littéral spécial nous permet de représenter la contrainte et ne peut être utilisé pour les inférences (aucune inférence, à partir de où vers ce littéral n'est admise). Le symbole $N$ devient donc un mot réservé par le système et une clause ne peut

3. Nous allons, par exemple, stocker (entre autres) les clauses de la forme $N(s^i(0))$ dans un des champs de *Glob*.

contenir plus d'un littéral de la forme $N(t)$ (où $t$ est un terme de sorte `nat`). Nous avons ajouté, aux règles de simplification mentionnées plus haut, une étape de normalisation des clauses générées retenues, voir la section 9.3.1.

**Exemple 113** L'ensemble de clauses (correspondant à l'exemple du chapitre 3) :

$$\begin{array}{ll} 1 & p(0,a) \simeq \mathtt{V} \\ 2 & p(x,y) \not\simeq \mathtt{V} \vee p(\mathrm{succ}(x), f(y)) \simeq \mathtt{V} \\ 3 & [q(x,y) \not\simeq \mathtt{V} \mid n \simeq x] \\ 4 & q(x,y) \simeq \mathtt{V} \vee p(x,y) \not\simeq \mathtt{V} \end{array}$$

est écrit dans le format de SuperInd comme suit :

```
 set(superind).
 formulas(sos).
  P(0,a).
-P(x,y) | P(s(x),f(y)).
 N(x) | -Q(x,y).
-P(x,y) | Q(x,y).
end_of_list.
```

On notera que les littéraux non-équationnels sont codés directement comme des atomes et non comme des équations. ♣

Pour activer la détection de boucles dans Prover9 il suffit d'ajouter au fichier d'entrée l'option **set(superind)** qui est activée par défaut (comme mentionné dans l'exemple ci-dessus).

### 9.2.1 Nouvelles fonctionnalités

Nous décrivons la majorité des fonctionnalités de Prover9 et indiquons celles qui sont supportées par SuperInd ainsi que les adaptations éventuelles.

#### Le mode de recherche

Il y a deux modes de recherche principaux, un mode manuel et un mode automatique. les deux modes fonctionnent avec SuperInd. Le mode automatique est activé par défaut, mais nous pouvons le désactiver avec la commande suivante :

```
clear(auto).
```

Pour définir entièrement une stratégie où il n y a aucun paramètre activé par défaut, nous utilisons la commande suivante :

```
clear(raw).
```

### L'ordre de réduction

Il y a trois ordres définis dans Prover9 : LPO, RPO et KBO. Les trois ordres sont supportés par SuperInd, et nous utilisons, le plus souvent dans les expérimentations, l'ordre KBO (Knuth-Bendix Ordering) qui consiste à utiliser un certain ordre sur les prédicats définis par l'utilisateur combiné avec une fonction fournie par le système, qui attribue à chaque symbole de fonction dans le fichier d'entrée, un poids entier. La commande suivante permet à l'utilisateur de choisir un ordre :

```
assign(order, string).  % default string=lpo, range [lpo,rpo,kbo]
```

Nous avons modifié, en amont, l'ordre, des littéraux sans modifier l'ordre des termes, afin d'assurer que $p(s(x), \dots) > q(x, \dots)$ pour tout $p$ et $q$.

### Les critères d'arrêt

Il y a plusieurs options dans Prover9 qui représentent les différentes conditions pour arrêter la recherche, par exemple : le nombre de clauses engendrées, le nombre de clauses actives, le temps d'exécution ... Toutes ces options sont supportées par SuperInd, voici une commande qui arrête la recherche après un nombre donné de clauses actives :

```
assign(max_given, n).  % default n=-1, range [-1 .. INT_MAX]
```

### Les règles d'inférence

Toutes les règles d'inférence définies dans Prover9 s'appliquent aussi dans SuperInd. Voici un exemple d'activation de la résolution binaire et la résolution ordonnée (qui est activée par défaut) :

```
set(binary_resolution).

set(ordered_res).    % default set
```

### La sélection de la clause active

Il existe quatre modes de sélection de la clause active dans Prover9 et SuperInd :

1. **Le mode aléatoire** (activé avec l'option $set(random\_given)$) : Une clause de *sos* est sélectionnée aléatoirement.
2. **L'âge** (activé par l'option $set(breadth\_first)$) : Les clauses dans *sos* forment une file d'attente de type FIFO (First In First Out), et la clause sélectionnée est toujours la première ajoutée dans *sos*.

3. **Le poids** (activé par l'option $set(lightest\_first)$.) : Les clauses dans Prover9 possèdent un poids assigné pendant le pré-processing (pour plus de détails voir [60]), et la clause sélectionnée est celle qui a le plus petit poids dans *sos*.
4. Le dernier mode permet de sélectionner, en premier lieu, les clauses de l'ensemble initial en entrée (activé par l'option $set(input\_sos\_first)$).

Nous pouvons aussi mixer tous ces paramètres (se référer au manuel de Prover9 pour plus de détails). Nous avons aussi ajouté un mode de sélection basé sur le rang et l'âge, c'est-à-dire que les clauses sont triées par rang et on sélectionne la clause avec le plus petit rang. Lorsque deux clauses sont de même rang on sélectionne la plus âgée (la première à avoir été générée). Ce mode est activé par l'option :

$$set(rank\_given)$$

### Le processing des clauses générées

Nous avons vu dans la section 9.1.2, que nous appliquons un certain traitement sur chaque clause générée en appliquant entre autres les règles de simplification et de suppression. Toutes les options correspondantes sont applicables dans SuperInd. Voici un exemple pour activer la subsomption en arrière [4] :

```
set(back_subsume).    % default set
```

Nous présentons aussi une autre option qui active la factorisation qui est appliquée dans Prover9 (et applicable dans SuperInd) comme une règle de simplification :

```
set(factor).
```

### Le fichier de sortie

Toutes les options d'affichage dans Prover9 sont supportées par SuperInd. Nous présentons ici un exemple d'activation d'une option qui permet d'afficher dans le fichier de sortie toutes les clauses actives :

```
set(print_given).    % default set
```

4. La subsomption à l'envers consiste à parcourir toute la liste *sos* et de marquer toutes les clauses qui sont subsumées par la nouvelle clause générée, si cette dernière est retenue alors toutes les clauses marquées seront supprimées.

**Les preuves multiples**

Les options concernant les preuves multiples ne sont pas supportées par SuperInd, car le format de la preuve à la sortie n'est pas le même que celui de Prover9. Par exemple l'option suivante ne fonctionne pas dans SuperInd :

```
set(reuse_denials).
```

**La guidance sémantique**

La guidance sémantique consiste à utiliser des interprétations finies (que l'utilisateur doit fournir) pour orienter et guider la recherche de preuve. Cette fonctionnalité n'est pas supportée par SuperInd, même s'il serait facile de l'adapter en considérant des interprétations finies pour les $n$-clauses. A noter toutefois qu'une interprétation fixant la valeur du paramètre $n$ ne serait utile que pour les clauses correspondant à cette valeur particulière du paramètre.

**La détection de boucles**

Les deux algorithmes présentés dans le chapitre 3 sont implémentés et activés avec les options suivantes :

```
set(cycle1).
```

qui active l'algorithme de recherche du plus petit point fixe : $\textsc{Cycle}_1$.

```
set(cycle2).
```

qui active l'algorithme de recherche du plus grand point fixe : $\textsc{Cycle}_2$.

Nous avons vu dans le chapitre 6, que nous pouvons restreindre l'application de la détection de boucles aux $n$-clauses $\mathcal{S}_I$-plates. Cette restriction est activée avec l'option suivante :

```
set(index_flat).
```

Nous allons maintenant présenter le fonctionnement de l'outil SuperInd sur un exemple.

### 9.2.2 Exemple d'application

Nous allons montrer le théorème suivant :

$$\forall n \in \mathbb{N}\ \forall a_1, \ldots, a_n \quad a_1 \times a_2 \times \cdots \times a_n = a_n \times a_{n-1} \times \cdots \times a_1 \qquad (1)$$

Nous allons, en premier lieu, écrire l'ensemble de clauses correspondant à ce théorème. Nous introduisons deux termes $p(x)$ et $q(x)$ qui vont coder, respectivement, la partie gauche et la partie droite de l'équation (1). Le terme $p(x)$ est défini inductivement comme suit :

$$p(0) = 1 \wedge \forall x\ p(s(x)) = p(x) * a(x)$$

Nous ajoutons aussi l'axiome $1 * x = x$ (qui code le fait que 1 est l'élément neutre de $*$). Nous procédons de la même manière pour définir $q(x)$ :

$$q(0) \wedge \forall x\ q(s(x)) = a(x) * q(x) \wedge x * 1 = x$$

Nous écrivons maintenant la formule (1) avec la clause suivante :

$$\forall n \in \mathbb{N}\ p(n) = q(n) \quad (2)$$

Comme nous utilisons la preuve par réfutation, nous ajoutons à notre ensemble de clauses la négation de la clause (2). Nous avons alors l'ensemble de clauses suivant (toutes les variables seront par la suite supposées quantifiées universellement et le quantificateur sera omis) :

$$\begin{array}{l} \exists n \in \mathbb{N}\ \neg(p(n) = q(n)) \\ p(0) = 1 \\ q(0) = 1 \\ p(s(x)) = p(x) * a(x) \\ q(s(x)) = a(x) * q(x) \\ x * 1 = x \\ 1 * x = x \end{array}$$

Cet ensemble est écrit avec le formalisme des $n$-clauses comme suit :

$$\begin{array}{l} [p(x) \not\simeq q(x) \mid n \simeq x] \\ p(0) = 1 \\ q(0) = 1 \\ p(s(x)) = p(x) * a(x) \\ q(s(x)) = a(x) * q(x) \\ x * 1 = x \\ 1 * x = x \end{array}$$

Voici le fichier d'entrée (de SuperInd) correspondant à l'ensemble de $n$-clauses ci-dessus :

```
set(loopdet).
formulas(sos).
```

```
N(x) | p(x) != q(x).
p(0)=1.
q(0)=1.
p(s(x)) = p(x)*a(x).
q(s(x)) = a(x)*q(x).
x*1=x.
1*x=x.
x*y= u*v | x!=u | y != v.
x*y = y*x.
end_of_list.
```

Nous avons ajouté la clause $x * y = u * v \mid x \mathrel{!}= u \mid y \mathrel{!}= v$ qui code le fait que si nous avons deux couples d'entiers $(u, v)$ et $(x, y)$ et que $u = x$ et $v = y$ alors nous pouvons déduire que leurs produits sont égaux c'est-à-dire $x*y = u*v$. Cette clause est nécessaire pour la détection de cycles. Si nous appliquons la paramodulation nous obtenons la clause $N(s(x)) \mid p(x) * a(x) \mathrel{!}= a(x) * q(x)$ et aucun cycle n'est engendré, car cette clause n'est pas identique à la clause $N(x) \mid p(x) \mathrel{!}= q(x)$ modulo un décalage, cependant il est clair que $N(x) \mid p(x) \mathrel{!}= q(x)$ est une conséquence logique de la clause obtenue (modulo un décalage). La clause ajoutée permet d'éliminer les termes $a(x)$ (en appliquant la résolution) et donc de retomber sur la première clause à un décalage près. Cet exemple montre que l'utilisation d'une procédure de preuve moins restrictive que les stratégies usuelles (c'est-à-dire celles permettant de garantir la complétude en logique du premier ordre) est parfois nécessaire pour générer un cycle et établir l'insatisfaisabilité d'un ensemble de $n$-clauses. Nous présentons maintenant le fichier de sortie généré par notre outil :

```
============================== PROOF =================================

% Proof 1 at 0.02 (+ 0.01) seconds.
% Given clauses 16.
% number of calls to fixpoint~: 1

 S_init ~:
 (2:  [ q(v0) != p(v0) if  n = v0 ].
)
 S_loop ~:
 (80:  [ q(v0) != p(v0) if  n = s(v0) ].
)
 The empty clauses ~:
 (12:  [  n = 0 ].
82:  [  n = s(0) ].
```

```
) max_rank 1

============================== end of proof ==========================
```

Le fichier de sortie contient le temps d'exécution, le nombre de clauses sélectionnées pour l'inférence, deux ensembles de clauses $S_{init}$, $S_{loop}$ et finalement les clauses $n = 0$ et $n = s(0)$ (pure contrainte) correspondant respectivement aux $n$-clauses $[\square \mid n \simeq 0]$ et $[\square \mid n \simeq s(0)]$.

