Résolution de contraintes symboliques

A. Boudet et H. Comon

26 mars 1996

Table des matières

Chapitre 1

Introduction

Les systèmes de contraintes permettent de définir des ensembles de données à l'aide de formules logiques. Par exemple, l'ensemble des entiers pairs est représenté par la contrainte $\exists x.y = 2 * x$, interprétée dans les entiers naturels.

En fait, une contrainte est exactement ce qu'on utilise pour définir un ensemble en compréhension comme dans l' exemple

$$\{y \in \mathbf{N} \mid \exists x.y = 2 * x\}$$

Ainsi, les contraintes sont des formules dont le domaine d'interprétation est fixé (les entiers naturels dans l'exemple). Leur utilisation dans un langage de programmation (logique) permet de manipuler des données éventuellement infinies, ce que ne permttrait pas les seules définitions en extension (types énumérés). Les systèmes de contraintes augmentent donc le pouvoir d'expression des langages de programmation. (NB: la plupart des langages de programmation sont Turing-complets, par "augmenter le pouvoir d'expression" il faut comprendre "la facilité d'expression"; mais ceci est aussi expliqué dans le paragraphe qui suit).

Il existe d'autres moyens de définir des ensembles infinis de données. Par exemple, on peut dire que les entiers pairs sont le plus petit ensemble contenant 0 et clos par addition de 2 à l'un de ses éléments. Mais supposons que les entiers impairs soient, de façon analogue, définis comme le plus petit ensemble contenant 1 et clos par addition de 2. Comment faire pour calculer les entiers à la fois pairs et impairs? Avec la définition procédurale que nous venons de donner, il faudrait énumérer les deux ensembles jusqu'à trouver un élément commun, ce qui ne termine pas dans cet exemple.

Ainsi, les contraintes possèdent un autre avantage: elles décrivent de manière *déclarative* (par opposition à *procédurale*) les données.

Résoudre des contraintes peut prendre un sens différent suivant les applications. Et, de fait, on trouve des approches très différentes en démonstration automatique et en IA. Le problème est d'abord de déterminer si un ensemble défini en compréhension comme ci-dessus est vide ou non (ce problème est appelé *problème de cohérence globale*). Mais dans les articles d'IA, on trouve souvent étudié un problème moins ambitieux mais plus réaliste: celui de la *cohérence locale*. En effet, pour des contraintes de la forme $\phi_1 \wedge \ldots \wedge \phi_n$, où n est grand, l'ensemble $\{d \in \mathcal{D} \mid \phi_1 \wedge \ldots \wedge \phi_n\}$ est une intersection $\{d \in \mathcal{D} \mid \phi_1\} \cap \ldots \cap \{d \in \mathcal{D} \mid \phi_n\}$. La cohérence locale exprime que ces ensembles sont non-disjoints deux à deux (ce qui est nécéssaire pour la cohérence globale). Evidemment, cette approximation du problème de cohérence globale peut être effectuée de manière plus efficace que le cas général.

1.1 Contenu du cours

Nous nous intéressons beaucoup ici à des systèmes de contraintes dites *symboliques*, c'est à dire dont le domaine d'interprétation est défini de façon syntaxique, comme un quotient de l'algèbre des termes (finis ou infinis). Nous considérerons aussi presque exclusivement des problèmes de cohérence globale.

Il y a peu de concepts généraux pour les problèmes de résolution de contraintes. Dans un premier temps nous donnons précisément les définitions de base sur les contraintes ou leur combinaison avec d'autres langages, ainsi que plusieurs exemples d'utilisation (chapitre 2).

La suite du cours, plutôt que d'être organisée en un catalogue de système de contraintes symboliques connus, s'articule suivant trois axes principaux correspondant à trois techniques de résolution de contraintes:

1. les *méthodes syntaxiques* sont étudiées dans le chapitre 3. Elles sont très simples dans leur principe et ce sont les plus utilisées. L'idée est de transformer la contrainte (formule) en une formule équivalente, donc qui décrit le même ensemble, mais plus simple. Ce processus est répété jusqu'à obtenir une *forme résolue*. Par exemple, si l'on s'intéresse à la contrainte sur les nombres réels décrite par le système de deux équations à trois inconnues (qui exprime "j'ai deux fois l'âge que vous

aviez quand j'avais l'âge que vous avez"):

$$\begin{cases} x = 2(y - t) \\ x - t = y \end{cases}$$

Par élimination dite de Gauss, on obtient une forme résolue:

$$\begin{cases} x = 4t \\ y = 3t \end{cases}$$

les méthodes syntaxiques ne tirent donc parti que des formules et de propriétés du domaine, c'est à dire, pour être plus précis, d'une axiomatisation du domaine.

les exemples de systèmes de contraintes utilisant ces méthodes sont extrèmement nombreux. Nous les illustrerons sur quelques exemples en particulier:

- la résolution de problèmes de programmation linéaire (inéquations linéaires en nombres rationnels), par la méthode dite "du simplexe.
- l'*unification*, procédure bien connue en informatique, qui consiste à résoudre des conjonctions d'équations dans les termes ou dans certains de leurs quotients.
- Plus généralement les problèmes équationnels, qui ne se limitent plus à des conjonction d'équations entre termes, mais dans lesquels sont aussi autorisés de diséquations ou des quantifications. En fait la théorie du premier ordre des termes est complètement axiomatizable, comme nous le verrons.
- Les arbres à traîts sont utilisés pour la représentation des connaissances (en particulier en linguistique). A nouveau la théorie des arbres à traits est complètement axiomatisable.

Mais, même pour les systèmes de contraintes résolus d'une autre façon, les méthodes syntaxiques constituent souvent une étape préliminaire de simplification.

2. Les méthodes *sémantiques* sont étudiées dans le chapitre 4. Elles tirent parti d'une autre représentation des solution de la formule. Un exemple typique est celui des automates: depuis les travaux de Büchi dans les

années 60 on sait qu'il y a correspondance entre la logique monadique de second ordre et les automates d'états finis. Dans cet esprit on peut associer différents types d'automates à différents systèmes de contraintes. Par exemple considérons la contrainte $\exists x.y = 2 * x$. Si les entiers sont représentés par des entiers écrits en base 2, cet ensemble est celui des mots se terminant par 0, ensemble reconnu par un automate très simple à 2 états. Mais cette construction est plus générale: on peut associer à n'importe quelle formule bâtie sur les prédicats $=$ et $\geq$ et dont les membres sont des combinaisons linéaires à coefficients entiers de variables (ce sont les formules de *l'arithmétique de Presburger*) un automate d'états finis qui accepte les entiers (ou les $n$uples d'entiers) solution de la formule.

Résoudre la contrainte, dans ce contexte, signifie trouver un automate réduit correspondant à la formule.

Nous envisagerons d'une part plusieurs systèmes de contraintes associés à des automates (arithmétique de Presburger, contraintes d'entourage, contraintes d'ensemble,...) après en avoir montré l'esprit, mais aussi d'autres représentations des solutions. Par exempl, les *bases de solution* pour les systèmes d'équations diophantiennes linéaires, ou bien les techniques d'unification modulo l'associativité et la commutativité de symboles de fonction.

3. Dans la troisième partie, nous abordons les techniques de *combinaison*: comment, à partir de deux techniques de résolution de contraintes pour des systèmes S_1 et S_2 peut-on construire une technique de résolution de contrainte dans leur réunion ? Il existes quelques concepts généraux comme l'abstraction de variable et les choix non déterministes d'interactions. Nous illustrerons par deux exemples ces techniques: la combinaison des algorithmes d'unification et la résolution de contraintes d'ordre sur les termes. Ces dernières étant importante en démonstration automatique par des techniques de réécriture.

4. Dans la dernière partie nous abordons les méthodes incomplètes (i.e. qui ne prouvent que la cohérence locale). En fait, ces méthodes sont généralement rendues complètes par des mécanismes d'énumération. Un exemple typique est celui des domaines finis; les variables de la contrainte varient chacune dans un ensemble fini et il existe donc une méthode triviale de résolution de la contrainte: l'énumération de toutes les possibilités. Mais cette méthode naïve est souvent impossible à

mettre en œuvre en pratique à cause du trop grand nombre de cas à envisager. On utilise alors des méthodes syntaxiques (ou sémantiques lorsque les contraintes sont représentées sous forme de graphes de dépendance). Quand aucune règle simple n'est plus applicable, on commence une énumération avant de simplifier à nouveau. Par exemple, supposons donnée la contrainte suivante dans laquelle toutes les variables sont des entiers de l'intervalle 0..9, toutes différentes:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Chacun des mots SEND, MORE, MONEY représentant un entier écrit en base 10. Un erecherche exhaustive conduirait à tester 10^8 possibilités, ce qui est trop élevé. À l'inverse, le système diophantien résultant comporterait 11 inconnues (incluant les retenues), ce qui le rend aussi assez compliqué. Mais le plus simple est de tirer parti des bornes sur les inconnues et des équations pour réduire l'espace de recherche. Par exemple, $S + M + r_3 = 10 * M + O$ (avec $0 \leq r_3 \leq 1$) donc $M \in \{0, 1\}$ et $S \in \{8, 9\}$ et $O \in \{0, 1\}$.

Chaque méthode (et donc chaque chapitre à partir du chapitre 3) comportera d'abord une partie dans laquelle seront rappelés tous les outils nécessaires (par exemple, dans le chapitre 3, les outils de réécriture nécessaires), puis une partie de description de la méthode proprement dite et enfin un certain nombre d'exemples de mise en œuvre. Certain systèmes de contraintes seront étudiés dans plusieurs chapitres, par des méthodes différentes.

1.2 Applications

Les systèmes de contraintes permettent parfois d'exprimer à eux seul un problème intéressant dont on recherche une solution, ce qui est le cas pour beaucoup de problèmes de recherche opérationnelle. La résolution des contraintes est un outil adéquat. C'est ainsi que la *programmation par contraintes* basée exclusivement sur ces mécanismes a permis de résoudre des questions pratique comme des problèmes de gestion, d'ordonnancement, ... Mais les contraintes sont aussi beaucoup utilisées combinées avec un langage de programmation logique. C'est le cas par exemple des langages de programmation comme PROLOG III [27], CHIP [50] ou CLP [71, 72] et bien d'autres.

Les applications industrielles développées dans ces langages utilisent de façon essentielle la résolution de contraintes, les constructions du langage de programmation permettant quant à elles de structurer les recherches et d'introduire les fonctions et prédicats spécifiques au problème à traiter. Mais il faut aussi mentionner l'utilisation des systèmes de contraintes en démonstration automatique.

En programmation logique comme en démonstration automatque les contraintes symboliques permettent d'exprimer de manière concise des ensembles de formules [79]. Par exemple, la formule $\phi(x, x, x, x, x)$ contrainte par $x = t$, ce que l'on notera $\phi(x, x, x, x, x) \mid x = t$ représente $\phi(t, t, t, t, t)$. Cette représentation évite la duplication cinq fois du terme t qui peut-être grand: les contraintes permettent d'exprimer du partage. Cette utilisation des contraintes d'égalité est en fait exploitée (explicitement ou non) dans la plupart des démonstrateurs et des interprètes de langages de programmation en logique. Il est aussi possible d'exprimer des ensembles de formules grands ou infinis.

Enfin, les contraintes permettent d'exprimer des stratégies de déduction dans les formules elles-mêmes (au lieu de les exprimer au "méta-niveau"). Ceci permet de garder dans les formules une "histoire" des raisons pour lesquelles les règles antérieures ont été appliquées, permettant ainsi de réduire encore l'espace de recherche. De nombreux travaux sont consacrés à cette approche, en particulier dans le cas des *stratégies basiques* (cf. par exemple [99, 5]) et des *stratégies ordonnées* [100, 102, 124, 92].

Chapitre 2

Les contraintes symboliques: définitions de base

Ce chapitre assez court donne d'une part les définitions de base de *contraintes* et de *logique contrainte* ainsi qu'une idée succinte de leurs utilisations possibles. On pourra aussi se reporter à [30].

Par souci de complétude nous rappelons d'abord les notions classiques de logique du premier ordre (classiques).

2.1 Rappels de logique du premier ordre

Les définitions suivantes sont très standard et peuvent être trouvées dans les premières pages de n'importe quel livre de logique. Nous en profitons néanmoins pour introduire nos notations qui sont celles de [48].

2.1.1 Termes et arbres

$\mathbf{N}_+^*$ est l'ensemble de tous les mots finis construits sur l'alphabet des entiers naturels strictement positifs. La concaténation de deux mots est notée $\cdot$ et le mot vide est noté Λ. $\succeq_{pref}$ est l'ordre préfixe sur les mots: $w \succeq_{pref} w'$ s'il existe un mot w'' tel que $w = w' \cdot w''$. Λ est minimal pour $\succeq_{pref}$.

Un *ensemble de positions* P est un sous-ensemble de $\mathbf{N}_+^*$ qui est stable par préfixe (i.e. si $w \in P$ et $w \succeq_{pref} w'$, alors $w' \in P$) et qui vérifie: si $w \cdot i \cdot w' \in P$ et $0 < j < i$, alors $w \cdot j \cdot w' \in P$.

Un *alphabet de symboles de fonction* est un ensemble $\mathcal{F}$ muni d'une application a de $\mathcal{F}$ dans l'ensemble $\mathbf{N}$ des entier naturels. a est aussi appelée *arité*. Les symboles d'arité 0 sont aussi appelés *constantes*.