## 9.3 Implémentation

### 9.3.1 Adaptation des procédures de Prover9

Avant de présenter comment est implémentée la détection de boucles, nous allons montrer les différentes modifications que nous avons réalisées dans le système de Prover9.

#### La normalisation

Chaque clause retenue est normalisée en utilisant la factorisation sur les littéraux $N(t)$ qui correspondent aux contraintes. Pendant le processing, nous vérifions si une clause est normalisée avec la fonction suivante :

```
BOOL is_normalized(Topform p)
```

Elle est implémentée dans $topform.c$ et elle vérifie si la clause $p$ est normalisée. Si la clause n'est pas normalisée, elle sera supprimée.

#### Le rang

Nous avons ajouté dans la structure $Topform$, qui représente une clause, un champ $rank$. Nous avons implémenté une fonction qui calcule le rang pour la stocker dans ce champ et elle possède la signature suivante :

```
void init_rank(Topform c)
```

#### Le blocage des inférences

Nous avons modifié les règles d'inférence de manière à ce que le rang n'augmente pas arbitrairement c'est-à-dire que nous considérons que les inférences qui augmentent le rang d'au plus 1. Nous utilisons, pour cela, l'écart entre le rang d'une nouvelle clause générée et le rang de ses clauses parentes qui ne doit pas être supérieur à 1 (sauf si une des clauses parentes est de rang $\perp$).

**Remarque 114** *Comme nous le verrons à la section 9.3.2, cette heuristique simplifie notablement l'implémentation de la détection de cycle. Elle préserve les résultats de complétude énoncés dans le chapitre 6.*

#### Construction de la clause $C{\downarrow_j}$

L'algorithme du point fixe CYCLE$_2$, présenté dans le chapitre 3, vérifie dans sa boucle principale si une clause $C{\downarrow_j}$ est subsumée par une autre clause de $S_{loop}$(où $C$ est une clause de $S_{init}$). Afin de simplifier cette vérification, nous construisons la clause $C{\downarrow_j}$ à partir de $C$, avec la fonction suivante :

```
Topform shift_cl(Topform C, int J)
```

L'algorithme consiste à récupérer le littéral correspondant à la contrainte (qui est de la forme $N(s^i(x))$), puis construire le terme $s^{i+j}(x)$ et enfin ajouter le nouveau littéral $N(s^{i+j}(x))$ à la partie clausale de $C$ qui, pour sa part, reste inchangée. Nous avons donc implémenté les outils suivants :

```
Literals param_of_topform(Literals lits)
```

Cette fonction (écrite dans *literal.c*) retourne le littéral $N(\dots)$ (qui représente la contrainte) de l'ensemble des littéraux d'une clause. Afin de construire le terme $s^{i+j}(x)$, il faut récupérer la $\mathcal{S}_I$-variable $x$ qui apparaît dans $N(s^i(x))$ et la fonction suivante (écrite dans *term.c*) retourne cette $\mathcal{S}_I$-variable :

```
Term var_term(Term a)
```

Lorsque nous récupérons la $\mathcal{S}_I$-variable nous construisons le terme $s^{i+j}(x)$ grâce à la fonction suivante :

```
Term shift(int J, Term x)
```

### 9.3.2 Implémentation de l'algorithme du point fixe

Nous avons présenté, dans le chapitre 3, deux algorithmes de recherche d'un point fixe (que nous avons nommés CYCLE$_1$ et CYCLE$_2$). Nous avons implanté ces deux algorithmes dans deux fonctions récursives écrites en langage $C$ en utilisant les outils de la bibliothèque *LADR* utilisée par Prover9. La signature de la fonction, correspondant à l'activation de la détection de boucle, est comme suit :

```
BOOL fixed_point(int I, int J, Clist sos, Clist usable)
```

Cette fonction retourne `V`si $(I, J)$ est un cycle, et cela en appelant CYCLE$_1$ ou CYCLE$_2$. Les deux ensembles $S_{init}$ et $S_{loop}$ sont obtenus et stockés de manière statique c'est-à-dire que nous avons modifié la structure de *Glob* définie dans *search.c* pour y ajouter les listes $Si$, $Sij$ et la liste des clauses de la forme $[\square \mid n \simeq \text{succ}^k(0)]$ avec $k \in [\![0, i+j]\!]$. Ainsi lorsqu'un cycle est obtenu, on affecte aux listes $Si$ et $Sij$ les ensembles de clauses correspondant au cycle. L'ensemble de clauses total $S$ est obtenu en additionnant les deux listes *sos* et *usable*[5]. La liste des clauses vides par rang, c'est-à-dire des clauses de la forme $[\square \mid n \simeq \text{succ}^k(0)]$ (où $k$ est le rang), est construite, et stockée dans le champ *empty_clauses* de la structure *Glob*.

Nous avons implanté les différents outils dans le fichier *loop.c*, nous présentons maintenant les plus importants :

### La génération des clauses

Nous avons vu dans le chapitre 3 que l'espace de recherche $S$ est hiérarchisé, grâce à la notion de rang, sous forme d'une partition $S[1], S[2], \ldots, S[i], \ldots$ (où $1, 2, i$ représentent les différents rangs). Nous avons vu aussi que les inférences qui augmentent le rang de plus de 1 ne sont pas retenues. Nous avons, donc, pour tout entier $i$, $(S[i] \cup S[\top]) \vdash (S[i+1] \cup S[\top])$. Cela simplifie la vérification de la relation de génération entre sous ensembles de rangs différents, car implicitement en bloquant les inférences, chaque rang $i$ (toutes les clauses de rang $i$) génère un rang $j$ si $i < j$. Dans le cadre d'une programmation dite par contrats, nous avons implémenté une fonction qui vérifie la relation de génération $\vdash_\delta$, dont la signature est comme suit :

```
BOOL generates(int I, Plist Si, Plist Sij)
```

Elle retourne `V` ssi $(S[i] \cup S[\top]) \vdash (S[ij] \cup S[\top])$[6]. $Si$ est un ensemble de $n$-clauses de rang $I$. Nous l'utilisons comme une assertion qui garantit qu'après les modifications dues à la procédure d'inférence ou l'algorithme du point fixe, nous avons toujours $(S[i] \cup S[\top]) \vdash S[ij]$. L'algorithme consiste à parcourir toutes les clauses de $Sij$ et à identifier tous les ancêtres, la fonction retourne `V`si chaque ancêtre vérifie un de ces trois cas de figure :

1. La clause ancêtre est de rang $\bot$.
2. La clause ancêtre est de rang $I$ et incluse dans $Si$.

5. *sos* et *usable* sont des *Clist* qui est un type de liste dans Prover9 et qui représente entre autres une liste de clauses.

6. *Si* et *Sij* sont des Plist. Nous utilisons ce type comme une liste de clauses de travail car contrairement au type *Clist* la *Plist* ne contient que des clauses et donc elle est plus légère et plus simple à manipuler.

3. La clause ancêtre est de rang $I$ et elle n'est pas incluse dans $Si$ mais elle est complètement générée par $Si$ avec les clauses de rang $\perp$ c'est-à-dire les clauses de $Si$ (avec les clauses de rang $\perp$) sont suffisantes pour dériver cette clause.

Le troisième cas est vérifié avec une fonction récursive :

```
BOOL parents_in_Si(Plist Si, Topform Ci, int I)
```

Cette fonction vérifie dans le cas où $Ci$ n'est pas incluse dans $Si$ que ses clauses parentes vérifient récursivement l'une des trois conditions définies ci-dessus. Ce cas apparaît lorsqu'une clause $Ci$ de rang $I$, ancêtre d'une clause de $Sij$, est supprimée (parce que son décalage n'est subsumée par aucune clause de $Sij$ ) alors que, par exemple, ses deux parents sont dans $Si$.

**La subsomption**

Le test de subsomption est déjà implanté dans une fonction, dans Prover9, nous avons donc utilisé cette fonction pour implémenter la relation de subsomption $\leq_{sub}$. Nous avons aussi implémenté une fonction de signature :

```
Topform find_notsubsumedcl(Plist Si, Plist Sij, int J)
```

Cette fonction retourne une clause $C$ de $Si$ telle que $C{\downarrow_j}$ n'est subsumée par aucune clause de $Sij$. Si aucune clause de ce type n'est obtenue, la fonction retourne un pointeur **NULL**.

### 9.3.3 Stratégies d'application de l'algorithme du point fixe

Nous avons implanté l'algorithme de recherche d'un point fixe qui vérifie si un couple d'entiers $(i, j)$ est un cycle. Afin d'utiliser cet algorithme, il est nécessaire de trouver une stratégie pour, d'une part, choisir les couples d'entiers sur lesquels appliquer la fonction, et d'autre part, trouver le moment d'activer et renouveler l'application de cet algorithme pour limiter les vérifications et les calculs inutiles. L'activation est contrôlée par l'option $assign(start\_rank, n)$ qui active la détection de boucle à partir du rang $n$. Nous utilisons une heuristique qui est une sorte de recherche exhaustive. La stratégie consiste à appliquer la détection de boucles dans la grande boucle **Tant que** (voir section 9.1.2), plus exactement, à chaque sélection d'une clause active. La raison est simple, nous pouvons supposer que l'ajout d'une nouvelle clause à *usable* et l'application des inférences correspondantes est susceptible de compléter un cycle. L'algorithme du point fixe teste tous les couples d'entiers $(i, j)$ tels que $i + j \leq Rank\_max$ où $Rank\_max$ correspond au plus grand rang d'une clause tel que toutes les clauses de la forme $n \not\simeq \text{succ}^k(0)$ (avec

$k \leq Rank\_max$) sont générées. Nous utilisons un tableau de booléens $tab\_rank$ défini comme suit :

— $tab\_rank[k] = \texttt{V}$ si la clause $n \not\simeq \text{succ}^k(0)$ a bien été générée.

— $tab\_rank[k] = \texttt{F}$ sinon.

Ainsi la fonction $compute\_rank\_max(int[]tab)$ renvoie $Rang\_max$ en faisant un simple parcours du tableau $tab\_rank$ passé en argument. Nous avons aussi implémenté une stratégie aléatoire (activée avec l'option $set(random\_loopdet)$) c'est-à-dire que le choix du couple d'entiers est réalisé de manière probabiliste. Nous utilisons la fonction du langage $C$ $random()$ pour tirer aléatoirement un entier $i$ entre 1 et le rang maximal $Rank\_max$ puis tirer un autre entier $j'$ entre $i$ et le rang maximal, correspondant à $i + j$ dans SuperInd.

### 9.3.4 Détection de la satisfaisabilité

L'outil SuperInd peut détecter la satisfaisabilité d'un ensemble de clauses si certaines conditions sont vérifiées. Ces conditions sont les suivantes. Premièrement, les clauses doivent appartenir à une classe admissible. Deuxièmement, une clause de la forme $n \not\simeq \text{succ}^k(0)$ pour un certain $k \in \mathbb{N}$, n'est pas générée. Finalement, l'ensemble des clauses de rang $k$ est saturé par rapport au rang c'est-à-dire que toute clause générée de rang $k$ est redondante. L'interprétation $n = k$ est donc un modèle possible pour l'ensemble de départ. L'implémentation de ces conditions consiste, dans un premier temps, à détecter si la classe est admissible. Nous vérifions cette propriété en surveillant les clauses supprimées et en testant si elles n'ont pas été éliminées parce qu'elles ne respectaient pas les conditions d'une classe admissible. Ensuite, comme les clauses sont ordonnées par rang, lorsque qu'on sélectionne une clause active, si celle-ci est de rang supérieur à $Rank\_max$, on arrête la recherche car la clause de la forme $n \not\simeq \text{succ}^k(0)$ où $k = Rank\_max$ n'a pas été générée et surtout ne peut plus être dérivée (car les inférences augmentent la valeur du rang).

#### Le cas des $\alpha$-clauses et les clauses multi-paramètres

SuperInd ne traite pas encore le cas des $\alpha$-clauses, mais peut traiter les exemples isomorphe à $\mathbb{N}$ où $\mathcal{S}_I$ se réduit à une seule sorte de la forme $\{a, f\}$. SuperInd ne traite pas le cas des clauses avec plusieurs paramètres, comme le cas de deux variables existentielles.

# Chapitre 10

# Expérimentations

Nous présentons dans ce chapitre les différentes expérimentations que nous avons menées pour tester l'outil SuperInd. Dans un premier temps, nous décrivons les "benchmarks" que nous avons utilisés. Ensuite, nous présentons les résultats des différents tests et nous les analysons en vérifiant, entre autres, leurs conformités avec les résultats théoriques. Pour finir, nous comparons SuperInd avec d'autres outils.

## 10.1 Benchmarks

### 10.1.1 Schémas de formules propositionnelles

Dans le chapitre 4, nous avons introduit les schémas de formules en logique propositionnelle et leur traduction vers notre formalisme c'est-à-dire sous forme d'ensembles de $n$-clauses. Nous présentons, dans cette section, deux types de schémas que nous avons testés, le premier étant le circuit additionneur et le deuxième est la comparaison entre vecteurs de $n$ bits. Ces schémas permettent de traduire n'importe quelle formule (sans quantificateur) de l'arithmétique de Presburger en un ensemble de $n$-clauses sat-équivalent, ce qui permet d'obtenir facilement un grand nombre d'exemples.

#### Le circuit additionneur à propagation de retenue

Nous considérons un circuit additionneur à propagation de retenue qui calcule la somme de deux vecteurs de $n$ bits qui lui même est composé de $n$ additionneurs à 1 bit. On note $p_i$ et $q_i$ le $i$ ème bit des deux opérandes, $r_i$ le $i$ ème bit du résultat et $c_{i+1}$ est la retenue qui va être propagée du rang $i$ au rang $i+1$ (pour cela $c_1 = 0$).

Nous allons introduire les notations suivantes ($\oplus$ correspond au "ou exclusif") :

$$Sum_i(p, q, c, r) \stackrel{\text{def}}{=} r_i \Leftrightarrow (p_i \oplus q_i) \oplus c_i$$

$$Carry_i(p, q, c) \stackrel{\text{def}}{=} c_{i+1} \Leftrightarrow (p_i \wedge q_i) \vee (c_i \wedge p_i) \vee (c_i \wedge q_i).$$

Pour ensuite définir la formule suivante :

$$Adder(p, q, c, r) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} Sum_i(p, q, c, r) \wedge \bigwedge_{i=1}^{n} Carry_i(p, q, c) \wedge \neg c_1$$

qui représente, avec la contrainte $n \geq 1$, un schéma de formules propositionnelles correspondant au circuit additionneur (qui modélise l'addition $r = p + q$).

### Le circuit additionneur à retenue anticipée

Nous considérons un circuit additionneur à retenue anticipée qui garde la même formule ci-dessus, excepté pour le calcul de la retenue $Carry_i$. La retenue anticipée est calculée en utilisant deux circuits intermédiaires :

— La retenue *propagée* définie par la formule :

$$Propagate_i(p, q) \stackrel{\text{def}}{=} p_i \oplus q_i$$

— La retenue *générée* définie avec la formule :

$$Generate_i(p, q) \stackrel{\text{def}}{=} p_i \wedge q_i$$

La retenue anticipée s'écrit alors :

$$Carry_i(p, q, c) \stackrel{\text{def}}{=} c_{i+1} \Leftrightarrow (Generate_i(p, q) \vee Propagate_i(p, q) \wedge c_i).$$

L'additionneur à circuit anticipé permet de réduire le temps de calcul en estimant la valeur de la retenue.

### Comparaisons entre vecteurs de $n$ bits

Comparer deux vecteurs de $n$ bits consiste à les comparer bit à bit suivant un ordre lexicographique, ce qui peut être décrit par un schéma de formules propositionnelles. Nous pouvons, par exemple, coder la relation $X \leq Y$ (où $X$ et $Y$ sont des vecteurs de $n$ bits) avec le schéma suivant :

$$Temp_0 \wedge Temp_n \wedge \bigwedge_{i=1}^{n} (Temp_i \Leftrightarrow ((X_i \Leftrightarrow Y_i) \wedge Temp_{i-1} \vee \neg X_i \wedge Y_i))$$

Où $X_i$ (respectivement) $Y_i$ dénote le $i^{eme}$ bit de $X$ (respectivement) de $Y$, et $Temp$ sert à stocker le résultat de comparaison au bit précédent.

Nous utilisons par la suite SuperInd pour montrer que les fonctions et relations définies ci-dessus satisfont les propriétés usuelles : neutralité de 0, commutativité, associativité, équivalence entre les deux définitions de l'addition, transitivité, antisymétrie et totalité de $\leq$. Nous prouvons également certaines égalités arithmétiques élémentaires. Afin de tester la satisfaisabilité des schémas que nous avons décrits, nous avons implémenté un algorithme qui transforme automatiquement (en un temps polynomial) un schéma propositionnel construit avec des connecteurs itérés de la forme $\bigwedge_{i=a}^{n+b} \phi$ or $\bigvee_{i=a}^{n+b} \phi$ (comme celui utilisé dans la modélisation du circuit additionneur) en un ensemble de $n$-clauses sat-équivalent. Nous avons décrit cette transformation dans le chapitre 4. Nous utilisons les abréviations suivantes : un additionneur parallèle à propagation de retenue est noté *Additionneur PPR* et un additionneur parallèle à retenue anticipée est noté *additionneur PRA*.

### 10.1.2 Schémas de formules provenant de CERES

Nous avons aussi considéré des exemples provenant de l'outil CERES (Cut Elimination by RESolution), un outil développé par le groupe "Théorie et Logique" de l'université technique de Vienne" (voir `http://www.logic.at/ceres/`). La méthode CERES (voir par exemple [12, 13]) est un algorithme de transformation de preuves non-analytiques, basé sur l'élimination de coupures dans la logique du premier ordre. L'élimination de coupures est habituellement fondée sur le théorème du même nom, énoncé et montré par Gentzen [42]. Cette méthode consiste à éliminer les coupures d'une preuve, dans le calcul des séquents, et générer une nouvelle preuve sans les lemmes qui la composent.

L'algorithme de CERES consiste à extraire de la preuve $\pi$, que nous voulons transformer, un ensemble de clauses insatisfaisable $S(\pi)$, appelé *ensemble caractéristique*. La réfutation de $S(\pi)$ par résolution permet, en combinant avec une projection de la preuve $\pi$, de construire une preuve sans coupures. L'ensemble caractéristique est étendu aux schémas de preuves du premier ordre dans [37, 76, 38], et cela dans le but de traiter les preuves mathématiques faisant appel à l'induction [1](car les preuves mathématiques par induction ne peuvent être exprimées en logique du premier ordre). L'ensemble caractéristique obtenu pour ces preuves n'est pas un simple ensemble de clauses, mais un schéma d'ensembles de clauses que nous pouvons coder avec un ensembles de $n$-clauses, et que nous pouvons ainsi traiter (réfuter où valider) avec notre outil SuperInd. Nous présentons un exemple d'ensemble caractéristique provenant de [37] en utilisant les notations de [37], c'est-

1. Cette transformation en mathématique correspond à analyser une preuve et supprimer les lemmes intermédiaires, pour réduire sa taille et détecter des concepts analytiques valides.

à-dire, une clause de la forme $\neg A_1 \vee \ldots \neg A_n \vee B_1, \ldots, B_m$ où $A_1, \ldots, A_n, B_1, \ldots, B_m$ sont des atomes sera représentée par le séquent : $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$. :

$$\vdash cl_{k+1}^{\psi_1,(|)}$$

$$cl_0^{\psi_2,(|\forall x(P(x)\rightarrow P(f(g(k,x)))))}, P(z_0) \vdash P(f(g(0,z_0)))$$

$$cl_0^{\psi_1,(|)} \vdash$$

$$cl_{k+1}^{\psi_2,(|\forall x(P(x)\rightarrow P(f(g(k,x)))))} \vdash cl_k^{\psi_2,(|\forall x(P(x)\rightarrow P(f(g(k,x)))))}$$

$$cl_{k+1}^{\psi_2,(|\forall x(P(x)\rightarrow P(f(g(k,x)))))}, P(g(k+1,z_{k+1})) \vdash P(f(g(k+1,z_{k+1})))$$

$$cl_{k+1}^{\psi_1,(|)} \vdash cl_{k+1}^{\psi_2,(|\forall x(P(x)\rightarrow P(f(g(k,x)))))}$$

$$cl_{k+1}^{\psi_1,(|)} \vdash P(a)$$

$$cl_{k+1}^{\psi_1,(|)}, P(f(g(k+1,a))) \vdash$$

qui est simplifié et codé par l'ensemble de $n$-clauses suivant :

$$\neg r(0) \vee \neg p(Z) \vee p(f(g(0,Z)))$$

$$-q(0)$$

$$-r(s(X)) \vee r(X)$$

$$\neg r(s(X)) \vee \neg p(g(X,Z)) \vee p(f(g(X,Z)))$$

$$-q(s(X)) \vee r(s(X))$$

$$p(a)$$

$$\neg q(s(X))$$

$$p(f(g(X,a)))$$

$$g(0,Z) = Z$$

$$g(s(X),Z) = f(g(X,Z))$$

$$[q(X) \mid n \simeq X]$$

où nous avons renommé les prédicats suivants :

— $cl_{k+1}^{\psi_1,(|)} \rightarrow q(k+1)$

— $cl_k^{\psi_2,(|\forall x(P(x)\rightarrow P(f(g(k,x)))))} \rightarrow r(k)$

Nous montrons aussi un lemme inductif utilisé dans la preuve de l'infinité des nombres premiers de Furstenberg [11] et exprimant le fait que le produit de nombres premiers est positif.