Un *arbre étiqueté* (par $\mathcal{F}$) t est une application d'un ensemble non vide de positions $\mathrm{Pos}(t)$ dans $\mathcal{F}$ qui est compatible avec l'arité: si $t(p) = f$ et $a(f) = n$, alors, pour tout $i \in N_+$, ($p \cdot i \in \mathrm{Pos}(t)$ ssi $1 \leq i \leq n$). Un arbre t est *fini* si $\mathrm{Pos}(t)$ est fini. Il est infini sinon. $T(\mathcal{F})$ est l'ensemble des arbres finis étiquetés par $\mathcal{F}$. $IT(\mathcal{F})$ est l'ensemble des arbres finis ou infinis étiquetés par $\mathcal{F}$. Pour $t \in T(\mathcal{F})$, la *taille* de t est le cardinal de $\mathrm{Pos}(t)$ (quelquefois notée $|t|$) et sa *profondeur* est la longueur maximale d'un élément de $\mathrm{Pos}(t)$.

Quelquefois, il convient de distinguer un ensemble particulier (infini) de symboles d'arité 0, qui sont appelés *variables*. $T(\mathcal{F}, \mathcal{X})$ (resp. $IT(\mathcal{F}, \mathcal{X})$ est alors l'ensemble $T(F \cup \mathcal{X})$ (resp $IT(\mathcal{F} \cup \mathcal{X})$), mais les symboles de $\mathcal{X}$ sont des constantes distinguées sous le nom de variables. Si $t \in T(\mathcal{F}, \mathcal{X})$, $\mathcal{V}ar(t)$ est défini par récurrence sur la structure de t: $\mathcal{V}ar(x) = \{x\}$ si $x \in \mathcal{X}$ et $\mathcal{V}ar(f(t_1, \ldots, t_n)) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$ (par convention les unions sur un ensemble vide sont toujours égales à l'ensemble vide et les intersections sur un ensemble vide sont égales à l'ensemble plein).

Etant donné un arbre t et une position $p \in \mathrm{Pos}(t)$, le *sous-arbre* de t à la position p, noté $t|_p$ est défini par: $\mathrm{Pos}(t|_p) = \{q \in \mathbf{N}_+^* \mid p \cdot q \in \mathrm{Pos}(t)\}$ et $t|_p(q) = t(p \cdot q)$.

L'orsqu'un arbre ne possède qu'un nombre fini de sous-arbres distincts, il est dit *rationel*. Le sous-ensemble de $IT(\mathcal{F})$ des arbres rationnels est noté $RT(\mathcal{F})$.

Etant donnés deux arbres t, u et une position $p \in \mathrm{Pos}(t)$, l'arbre $t[u]_p$ obtenu en *remplaçant $t|_p$ par u à pa position p* est défini par: $\mathrm{Pos}(t[u]_p) = (\mathrm{Pos}(t) \setminus \mathrm{Pos}(t|_p)) \cup \{p \cdot q \mid q \in \mathrm{Pos}(u)\}$ et $t[u]_p(p \cdot q) = u(q)$ et $t[u]_p(q) = t(q)$ si p n'est pas un préfixe de q.

On représente aussi classiquement les arbres finis par des *termes* en utilisant un parenthésage pour la liste des sous-arbres immédiats. De façon équivalente on définit donc $T(\mathcal{F})$ comme le plus petit ensemble tel que: si $t_1, \ldots, t_n \in T(\mathcal{F})$, $f \in \mathcal{F}$ et $a(f) = n$, alors $f(t_1, \ldots, t_n) \in T(\mathcal{F})$.

Nous emploierons indifféremment les mots "terme" et "arbre fini" dans la suite. L'égalité des arbres sera écrite $\equiv$ (pour distinguer entre la syntaxe et la sémantique dans le cas où les arbres sont utilisés dans à la fois dans les formules et dans leur interprétation).

2.1.2 $\mathcal{F}$-algèbres

Soit $\mathcal{F}$ un ensemble de symboles de fonction. Une *$\mathcal{F}$-algèbre* $\mathcal{A}$ est un ensemble A muni, pour chaque symbole $f \in \mathcal{F}$ d'arité $a(f)$, d'une application

$f_{\mathcal{A}}$ de $A^{a(f)}$ dans A.

$T(\mathcal{F}), RT(\mathcal{F}), IT(\mathcal{F})$ sont des $\mathcal{F}$-algèbres de façon immédiate.

Un *morphisme* d'une $\mathcal{F}$-algèbre $\mathcal{A}$ dans une $\mathcal{F}$-algèbre $\mathcal{B}$ est une application h de A dans B qui est compatible avec la structure: pour tout $f \in \mathcal{F}$ et tous $\alpha_1, \ldots, \alpha_{a(f)} \in A$,

$$h(f_{\mathcal{A}}(\alpha_1, \ldots, \alpha_{a(f)})) = f_{\mathcal{B}}(h(\alpha_1), \ldots h(\alpha_{a(f)}))$$

2.1.3 Substitutions et assignations

Soient $t \in T(\mathcal{F}, \mathcal{X})$ et $\mathcal{A}$ une $\mathcal{F}$-algèbre. Une *assignation* σ dans $\mathcal{A}$ est une application de $\mathcal{X}$ dans $\mathcal{A}$. Une telle application peut-être étendue de manière unique en un morphisme de $T(\mathcal{F}, \mathcal{X})$ dans $\mathcal{A}$, encore noté σ.

Lorsque $\mathcal{A} = T(\mathcal{F}, \mathcal{X})$ une assignation dans $\mathcal{A}$ est aussi appelée *substitution.* Lorsque $\mathcal{A} = T(\mathcal{F})$, σ est appelée *substitution close.* Si $x_1, \ldots, x_n \in \mathcal{X}$ sont des variables distinctes et $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$, $\{x_1 \mapsto t_1; \ \ldots \ ; x_n \mapsto t_n\}$ désigne la substitution σ telle que $\sigma(x_i) \equiv t_i$ pour $1 \leq i \leq n$ et $\sigma(x) = x$ si $x \notin \{x_1, \ldots, x_n\}$. (Par abus, cette notation est aussi utilisée pour les substitution closes lorsque l'assignation des variables autres que $x_1, \ldots, x_n$ n'a pas d'importance).

Nous utiliserons la notation post-fixée pour l'application des assignations et des substitutions.

2.1.4 Théorie équationnelle

Une *équation* est une paire de termes (non orientée) de $T(\mathcal{F}, \mathcal{X})$. L'équation (t_1, t_2) est indifféremment notée $t_1 = t_2$ ou $t_2 = t_1$.

Une $\mathcal{F}$-algèbre $\mathcal{A}$ *satisfait* E (ce que l'on écrit $\mathcal{A} \models E$) si, pour toute équation $t_1 = t_2 \in E$ et toute assignation σ dans $\mathcal{A}$, $t_1\sigma = t_2\sigma$.

Si E est un ensemble d'équations, $=_E$ est la plus petite congruence sur $T(\mathcal{F}, \mathcal{X})$ telle que pour toute substitution σ et toute équation $t_1 = t_2 \in E$, $t_1\sigma =_E t_2\sigma$. Le quotient $T(\mathcal{F}, \mathcal{X})/_{=_E}$ peut alors être muni d'une structure de $\mathcal{F}$-algèbre de façon canonique. Ces définitions valent aussi en remplaçant $T(\mathcal{F}, \mathcal{X})$ par $T(\mathcal{F})$ ou $RT(\mathcal{F}, \mathcal{X})$, ou $RT(\mathcal{F})$, ... Noter que, pour toute $\mathcal{F}$ algèbre qui satisfait E, il existe un unique morphisme h de $T(F\mathcal{F})/_{=_E}$ dans $\mathcal{A}$.

2.1.5 Formules du premier ordre: syntaxe

Un *alphabet de symboles de prédicats* $\mathcal{P}$ est un ensemble, muni d'une fonction d'arité, comme pour les symboles de fonction. Dans la suite, lorsque $\mathcal{F}, \mathcal{X}, \mathcal{P}$ apparaissent simultanément dans le même contexte, ils sont toujours supposés disjoints.

Une *formule atomique* est un terme $p(t_1, \ldots, t_n)$ où p est un symbole de prédicat tel que $a(p) = n$ et $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$. Dans le cas de prédicats binaires, on utilisera aussi la notation infixée $t_1 \, p \, t_2$ (au lieu de $p(t_1, t_2)$). Enfin, lorsque l'égalité (symbole de prédicat binaire distingué) est présente dans $\mathcal{P}$, on ne fera aucune distinction entre $t_1 = t_2$ et $t_2 = t_1$.

L'*algèbre des formules du premier ordre* sur $\mathcal{F}$ et $\mathcal{P}$ est formée sur l'alphabet de symboles de fonction $\mathcal{H}$ contenant

- toutes les formules atomiques sur $\mathcal{F}$ et $\mathcal{P}$, qui sont considérées comme des constantes
- les constantes $\top$ et $\bot$
- les connecteurs $\neg$ (arité 1), $\wedge, \vee$ (arité 2)
- pour toute variable x, les connecteurs $\exists x$ et $\forall x$ (d'arité 1).

Cette algèbre est en outre un quotient par une relation de congruence engendrée par les axiomes d'associativité, commutativité et idempotence de $\wedge$ et $\vee$, élément absorbant de $\top$ pour $\vee$, $\exists x$, $\forall x$ et de $\bot$ pour $\wedge$, $\exists x$, $\forall x$ ainsi que les axiomes d'élément neutre de $\top$ pour $\wedge$ et de $\bot$ pour $\vee$, et enfin les relations $\neg\top = \bot$ et $\neg \bot = \top$. Donnons par exemple les axiomes pour $\wedge$:

$$\begin{array}{rcll}
(x \wedge y) \wedge z & = & x \wedge (y \wedge z) & \text{associativité de } \wedge \\
x \wedge y & = & y \wedge x & \text{commutativité de } \wedge \\
x \wedge \bot & = & \bot & \bot \text{ est absorbant pour } \wedge \\
x \wedge \top & = & x & \top \text{ est élément neutre pour } \wedge
\end{array}$$

D'autres axiomes peuvent éventuellement être ajoutés (comme la distributivité de $\wedge$ par rapport à $\vee$ ou le fait que $\exists x$ est une projection: $\exists x.\exists x.y = \exists x.y$.) Ce quotient sera précisé complètement au chapitre 3. Mais, pour l'instant, nous n'avons pas besoin de plus de précisions. On peut enfin éventuellement ajouter d'autres connecteurs définis en fonction des précédents.

Les axiomes de l'algèbre des formules du premier ordre nous autorisent à écrire les conjonctions sans parenthésage: $x \wedge y \wedge z$ représente indifféremment

$(x \wedge y) \wedge z$ ou $x \wedge (y \wedge z)$. Ils nous autorisent aussi á supposer que les seules formules du premier ordre contenant $\perp$ ou $\top$ sont précisément $\perp$ et $\top$.

L'ensemble des *variables libres* $\mathcal{V}ar(\phi)$ d'une formule du premier ordre ϕ est définie par récurrence sur la structure de ϕ:

- $\mathcal{V}ar(\top) = \mathcal{V}ar(\perp) = \emptyset$
- $\mathcal{V}ar(p(t_1, \ldots, t_n)) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$
- $\mathcal{V}ar(P \wedge Q) = \mathcal{V}ar(P \vee Q) = \mathcal{V}ar(P) \cup \mathcal{V}ar(Q)$
- $\mathcal{V}ar(\neg P) = \mathcal{V}ar(P)$
- $\mathcal{V}ar(\exists x.P) = \mathcal{V}ar(\forall x.P) = \mathcal{V}ar(P) \setminus \{x\}$

2.1.6 Formules du premier ordre: sémantique

Etant donnés un alphabet de symboles de fonction $\mathcal{F}$ et un alphabet de symboles de prédicats $\mathcal{P}$, une $(\mathcal{F}, \mathcal{P})$*-structure* est une $\mathcal{F}$-algèbre $\mathcal{A}$ munie, pour chaque $p \in \mathcal{P} \setminus \{=\}$ d'un sous-ensemble $p_{\mathcal{A}}$ de $A^{a(p)}$.

La relation de satisfaction $\models$ associe une formule du premier ordre ϕ telle que $\mathcal{V}ar(\phi) = \{x_1, \ldots, x_n\}$ à une structure $\mathcal{A}$ (définie sur les mêmes alphabets de symboles et de prédicats que ϕ) et une assignation dans A. Elle est définie par récurrence sur la structure des formules de la façon suivante:

- Pour tous $\sigma, \mathcal{A}$, $\mathcal{A}, \sigma \models \top$ et $\mathcal{A}, \sigma \not\models \perp$
- $\mathcal{A}, \sigma \models P(t_1, \ldots, t_n)$ si et seulement si $(t_1\sigma, \ldots t_n\sigma) \in p_{\mathcal{A}}$
- $\mathcal{A}, \sigma \models t_1 = t_2$ ssi $t_1\sigma =_{\mathcal{A}} t_2\sigma$
- $\mathcal{A}, \sigma \models \phi_1 \wedge \phi_2$ ssi $\mathcal{A}, \sigma \models \phi_1$ et $\mathcal{A}, \sigma \models \phi_2$
- $\mathcal{A}, \sigma \models \phi_1 \vee \phi_2$ ssi $\mathcal{A}, \sigma \models \phi_1$ ou $\mathcal{A}, \sigma \models \phi_2$
- $\mathcal{A}, \sigma \models \neg\phi$ ssi $\mathcal{A}, \sigma \not\models \phi$
- $\mathcal{A}, \sigma \models \exists x.\phi$ ssi il existe une assignation σ' identique à σ, sauf peut-être en x, telle que $\sigma' \models \phi$.
- $\mathcal{A}, \sigma \models \forall x.\phi$ ssi pour toute assignation σ', identique à σ sauf peut-être en x, $\sigma' \models \phi$.