### 10.1.3 Exemples provenant de SPASS-FD

Nous avons introduit dans le chapitre 8 le calcul SFD (Superposition for Fixed Domains), qui est implémenté dans le démonstrateur SPASS. Nous avons testé SuperInd sur quelques exemples provenant des entrées de SFD. Nous présentons quelques-uns des ces exemples :

**Exemple 115** Nous considérons l'ensemble $S = \{G(s(0),0), G(x,y) \Rightarrow G(s(x),s(y))\}$ qui correspond aux axiomes qui codent l'ordre sur les entiers. Nous voulons montrer que $S \models \forall x\ G(s(x),x)$. L'ensemble $S$ est codé simplement avec notre formalisme (avec des $n$-clauses). La clause but devient, après sa négation, $\exists x\ \neg G(s(x),x)$ et elle est codée par la $n$-clause $[\neg G(s(x),x) \mid n \simeq x]$. On procède de la même manière pour $S = \{0 + y = y, s(x) + y = s(x + y)\}$ afin de montrer $S \models \forall x\ x + 0 = x$. ♣

**Exemple 116** Nous considérons l'ensemble $S = \{E(0), E(x) \Rightarrow E(s(s(x)))\}$ qui définit l'ensemble des nombres pairs. Nous voulons montrer que $S \not\models \forall x\ E(x)$. L'ensemble $S$ est codé simplement avec notre formalisme (avec des $n$-clauses). La clause but devient, après sa négation, $\exists x\ \neg E(x)$ et elle est codée par la $n$-clause $[\neg E(x) \mid n \simeq x]$. Notez que nous cherchons pas ici une réfutation, mais plutôt un contre exemple.

### 10.1.4 Autres exemples du premier ordre

Nous avons construit quelques exemples qui ne peuvent être traités par des démonstrateurs du premier ordre :

**Exemple 117** (*Permutation*) Nous considérons un terme $t$ de la forme $f(t_1, \ldots, t_m)$ (avec un $m$ fixé). Si l'un des termes $t_i$ est une constante $a$ et que nous réalisons différentes permutations sur les termes $t_1, \ldots, t_m$ alors nous pouvons montrer que la constante $a$ apparaît dans l'une des positions du terme $t$. Nous formalisons le problème pour $m = 3$ par les formules suivantes :

$$-e(x,x_1,x_2,x_3) \vee (e(s(x),x_1,x_2,x_3) \vee e(s(x),x_2,x_1,x_3) \vee e(s(x),x_3,x_2,x_1) \vee e(s(x),x_1,x_3,x_2)).$$

$$e(0,a,b,c).$$

La formule but est alors :

$$\forall n \in \mathbb{N}\ e(n,a,x_2,x_3) \vee e(n,x_2,a,x_3) \vee e(n,x_2,x_3,a)$$

**Exemple 118** (*Longueur d'une liste*) Nous montrons dans cet exemple qu'une liste de longueur $n$ contient un $n$-ème élément. Nous codons la longueur d'une liste comme suit :

$$length(l, n) \Leftrightarrow n = 0 \vee \exists x, l', n'(l = cons(x, l') \wedge n = s(n') \wedge length(l', n'))$$

Le $n$-ème élément est défini comme suit :

$$nth(x, l, n) \Leftrightarrow \exists y, l'\ l = cons(y, l') \wedge (n = s(0) \wedge x = y) \vee \exists n'\ (n = s(n') \wedge nth(x, l', n'))$$

La formule but est comme suit :

$$\forall n \in \mathbb{N}, \forall l\ (length(l, n) \wedge n \neq 0 \Rightarrow \exists x\ nth(x, l, n))$$

## 10.2 Résultats

Nous avons testé notre outil avec plusieurs paramètres mais aucune validation n'a été réalisée avec des tests structurels. Nous présentons les résultats des tests pour chacun des deux algorithmes du point fixe que nous avons implémentés (CYCLE$_1$ et CYCLE$_2$). En utilisant une stratégie basée sur une recherche exhaustive (voir le chapitre 9), chaque algorithme est testé avec l'option $set(index\_flat)$ qui restreint la détection de boucles aux clauses $\mathcal{S}_I$-plates. Nous avons vu dans le chapitre 6 que la détection de boucles est enclenchée à partir d'un certain rang $start\_rank$ qui est un des paramètres de SuperInd. Nous avons donc testé les deux algorithmes du point fixe avec différentes valeurs de $start\_rank$. Nous avons aussi testé la stratégie probabiliste que nous avons décrite au chapitre 9, pour comparer avec la recherche exhaustive de cycles.

### 10.2.1 Résultats préliminaires

Les tableaux des figures 10.1 et 10.2 résument les différents résultats que nous avons obtenus en appliquant les deux algorithmes de recherche d'un point fixe (CYCLE$_1$ et CYCLE$_2$). Ces résultats sont obtenus en appliquant la détection de boules sur les clauses $\mathcal{S}_I$-plates (c'est-à-dire que l'option $set(index\_flat)$ est activée). Chaque ligne contient le théorème que nous voulons montrer, le temps d'exécution, le nombre d'appels à l'algorithme du point fixe CYCLE$_1$ ou CYCLE$_2$, le nombre total de clauses générées et le nombre de clauses que contient l'invariant obtenu.

| Théorème | Temps d'exécution (s) | # d'appels à $\text{CYCLE}_1$ | # clauses générées | # des clauses de l'invariant |
|---|---|---|---|---|
| Additionneur PRA | 0.01 | 4 | 255 | 3 |
| $(A + 0 = A)$ | 0.01 | 4 | 255 | 3 |
| Additionneur PRA (commutativité) | 0.02 | 6 | 963 | 4 |
| Additionneur PRA (associativité) | 1.73 | 4 | 142527 | 10 |
| Additionneur PRA $(3 + 4 = 7)$ | 1.06 | 10 | 102397 | 7 |
| Unicité du résultat (Additionneur PRA) | 0.03 | 4 | 1885 | 4 |
| Additionneur PPR $(A + 0 = A)$ | 0.76 | 3 | 116533 | 3 |
| Additionneur PPR (commutativité) | 0.09 | 4 | 3487 | 4 |
| Additionneur PPR (associativité) | 0.68 | 3 | 52066 | 10 |
| Equivalence entre un additionneur PPR et un additionneur PRA | 0.06 | 4 | 1786 | 4 |
| Comparaisons $(n > 0)$ | 0.01 | 1 | 143 | 4 |
| $n_1 \geq n_2 \wedge n_2 \geq n_1 \Rightarrow n_1 = n_2$ | 0.01 | 1 | 288 | 4 |
| Totalité : $n_1 > n_2 \vee n_2 \geq n_1$ | 0.01 | 1 | 288 | 5 |
| Transitivité : $n_1 \leq n_2 \wedge n_2 \leq n_3 \Rightarrow n_1 \leq n_3$ | 0.02 | 1 | 877 | 7 |
| Presburger $(n + p > n)$ | 0.43 | 4 | 31818 | 5 |
| CERES ex1 (Propositional) | 0.01 | 4 | 92 | 3 |
| CERES ex1Original (Propositional) | 0.01 | 4 | 153 | 3 |
| CERES ex2 (Propositional) | 0.01 | 4 | 325 | 3 |
| CERES (First order) | 0.01 | 41 | 78 | 4 |
| Furstenberg inductive lemma $\bigwedge_{i=1}^{n} p_i > 0 \Rightarrow p_1 \times \cdots \times p_n > 0$ | 0.03 | 1 | 1176 | 1 |
| SPASS-FD Addition | 0.01 | 1 | 399 | 2 |
| SPASS-FD int order | 0.01 | 1 | 23 | 1 |
| Permutation | 0.03 | 3 | 1563 | 2 |
| Longueur d'une liste | 0.01 | 8 | 213 | 1 |

FIGURE 10.1 – Résumé des tests pour $\text{CYCLE}_1$

| Théorème | Temps d'exécution (s) | # d'appels à CYCLE$_2$ | # clauses générées | # des clauses de l'invariant |
|---|---|---|---|---|
| Additionneur PRA ($A + 0 = A$) | 0.01 | 4 | 255 | 3 |
| Additionneur PRA (commutativité) | 0.02 | 6 | 963 | 4 |
| Additionneur PRA (associativité) | 1.75 | 4 | 142527 | 10 |
| Additionneur PRA ($3 + 4 = 7$) | 1.32 | 10 | 102397 | 68 |
| Unicité du résultat (Additionneur PRA) | 0.04 | 4 | 1885 | 4 |
| Additionneur PPR ($A + 0 = A$) | 0.76 | 3 | 116533 | 3 |
| Additionneur PPR (commutativité) | 0.09 | 4 | 3487 | 4 |
| Additionneur PPR (associativité) | 0.71 | 3 | 52066 | 10 |
| Equivalence entre un additionneur PPR et un additionneur PRA | 0.06 | 4 | 1786 | 4 |
| Comparaisons ($n > 0$) | 0.01 | 1 | 143 | 4 |
| $n_1 \geq n_2 \wedge n_2 \geq n_1 \Rightarrow n_1 = n_2$ | 0.01 | 1 | 288 | 4 |
| Totalité : $n_1 > n_2 \vee n_2 \geq n_1$ | 0.01 | 1 | 288 | 5 |
| Transitivité : $n_1 \leq n_2 \wedge n_2 \leq n_3 \Rightarrow n_1 \leq n_3$ | 0.02 | 1 | 877 | 7 |
| Presburger ($n + p > n$) | 0.43 | 4 | 31818 | 14 |
| CERES ex1 (Propositional) | 0.01 | 4 | 92 | 3 |
| CERES ex1Original (Propositional) | 0.01 | 4 | 153 | 4 |
| CERES ex2 (Propositional) | 0.01 | 4 | 325 | 6 |
| CERES (First order) | 0.01 | 41 | 78 | 4 |
| Furstenberg inductive lemma $\bigwedge_{i=1}^{n} p_i > 0 \Rightarrow p_1 \times \cdots \times p_n > 0$ | 0.03 | 1 | 1176 | 1 |
| SPASS-FD addition | 0.01 | 1 | 399 | 1 |
| Permutation | 0.03 | 3 | 1563 | 2 |
| Longueur d'une liste | 0.01 | 8 | 213 | 1 |

FIGURE 10.2 – Résumé des tests pour CYCLE$_2$

### 10.2.2 Analyse

La première remarque que nous pouvons faire sur les tableaux des figures 10.1 et 10.2, est que plusieurs résultats sont les mêmes dans les deux tableaux. En particulier, le nombre d'appels à l'algorithme de recherche d'un point fixe et le nombre de clauses générées, sont identiques pour les deux algorithmes. Cela est dû au fait que l'appel aux deux algorithmes intervient au même moment car il dépend dans un premier temps du rang maximum de départ, qui est une valeur fixe (stockée dans le paramètre $start\_rank$). Puis l'appel à l'algorithme se fait à chaque sélection d'une clause active. Nous remarquons aussi que $\textsc{Cycle}_1$ est plus efficace que $\textsc{Cycle}_2$, ce qui est surtout significatif lorsque le nombre de clauses générées est très grand, comme l'exemple de l'additionneur PRA ($3 + 4 = 7$). Cette différence de performance est due aux deux facteurs suivants. Premièrement, la relation d'implication immédiate $\sqsupseteq$ pour $\textsc{Cycle}_1$ est l'identité (à un renommage de variables près), tandis que c'est la *subsomption* pour $\textsc{Cycle}_2$, et la subsomption est bien plus coûteuse. Deuxièmement, la structure des deux algorithmes est différente, pour tester un cycle $(i, j)$ par exemple, $\textsc{Cycle}_1$ commence la recherche avec l'ensemble des clauses de rang $i$ qui sont ancêtres des clauses de la forme $[\square \mid n \simeq \text{succ}^k(0)]$ (pour tout $k \leq Rank\_max$), alors que $\textsc{Cycle}_2$ commence la recherche avec l'ensemble entier des clauses de rang $i$, ce qui peut être très coûteux lorsque l'ensemble de clauses est très important. Les deux algorithmes étant corrects et complets, ils retournent les mêmes ensembles de cycles, ce qui montre que la subsomption ne joue pas un rôle fondamental, dans le sens où elle ne permet pas d'obtenir plus de cycles en pratique.