Lorsque $\mathcal{A}, \sigma \models \phi$, on dira aussi (suivant la terminologie "contraintes") que *σ est une solution de ϕ dans $\mathcal{A}$.*

Lorsqu'il existe une assignation σ et une structure $\mathcal{A}$ telle que $\mathcal{A}, \sigma \models \phi$, on dit que *$\phi$ est satisfaisable.* Symétriquement, lorsque, pour toute assignation σ et toute structure $\mathcal{A}$, $\mathcal{A}, \sigma \models \phi$, on dit que ϕ est *valide.*

La relation de *conséquence logique* est définie sur les ensembles de formules du premier ordre par $\Phi \models \Psi$ si et seulement si, pour toutes formules $\phi \in \Phi$ et $\psi \in \Psi$, $\mathcal{V}ar(\phi) \subseteq \mathcal{V}ar(\psi)$ et, pour tous $\mathcal{A}$, σ, $\mathcal{A}, \sigma \models \phi$ entraine $\mathcal{A}, \sigma \models \psi$.

Dans le cas des contraintes, c'est-à dire lorsque $\mathcal{A}$ est fixée, les notions de satisfaisabilité et de validité doivent se comprendre sans la quantification sur $\mathcal{A}$.

2.1.7 Axiomatisations

Une *axiomatisation* d'une $(\mathcal{F}, \mathcal{P})$-structure $\mathcal{S}$ est un ensemble récursif de formules du premier ordre construites sur $\mathcal{F}, \mathcal{P}$ et qui sont valides dans $\mathcal{S}$.

La *théorie* d'une $(\mathcal{F}, \mathcal{P})$-structure $\mathcal{S}$ (resp. d'un ensemble de formules Φ), notée $Th(\mathcal{S})$ (resp. $Th(\Phi)$) est l'ensemble des formules ϕ telles que $\mathcal{S} \models \phi$ (resp. $\Phi \models \phi$). Remarquons que, si Φ est récursivement énumérable, $Th(\Phi)$ est aussi récursivement énumérable (il suffit d'utiliser n'importe quel système de déduction complet en logique du premier ordre).

Une *théorie du premier-ordre* est un ensemble de formules Φ tel que $Th(\Phi) = \Phi$.

Une théorie Φ est *complète* si, pour toute formule ϕ, ϕ ou $\neg\phi$ (mais pas les deux) sont dans Φ. Par exemple, $Th(\mathcal{S})$ est toujours une théorie complète.

Une axiomatisation *complète* est un ensemble récursif de formules Φ tel que $Th(\Phi)$ est une théorie complète. Remarquons que, si Φ est une axiomatisation complète, alors $Th(\Phi)$ est récursif (i.e. la théorie est décidable).

2.2 Systèmes de contraintes

Definition 2.2.1 *Un* système de contraintes *est un triplet formé de*

1. *un ensemble $\mathcal{L}$ de formules du premier ordre sur un alphabet $\mathcal{F}$ de symboles de fonction et un alphabet $\mathcal{P}$ de symboles de prédicats (contenant ou non l'égalité)*

2. *une $(\mathcal{F}, \mathcal{P})$-structure $\mathcal{S}$*

3. *une fonction (récursive totale) qui associe à chaque formule $\phi \in \mathcal{L}$ ou bien $\emptyset$ si ϕ n'a pas de solution dans $\mathcal{S}$, ou bien un ensemble non vide de solutions de ϕ dans $\mathcal{S}$ si un tel ensemble existe*

La condition 3 ci-dessus peut se lire: la satisfaisabilité d'une formule de $\mathcal{L}$ dans $\mathcal{S}$ doit être décidable. On doit de plus pouvoir exhiber au moins une solution lorsqu'une telle solution existe.

On pourra aussi dans la suite considérer des systèmes de contraintes de deuxième ordre (voire d'ordre supérieur), mais les définitions correspondantes seront alors introduites par nécessité.

2.2.1 Exemple: les systèmes d'équations dans les réels

Reprenant l'exemple de l'introduction:

$$\begin{cases} x = 2(y - t) \\ x - t = y \end{cases}$$

Les formules sont construites à partir de l'alphabet $\mathcal{F}$ constitué des nombres rationnels des symboles $+, -$, et, pour tout rationnel q, du produit par q. Les formules atomiques sont des égalités entre termes et les formules sont des conjonctions d'égalités.

le domaine d'interprétation est l'ensemble des nombres réels (avec l'interprétation habituelle des symboles de fonction).

La technique de résolution de contrainte est, par exemple, l'élimination de Gauss, qui permet aussi de savoir si un système d'équations possède au moins une solution.

2.2.2 Exemple: l'arithmétique de Presburger

Dans l'arithmétique de Presburger, $\mathcal{F}$ contient les deux constantes $0, 1$ et le symbole binaire $+$, $\mathcal{P}$ contient l'égalité (et, si l'on veut, le symbole d'inégalité $\geq$). $\mathcal{L}$ est l'ensemble de toutes les formules du premier ordre sur $\mathcal{F}, \mathcal{P}$.

La structure $\mathcal{S}$ est donnée par l'ensemble des entiers naturels, avec les opérations usuelles (et éventuellement l'ordre sur les entiers naturels).

La vérification du point 3 n'est pas triviale: c'est justement l'objet des chapitres qui vont suivre de mettre en place des techniques dites "de résolution de contrainte". Pour l'instant, nous pouvons simplement nous référer à [106, 107].

2.2.3 Exemple: Problèmes d'unification

On considère un alphabet de symboles de fonction quelconque (en fait, ces contraintes sont *paramétrées* par l'alphabet de symboles de fonction). Le seul symbole de prédicat est l'égalité. Les formules considérées sont $\top, \bot$ ou bien sont des disjonctions de formules qui s'écrivent:

$$\exists \vec{x}.\ s_1 = t_1 \wedge \ldots \wedge s_n = t_n$$

où $\exists \vec{x}$ est une abréviation pour $\exists x_1.\exists x_2.\ldots\exists x_m$ (m quelconque) et $s_1, t_1, \ldots, s_n, t_n \in T(\mathcal{F}, \mathcal{X})$.

La structure dans laquelle ces formules sont interprétées est $T(\mathcal{F})$ ou $RT(\mathcal{F})$(ou bien l'un des quotients $T(\mathcal{F})/_{=_E}$, mais dans ce cas, le point 3. de la définition exige certaine conditions sur $=_E$).

Le point 3. est assuré par l'existence d'algorithmes d'unification (qui terminent) dont nous discuterons plus en détail dans le chapitre 3.

2.2.4 Les théories décidables

En fait, n'importe quelle théorie décidable (ou fragment décidable de théorie) peut être vu comme un système de contraintes. (Voir l'article de synthèse de M.O. Rabin sur les théories décidables [107]). Par exemple, la théorie des nombres réels conduit à un système de contraintes. Dans cette théorie, les symboles de fonction sont, par exemple, $0, 1, +, *$ (ce qui permet de définir les fonctions polynômes) et le symbole de prédicat est l'égalité. $\mathcal{S}$ est la structure correspondante des nombres réels. Le résultat de décidabilité est dû à Tarski dans les années 50 [119].

La seule différence entre "fragment décidable de théorie" et "système de contraintes" est le caractère effectif exigé pour le calcul d'une solution. En fait c'est vraiment le moins qu'on puisse exiger pour une utilisation en informatique. D'autres considérations auront à être prises en compte de façon plus ou moins formelle.

2.2.5 Contraintes symboliques

Definition 2.2.2 *Un système de contraintes $(\mathcal{L}, \mathcal{S}, f)$ est dit* symbolique *lorsque $\mathcal{S}$ est $T(\mathcal{F}')$, $RT(\mathcal{F}')$, $IT(\mathcal{F}')$ ou l'un de ses quotients par une congruence $=_E$, pour un certain $\mathcal{F}'$ contenant l'alphabet $\mathcal{F}$ de symboles de fonction sur lequel sont construites les formules de $\mathcal{L}$.*

Les problèmes d'unification sont des exemples de systèmes de contraintes symboliques. L'arithmétique de Presburger en est un autre exemple, à condition de préciser correctement la congruence $=_E$ et l'alphabet $\mathcal{F}'$ (voir chapitre 3).

L'intérêt des contraintes symboliques est la transparence de leur combinaison avec une logique, ce que nous voyons dans le paragraphe qui suit.

2.3 Logique contrainte

Les systèmes de contraintes possèdent leur intérêt mathématique propre. D'ailleurs nombre d'entre eux ont été étudiés bien avant que l'informatique n'existe. Mais, les systèmes de contraintes connaissent un regain d'intérêt à cause de leur utilisation potentielle en conjonction avec une logique, par ce qu'ils permettent de représenter aisément des ensembles très grands (éventuellement infinis) de formules. Ils permettent aussi d'exprimer les stratégies en démonstration automatique, dans les formules elles-mêmes, ce qui présente de nombreux avantages que nous allons expliquer dans le paragraphe suivant.

2.3.1 Définitions

Definition 2.3.1 *Une* logique contrainte *est constituée de*

- *une logique : un ensemble de formules Φ_L, une classe de modèles $\mathcal{M}_L$ et une relation de satisfaction $\models_L$*
- *un système de contraintes $(\mathcal{L}, \mathcal{S}, f)$*
- *pour chaque $M \in \mathcal{M}_L$ une fonction de* recollage *H_S du domaine D_S de $\mathcal{S}$ dans le domaine D_M de M*

Definition 2.3.2 *Les* formules *de la logique contraintes sont formées d'une paire d'un élément ϕ de Φ_L et d'une contrainte c de $\mathcal{L}$, notée $\phi \mid c$.*

Definition 2.3.3 *L'ensemble $\mathcal{V}ar(\phi \mid c)$ des* variables libres *de $\phi \mid c$ est la différence $\mathcal{V}ar(\phi) \setminus \mathcal{V}ar(c)$ (les "variables contraintes" ne sont pas libres).*

Definition 2.3.4 *La* relation de satisfaction *dans la logique contrainte est définie de la façon suivante: si M est une structure de $\mathcal{M}_L$ et σ est une* *assignation de $\mathcal{V}ar(\phi \mid c)$ dans D_M,*

$$\sigma, M \models \phi \mid c$$
$$\textit{si et seulement si, pour toute assignation } \theta \textit{ telle que } \mathcal{S}, \theta \models c,$$
$$M, H_S(\theta) \oplus \sigma \models_L \phi.$$

Où $H_S(\theta)$ est l'assignation qui à une variable x associe $H_S(\theta(x))$. $\oplus$ est ici la *juxtaposition* des assignations. Plus précisément, $(H_S(\theta) \oplus \sigma)(x) = x\sigma$ si $x \in \mathcal{V}ar(\phi \mid c)$ et $H_S(x\theta)$ sinon.

Exemple 2.3.5 Soit $\mathcal{F}_1 = \{\times\} \cup \mathbf{N}$ (chaque élément de $\mathbf{N}$ est une constante, $\times$ étant binaire), $\mathcal{F}_2 = \{0, S, +\}$ et $\mathcal{P}_1 = \{=, \geq\}$, $\mathcal{P}_2 = \{=\}$. On considère la logique du premier ordre $\mathcal{L}_1$ sur $\mathcal{F}_1, \mathcal{P}_1$, dans laquelle $\geq$ est interprété comme un ordre total bien fondé sur un ensemble infini d'une part et le système de contraintes $\mathcal{C}$ de l'arithmétique de Presburger sur $\mathcal{F}_2, \mathcal{P}_2$ d'autre part. La logique contrainte dans laquelle les formules de $\mathcal{L}_1$ sont contraintes par $\mathcal{C}$ peut être définie par le recollage H_S^1 qui associe à un entier n, l'interprétation de n dans M. Mais d'autres logiques contraintes peuvent être définies à partir de ces composantes, par exemple à l'aide de la fonction de recollage H_S^2 qui interprète n comme l'ordinal n. (Ce qui donne en général des interprétations différentes).

Par exemple, considérons le modèle de $\mathcal{L}_1$ dont le domaine est l'ensemble des paires d'entiers, $\times$ est le produit composante à composante, $\geq$ est l'ordre lexicographique sur les paires et n est la paire (n, n). Alors $H_S^2(n)$ est cette fois la paire $(0, n)$.

Exemple 2.3.6 Prenons un exemple plus significatif: on considère la même logique que dans l'exemple précédant, et la même syntaxe des contraintes, mais cette fois interprétée dans les Booléens, de domaine $\{\top, \bot\}$, S étant la négation, 0 étant $\top$ et $+$ la disjonction. Il n'y a plus maintenant d'interprétation "naturelle" de la logique contrainte et la fonction de recollage est nécessaire pour expliquer ce qu'est la combinaison.