Nous pouvons aussi remarquer que pour certains théorèmes le nombre de clauses dans l'invariant diffère d'un algorithme à un autre. En effet, si nous considérons par exemple l'inégalité de vecteurs de $n$ bits $N + P > N$, nous constatons que le nombre de clauses dans l'invariant est de l'ordre de 5 clauses pour $\textsc{Cycle}_1$ et de 14 clauses pour $\textsc{Cycle}_2$. Nous montrons sur les figures 10.3 et 10.4 les deux invariants, plus exactement l'ensemble $S_{init}$ obtenu avec les deux algorithmes. L'invariant obtenu avec $\textsc{Cycle}_2$ contient toutes les clauses de l'invariant obtenu avec $\textsc{Cycle}_1$ mais contient des clauses en plus. Ce surplus de clauses peut s'expliquer par le fonctionnement de $\textsc{Cycle}_2$ qui, grossièrement, considère toutes les clauses d'un certain rang, puis supprime les clauses qui ne peuvent être dans l'invariant. Ce résultat est conforme à l'analyse théorique, car $\textsc{Cycle}_1$ et $\textsc{Cycle}_2$ retournent respectivement les invariants minimal et maximal correspondant à un cycle donné.

```
  S_init ~:
 (976:  [ S1(v0) if  n = s(v0) ].
  984:  [ S2(v0) if  n = s(v0) ].
1630:  [ -$T1$(v0) if  n = s(v0) ].
1631:  [ S3(v0) if  n = s(v0) ].
1632:  [ T2(v0) if  n = s(v0) ].
)
```

FIGURE 10.3 – L'invariant obtenu avec CYCLE$_1$

### 10.2.3 Le choix du paramètre *start_rank*

Nous avons vu dans le chapitre 9, que la détection de boucles est enclenchée lorsque *Rank_max* est supérieur au paramètre *start_rank*. Nous avons mené quelques expérimentations pour déterminer la valeur optimale de *start_rank* pour un théorème donné. Nous avons donc considéré deux théorèmes de tailles différentes. Nous avons lancé SuperInd avec différentes valeurs de *start_rank* et nous avons mesuré le temps d'exécution, le nombre d'appels à CYCLE$_1$ et CYCLE$_2$ et le nombre de clauses générées.

Le tableau de la figure 10.5 correspond aux tests réalisés avec l'additionneur PRA ($3+4=7$). Nous avons choisi ce théorème pour sa grande taille (plus de $10^5$ clauses générées). Nous pouvons remarquer, que la valeur optimale de *start_rank* est 3 pour les deux algorithmes. Les résultats se dégradent pour des valeurs inférieurs où supérieurs à 3 même si cette dégradation est plus faible pour des valeurs supérieurs à 3. Nous pouvons, aussi, remarquer que le nombre d'appels à CYCLE$_1$ et CYCLE$_2$ est très grand pour les valeurs de *start_rank* inférieures à 3, ce qui augmente le temps d'exécution. Ce nombre d'appels est très grand car les clauses qui constituent le cycle ne sont pas encore toutes générées et comme l'appel se fait à chaque sélection d'une clause active et que le nombre de clauses générées au début de la procédure n'est pas très grand, nous avons un bon nombre d'appels qui sont inutiles. Notez aussi que que le nombre de clauses générées augmente à partir de la valeur 3 alors que toutes les clauses du cycle sont déjà générées (car nous avons déjà détecté un cycle avec 102397 clauses générées). Cela est dû au fait que la détection de boucles est enclenchée lorsque toutes les clauses de la forme $[\Box \mid n \simeq \mathrm{succ}^k(0)]$ (pour tout $k \leq Rank\_max$) sont générées. Par exemple, si $R$ est le *Rang_max* à partir duquel toutes les clauses du cycle sont générées, comme la condition $Rank\_max > start\_rank$ doit être vérifiée, il faudra dériver toutes les clauses de la forme $[\Box \mid n \simeq \mathrm{succ}^k(0)]$ où $R \leq k \leq start\_rank$ pour lancer la détection de boucles. Cela explique aussi l'augmentation du nombre d'appels à l'algorithme du point fixe, car la recherche d'un cycle est exhaustive et donc tous

```
   S_init ~:
  (1633:  [ $MplusP$(v0) | -$M$(v0) if  n = s(v0) ].
1632:  [ T2(v0) if  n = s(v0) ].
1631:  [ S3(v0) if  n = s(v0) ].
1630:  [ -$T1$(v0) if  n = s(v0) ].
1629:  [ -$P$(v0) if  n = s(v0) ].
1538:  [ -$MplusP$(v0) | $M$(v0) if  n = s(v0) ].
1518:  [ -$T1$$(v0) if  n = s(v0) ].
1458:  [ $S3$(v0) if  n = s(v0) ].
1416:  [ $T2$(v0) if  n = s(v0) ].
984:  [ S2(v0) if  n = s(v0) ].
983:  [ $S2$(v0) if  n = s(v0) ].
976:  [ S1(v0) if  n = s(v0) ].
975:  [ $S1$(v0) if  n = s(v0) ].
111:  [ -$P$$(v0) if  n = s(v0) ].
)
```

FIGURE 10.4 – L'invariant obtenu avec $\text{CYCLE}_2$

les cycles jusqu'au $Rank\_max$ sont testés et lorsqu'on augmente $Rank\_max$, on augmente le nombre de cycles possibles. Ces résultats confortent aussi le fait que $\text{CYCLE}_1$ est plus efficace que $\text{CYCLE}_2$, car le temps d'exécution de $\text{CYCLE}_2$ est presque de l'ordre de trois fois le temps d'exécution de $\text{CYCLE}_1$, lorsque le nombre d'appels est très grand.

Le tableau de la figure 10.6 correspond aux tests réalisés sur le théorème attestant de la symétrie de l'égalité. Ce théorème est de petite taille et les résultats ne sont pas significatifs car le nombre de clauses n'est pas très grand, même si les résultats correspondent à l'allure générale du premier tableau c'est-à-dire le nombre de clauses générées, le temps d'exécution et le nombre d'appels à l'algorithme du point fixe augmentent avec l'augmentation de la valeurs de $start\_rank$.

Pour conclure, la valeur de $start\_rank$ ne doit pas être très petite, car l'augmentation du nombre d'appels à l'algorithme du point fixe peut s'avérer très coûteuse. Elle ne doit pas être très grande non plus car, d'une part, le nombre de cycles potentiels augmentent, et d'autre part, le nombre de clauses générées est plus important.

### 10.2.4 Détection de la satisfaisabilité

Nous présentons maintenant quelques exemples satisfaisables :

| Additionneur PRA $(3 + 4 = 7)$ | | | | | | |
|---|---|---|---|---|---|---|
| valeurs de *start_rank* | CYCLE$_1$ | | | CYCLE$_2$ | | |
| | nombre d'appels | # de clauses générées | Temps d'exécution (s) | nombre d'appels | # de clauses générées | Temps d'exécution (s) |
| 1 | 8953 | 102397 | 7.26 | 8953 | 102397 | 23.44 |
| 2 | 5284 | 102397 | 6 | 5284 | 102397 | 21.51 |
| 3 | 10 | 102397 | 1.06 | 10 | 102397 | 1.34 |
| 4 | 13 | 166434 | 1.67 | 13 | 166434 | 1.96 |
| 5 | 16 | 245548 | 2.44 | 16 | 245548 | 2.75 |
| 6 | 19 | 339739 | 3.36 | 19 | 339739 | 3.74 |
| 7 | 22 | 449007 | 4.47 | 22 | 449007 | 4.84 |

FIGURE 10.5 – Résultats des tests correspondant aux différentes valeurs de *start_rank* pour un théorème de grande taille

| $n_1 \geq n_2 \wedge n_2 \geq n_1 \Rightarrow n_1 = n_2$ | | | | | | |
|---|---|---|---|---|---|---|
| valeurs de *start_rank* | CYCLE$_1$ | | | CYCLE$_2$ | | |
| | nombre d'appels | # de clauses générées | Temps d'exécution (s) | nombre d'appels | # de clauses générées | Temps d'exécution (s) |
| 1 | 3 | 199 | 0.01 | 3 | 199 | 0.01 |
| 2 | 4 | 255 | 0.01 | 4 | 255 | 0.01 |
| 3 | 5 | 316 | 0.01 | 5 | 316 | 0.01 |
| 8 | 10 | 696 | 0.02 | 10 | 696 | 0.02 |
| 10 | 12 | 883 | 0.02 | 12 | 883 | 0.02 |

FIGURE 10.6 – Résultats des tests correspondant aux différentes valeurs de *start_rank* pour un théorème de taille moyenne

**Exemple 119** Cet exemple est adapté de [20]. L'ensemble de clauses est le suivant :

$$\begin{array}{ll} (1) & P(f(a,\diamond)^{n+1}.b) \\ (2) & \neg P(c) \\ (3) & f(a,b) \simeq d \\ (4) & f(a,d) \simeq c \end{array}$$

Le terme $f(a,\diamond)^n.b$ est un terme avec exposant entier (au sens de [33]), dénotant le terme $f(a, f(\ldots, f(a,b)\ldots))$, avec $n$ occurrences de $f$. La clause 1 est ensuite codée avec l'ensemble des $n$-clauses suivantes :

$$\begin{array}{ll} (5) & S(0) = f(a,b) \\ (6) & S(s(x)) = f(a, S(x)) \\ (7) & [P(S(x)) \not\simeq \mathtt{V} \mid n \simeq x] \end{array}$$

$S(n)$ encode le terme $f(a,\diamond)^b.n+1$. A noter que la variable $n$ est quantifiée existentiellement (à la différence de [20] où elle est quantifiée universellement), ce qui explique que l'ensemble est satisfaisable. ♣

**Exemple 120** Nous construisons des exemples satisfaisables à partir des schémas de formules propositionnelles. Si nous considérons l'exemple de l'additionneur où on montre la commutativité nous avons un schéma qui calcule $X+Y$ et un autre qui calcule $Y+X$ et la formule but vérifie l'égalité des deux résultats. Il suffit donc d'enlever la formule but pour obtenir un exemple satisfaisable. Nous procédons de la même manière pour les autres exemples insatisfaisables testés ci-dessus. ♣

La figure 10.7 résume les résultats obtenus pour les exemples satisfaisables dans un tableau contenant respectivement l'exemple testé, la valeur possible du paramètre et le temps d'exécution.

| Exemples | Valeur possible pour $n$ | Temps d'exécution (s) |
|---|---|---|
| Additionneur PPR Commutativité | 0 | 0.01 |
| Additionneur PPR Associativité | 0 | 0.02 |
| Equivalence entre additionneurs PPR et PRA | 0 | 0.01 |
| Exemple 116 (Even) | $s(0)$ | 0.01 |
| Exemple 119 (schémas de termes) | $s(0)$ | 0.01 |