Une variante plus restrictive mais plus simple des logiques contraintes et qui permet d'éviter en particulier le recollage est de ne considérer comme modèles de la logique contrainte que les modèles de la logique contenant le domaine des contraintes (ou une de ses images homomorphes).

2.3.2 Simplifications dans le cas des contraintes symboliques

Les définitions qui précèdent se simplifient beaucoup dans le cas de contraintes symboliques, qui est le cas qui nous intéresse ici.

Si les formules de la logique sont construites sur un alphabet $\mathcal{F}''$ qui contient l'alphabet $\mathcal{F}'$ sur lequel est bâtie l'interprétation des contraintes (cf. définition 2.2.2), alors une formule containte $\phi \mid c$ peut être vue comme une représentation de l'ensemble de formules

$$\{\phi\sigma \mid \mathcal{S}, \sigma \models c\}$$

Les notions de variables libres et de satisfaction du paragraphe précédent coïncident alors avec les notions correspondantes sur les ensembles de formules. En fait, dans le cas de contraintes symboliques, la fonction de recollage est implicitement l'identité.

2.3.3 Contraintes vs conditions

Les contraintes sont différentes de ce que l'on appelle souvent *condition* en démonstration automatique. Autrement dit, il ne faut pas confondre $\phi \mid c$ avec $\phi \Leftarrow c$. La différence vient des interprétations qui sont faites de chacune de ces formules. Dans le premier cas, la containte c est interprétée dans un domaine fixé qui ne dépend pas de la formule ϕ ou d'autres formules. Dans le deuxième cas, la condition c est interprétée dans la même structure que ϕ. Prenons un exemple qui montre que cette distinction est essentielle.

Exemple 2.3.7 Considérons les formules

$$\begin{cases} c = d \quad \mid \quad a = b \\ a = b \end{cases}$$

$\mathcal{F}$ est constitué des 4 constantes a, b, c, d. Le système de contraintes étant les problèmes d'unification.

Peut-on déduire $c = d$ de ces formules? Clairement oui dans le cas où $a = b$ serait une condition de la première équation (i.e. en remplaçant $\mid$ par $\Leftarrow$. Mais, avec les définitions que nous avons données, la réponse est non. En effet, comme a et b sont deux constantes distinctes, la contrainte $a = b$ dans la première formule est fausse. $c = d \mid a = b$ représente ainsi un ensemble vide de formules et le système ci-dessus se réduit à $a = b$.

Le problème vient en fait d'une confusion de notations: le signe $=$ à gauche et à droite de $\mid$ n'étant pas interprétés de la même manière.

2.4 Exercices

1. Montrer que $T(\mathcal{F}) \subset RT(\mathcal{F}) \subset IT(\mathcal{F})$ mais que les inclusions sont strictes dès que $\mathcal{F}$ contient au moins un symbole d'arité au moins deux ou bien au moins deux symboles d'arité 1. Que se passe-t-il dans les autres cas?

2. Montrer que, s'il existe un homomorphisme de $T(F)$ dans $RT(F)$ qui est l'identité sur $T(F)$, alors $T(F) = RT(F)$.

3. Montrer que, étant donné un terme $t \in RT(\mathcal{F})$ et $p \in \mathrm{Pos}(t)$ telle que $p \neq \Lambda$, il existe un unique $u \in T(\mathcal{F})$ tel que $u \equiv t[u]_p$.

4. Existe-t-il des $\mathcal{F}$-algèbres vides? Pourquoi?

5. Un terme $t \in T(\mathcal{F}, \mathcal{X})$ est *linéaire* si, pour tout $x \in \mathcal{X}$, il existe au plus une position $p \in \mathrm{Pos}(t)$ telle que $t(p) = x$. Deux termes $t, u \in T(\mathcal{F}, \mathcal{X})$ sont *unifiables* s'il existe une substitution σ telle que $t\sigma \equiv u\sigma$.

   (a) soient $t_1, \dots, t_n \in T(\mathcal{F}, \mathcal{X})$ n termes linéaires tels que $\mathcal{V}ar(t_i) \cap \mathcal{V}ar(t_j) = \emptyset$ pour tous $i \neq j$. Montrer que

   $$(\forall i, j,\ t_i \text{ et } t_j \text{ sont unifiables }) \leftrightarrow (\exists \sigma, t_1\sigma \equiv t_2\sigma \equiv \dots \equiv t_n\sigma)$$

   (b) Montrer par des contre-exemples que le résultat ci-dessus n'est plus vrai dès qu'on enlève l'hypothèse de linéarité ou l'hypothèse de non partage de variables.

6. Soit $E = \{0 + x = x;\ s(x) + y = s(x + y)\}$ et $\mathcal{F} = \{0, s, +\}$ (0 est une constante, s est unaire et $+$ est binaire utilisé en notation infixée). Montrer que $T(\mathcal{F}, \mathcal{X})/_{=_E}$ ne satisfait pas $x + 0 = x$ mais que $T(\mathcal{F}) \models x + 0 = x$.

7. Donner, pour les formules de l'exemple 2.3.7 un modèle de l'ensemble d'axiomes qui ne satisfasse pas $c = d$.

8. Si A est une partie du domaine D d'une $\mathcal{F}$-algèbre $\mathcal{A}$, on note $[A]$ la plus petite sous-algèbre de $\mathcal{A}$ contenant A. A est alors appelé *ensemble de générateurs* de $[A]$.

   (a) Montrer que $[A]$ existe toujours.

$$
\begin{array}{lrcl}
\forall x. & x = x & & \\
\forall x, y. & x = y & \Rightarrow & y = x \\
\forall x, y, z & x = y \wedge y = z & \Rightarrow & x = z \\
\forall \vec{x}, \forall \vec{y}. & x_1 = y_1 \wedge \ldots \wedge x_{a(f)} = y_{a(f)} & \Rightarrow & f(x_1, \ldots, x_{a(f)}) = f(y_1, \ldots, y_{a(f)})
\end{array}
$$

Pour tout symbole $f \in \mathcal{F}$.

FIG. 2.1 - *Axiomes de l'égalité*

(b) Montrer que, si $\mathcal{F}$ contient au moins un symbole non constant, alors il n'existe pas de sous-ensemble fini A de $RT(\mathcal{F})$ tel que $[A] = RT(F)$.

(c) Un sous-ensemble A du domaine de la $\mathcal{F}$-algèbre $\mathcal{A}$ est appelé *système libre* si, pour tous termes $t, u \in T(\mathcal{F}, \mathcal{X})$ et toute assignation σ dans $\mathcal{A}$,

$$t\sigma =_{\mathcal{A}} u\sigma \;\wedge\, ; \forall x \in \mathcal{X}, x\sigma \in A \quad t \equiv u$$

Montrer qu'il existe dans $T(\mathcal{F})$ un système libre infini si et seulement si il existe dans $RT(\mathcal{F})$ un système libre infini.

(d) Un ensemble A est appelé *base* de $\mathcal{A}$ si A est un ensemble de générateurs libre de $\mathcal{A}$. Une $\mathcal{F}$-algèbre est *libre* si elle admet une base. Montrer que $T(\mathcal{F})$ et $T(\mathcal{F}, \mathcal{X})$ sont des algèbres libres, mais que, dès que $\mathcal{F}$ contient un symbole non constant, $RT(\mathcal{F})$ n'est pas libre.

(e) Une $\mathcal{F}$-algèbre $\mathcal{A}$ est *localement libre* si toute sous-algèbre finiment engendrée de $\mathcal{A}$ est libre. Montrer que toute algèbre libre est localement libre mais que la réciproque est fausse. $RT(\mathcal{F})$ est-elle localement libre?

9. Les *axiomes de l'égalité* Eq sont constitués de l'ensemble (récursif) de formules de la figure 2.1. Montrer que $Eq \models (\forall x.(x \neq t \vee \phi)) \leftrightarrow \phi\{x \mapsto t\}$.

10. Définir une logique contrainte qui permet de combiner un langage du style de PROLOG (i.e. les clauses de Horn pures) avec l'arithmétique de Presburger.

11. Montrer que tout système de contraintes peut en fait être vu comme un système de contraintes symboliques. Y a-t-il alors coincidence entre la

simplification du paragraphe 2.3.2 et la definition 2.3.4? Donner une définition alternative des interprétations d'une logique contrainte et expliquer pourquoi nous n'avons pas retenu cette définition.

12. Soit $\mathcal{F}$ l'ensemble de symboles composé de a (constante) et f (unaire). On considère l'équation contrainte $e : f(x) = a \mid x \neq a$. L'égalité dans les contraintes étant interprétée comme l'identité des arbres $\equiv$. A-t-on $e \models f(a) = a$? Justifier sa réponse.

Chapitre 3

Résolution de contraintes par des méthodes syntaxiques

Comme nous l'avons expliqué, les méthodes syntaxiques consistent essentiellement à réécrire les formules jusqu'à obtenir une forme résolue. C'est la méthode la plus utilisée (consciemment ou non). C'est en particulier sur ce principe que sont conçus les algorithmes d'unification ou les preuves de décidabilité par élimination de quantificateurs. Même pour les contraintes qui sont résolues par d'autres méthodes, on utilise systématiquement des étapes de simplification de formules par réécriture. L'idée est d'ailleurs extrêmement naturelle: elle consiste à appliquer quelques règles de transformations permettant de simplifier le problème à résoudre.

Nous commençons ci-dessous (après un bref aperçu de la méthode) par rappeler les outils nécessaires de la réécriture (paragraphe 3.2) et renvoyons pour plus de détails à l'article de synthèse [48]. Nous donnons aussi quelques outils de preuve de terminaison dans le paragraphe 3.3. Après une description plus détaillée des techniques de résolution de contraintes par réécriture de formules (paragraphe 3.4 nous étudions un certains nombre d'exemples: la programmation linéaire, méthode du simplexe dans le paragraphe 3.5, l'unification standard (paragraphe 3.6), l'unification syntaxique (au sens de C. Kirchner [77], paragraphe 3.7), l'arithmétique de Presburger (paragraphe 3.8), les problèmes équationnels (paragraphe 3.9), la logique des traîts (paragraphe 3.10). Cette liste n'est pas exhaustive. Elle n'a pas non plus vocation à être lue linéairement. Pour chaque système nous nous efforcerons aussi d'évoquer son utilisation dans d'autres domaines de l'informatique.

3.1 Un bref Aperçu de la méthode

L'idée générale est extrèmement simple: pour résoudre une contrainte, on définit un ensemble (récursif) de règles qui transforment une formule en une formule équivalente (c'est-à dire qui a même ensemble de solutions). Le système de règles doit être défini de façon à toujours terminer.

On définit d'autre part un ensemble de *formes résolues* qui sont des contraintes particulières, qui sont ou bien trivialement insatisfaisables (i.e. la formule $\perp$) ou bien pour lesquelles il est aisé de construire, ou même d'énumérer les solutions. Par exemple, dans le cas de l'unification standard, une forme résolue représentera un unificateur le plus général. (Le choix de définition d'une forme résolue n'est pas unique.)

Le système de réécriture de formules devra être *complet*: toute formule irréductible doit être une forme résolue. Dans ce cas, la résolution de contrainte consiste en une réduction arbitraire de la formule jusquà une de ses formes irréductibles.

En général, les règles de réécriture ne sont pas "déterministes": plusieurs règles peuvent être appliquées à une même formule. Mais il s'agira ici d'un non-déterminisme dans lequel un seul (n'importe lequel) choix suffit. Ce non-déterminisme est donc un avantage et non un inconvénient par rapport à un algorithme déterministe résolvant le même problème puisqu'il fournit une famille d'algorithmes de résolution de contraintes (tous ceux qui sont obtenus en renforçant le contrôle pour le rendre déterministe). Ceci s'est déjà montré interessant dans le cas de l'unification, un même systèmes de règles pouvant être utilisé de plusieurs façons selon le problème à résoudre, permettant ainsi de choisir le plus efficace sur chaque instance. Nous nous attachons donc à prouver la terminaison pour un système de règles aussi non-déterministe que possible.

Les règles de réécritures, si elles sont vues comme des équivalences de formules, fournissent une axiomatisation de la structure $\mathcal{S}$. On obtient donc en retour une axiomatisation complète de la structure.

3.2 Systèmes de réécriture: notions de base

3.2.1 Relations de réduction, confluence et terminaison

Étant donné une $\mathcal{F}$-algèbre $\mathcal{A}$, une *relation de réduction* sur $\mathcal{A}$ est une relation binaire $\rightarrow$, compatible avec la structure d'algèbre: pour tout $f \in \mathcal{F}$, pour tous $t_1, \ldots, t_{a(f)}, t'_i \in \mathcal{A}$, si $t_i \to t'_i$, alors

$$f_{\mathcal{A}}(t_1, \ldots, t_n) \to f_{\mathcal{A}}(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n)$$

La *fermeture transitive* (resp. *fermeture réflexive*, resp. *fermeture symétrique*, resp. *fermeture réflexive transitive*) d'une relation $\to$ est la plus petite relation transitive (resp. réflexive, resp. symétrique, resp. réflexive et transitive) contenant $\to$. Elle est notée $\xrightarrow{+}$ (resp. $\xrightarrow{=}$, resp. $\leftrightarrow$, resp. $\xrightarrow{*}$).

Une relation $\to$ est *confluente* si, pour tous t_1, t_2, t_3, $t_1 \xrightarrow{*} t_2$ et $t_1 \xrightarrow{*} t_3$ implique l'existence d'un t_4 tel que $t_2 \xrightarrow{*} t_4$ et $t_3 \xrightarrow{*} t_4$.