FIGURE 10.7 – Résumé des exemples satisfaisables

La première remarque que nous pouvons faire est que les résultats n'indiquent pas le nombre d'appels à l'algorithme du point fixe. Cela est dû au fait que la preuve de satisfiabilité ne nécessite pas la détection de boucle : elle est réalisée

| Théorème | Temps d'exécution (s) | # d'appels à $\text{CYCLE}_1$ | # clauses générées | # des clauses de l'invariant |
|---|---|---|---|---|
| Additionneur PRA ($A + 0 = A$) | 0.01 | 1 | 255 | 3 |
| Additionneur PRA (commutativité) | 0.02 | 1 | 963 | 4 |
| Additionneur PRA (associativité) | 1.72 | 1 | 142527 | 10 |
| Additionneur PRA ($3 + 4 = 7$) | 2.16 | 883 | 102397 | 7 |
| Unicité du résultat (Additionneur PRA) | 0.03 | 1 | 1885 | 4 |
| Additionneur PPR ($A + 0 = A$) | 0.77 | 1 | 142692 | 3 |
| Additionneur PPR (commutativité) | 0.09 | 1 | 3487 | 4 |
| Additionneur PPR (associativité) | 0.67 | 1 | 52066 | 10 |
| Equivalence entre un additionneur PPR et un additionneur PRA | 0.04 | 1 | 1855 | 4 |
| ... | | | | |

FIGURE 10.8 – Résumé des tests pour $\text{CYCLE}_1$ avec la méthode probabiliste

en saturant l'ensemble de clauses pour une valeur donnée du paramètre, ce qui peut-être réalisé en utilisant les règles d'inférence de Prover9. Nous remarquons aussi que tous les exemples avec l'additionneur ont des modèles où $n = 0$ même si toutes les valeurs de $\mathbb{N}$ sont possibles, ce qui est un simple choix (de renvoyer la première valeur trouvée). Il suffit par exemple d'ajouter la clause $n \neq 0$, pour que SuperInd retourne $n = s(0)$. Le même raisonnement est appliqué pour l'exemple 116, où tous les nombres impairs sont des valeurs possibles pour $n$.

### 10.2.5 Comparaisons avec la méthode probabiliste

Le tableau de la figure 10.9, résume les expérimentations réalisées pour comparer la méthode probabiliste à la recherche exhaustive de cycles. Nous remarquons que les temps d'exécution sont légèrement meilleurs avec la méthode probabiliste excepté pour certains théorèmes où les résultats se dégradent fortement. Le nombre d'appels à l'algorithme du point fixe est est réduit, ce qui fait que le temps d'exécution est meilleur. Néanmoins, ce nombre d'appel peut exploser dans certains cas, notamment lorsque nous retardons la détection de boucles (en augmentant la valeur de $start\_rank$), ce qui augmente les temps d'exécution.

### 10.2.6 Comparaison avec RegStab

La figure 10.9 compare sur quelques exemples les temps d'exécutation de SuperInd à ceux obtenus avec le système RegSTAB développé par V. Aravantinos [5, 2]. Notez que les temps d'exécution de SuperInd sont légèrement meilleurs que les résultats préliminaires de la figure 10.1. La différence est dû au fait que le temps est calculé à partir du lancement de SuperInd pour les résultats préliminaires, tandis qu'il correspond au début de la recherche, notamment après la clausification,

| Théorème | Temps d'exécution de SuperInd (s) | Temps d'exécution de RegStab (s) |
|---|---|---|
| Additionneur PRA ($A + 0 = A$) | 0.001 | 0.002 |
| Additionneur PRA (commutativité) | 0.003 | 0.008 |
| Additionneur PRA (associativité) | 1.67 | 0.375 |
| Additionneur PRA ($3 + 4 = 7$) | 0.96 | 0.197 |
| Additionneur PPR ($A + 0 = A$) | 0.75 | 0.002 |
| Additionneur PPR (commutativité) | 0.09 | 0.016 |
| Additionneur PPR (associativité) | 0.68 | 2.219 |
| Equivalence entre un additionneur PPR et un additionneur PRA | 0.027 | 0.011 |
| Comparaisons ($n > 0$) | 0.001 | 0.001 |
| $n_1 \geq n_2 \wedge n_2 \geq n_1 \Rightarrow n_1 = n_2$ | 0.001 | 0.001 |
| Totalité : $n_1 > n_2 \vee n_2 \geq n_1$ | 0.001 | 0.001 |
| Transitivité : $n_1 \leq n_2 \wedge n_2 \leq n_3 \Rightarrow n_1 \leq n_3$ | 0.002 | 0.001 |
| Presburger ($n + p > n$) | 0.33 | 0.002 |

FIGURE 10.9 – Résumé des tests pour $\text{CYCLE}_2$

pour les résultats de la figure 10.9. Nous remarquons que les résultats obtenus avec SuperInd et RegStab sont très proches pour les théorèmes simples. Pour les autres exemples, nous remarquons que RegStab est légèrement meilleur (même si SuperInd est meilleur pour certains exemples). Cela peut s'expliquer par le fait que la détection de boucles dans RegStab est appliquée sur des ensembles de littéraux, tandis qu'elle est appliquée sur des ensembles de clauses (donc des ensembles d'ensembles de littéraux) dans SuperInd, ce qui fait que la complexité de RegStab est meilleure que celle de SuperInd.

## Le cas des schémas de termes

Nous avons présenté dans le chapitre 8, les travaux concernant les schémas de termes qui sont décrits dans [20]. SuperInd ne termine pas dans la plupart des exemples contenant des schémas de termes. Cela est dû essentiellement à la définition inductive que nous utilisons pour représenter ces schémas de termes au premier ordre. Nous illustrons cela dans l'exemple simple suivant (provenant de [20]) :

**Exemple 121** Nous considérons un prédicat $p$ qui code le fait qu'une liste donnée contient un seul élément (c'est-à-dire que tous les membres de la liste sont égaux). $p$ peut être défini par $p(cons(x, \diamond)^N.nil)$. Nous testons ensuite, que $p(cons(a, \diamond)^N.nil)$ est vrai pour tout $n$ où $a$ est une constante. Ce problème peut être formalisé par

l'ensemble des clauses du premier ordre suivante :

1. $p(x) \vee \neg q(x, y)$
2. $q(nil, x)$
3. $q(cons(x, y), x) \vee \neg q(y, x)$
4. $\neg p(\underbrace{cons(a, cons(a, \ldots, cons(a, nil))))}_{n fois}$

où *nil* est la liste vide, *cons* est le constructeur d'une liste et $q(y, x)$ code le fait que $y$ est une liste ne contenant que l'élément $x$.

Nous allons définir inductivement la clause 4 en introduisant un nouveau terme $Si$ comme suit :

5. $Si(0) = nil$
6. $Si(s(x)) = cons(a, Si(x))$

La clause 4 s'écrit alors

$$\neg p(Si(n))$$

qui est codée par la $n$-clause suivante :

7. $[\neg p(Si(x)) \mid n \simeq x]$

Si nous appliquons SuperInd sur l'ensemble de clauses $\{1, 2, 3, 5, 6, 7\}$, il ne termine pas et retourne à chaque cycle testé :

```
EmptyCl (167: N(s(s(0))).)
Si (16: N(s(v0)) | -p(cons(a,Si(v0))).)
Sij (162: N(s(s(v0))) | -p(cons(a,cons(a,Si(v0)))).)
This clause has not a shift in S_loop~: 16: N(s(v0)) | -p(cons(a,Si(v0))).
```

Les clauses vides par rang (les clauses de la forme $[\square \mid n \simeq \text{succ}^k(0)]$ où $k \in \mathbb{N}$) sont toutes générées mais aucun cycle n'est détecté car les décalages des clauses correspondant aux clauses ancêtres des clauses vides ne sont jamais générées. Le problème vient du fait que lorsqu'on dérive une clause de rang supérieur, on est contraint de déplier le schéma de termes et on obtient une clause qui ne peut être ni égale à une clause (modulo un décalage) ni la subsumer. Ce comportement est étroitement lié avec la stratégie utilisée, et en particulier avec la façon dont les littéraux sont ordonnés : ici le littéral $p(x)$ est supérieur au littéral $q(x, y)$, ce qui implique que les clauses 1 et 4 ne peuvent interagir. La stratégie choisie par le système consiste à normaliser la clause $p(Si(x))$ avant d'appliquer la résolution avec les conséquences des autres axiomes, ce qui ne termine pas ici puisque la valeur du paramètre n'est pas fixe. En utilisant une stratégie différente, il serait possible d'engendrer la $n$-clause $[q(Si(x), y) \mid n \simeq x]$, d'où il est facile de déduire $[q(Si(x), a) \mid n \simeq \text{succ}(x)]$ avec la clause 3, ce qui engendre un cycle. ♣

# Quatrième partie

# Conclusion

# Chapitre 11

# Travaux accomplis et perspectives

## 11.1 Résumé de nos travaux

### 11.1.1 Partie théorique

Nous avons défini un formalisme, appelé les $n$-clauses, pour représenter des formules du premier ordre contenant des constantes de type entier. Une extension du calcul de superposition aux $n$-clauses a été proposée. Ce calcul est incomplet en général car, comme nous l'avons montré, la classe des $n$-clauses n'est pas semi-décidable. Nous avons introduit une nouvelle règle d'inférence permettant de prouver certaines propriétés des $n$-clauses en faisant appel à l'induction mathématique. A la différence des règles d'inférence usuelles, cette règle (appelée règle de détection de boucle) s'applique en prenant en compte l'ensemble des clauses engendrées par le démonstrateur. Elle permet d'engendrer des lemmes inductifs de manière automatique et s'appuie sur une analyse de l'espace de recherche visant à mettre en évidence des cycles dans le graphe d'inférence.

Nous avons proposé deux algorithmes de recherche de boucles : un algorithme de recherche d'un plus petit point fixe appelé $\textsc{Cycle}_1$ et un algorithme de recherche d'un plus grand point fixe appelé $\textsc{Cycle}_2$. Nous avons prouvé la correction et la complétude de ces deux algorithmes (qui n'implique cependant pas la complétude réfutationnelle du calcul – nous prouvons que tous les cycles peuvent être engendrés, ce qui n'implique pas que toute formule admet éventuellement un cycle). Les deux algorithmes calculent les mêmes ensembles de cycles, mais ils ne sont pas équivalents : les lemmes inductifs engendrés sont différents en général. L'algorithme $\textsc{Cycle}_1$ est le plus naturel et le plus simple à mettre en oeuvre, mais il est exponentiel dans certains cas, si la subsomption est utilisée pour comparer les ensembles de clauses lors de la recherche de cycles. L'algorithme $\textsc{Cycle}_2$ est polynomial, mais il s'avère moins efficace en pratique car les ensembles de clauses

considérés et les lemmes engendrés sont de taille plus importante.

Nous avons ensuite considéré le cas où les constantes ne sont plus de type entier, mais de type mot. Nous introduisons ainsi le formalisme des $\alpha$-clauses, généralisant les $n$-clauses, et étendons certains des résultats théoriques précédents. La définition de la notion de cycle s'avère toutefois beaucoup plus ardue. La structure de l'espace de recherche qui était linéaire dans le cas des termes entiers devient arborescente dans le cas des mots. Ainsi, la notion de décalage devient une opération d'instanciation dépendant d'un terme. En raison de la structure arborescente de l'espace de recherche, la règle de détection de boucle est beaucoup plus complexe et difficile à appliquer dans le cas des $\alpha$-clauses.

Nous avons ensuite défini des conditions sémantiques assurant la complétude (classe dite *admissible*). En effet, après avoir hiérarchisé l'espace de recherche en introduisant une notion de *rang*, nous proposons certaines conditions permettant d'assurer qu'un cycle finira systématiquement par être détecté entre deux rangs. Ces conditions permettent, d'une part, de maîtriser la profondeur des termes de type inductif pour qu'elle n'augmente pas arbitrairement, et d'autre part de restreindre la détection de boucles à un sous-ensemble *fini* de clauses (noté $\mathfrak{F}$). Si, de plus, le calcul de superposition termine pour toute valeur du paramètre, le problème de la satisfiabilité est alors décidable pour le fragment considéré des $\alpha$-clauses.

En instanciant ces derniers résultats, nous avons identifié deux classes syntaxiques pour lesquelles la procédure de preuve que nous avons définie est complète : la classe des clauses $\mu$-contrôlées et la classe des clauses quasi-fermées. Les clauses $\mu$-contrôlées (inspirées de [56]) sont définies modulo une certaine mesure de complexité $\mu$ qui peut être choisie arbitrairement (par exemple la profondeur ou la taille des clauses). En utilisant les résultats de [68], cette fonction de complexité nous permet de définir des conditions syntaxiques pour que la classe soit admissible. La classe des clauses quasi-fermées est un fragment des $n$-clauses où tous les termes sont constants à l'exception des termes de type inductifs. Le problème de la satisfaisabilité est uniquement semi-décidable pour cette classe (i.e., il existe un algorithme permettant d'énumérer tous les modèles). Nous avons proposé une restriction de la classe des clauses quasi-fermées, les clauses $\mathcal{S}_I$*-fermées*. Ces clauses sont définies comme une restriction des clauses quasi-fermées au cas où toutes les fonctions dépendent du paramètre inductif. Cette classe n'est pas stable par superposition, aussi avons-nous défini un nouveau calcul permettant de tester la satisfiabilité des ensembles de clauses $\mathcal{S}_I$-fermées : la superposition à la racine, qui combine l'application de la transitivité de l'égalité avec des règles de décomposition des termes à la racine. En utilisant les résultats précédents (qui se généralisent à n'importe quel calcul correct et complet) nous avons établi que cette classe est admissible, et donc décidable.

### 11.1.2 Partie pratique

Nous avons implémenté ces différents résultats théoriques dans un prototype de recherche nommé SuperInd, basé sur le démonstrateur Prover9. Ce système est utilisé comme moteur d'inférence, nos ajouts consistant en l'adaptation des règles d'inférence et de la stratégie, ainsi que dans l'implémentation des algorithmes de détection de cycles. Toutes les règles d'inférence de Prover9 ont été adaptées pour s'appliquer sur les $n$-clauses et la plupart des fonctionnalités du système ont été conservées. Les deux algorithmes de détection de boucles $\textsc{Cycle}_1$ et $\textsc{Cycle}_2$ sont implémentés dans SuperInd, et deux stratégies d'appel à ces algorithmes sont proposées : une stratégie aléatoire et une stratégie basée sur une recherche exhaustive de cycles. Nous avons mené différentes expérimentations pour tester notre outil, qui s'avère assez efficace pour réfuter différents théorèmes du premier ordre provenant de certains outils existants : Regstab [5], SPASS-FD [47] et CERES [38]. En particulier, afin d'utiliser SuperInd pour tester la satisfaisabilité de schémas de formules propositionnelles nous avons implémenté un traducteur des entrées de RegStab vers les $n$-clauses, utilisant l'algorithme décrit au chapitre 4.

## 11.2 Critiques et ouvertures

Le formalisme des $\alpha$-clauses nous parait adéquat pour représenter les formules dont certaines constantes sont interprétées dans un domaine inductif. En effet, nous avons besoin d'abstraire ces constantes particulières en utilisant des variables pour pouvoir appliquer les différentes règles sans avoir à instancier a priori les paramètres. Ce type de formalisme est déjà très utilisé dans la littérature [15, 1, 17, 41, 49] pour combiner la superposition avec d'autres procédures de preuve et il nous a semblé pertinent dans notre cas.

Nous avons imposé plusieurs restrictions tout le long de notre travail car d'une part nous voulions obtenir une procédure susceptible d'être implémentée suffisamment efficacement pour être utilisable en pratique, et d'autre part nous voulions pouvoir identifier des classes syntaxiques pour lesquelles nous pouvons garantir la semi-décidabilité ou la terminaison. Une autre démarche serait par exemple de se placer dans le cadre le plus général possible (en autorisant des domaines inductifs définis sur un ensemble de constructeurs quelconques, voir un domaine arbitraire muni d'un ordre noethérien), et de définir la détection de boucles de manière théorique quitte à se retrouver confronté à l'impossibilité de son application en pratique.

Pour la partie pratique, nous voulions à la base implémenter la procédure de preuve pour les $n$-clauses en utilisant un code générique pour pouvoir étendre aux $\alpha$-clauses. Il s'est avéré que la tâche était plus délicate que prévu et n'a pu être

menée à bien dans le temps imparti.

Nous présentons ci-dessous plusieurs pistes qui nous paraissent intéressantes pour des travaux futurs.

## Traitement des $\alpha$-clauses avec plusieurs $\mathcal{S}_I$-variables

Nous avons vu que dans le cas des $\alpha$-clauses les $\mathcal{S}_I$-termes sont des fonctions monadiques. Une évolution possible de ce travail pourrait être d'étendre aux termes non monadiques. Nous avions envisagé ce cas, mais la détection de boucles s'était avérée plus complexe car un cycle pouvait dépendre de plusieurs positions. Si nous considérons l'exemple très élémentaire suivant :

**Exemple 122**

$$\begin{array}{ll} 1 & [p(x) \not\simeq \mathtt{V} \mid \alpha \simeq x] \\ 2 & [p(x) \not\simeq \mathtt{V} \vee p(y) \not\simeq \mathtt{V} \vee p(f(x,y)) \simeq \mathtt{V} \mid \mathtt{V}] \\ 3 & [p(a) \mid \mathtt{V}] \end{array}$$

Nous appliquons la superposition et nous obtenons :

$$\begin{array}{lll} 4 & [p(x) \not\simeq \mathtt{V} \vee p(y) \not\simeq \mathtt{V} \mid \alpha \simeq f(x,y)] & \text{superposition } (1,2) \\ 5 & [\Box \mid \alpha \simeq a] & \text{superposition } (1,3) \end{array}$$

Nous remarquons ici que la clause 4 ne correspond pas à un simple décalage de la clause 1 : le littéral $p(x)$ est dupliqué car l'induction doit être réalisée simultanément sur les deux positions 1 et 2 (autrement dit, l'hypothèse d'induction doit être appliquée sur chacun des arguments de $f$). ♣

Il serait donc intéressant d'adapter la détection de boucles aux fonctions non monadiques en étendant ses conditions d'applicabilité, notamment en modifiant par exemple la fonction décalage afin d'autoriser la copie (duplication) de littéraux, ou bien d'appliquer une forme particulière de factorisation avant le décalage.

## Formules avec des paramètres multiples

Nous nous sommes restreint dans cette thèse au cas où les formules contiennent un seul paramètre. Un futur travail consisterait à étendre les $\alpha$-clauses aux formules avec plusieurs paramètres, permettant ainsi de traiter des problèmes faisant appel soit à plusieurs applications du principe d'induction sur plusieurs variables différentes, soit à des inductions imbriquées. Le premier cas nous semble relativement aisé : il suffit de généraliser le formalisme en autorisant des contraintes de la forme $\vec{\alpha} = \vec{t}$, et d'étendre la règle de détection de boucles en supposant que le décalage ne s'applique que sur une seule composante de $\vec{t}$. Le second cas est en revanche

beaucoup plus complexe. Nous avions envisagé un cas particulier où les formules contiennent deux paramètres entiers. L'application en mathématique est intéressante notamment en analyse combinatoire où les deux paramètres représenteraient par exemple un coefficient binomial qui est souvent utilisé en dénombrement et en probabilité.

## Integration des schémas de termes

Nous pouvons réfléchir au moyen d'intégrer les schémas de termes (au sens de [32, 33, 46]) dans les procédures de preuve que nous avons définis. Ces formalismes permettent en effet de représenter formellement des schémas de termes de manière à la fois concise et naturelle. Une première approche consisterait à coder la définition des schémas de termes par des axiomes du premier ordre comme dans l'exemple 121 du chapitre 10 et adapter la détection de boucles en modifiant l'opération de décalage. Une approche alternative (mais plus coûteuse) pourrait consister à garder le même formalisme que [20], et implémenter la règle de H-superposition (une extension de la règle de superposition permettant de garantir la complétude dans le cas où les clauses contiennent des schémas de termes) dans SuperInd.

## Classes décidables

Il serait évidemment intéressant d'identifier d'autres classes décidables plus expressives que la classe des clauses $\mu$-contrôlées. Nous pouvons aussi tenter d'étendre la classe des clauses quasi-fermées en autorisant par exemple des variables, tout en préservant la décidabilité.

## Evolution de SuperInd

Nous avons proposé deux stratégies d'application de la détection de boucles dans SuperInd : la première est basée sur une recherche exhaustive de cycles et la deuxième est probabiliste et permet de ne pas tester tous les cycles. La méthode probabiliste s'avère souvent plus efficace mais explose pour certains exemples. Une évolution possible consiste à définir des stratégies plus élaborées pour réduire le nombre d'applications de la détection de boucles. Nous avons, par exemple, pensé à construire une table de hachage qui va sauvegarder les entiers qui ne peuvent former un cycle avec les clauses manquantes. Cela aurait deux avantages : d'une part identifier lors de la génération de nouvelles clauses celles qui seraient susceptibles de compléter un cycle, et d'autre part d'effectuer les calculs de manière incrémentale, au lieu de repartir de zéro à chaque appel de la procédure de détection de boucles.

Nous avons vu aussi que SuperInd est restreint aux $n$-clauses, une évolution possible est donc l'extension de SuperInd aux $\alpha$-clauses, ce qui n'est pas trivial pour plusieurs raisons. Nous présentons les évolutions principales à prévoir et les difficultés à étendre le code existant :

1. Comme dans le cas des $n$-clauses il faudra définir un algorithme de recherches de cycles, ce qui passe par l'adaptation de l'algorithme de recherche d'un point fixe. Pour cela nous devons, dans un premier temps, revoir la fonction qui calcule le décalage, qui dépend de plus de paramètres dans le cas des $\alpha$-clauses (car comme nous avons vu dans le chapitre 5, le décalage pour une $\alpha$-clause dépend d'une substitution et d'un terme). Le choix de ces paramètres augmente d'autant l'espace de recherche, il apparaît donc nécessaire de disposer de bonnes heuristiques permettant de suggérer le terme et la substitution à considérer (dans le cas des entiers il reste possible de tester successivement chacun des couples d'entier, mais cette solution n'est pas envisageable avec un espace de recherche exponentiellement plus important).
2. Ajouter une fonction qui calcule l'ensemble couvrant sur lequel on applique la détection de boucles. Cet ensemble est facile à construire dans le cas entier car tous les ensembles couvrants contiennent un sous-ensemble dans $\{0, \text{succ}(0), \text{succ}^n(0), \text{succ}^{n+1}(x)\}$.
3. Adapter la fonction qui calcule le rang qui doit retourner un terme dans le cas des $\alpha$-clauses.
4. Une fois l'algorithme de détection de boucles implémenté, il faudra définir une stratégie d'application de cet algorithme. Contrairement à la détection de boucles pour les $n$-clauses où le fait de trouver un cycle permettait d'établir l'insatisfaisabilité, le cas des $\alpha$-clauses est plus complexe car la détection de boucles peut être appliquée plusieurs fois afin de fermer des branches infinies distinctes de l'espace de recherche (dans le cas des entiers il suffit d'ajouter une seule clause de la forme $n < i + j$ pour éliminer toutes les branches infinies).