Une relation $\to$ *termine* (ou est *bien fondée*), s'il n'esiste pas de chaîne infinie $t_1 \to t_2 \ldots \to t_n \to \ldots$.

3.2.2 Systèmes de réécriture standards

Étant donné un alphabet $\mathcal{F}$ de symboles de fonction, un *système de réécriture* est un ensemble (récursif) de paires (orientées) de termes de $T(\mathcal{F}, \mathcal{X})$. Chacune des paires du système est une *règle de réécriture* notée $t_1 \to t_2$.

La *relation de réduction* associée à un système de réécriture $\mathcal{R}$ est la plus petite relation de réduction $\underset{\mathcal{R}}{\longrightarrow}$ sur $T(\mathcal{F}, \mathcal{X})$ telle que, pour toute subsitution σ et toute règle de réécriture $t_1 \to t_2$, $t_1\sigma \underset{\mathcal{R}}{\longrightarrow} t_2\sigma$. De façon plus opérationnelle, $t \underset{\mathcal{R}}{\longrightarrow} t'$ s'il existe une position p de t, une règle $l \to r$ de $\mathcal{R}$ et une substitution σ telles que $t|_p \equiv l\sigma$ et $t' \equiv t[r\sigma]_p$.

Un système de réécriture termine (resp. est confluent) si la relation de réduction associée termine (resp. est confluente). Un système de réécriture est *convergent* s'il termine et est confluent. Lorsque $\mathcal{R}$ est convergent, pour tout terme $t \in T(\mathcal{F}, \mathcal{X})$ il existe un unique terme $t \downarrow_{\mathcal{R}}$ tel que $t \xrightarrow[\mathcal{R}]{*} t \downarrow_{\mathcal{R}}$ et il n'existe aucun terme t' tel que $t \downarrow_{\mathcal{R}} \xrightarrow[\mathcal{R}]{+} t'$. $t \downarrow_{\mathcal{R}}$ est appelé *forme normale de t* pour $\mathcal{R}$. Dans le cas de systèmes convergents, on peut aussi décider l'égalité engendrée: $t \xleftrightarrow[\mathcal{R}]{*} t'$ ssi $t \downarrow_{\mathcal{R}} \equiv t' \downarrow_{\mathcal{R}}$.

3.2.3 Réécriture modulo E

La réécriture standard pose des problèmes en présence de symboles commutatifs: l'équation $x + y = y + x$ ne peut pas être orientée en une règle de réécriture sans perdre la propriété de terminaison. Plutôt que de tenter d'orienter ces axiomes, on préfère alors réécrire dans un quotient de l'alègre des termes par une congruence $=_E$. Mais, pour que la réécriture reste effective dans le cas de classes d'équivalences infinies, on est obligé d'utiliser une restriction de la réécriture des classes:

Soit E un ensemble d'équations et $\mathcal{R}$ un système de réécriture. Un terme t *se réécrit par* $\mathcal{R}$ *modulo* E en t' (ce que l'on note $t \xrightarrow[\mathcal{R}/E]{} t'$) s'il existe une position p de t, une substitution σ et une règle $l \to r$ de $\mathcal{R}$ telles que $t|_p =_E l\sigma$ et $t' \equiv t[r\sigma]_p$.

$\xrightarrow[\mathcal{R}/E]{}$ possède *la propriété de Church-Rosser* si, pour tous termes s, t, $s \xleftrightarrow[\mathcal{R}/E]{*} t$ ssi il existe des termes u, v tels que

$$s \xrightarrow[\mathcal{R}/E]{*} u \xleftrightarrow[E]{*} v \xleftarrow[\mathcal{R}/E]{*} t$$

Supposant que $\xrightarrow[\mathcal{R}/E]{}$ est bien fondée, la propriété de Church-Rosser est équivalente à la *locale confluence* plus la *locale cohérence* de $\xrightarrow[\mathcal{R}/E]{}$, propriétés qui sont décidables.

$\xrightarrow[\mathcal{R}/E]{}$ est localement confluente si, pour tous termes s, t, u, $u \xleftarrow[\mathcal{R}/E]{} s \xrightarrow[\mathcal{R}/E]{} t$ implique qu'il existe v_1, v_2 t.q. $u \xrightarrow[\mathcal{R}/E]{*} v_1 \xleftrightarrow[E]{*} v_2 \xleftarrow[\mathcal{R}/E]{*} t$ (i.e. les "pics" se réduisent en "vallées"). La relation est localement cohérente ssi, pour tous termes s, t, u, $s \xleftrightarrow[E]{} t \xrightarrow[u]{\mathcal{R}/E}$ implique qu'il existe des termes v_1, v_2 tels que $s \xrightarrow[\mathcal{R}/E]{*} v_1 \xleftrightarrow[E]{*} v_2 \xleftarrow[\mathcal{R}/E]{*} t$

$\mathcal{R}$ est *convergent modulo* E si $\xrightarrow[\mathcal{R}/E]{}$ est bien fondée, localement confluente et localement cohérente. De façon équivalente, $\mathcal{R}$ est convergent modulo E ssi pour tout terme t, il existe un terme u irréductible tel que d'une part $t \xrightarrow[\mathcal{R}/E]{*} u$ et d'autre part $t \xrightarrow[\mathcal{R}/E]{*} v$ et v irréductible entraine $v \xleftrightarrow[u]{*}$ (Unicité de la forme normale modulo E).

Mais ces notions ne nous intéresserons que de façon annexe. L'important pour la résolution de contraintes, ce sont les preuves de terminaison.

3.3 Preuves de terminaison

La terminaison des systèmes de réécriture est indécidable. Pour prouver la terminaison d'un ensemble de règles on ne dispose donc pas de procédé systématique. Mais le problème de terminaison se ramène à la recherche d'ordres bien fondés contenant la relation de réduction. C'est pourquoi il est utile de connaitre quelques rudiments sur la construction d'ordres bien fondés.

3.3.1 Ordres bien fondés

Un préordre est une relation réflexive et transitive. Si $\geq$ est un préordre sur A, la relation d'équivalence associée $\approx_{\geq}$ est dfinie pas $s \approx_{\geq} t$ ssi $s \geq t$ et $t \geq s$. La relation d'ordre strict associée à $\geq$ est alors définie par $> \stackrel{\text{def}}{=} \geq \setminus \approx_{\geq}$.

Un préordre $\geq$ est bien fondé sur un ensemble A s'il n'existe pas de chaîne infinie $a_1 > a_2 \ldots > a_n \ldots$.

L'ordre sur les entiers naturels est bien fondé. Sur les termes, l'ordre de sous-terme est bien fondé.

On peut encore définir des classes d'ordres bien fondés à partir d'un *pré bel ordre*. Un pré bel ordre sur A est un préordre $\geq$ tel que, dans tout sous-ensemble infini de A, il existe deux éléments comparables pour $\geq$. Tout préordre contenant un pré bel ordre est bien fondé. Cette technique permet de construire des ordres bien fondés. Par exemple, le *plongement* qui est défini sur les arbres finis construits sur un alphabet fini $\mathcal{F}$ comme la relation de réduction associée à l'ensemble de règles de réécriture $f(x_1, \ldots, x_{a(f)}) \rightarrow x_i$, pour $1 \leq i \leq a(f)$ et $f \in \mathcal{F}$, est un bel ordre. Ce résultat n'est pas trivial. C'est une conséquence du théorème de Kruskal (voir par exemple [48] pour une description un peu plus détaillée). Ce résultat se généralise au cas où $\mathcal{F}$ est infini mais muni lui-même d'un pré bel ordre (voir à nouveau [48]). Donc tout ordre sur $T(\mathcal{F})$ contenant le plongement est bien fondé.

3.3.2 Les fonctionnelles lexicographique et multi-ensemble

Pour construire des ordres bien fondés, on dispose également de deux outils fondamentaux qui sont les extensions lexicographique et multi-ensemble.

Si $(A_1, \geq_1), \ldots, (A_n, \geq_n)$ sont n ensembles, chacun étant muni d'un préordre, la *composée lexicographique* de $\geq_1, \ldots, \geq_n$ est le préordre défini sur $A_1 \times \ldots \times A_n$ par: $(a_1, \ldots, a_n) \geq (a'_1, \ldots, a'_n)$ ssi $\exists 0 \leq i \leq n$, $a_1 \approx_{\geq_1} a'_1 \ldots a_i \approx_{\geq_i} a'_i$ et $a_{i+1} >_i a'_{i+1}$. Le résultat suivant n'est pas très difficile à prouver:

Theorem 3.3.1 *La composée lexicographique de n préordres est bien fondée si et seulement si chacun des préordres composantes est bien fondé.*

Preuve:
En raisonnant par l'absurde, à partir d'une suite infinie strictement décroissante sur $A_1 \times \ldots \times A_n$ on peut construire une suite strictement décroissante sur l'un des A_i. $\square$

Un *multi-semble* (finis) sur un ensemble E est une application de E dans l'ensemble des entiers naturels, qui est nulle pour tous les éléments de E sauf un nombre fini d'entre eux. Une telle application est notée en énumérant les éléments de E sur lesquelles l'application prend une valeur non nulle, chacun d'entre eux étant répété autant de fois que sont image. Par exemple, le multi-ensemble d'entiers défini par $M(0) = 1, M(3) = 3, M(4) = 2$ et nul ailleurs se note $\{0, 3, 3, 3, 4, 4\}$ (ou $\{3, 0, 3, 4, 4, 3\}$ ou tout autre permutation: l'ordre d'énumération n'est pas significatif ici).

L'union de multi-ensembles M_1, M_2 est le multi-ensemble défini, pour tout $e \in E$, par $(M_1 \cup M_2)(e) = M_1(e) + M_2(e)$. La soustraction de multi-ensembles est définie par $(M_1 \setminus M_2)(e) = \max(M_1(e) - M_2(e), 0)$. Le multi-ensemble vide ($\emptyset$) est l'application identiquement nulle. $m \in M$ si $M(m) \neq 0$. $M \subseteq N$ si, pour tout n, $M(n) \geq N(n)$.

Si E est muni d'un préordre $\geq$, l'ensemble des multi-ensembles finis sur E est muni de l'*extension multi-ensemble* notée $\underline{\gg}$. définie par:

$\emptyset \underline{\gg} \emptyset$ et, si M ou N est non vide, $M \underline{\gg} N$ ssi

- ou bien il existe $m \in M$, il existe $n \in N$ tels que $m \approx_{\geq} n$ et $M \setminus \{m\} \underline{\gg} N \setminus \{n\}$
- ou bien il existe $m \in M$ et il existe $N' \subseteq N$ tels que, pour tout $n \in N'$, $m > n$ et $M \setminus \{m\} \underline{\gg} N \setminus N'$

On peut aussi définir $\gg$ (i.e. $\underline{\gg} \setminus \approx_{\underline{\gg}}$) de manière équivalente comme la fermeture transitive de la relation $\rightsquigarrow$ définie par:

$$M \cup \{m\} \rightsquigarrow M \cup N \text{ ssi } \forall n \in N, m > n$$

Autrement dit, un multi-ensemble M est strictement plus grand qu'un multi-ensemble N, si N peut être obtenu à partie de M en itérant le remplacement d'un élément de M par un nombre quelconque d'éléments strictement plus petits. Par exemple, $\{1, 2, 3, 4\} \gg \{2, 2, 2, 2, 4\}$. Si E est totalement ordonné, il suffit d'ailleurs, après avoir retiré les éléments communs (avec leur multiplicité) de comparer les éléments maximaux de chaque multi-ensemble.

Théorème 3.3.2 *$\geq$ est bien fondé ssi $\underline{\gg}$ est bien fondé.*

Preuve:

On raisonne par l'absurde: si $\gg$ n'est pas bien fondée, il existe une suite infinie strictement décroissante $M_1 \gg \ldots \gg M_k \ldots$ de multi-ensembles.

Appelons *persistant* un élément $m \in M_i$ tel que, pour tout $j > i$, $m \in M_j$. Soit alors pour tout i, M'_i le multi-ensemble obtenu à patir de M_i en retirant tous les éléments persistants. On a encore $M'_1 \rightsquigarrow \ldots \rightsquigarrow M'_k \ldots$. Si bien qu'on peut en fait considérer que la suite M_i ne contient pas d'élément persistant.

Soit $m \in M_j$. Comme m n'est pas persistant, il existe un indice $i \geq j$ tel que $m \in M_i$ et $m \notin M_{i+1}$. On lui associe alors le multi-ensemble $S(m)$ des éléments de M_{i+1} qui lui sont inférieurs. S est étendu aux multi-ensembles par

$$S(M) = \bigcup_{m \in M} S(m)$$

On note encore $S^\infty(m)$ la suite $\{m\}, S(m), S(S(m)), \ldots$. L'une des suites $S^\infty(m)$, pour $m \in M_1$ est donc infinie. (Sans quoi, il existerait un k tel que $S^k(m) = \emptyset$ pour tout $m \in M_1$, ce qui entrainerait la finitude de la suite M_i). Ainsi, pour toute suite infinie strictement décroissante M_i nous avons construit une suite infinie strictement décroissante $\{m\} \gg S(m) \gg \ldots$ où $m \in M_1$. On peut donc par récurrence construire une suite $m_1, \ldots, m_k, \ldots$ telle que $m_1 \in M_1$, $m_2 \in S(m_1), \ldots, m_{k+1} \in S(m_k), \ldots$. Ce qui fournit une suite infinie strictement décroissante pour $>$, d'où une contradiction. $\Box$

3.3.3 Ordres de réduction. Exemples

Un ordre $\geq$ sur $T(\mathcal{F})$ est *monotone* si, pour tout terme $t \in T(\mathcal{F})$ et tout $p \in \mathrm{Pos}(t)$, $u > v \Rightarrow t[u]_p > t[v]_p$. Un *ordre de réduction* sur $T(\mathcal{F})$ est un ordre bien fondé et monotone.

Un ordre $\geq$ *a la propriété de sous-terme* s'il contient l'ordre de sous-terme. Un *ordre de simplification* est un ordre monotone sur $T(\mathcal{F})$ qui a la propriété de sous-terme. Un tel ordre est toujours un ordre de réduction car il contient le plongement (cf. remarque du paragraphe 3.3.1).

Parmi les ordres de simplification les plus utilisés en réécriture, se trouvent les *ordres récursifs sur les chemins* introduits par N. Dershowitz [47]. Etant donné un alphabet $\mathcal{F}$ et un ordre bien fondé $\succeq_{\mathcal{F}}$ sur $\mathcal{F}$ et une application statut de $\mathcal{F}$ dans $\{\mathrm{lex}, \mathrm{mult}\}$, pour tous $s, t \in T(\mathcal{F})$,

$s \equiv f(s_1, \ldots, s_m) >_{rpo} g(t_1, \ldots, t_n) \equiv t$ ssi

- ou bien il existe un indice i tel que $s_i \geq_{rpo} t$

- ou bien $f \succ_{\mathcal{F}} g$ et, pour tout j, $s \succ_{rpo} t_j$
- ou bien $f = g$ et statut(f) = lex et
  - pour tout j, $s \succ_{rpo} t_j$ et
  - $(s_1, \ldots, s_n) \gg_{lex} (t_1, \ldots, t_n)$

  où $\gg_{lex}$ est l'extension lexicographique de n exemplaires de $>_{rpo}$
- ou bien $f = g$ et statut(f) = mult et $\{s_1, \ldots, s_n\} \gg_{mult} \{t_1, \ldots, t_n\}$ où $\gg_{mult}$ est l'extension multiensemble de $>_{rpo}$.

$\geq_{rpo}$ est un ordre de simplification (ce qui demande une petite preuve, en particulier pour montrer la transitivité de la relation).

$\geq_{rpo}$ s'étend à $T(\mathcal{F}, \mathcal{X})$ en déclarant que, dans la préséance $\succ_{\mathcal{F} \cup X}$ les variables sont incomparables avec tout autre symbole. $\mathcal{F} \cup X$ est ainsi muni d'un ordre bien fondé. L'ordre $\succ_{rpo}$ ainsi défini sur $T(\mathcal{F}, \mathcal{X})$ a une propriété supplémentaire: il est *stable par instanciation*: si σ est une substitution et si $s \geq_{rpo} t$, alors $s\sigma \geq_{rpo} t\sigma$. Ceci permet d'effectuer les preuves de terminaison des systèmes de réécriture en comparant simplement les membres gauche et droit de chaque règle.

Malheureusement, l'ordre $\geq_{rpo}$ est insuffisant pour les preuves de terminaison de la réécriture modulo, et en particulier de la réécriture modulo associativité-commutativité qui est essentielle dans la réécriture des formules. Il est en effet nécessaire d'avoir un ordre *compatible avec* E, c'est-à-dire un ordre tel que, pour tous termes s, t, u, v, $s =_E t$, $t \geq u$, $u =_E v$ implique $s \geq v$. Mais, en présence de symboles associatifs-commutatifs, $\geq_{rpo}$, les symboles associatifs-commutatifs ont nécessairement le statut multi-ensemble. Mais c'est insuffisant pour obtenir la compatibilité:

Exemple 3.3.3 Soit $\mathcal{F} = \{0, +\}$ où $+$ est un symbole binaire associatif et commutatif.

$$t_1 \equiv 0 + (0 + (0 + (0 + 0))) >_{rpo} (0 + (0 + 0)) + (0 + (0 + 0)) \equiv t_2$$

puisque $0 + (0 + (0 + 0)) >_{rpo} 0 + (0 + 0)$ par la propriété de sous-terme et $\{0, 0 + (0 + (0 + 0))\} \gg_{rpo} \{0 + (0 + 0), 0 + (0 + 0)\}$ par définition de l'extension multi-ensemble.

Mais $t_2 =_{AC} 0 + t_1$ et on aurait donc aussi $t_2 >_{rpo} t_1$ si l'on avait la compatibilité, ce qui est absurde.

Une façon de résoudre (partiellement) cette difficulté, et qui a été proposé par d'innombrables auteurs, est d'"aplatir" les termes; les symboles associatifs et commutatifs sont alors considérés comme des symboles d'arité variable et leurs arguments sont regroupés suivant une règle $+(x, +(x_1, \ldots, x_n)) \rightarrow +(x, x_1, \ldots, x_n)$. (Ainsi que la règle symétrique). Par exemple, les termes t_1 et t_2 dans l'exemple 3.3.3 s'écriraient respectivement $t_1 = +(0,0,0,0,0)$ et $t_2 = +(0,0,0,0,0,0)$. Et dans ce cas on a $t_2 >_{rpo} t_1$ puisque le multi-ensemble $\{0,0,0,0,0,0\}$ est strictement plus grand que $\{0,0,0,0,0\}$. Malheureusement, cette solution est insuffisante car on perd alors la monotonie de $\geq_{rpo}$

Exemple 3.3.4 Supposons que $\mathcal{F}$ contient un symbole associatif et commutatif $+$ ainsi qu'une constante 0 et un symbole unaire g. Supposons $+ > g$. On a alors $+(0,0) >_{rpo} g(0)$ mais si l'on "ajoute 0" aux deux membres, on obtient à gauche $+(0,0,0)$ et à droite $+(0, g(0))$ or, par définition de l'extension multi-ensemble, le membre droit est supérieur au membre gauche.

Plusieurs ordres ont été proposés pour pallier ces inconvénients. Nous en citons deux

Definition 3.3.5 (Ordre associatif sur les chemins, [7]) *Soit $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_{AC}$ où les symboles de $\mathcal{F}_{AC}$ sont les symboles associatifs et commutatifs. On suppose que $\mathcal{F}$ est fini et muni d'un ordre tel que les symboles de $\mathcal{F}_{AC}$ soient inférieurs aux symboles de $\mathcal{F}_1$ et tel que l'on n'ait pas de chaîne $f_1 > f_2 > f_3$ avec $f_1, f_2, f_3 \in \mathcal{F}_{AC}$ (Cette condition est toujours vérifiée lorsqu'il y a au plus deux symboles associatifs et commutatifs).*

On note alors D l'ensemble des règles de distributivité:

$$f(x, g(y,z)) \rightarrow g(f(x,y), f(x,z)) \qquad \text{Si } f, g \in \mathcal{F}_{AC} \text{ et } f > g$$

Ces règles forment un système de réécriture convergent modulo les équations d'associativité et commutativité dont la relation de réduction associée est notée $\underset{D}{\longrightarrow}$. La forme irréductible et aplatie d'un terme t est notée $t\downarrow_D$.

$t \geq_{apo} u$ ssi $t\downarrow_D >_{rpo} u\downarrow_D$ ou bien ($t\downarrow_D =_{AC} u\downarrow_D$ et $t \geq_D u$.

Où $\geq_D$ est n'importe quel ordre compatible avec $=_{AC}$ et contenant $\underset{D}{\longrightarrow}$.

L'ordre associatif sur les chemins ainsi défini possède les bonnes propriétés: il est monotone, bien fondé et stable par substitution (voir [7, 46]).

Une généralisation (en ce sens qu'aucune restriction n'est imposée sur la préséance) est proposée dans [101]:

Definition 3.3.6 ([101]) *Soit $\mathcal{F}$ un ensemble fini de symboles de fonction et $\geq$ une relation d'ordre sur $\mathcal{F}$. Soit $I(t)$ la forme irréductible et aplatie de $t \in T(\mathcal{F})$ par rapport au système:*

$$f(t_1,\ldots,t_{i-1},g(u_1,\ldots,u_m),t_{i+1},\ldots,t_n) \to f(t_1,\ldots,t_{i-1},u_k,t_{i+1},\ldots,t_n)$$
$$\text{Si } f > g \text{ et } u_k = max\{u_1,\ldots,u_m\}$$
$$f(t_1,\ldots,t_{i-1},a,t_{i+1},\ldots,t_n) \to f(t_1,\ldots,t_{i-1},0,t_{i+1},\ldots,t_n)$$
$$\text{Si } f > a > 0, \text{ } a \text{ est une constante et } 0 \text{ est la plus petite constante}$$

On définit alors $s \geq_{acrpo} t$ ssi ou bien $I(s) >_{rpo} I(t)$ ou bien $I(s) = I(t)$ et $s \geq_{rpo} t$.

Cet ordre possède les bonnes propriétés (monotonie, sous-terme, bonne fondaison, compatibilité. Il est de plus total et peut être étendu aux termes avec variables. (Voir [101]).

3.4 Réécriture de formules

Dans le cadre de la résolution de contraintes par des méthodes syntaxiques, nous nous intéressons essentiellement à la réécriture de formules. Ces formules utilisent habituellement des connecteurs logiques comme $\vee, \wedge$ qui possèdent des propriétés d'associativité et de commutativité qui sont en général utilisées sans même être mentionnées. Cependant, pour ce qui concerne les preuves de terminaison, il faut faire attention à ces propriétés qui peuvent le cas échéant nuire à la terminaison. Bien sûr, on peut aussi utiliser une *représentation* des contraintes dans laquelle ce problème disparaît. Mais d'une part nous ne voulons pas nous compromettre avec une représentation en particulier (comme nous le verrons dans le paragraphe 3.6 la complexité de la résolution de contrainte dépend de la représentation choisie et selon les problèmes à traîter l'une ou l'autre représentation est mieux adaptée), d'autre part, c'est le propre des méthodes syntaxiques que de ne pas exploiter de représentation particulière.

Il nous faut ainsi préciser un peu plus de quelle réécriture de termes nous parlons quand nous réécrivons des contraintes.

3.4.1 L'algèbre des formules

Nous définissons ainsi l'algèbre des formules du premier ordre sur un alphabet de symboles de prédicats $\mathcal{P}$ et un alphabet de symboles de fonction $\mathcal{F}$ comme une algèbre quotient. Les termes sont formés à partir des formules atomiques en utilisant (en totalité ou en partie) les connecteurs logiques $\top, \bot, \wedge, \vee, \neg$ ainsi que les quantificateurs $\exists x$ et $\forall x$. (Qui constituent en fait un alphabet infini de symboles unaires, autant qu'il y a de variables.[1])

Les identités et règles de la figure 3.1 sont des transformations valides (l'équivalence entre les deux membres est une tautologie). De plus, l'ensemble de règles $\mathcal{S}_1$ est convergent modulo les identités. L'ensemble $\mathcal{S}_2$ n'est pas quant à lui convegent, les deux règles de distrbutivité conduisant à la non terminaison. Cependant, nous les donnons toutes deux car, ,suivant les contraintes à résoudre, on peut préférer une forme conjonctive ou une forme disjonctive. De même, les règles de mise n forme prénexe peuvent ne pas être souhaitées.

De façon générale, dans la suite, nous supposerons toujours les formules normalisées pour le système $\mathcal{S}_1$: la transformation de formules a lieu sur les formes normales pour ce système. Le cas échéant, selon le système de contraintes, les règles de $\mathcal{S}_1$ seront enrichies par certaines règles de $\mathcal{S}_2$ et le cas échéant par des règles ou identités supplémentaires. Par exemple, la symétrie de l'égalité $s = t \sim t = s$.

Les preuves de terminaison devront tenir compte de ces règles de normalisation appliquées de façon prioritaire. En général l'ordre utilisé est compatible avec les règles de normalisation. Il n'y a guère que les règles de distributivité (mise en forme normale conjonctive ou disjonctive) qui posent de vrais problèmes.

3.5 Programmation linéaire

Pour plus de détail consulter les ouvrages plus complets comme [112, 24].

3.5.1 Exemples

Commençons par une exemple. Supposons que monsieur Gérard, négociant en vins dispose de 180 000FF pour investir dans l'achat de vin sur

1. Attention: les "variables" des formules sont des constantes pour l'algèbre des formules. Les variables de l'algèbre des formules (souvent appelées *variables logiques*) représentant des termes ou des formules, selon leur sorte.