## Implémentation de la superposition à la racine

Une autre évolution possible de ce travail serait d'implémenter le calcul de superposition à la racine, en ajoutant une nouvelle règle d'inférence dans Prover9. Nous pourrions ainsi utiliser SuperInd sans avoir à modifier son code, car la détection de boucles est indépendante du calcul utilisé.

# Index

# Table des figures

TABLE DES FIGURES

# Bibliographie


[1] E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic sup(la). In S. Ghilardi and R. Sebastiani, editors, *FroCoS 2009*, volume 5749 of *LNCS*, pages 84–99. Springer, 2009.

[2] V. Aravantinos. *Schémas de formules et de preuves en logique propositionnelle*. PhD thesis, Institut National Polytechnique de Grenoble, 2010.

[3] V. Aravantinos, R. Caferra, and N. Peltier. A Decidable Class of Nested Iterated Schemata. In *IJCAR 2010 (International Joint Conference on Automated Reasoning)*, LNCS, pages 293–308. Springer, 2010.

[4] V. Aravantinos, R. Caferra, and N. Peltier. Complexity of the satisfiability problem for a class of propositional schemata. In *LATA 2010 (Language, Automata Theory and Applications)*, LNCS, pages 58–69. Springer, 2010.

[5] V. Aravantinos, R. Caferra, and N. Peltier. RegSTAB : a SAT-Solver for Propositional Schemata. In *IJCAR 2010 (International Joint Conference on Automated Reasoning)*, LNCS, pages 309–315. Springer, 2010.

[6] V. Aravantinos, R. Caferra, and N. Peltier. Decidability and undecidability results for propositional schemata. *Journal of Artificial Intelligence Research*, 40 :599–656, 2011.

[7] V. Aravantinos, R. Caferra, and N. Peltier. Linear Temporal Logic and Propositional Schemata, Back and Forth. In *TIME'2011 (18th International Symposium on Temporal Representation and Reasoning)*, 2011.

[8] V. Aravantinos, M. Echenim, and N. Peltier. A resolution calculus for first-order schemata. *Fundamenta Informaticae*, 2013. Accepted for publication, to appear.

[9] V. Aravantinos and N. Peltier. Schemata of SMT problems. In *TABLEAUX 11 (International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, LNCS. Springer, 2011.

[10] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2) :140–164, 2003.

[11] M. Baaz, S. Hetzl, A. Leitsch, C. Richter, and H. Spohr. CERES : An analysis of Fürstenberg's proof of the infinity of primes. *Theor. Comput. Sci.*, 403(2-3) :160–175, 2008.

[12] M. Baaz and A. Leitsch. Cut-elimination and Redundancy-elimination by Resolution. *Journal of Symbolic Computation*, 29(2) :149–176, 2000.

[13] M. Baaz and A. Leitsch. Towards a clausal analysis of cut-elimination. *J. Symb. Comput.*, 41(3-4) :381–410, 2006.

[14] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4) :217–247, 1994.

[15] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierachic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5 :193–212, 1994.

[16] D. Baelde, D. Miller, and Z. Snow. Focused inductive theorem proving. In *IJCAR*, pages 278–292, 2010.

[17] P. Baumgartner and C. Tinelli. Model evolution with equality modulo built-in theories. In Bjørner and Sofronie-Stokkermans [23], pages 85–100.

[18] P. Baumgartner and U. Waldmann. Hierarchic superposition with weak abstraction. In Bonacina [24], pages 39–57.

[19] E. Ben-Sasson and N. Galesi. Space complexity of random formulae in resolution. *Colloquium on Computational Complexity*, 2001.

[20] H. Bensaid. *Utilisation des schématisations de termes de déduction automatique*. PhD thesis, Institut National Polytechnique de Grenoble, 2011.

[21] H. Bensaid, R. Caferra, and N. Peltier. Towards systematic analysis of theorem provers search spaces : First steps. In *Proc. Wollic'07 (Workshop on Logic, Language, Information and Computation)*, pages 38–52. Springer, July 2007. LNCS 4576.

[22] H. Bensaid, R. Caferra, and N. Peltier. I-terms in ordered resolution and superposition calculi : retrieving lost completeness. In *AISC 2010 (10th International Conference on Artificial Intelligence and Symbolic Computation)*, LNCS, pages 19–33. Springer, 2010.

[23] N. Bjørner and V. Sofronie-Stokkermans, editors. *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*. Springer, 2011.

[24] M. P. Bonacina, editor. *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14,*

*2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*. Springer, 2013.

[25] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Spike, an automatic theorem prover. In *LPAR '92 : Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 460–462, London, UK, 1992. Springer-Verlag.

[26] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *J. Log. Comput.*, 5(5) :631–668, 1995.

[27] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *J. Autom. Reasoning*, 14(2) :189–235, 1995.

[28] R. S. Boyer and J. S. Moore. *A computational logic*. Academic Press, 1979.

[29] A. Bundy. The automation of proof by mathematical induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 845–911. Elsevier and MIT Press, 2001.

[30] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling : meta-level guidance for mathematical reasoning*. Cambridge University Press, New York, NY, USA, 2005.

[31] A. Bundy, F. v. Harmelen, C. Horn, and A. Smaill. The oyster-clam system. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK, UK, 1990. Springer-Verlag.

[32] H. Chen and J. Hsiang. Logic programming with recurrence domains. In *Automata, Languages and Programming (ICALP'91)*, pages 20–34. Springer, LNCS 510, 1991.

[33] H. Comon. On unification of terms with integer exponents. *Mathematical System Theory*, 28 :67–88, 1995.

[34] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 14, pages 913–962. North-Holland, 2001.

[35] H. Comon and R. Nieuwenhuis. Induction = i-axiomatixation+first-order consistency. Technical Report LSV-98-9, ENS Cachan, october 1998.

[36] J. Corcoran. Schemata : the concept of schema in the history of logic. *The Bulletin of Symbolic Logic*, 12(2) :219–240, June 2006.

[37] T. Dunchev. *Automation of cut-elimination in proof schemata*. PhD thesis, T.U. Vienna, 2012.

[38] T. Dunchev, A. Leitsch, M. Rukhaia, and D. Weller. Ceres for first-order schemata, 2013. Research Report, `http://arxiv.org/abs/1303.4257`.

[39] S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2012.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[41] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[42] G. Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39 :176–210, 1934.

[43] J. Giesl and D. Kapur. Decidable classes of inductive theorems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 469–484. Springer, 2001.

[44] J. Giesl and D. Kapur. Deciding inductive validity of equations. In F. Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2003.

[45] R. Hähnle. Tableaux and related methods. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 100–178. Elsevier Science, 2001.

[46] M. Hermann and R. Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theor. Comput. Sci.*, 176(1-2) :111–158, 1997.

[47] M. Horbach. System description : Spass-fd. In Bjørner and Sofronie-Stokkermans [23], pages 331–337.

[48] M. Horbach and C. Weidenbach. Deciding the inductive validity of for all there exists $^*$ queries. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *LNCS*, pages 332–347. Springer, 2009.

[49] M. Horbach and C. Weidenbach. Superposition for fixed domains. *ACM Trans. Comput. Logic*, 11(4) :1–35, 2010.

[50] D. Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *J. Autom. Reasoning*, 16(1-2) :39–78, 1996.

[51] D. Kapur and M. Subramaniam. Extending decision procedures with induction schemes. In D. A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 324–345. Springer, 2000.

[52] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (rrl). *J. of Computer and Mathematics with Applications*, 29 :91–114, 1995.

[53] A. Kersani and N. Peltier. Combining superposition and induction : A practical realization. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *FroCos*, volume 8152 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2013.

[54] A. Kersani and N. Peltier. Completeness and decidability results for first-order clauses with indices. In Bonacina [24], pages 58–75.

[55] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In Shrobe et al. [80], pages 240–245.

[56] A. Leitsch. Deciding clause classes by semantic clash resolution. *Fundamenta Informaticae*, 18 :163–182, 1993.

[57] A. Leitsch. *The resolution calculus*. Springer. Texts in Theoretical Computer Science, 1997.

[58] P. M. M. Kaufmann and J. S. Moore. *Computer-Aided Reasoning : An Approach*. Kluwer, 2000.

[59] W. McCune. Otter and mace4. `http://www.cs.unm.edu/~mccune/otter/`, 2003.

[60] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[61] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll(). *J. ACM*, 53(6) :937–977, 2006.

[62] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[63] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[64] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Form. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.

[65] C. S. Peirce. *Philosophical Writings of Peirce*. Dover Books, Justus Buchler editor, 1955.

[66] N. Peltier. A General Method for Using Terms Schematizations in Automated Deduction. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 578–593. Springer LNCS 2083, 2001.

[67] N. Peltier. The First Order Theory of Primal Grammars is Decidable. *Theoretical Computer Science*, 323 :267–320, 2004.

[68] N. Peltier. Some Techniques for Proving Termination of the Hyperresolution Calculus. *Journal of Automated Reasoning*, 35 :391–427, 2006.

[69] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2 :293–304, 1986.

[70] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[71] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *CADE*, volume 449 of *LNCS*, pages 162–177. Springer, 1990.

[72] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 376–380. Springer LNCS 2083, 2001.

[73] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.

[74] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie and R. Meltzer, editors, *Machine Intelligence*, volume 4, pages 135–150. Edinburg U. Press, 1969.

[75] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12 :23–41, 1965.

[76] M. Rukhaia. *CERES in Proof Schemata*. PhD thesis, T.U. Vienna, 2012.

[77] M. Rusinowitch. Démonstration automatique par des techniques de réécriture. Thèse d'état 1987, Nancy 1- France. Also available as textbook (Inter Editions, Paris 1989), 1987.

[78] G. Salzer. The unification of infinite sets of terms and its applications. In A. Voronkov, editor, *LPAR*, volume 624 of *LNCS*, pages 409–420. Springer, 1992.

[79] S. Schulz. System Description : E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[80] H. E. Shrobe, T. G. Dietterich, and W. R. Swartout, editors. *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, July 29 - August 3, 1990, 2 Volumes*. AAAI Press / The MIT Press, 1990.

[81] W. Sieg. Hilbert's programs : 1917-1922. *Bulletin of Symbolic Logic*, 5(1) :1–44, 1999.

[82] S. Stratulat. Automatic 'descente infinie' induction reasoning. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2005)*, volume 3702 of *LNCS*, pages 262–276. Springer, 2005.

[83] S. Stratulat. A unified view of induction reasoning for first-order logic. In A. Voronkov, editor, *Turing-100*, volume 10 of *EPiC Series*, pages 326–352. EasyChair, 2012.

[84] The Coq Development Team. *The Coq Proof Assistant Reference Manual V7.1*. INRIA-Rocquencourt, CNRS-ENS Lyon (France), October 2001. `http://coq.inria.fr/doc/main.html`.

[85] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. L. Lusk and R. A. Overbeek, editors, *CADE*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.