Identités

$$
\begin{array}{rclcrcl}
(P \wedge Q) \wedge R & \sim & P \wedge (Q \wedge R) & \qquad & P \wedge Q & \sim & Q \wedge P \\
(P \vee Q) \vee R & \sim & P \vee (Q \vee R) & & P \vee Q & \sim & Q \vee P \\
\exists x, \exists y\ P & \sim & \exists y, \exists x\ P & & \forall x, \forall y\ P & \sim & \forall y, \forall x\ P
\end{array}
$$

Premier ensemble de règles $\mathcal{S}_1$

$$
\begin{array}{rclcrcl}
\neg \bot & \to & \top & \qquad & \neg\top & \to & \bot \\
\top \wedge P & \to & P & & \bot \wedge P & \to & \bot \\
\top \vee P & \to & \top & & \bot \vee P & \to & P \\
P \wedge P & \to & P & & P \vee P & \to & P \\
\exists x, \exists x\ P & \to & \exists x\ P & & \forall x, \forall x\ P & \to & \forall x\ P \\
\neg(P \wedge Q) & \to & \neg P \vee \neg Q & & \neg(P \vee Q) & \to & \neg P \wedge \neg Q \\
\neg(\exists x\ P) & \to & \forall x\ \neg P & & \neg(\forall x\ P) & \to & \exists x\ \neg P \\
\neg\neg P & \to & P & & & &
\end{array}
$$

Deuxième ensemble de règles $\mathcal{S}_2$

$$
\begin{array}{rclcrcl}
P \wedge (Q \vee R) & \to & (P \wedge Q) \vee (P \wedge R) & & & & \\
P \vee (Q \wedge R) & \to & (P \vee Q) \wedge (P \vee R) & & & & \\
\exists x\ \phi[\exists y\ \psi]_u & \to & \exists x, \exists y\ \phi[\psi]_u & \qquad & \forall x\ \phi[\forall y\ \psi]_u & \to & \forall x, \forall y\ \phi[\psi]_u
\end{array}
$$

Pour ces deux dernières règles, on suppose qu'il n'y a aucun symbole $\forall$ (pour la première règle) ou aucun symbole $\exists$ (pour la deuxième règle) et aucun symbole $\neg$ (pour les deux règles) le long du chemin u.
Ces deux règles supposent de plus que y n'apparait pas libre dans $\phi[\top]_u$.

FIG. 3.1 - *Règles et identités de l'algèbre des formules*

10 ans. (On suppose que les commandes s'effectuent toutes simultément) Il dispose de 3 possibilités:

1. acheter du vin bon marché (4 FF le litre) qu'il doit conserver deux ans avant de pouvoir le revendre 9 francs le litre

2. acheter du vin d'excellente qualité 40FF le litre, qu'il doit garder 10 ans avant de le revendre 80FF le litre

3. du vin de qualité intermédiaire à 30FF le litre, qu'il doit garder 5 ans avant de le revendre 55 FF le litre.

Monsieur Gérard dispose d'une capacité limitée: il ne peut stocker que 5000 litres en moyenne par année.

Comment obtenir un bénéfice maximum?

Soit x la quantité de vin de basse qualité achetée au cours des 10 années, y celle de vin excellent et z celle de vin de moyenne qualité. Les contraintes financières et d'espace peuvent s'exprimer:

$$\begin{cases} 4x + 40y + 30z \leq 180000 \\ 2x + 10y + 5z \leq 50000 \end{cases}$$

La deuxième inégalité exprime le produit du nombre de litres stockés par le nombre d'années.

Il faut maximiser le profit, c'est à dire

$$P = 5x + 40y + 25z$$

en supposant bien sûr que $x, y, z \geq 0$.

Dans ce cas précis, on peut obtenir une solution en réécrivant les contraintes sous la forme:

$$\begin{cases} 4x + 40y + 30z + z_1 = 180000 \\ 2x + 10y + 5z + z_2 = 50000 \end{cases}$$

où z_1 et z_2 sont des *variables d'écart* représentant respectivement le capital non utilisé et l'espace non utilisé.

On peut résoudre le système par rapport à x et y, ce qui conduit à

$$\begin{cases} x = 5000 + 2.5z + 0.25z_1 - z_2 \\ y = 4000 - z - 0.05z_1 + 0.1z_2 \end{cases}$$

En reportant dans le profit, on trouve:

$$P = 185000 - 0.75z_1 - z_2 - 2.5z$$

Il est alors évident que P est maximal lorsque $z_1 = z_2 = 0$: ce qui conduit à un profit de 185000FF en achetant 5 000 litres de vin de basse qualité et 4000 litres de vin d'excellente qualité (aucun de moyenne qualité).

Mais pourquoi avoir choisi de résoudre par rapport à x et y? Si l'on choisit par exemple x et z on obtient:

$$\begin{cases} x = 15000 - 2.5y - z_1 - 0.75z_2 \\ z = 4000 - y - 0.05z_1 + 0.1z_2 \end{cases}$$

En reportant dans P on trouve:

$$P = 175000 + 12.5y - 6.25z_1 - 1.25z_2$$

Le coefficient de y étant positif, on ne peut pas conclure comme précédemment; il faut réitéerer.

D'une façon générale, les formules seront de la forme

$$S(\vec{x}) \wedge (\forall \vec{y}. S(\vec{y}) \Rightarrow P(\vec{y}) \leq P(\vec{x}))$$

où S est un système d'(in)équations linéaires en rationnels de variables libres $\vec{x}$. P est une combinaison linéaire des variables $\vec{x}$ (Fonction de profit). On suppose $\vec{b} \geq 0$.

En pratique, $S(\vec{x})$ s'écrira $A\vec{x} \leq \vec{b}$ où A est une matrice à n lignes et m colonnes et $P(\vec{x})$ s'écrira $\vec{p} \cdot \vec{x}$ où $\vec{p}$ est un vecteur à m lignes.

On peut tout d'abord se ramener à des égalités à l'aide de variables d'écart; la formule est transformée en:

$$M\vec{z} = \vec{b} \wedge \vec{z} \geq 0 \wedge (\forall \vec{y}. M\vec{y} = \vec{b} \Rightarrow \vec{q} \cdot \vec{z} \geq \vec{q} \cdot \vec{y})$$

où M est la matrice A complétée en ajoutant n colonnes qui correspondent à la matrice identité $n \times n$, $\vec{z}$ est le vecteur $\vec{x}$ complété par n variables d'écart et $\vec{q}$ est le vecteur $\vec{p}$ complété par n 0. Dans notre exemple cidessus,

$$M = \begin{pmatrix} 4 & 40 & 30 & 1 & 0 \\ 2 & 10 & 5 & 0 & 1 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 180000 \\ 50000 \end{pmatrix} \quad \vec{q} = \begin{pmatrix} 5 \\ 40 \\ 25 \\ 0 \\ 0 \end{pmatrix}$$

Pour plus de simplicité, on écrit aussi

$$\mathrm{Max}(\vec{q} \cdot \vec{z})$$

la formule

$$\forall \vec{y}. M\vec{y} = \vec{b} \Rightarrow \vec{q} \cdot \vec{z} \geq \vec{q} \cdot \vec{y}$$

3.5.2 Syntaxe et Interprétation

Les formules de notre système de contraintes s'écriront

$$MX = B \wedge \mathrm{Max}(P \cdot X + K)$$

où

- M est une matrice à coefficients dans les nombres rationnels, à n lignes et m colonnes, $m \geq n$ et contenant une sous-matrice identité $n \times n$. (Disons, la sous-matrice des colonnes d'indice $i \in I$ où I est un sous-ensemble à n éléments de $\{1, ..., m\}$)
- B est un vecteur à n lignes de nombres rationnels positifs ou nuls.
- P est un vecteur à m lignes de nombres rationnels dont les composantes d'indice $i \in I$ sont nulles.
- K est un nombre rationnel positif ou nul

Les formules sont interprétées dans les nombres rationnels positifs: $X = A$ est une solution de la formule ci-dessus lorsque

- $A \geq 0$
- $MA = B$
- pour tout vecteur positif A' tel que $MA' = B$, le produit scalaire $P \cdot A' \leq P \cdot A$. Dans ce cas, $P \cdot A + K$ est appelé *profit maximal*.

3.5.3 Formes résolues et règles de transformation

Les formes résolues sont $\perp$ ou bien des formules

$$MX = B \wedge \mathrm{Max}(P \cdot X + K)$$

comme ci-dessus où, de plus, $P \leq 0$.

Remarquons alors que, si I est un ensemble de n indices tels que la sous-matrice de M formée des colonnes d'indice $i \in I$ est l'identité, alors le vecteur A dont les composantes d'indice $i \notin I$ sont 0 et tel que $MA = B$ est une solution de la contrainte ci-dessus et K est le profit maximal (tout autre solution conduirait à une somme $P \cdot X + K$ inférieure). Nous avons donc la propriété fondamentale des formes résolues:

Lemme 3.5.1 *Les formes résolues différentes de $\perp$ ont toujours une solution.*

Maintenant, il nous faut construire un système de règles qui permet d'obtenir ces formes résolues:

$$MX = B \wedge \mathrm{Max}(P \cdot X + K) \quad \rightarrow \quad \perp$$
$$\text{Si } \exists i. p_i > 0 \text{ et } \forall j. m_{j,i} < 0$$

$$MX = B \wedge \mathrm{Max}(P \cdot X + K) \quad \rightarrow \quad M'X = B' \wedge \mathrm{Max}(P' \cdot X + K')$$
$$\text{Si } \exists i, p_i > 0 \text{ et } \exists j. \tfrac{b_j}{m_{j,i}} = \min\{\tfrac{b_k}{m_{k,i}} \mid m_{k,i} > 0\}$$
$$\text{et } M' \text{ est la matrice } m'_{k,p} \text{ telle que:}$$
$$m'_{j,p} = \tfrac{m_{j,p}}{m_{j,i}} \text{ et } m'_{k,p} = m_{k,p} - \tfrac{m_{k,i}}{m_{j,i}} m_{j,p} \text{ si } k \neq j\ .$$
$$B' \text{ est défini par } b'_j = \tfrac{b_j}{m_{j,i}} \text{ et } b'_k = b_k - \tfrac{m_{k,i}}{m_{j,i}} b_j \text{ si } k \neq j.$$
$$P' \text{ est défini par } p'_k = p_k - \tfrac{p_i}{m_{j,i}} m_{j,k}.$$
$$K' = K + \tfrac{p_i}{m_{j,i}} b_j$$

Exemple 3.5.2 Voyons comment ces règles fonctionnent sur notre exemple:

1. choisir un indice i de P tel que $p_i > 0$. Si aucun choix n'est possible, la solution courante est optimale.

   Dans notre exemple, on peut choisir $i = 1$.

2. choisir un indice j tel que $m_{j,i} > 0$ et $b_j/m_{j,i} = \min\{b_k/m_{k,i} \mid m_{k,i} > 0\}$. Si aucun choix n'est possible, le profit peut être rendu aussi grand que l'on veut: il n'y a pas de solution.

   Dans notre exemple, $j = 2$.

3. remplacer le problème en substituant M' à M, P' à P, B' à B.

M' est la matrice $m'_{k,p}$ telle que: $m'_{j,p} = m_{j,p}/m_{j,i}$ et $m'_{k,p} = m_{k,p} - \frac{m_{k,i}}{m_{j,i}} m_{j,p}$ si $k \neq j$.

B' est défini par $b'_j = b_j/m_{j,i}$ et $b'_k = b_k - \frac{m_{k,i}}{m_{j,i}} b_j$ si $k \neq j$

P' est défini par $p'_k = p_k - \frac{p_i}{m_{j,i}} m_{j,k}$.

Dans notre exemple, on obtient le nouveau problème:

$$M' = \begin{pmatrix} 0 & 20 & 20 & 1 & -2 \\ 1 & 5 & 2.5 & 0 & 0.5 \end{pmatrix} \quad B' = \begin{pmatrix} 80000 \\ 25000 \end{pmatrix} \quad P' = \begin{pmatrix} 0 \\ 15 \\ 12.5 \\ 0 \\ -2.5 \end{pmatrix}$$

À l'étape suivante, sur notre exemple, on choisirait $i = 3$, $j = 1$, ce qui conduit à:

$$M_2 = \begin{pmatrix} 0 & 1 & 1 & 0.05 & -0.1 \\ 1 & 2.5 & 0 & -0.125 & 0.75 \end{pmatrix} \quad B_2 = \begin{pmatrix} 4000 \\ 15000 \end{pmatrix} \quad P_2 = \begin{pmatrix} 0 \\ 2.5 \\ 0 \\ -0.625 \\ -1.25 \end{pmatrix}$$

puis à l'étape suivante, on choisit $i = 2$ (c'est la seule possibilité), $j = 2$

$$M_3 = \begin{pmatrix} 0 & 1 & 1 & 0.05 & -0.1 \\ 1 & 0 & -2.5 & -0.25 & 1 \end{pmatrix} \quad B_3 = \begin{pmatrix} 4000 \\ 5000 \end{pmatrix} \quad P_3 = \begin{pmatrix} 0 \\ 0 \\ -2.5 \\ -0.75 \\ -1 \end{pmatrix}$$

$P_3 \leq 0$ et on s'arrête donc avec un système en forme résolue.

Théorème 3.5.3 *Le système de transformation de formules défini ci-dessus est correct: il transforme un problème en un problème équivalent.*

<u>Preuve:</u>

Il nous faut montrer tout d'abord que les conditions imposées à M, B, P sont toujours satisfaites par M', B', P' lors d'une transformation du problème.

- On peut vérifier que la matrice M' contient une sous-matrice identité $n \times n$: celle qui correspond aux indices de I moins l'indice du vecteur colonne dont la $j$ème ligne est 1 (c'est la variable "sortante") et plus la colonne i (c'est la variable "entrante").

- $B' \geq 0$. En effet, $b'_k = b_k - \frac{m_{k,i}}{m_{j,i}} b_j$ mais, par définition, $b_j \leq m_{j,i} \frac{b_k}{m_{k,i}}$ dès que $m_{k,i} > 0$ (et puisque $m_{j,i} > 0$). Donc $b_k \geq 0$.

- Les composantes d'indice $k \in I'$ de P sont nulles; il suffit de le vérifier pour $i \in I'$ (seul indice nouveau dans I'). $p'_i = p_i - p_i \frac{m_{j,i}}{m_{j,i}} = 0$ par définition.

- $K' \geq K$ car $p_j > 0$.

Maintenant, il nous faut montrer d'une part que si la première règle s'applique, la contrainte n'a pas de solution et que, d'autre part, si la deuxième règle s'applique, elle transforme bien une contrainte en une contrainte qui a même ensemble de solutions (et même profit maximal).

Concernant la première règle: si $p_i > 0$ et $\forall j . m_{j,i} \leq 0$, soit $\lambda > 0$ arbitraire. On définit A^λ par: $a_i^\lambda = \lambda$, $a_k^\lambda = 0$ si $k \notin I$ et, si k est l'indice de I tel que $m_{j,k} = 1$, $a_k^\lambda = b_j - \lambda m_{j,i}$. On a alors $MA^\lambda = C$ où $c_j = m_{j,i} a_i^\lambda + m_{j,k} a_k^\lambda$ si $m_{j,k} = 1$. Soit $c_j = \lambda m_{j,i} + b_j - \lambda m_{j,i} = b_j$. Donc $MA^\lambda = B$. D'autre part, $A^\lambda \geq 0$ puisque $m_{j,i} \leq 0$. Enfin, $P \cdot A^\lambda = p_i \lambda$. Il en résulte que le problème admet des solutions arbitrairement grandes: quel que soit K, il existe un $A^\lambda \geq 0$ tel que $MA^\lambda = B$ et $P \cdot A^\lambda > K$; la contrainte n'a donc pas de solution.

Concernant la deuxième règle, il s'agit essentiellement d'exprimer la variable x_i ($i$ème composante de X) en fonction des autres, en utilisant la $j$ème équation. Puis de remplacer x_i par le résultat dans les autres équations. L'expression de ce "pivotagë se traduit par la règle en question qui est donc correcte.

□

Théorème 3.5.4 *Le système de transformation de formules défini ci-dessus se termine toujours.*

Exemple 3.5.5 [d'après [24]] Donnons un autre exemple de l'utilisation de la méthode. Une entreprise de charcuterie peut produire jusqu'à 480 saucissons, 400 jambons et 250 rotis chque jour. Chacun de ces produits peut être vendu frais ou fumé. Le nombre total maximal de saucissons, jambons et rôtis qui peuvent être fumés durant une journée normale de travail est 450. Mais, en plus, jusqu'à 280 produits peuvent être fumés durant des heures supplémentaires rémunérées au prix fort. Les profits de la compagnie s'établissent comme suit:

	Produit frais	Fumé durant les heures règlementaires	Fumé en heures complémentaires
Jambons	40 F	80 F	50 F
Saucissons	20 F	30 F	10 F
Rôtis	30 F	60 F	40 F

Comment réaliser le profit maximal?

Soient $x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3$ respectivement les quantités de chacun de ces produits confectionnés quotidiennement. On

$$\left\{\begin{array}{ll} x_1 + x_2 + x_3 & \leq 400 \\ y_1 + y_2 + y_3 & \leq 480 \\ z_1 + z_2 + z_3 & \leq 250 \\ x_2 + y_2 + z_2 & \leq 450 \\ x_3 + y_3 + z_3 & \leq 280 \end{array}\right.$$
$$\wedge \mathrm{Max}(40x_1 + 80x_2 + 50x_3 + 20y_1 + 30y_2 + 10y_3 + 30z_1 + 60z_2 + 40z_3)$$

Après introduction des variables d'écart, on trouve successivement les problèmes:

$$P = \begin{pmatrix} 40 & 80 & 50 & 20 & 30 & 10 & 30 & 60 & 40 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad K = \quad 0$$

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 400 \\ 480 \\ 250 \\ 450 \\ 280 \end{pmatrix}$$

$j = 1, i = 2.$

$$P_1 = \begin{pmatrix} -40 & 0 & -30 & 20 & 30 & 10 & 30 & 60 & 40 & -80 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad K_1 = \quad 32000$$

$$M_1 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad B_1 = \begin{pmatrix} 400 \\ 480 \\ 250 \\ 50 \\ 280 \end{pmatrix}$$

$i = 8, j = 4$

$$P_2 = \begin{pmatrix} 20 & 0 & 30 & 20 & -30 & 10 & 30 & 0 & 40 & -20 & 0 & 0 & -60 & 0 \end{pmatrix} \qquad K_2 = 35000$$

$$M_2 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & -1 & 0 \\ -1 & 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad B_2 = \begin{pmatrix} 400 \\ 480 \\ 200 \\ 50 \\ 280 \end{pmatrix}$$

$i = 3, j = 3$

$$P_3 = \begin{pmatrix} -10 & 0 & 0 & 20 & 0 & 10 & 0 & 0 & 10 & -50 & 0 & -30 & -30 & 0 \end{pmatrix} \qquad K_3 = 41000$$

$$M_3 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & -1 & 0 & -1 & 1 & 1 \end{pmatrix} \qquad B_3 = \begin{pmatrix} 200 \\ 480 \\ 200 \\ 250 \\ 80 \end{pmatrix}$$

$i = 4, j = 2$

$$P_4 = \begin{pmatrix} -10 & 0 & 0 & 0 & -20 & -10 & 0 & 0 & 10 & -50 & -20 & -30 & -30 & 0 \end{pmatrix} \qquad K_4 = 50600$$

$$M_4 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & -1 & 0 & -1 & 1 & 1 \end{pmatrix} \qquad B_4 = \begin{pmatrix} 200 \\ 480 \\ 200 \\ 250 \\ 80 \end{pmatrix}$$

$i = 9, j = 3$

$$P_5 = \begin{pmatrix} -20 & 0 & -10 & 0 & -10 & -10 & -10 & 0 & 0 & -60 & -20 & -40 & -20 & 0 \end{pmatrix} \qquad K_5 = 52600$$

$$M_5 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & -1 & 0 \\ -1 & 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & -1 & 0 & -1 & 1 & 1 \end{pmatrix} \qquad B_1 = \begin{pmatrix} 400 \\ 480 \\ 200 \\ 50 \\ 80 \end{pmatrix}$$

Ce qui conduit à $x_2 = 400, y_1 = 480, z_2 = 50, z_3 = 200, e_5 = 80$, toutes les autres variables étant nulles.

3.6 Unification standard

L'unification dans la théorie vide, appelée aussi *unification standard* ou *unification syntaxique* est le type de contrainte symbolique le plus utilisé. En substance, le mécanisme apparaissait déjà dans la thèse de Herbrand [66], mais le terme unification à été forgé par Robinson [109]. Avant de décrire formellement ce qu'est un problème d'unification et comment le résoudre, illustrons son utilité par deux exemples.

De $A \vee C$ et $\neg A \vee D$, on peut inférer la formule $C \vee D$. Considérons à présent les deux clauses

$$P(s_1, \ldots, s_n) \vee C \quad \text{et} \quad \neg P(t_1, \ldots, t_n) \vee D$$

où toutes les variables sont (implicitement) universellement quantifiées. Supposons qu'une substitution σ soit telle que pour $1 \leq k \leq n$ on ait $s_k\sigma \equiv t_k\sigma$. (La substitution σ est un unificateur de s_k et t_k.) Alors, des formules

$$P(s_1\sigma, \ldots, s_n\sigma) \vee C\sigma \quad \text{et} \quad \neg P(t_1\sigma, \ldots, t_n\sigma) \vee D\sigma$$

on pourra, comme précédemment inférer la formule

$$C\sigma \vee D\sigma$$

La fameuse règle de *résolution* de Robinson [109] consiste précisément à faire une telle inférence en utilisant une substitution σ qui soit la plus générale possible (dans un sens que nous allons définir au prochain paragraphe).

Un deuxième exemple est le calcul de paires critiques dans la complétion de Knuth et Bendix [82]. Considérons les deux axiomes équationnels

$$(x + y) + z = x + (y + z) \quad \text{et} \quad u + i(u) = e$$

(valides dans la théorie des groupes, par exemple).
La substitution $\sigma = \{x \mapsto v, y \mapsto i(v), u \mapsto v\}$ rend les deux termes $x + y$ et $u + i(u)$ égaux à $v + i(v)$. Alors le terme $((x + y) + z)\sigma \equiv (v + i(v)) + z$ est égal à $v + (i(v) + z)$ par le premier axiome et à $e + z$ par le deuxième. On peut donc inférer que l'identité $v + (i(v) + z = e + z$ est conséquence des deux axiomes de départ. C'est essentiellement le mécanisme du calcul des *paires critiques*. De nouveau, on souhaite une substitution σ qui soit la plus générale possible (et non pas inférer, par exemple $a + (i(a) + z = e + z$, $b + (i(b) + z = e + z$,...). D'où la notion d'*unificateur principal* ou *plus général unificateur*.

3.6.1 Définitions

En un premier temps, nous nous restreignons au cas des arbres finis. La classe d'algorithmes que nous définissons s'étendra facilement au cas des arbres rationnels.

Definition 3.6.1 (Problèmes d'unification, syntaxe) *Les problèmes d'unification sont définis récursivement comme suit:*

- $\top$ *et* $\bot$ *sont des problèmes d'unification.*
- *Une équation* $s = t$ *avec* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *est un problème d'unification.*
- *Si* P_1 *et* P_2 *sont des problèmes d'unification, alors* $P_1 \wedge P_2$, $P_1 \vee P_2$ *et* $(\exists x)P_1$ *sont des problèmes d'unification.*

Comme d'habitude, l'algèbre des formules n'est pas une algèbre libre. Les formules sont considérées modulo

- la commutativité de $=$,
- l'associativité-commutativité de $\wedge$ et $\vee$,
- distributivité de $\wedge$ par rapport à $\vee$,
- l'équivalence $(\exists x)P \sim (\exists y)P\{x \mapsto y\}$ si x est libre dans P et y n'apparaît pas dans P,
- les règles de réécriture $P \wedge \top \to P$, $P \vee \top \to \top$, $P \wedge \bot \to \bot$, $P \vee \bot \to P$.

Le domaine d'interprétation des formules étant l'algèbre des termes elle-même, les solutions sont des substitutions :

Definition 3.6.2 (Problèmes d'unification, sémantique) *Les solutions d'une équation* $s = t$ *sont les substitutions* σ *telles que* $\mathcal{T}(\mathcal{F}, \mathcal{X}) \models s\sigma = t\sigma$ *c'est-à-dire* $s\sigma \equiv t\sigma$. *Toute substitution est solution de* $\top$ *et* $\bot$ *n'a pas de solution. La définition s'étend aux problèmes d'unification de manière standard. Les solutions d'un problème d'unification s'appellent aussi* unificateurs. *Etant donné un problème d'unification* P, *un* unificateur principal *ou* plus général unificateur *de* P *(souvent noté* mgu *pour* most general unifier*) est un unificateur* σ *tel que pour tout unificateur* θ *il existe une substitution* ρ *telle que* $\theta = \sigma\rho$.

Dans la pratique, il suffira de résoudre des problèmes de la forme

$$(\exists \vec{y}) s_1 = t_1 \ \wedge \ \cdots \ \wedge \ s_n = t_n$$

Plutôt que calculer un unificateur principal, nous allons définir des formes résolues. Une *forme résolue* d'un problème d'unification est juste un unificateur écrit sous forme d'une conjonction d'équations.

Definition 3.6.3 (Formes résolues) *Un problème d'unification P est en* forme résolue *si $P \equiv \top$ ou $P \equiv \bot$ ou*

$$P \equiv (\exists y_1, \ldots, y_p) \ \ x_1 = t_1 \ \wedge \ \cdots \ \wedge \ x_n = t_n$$

avec

- $x_i \notin \{y_1, \ldots, y_p\}$ *pour* $1 \leq i \leq n$
- $\forall i, j \in \{1, \ldots, n\} \quad i \neq j \Rightarrow x_i \neq x_j$
- $\forall i, j \in \{1, \ldots, n\} \quad x_i \notin V(t_j)$

Lemme 3.6.4 *Soit $P \equiv (\exists y_1, \ldots, y_p) \quad x_1 = t_1 \ \wedge \ \cdots \ \wedge \ x_n = t_n$ un problème d'unification en forme résolue. Alors $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ est un unificateur principal de P.*

On peut donc considérer un algorithme d'unification comme un processus qui transforme un problème d'unification en un problème équivalent en forme résolue. De manière générale, un algorithme d'unification sera donné dans ce chapitre par un ensemble de règles de transformation muni d'un contrôle. On essaiera de donner des contrôles aussi laxistes que possible afin d'obtenir non pas un algorithme, mais une classe d'algorithmes.

3.6.2 Unification dans les arbres finis

Depuis l'algorithme de Robinson, plusieurs algorithmes ont été proposés, tous basés sur les mêmes mécanismes et différant essentiellement par le contrôle avec lequel ces mécanismes sont utilisés (par exemple l'algorithme de Robinson n'est pas donné sous forme d'un ensemble de règles), et par les structures de données adoptées. Martelli et Montanari utilisent des multi-équations [95], Corbin et Bidoit [41] reprennent l'algorithme de Robinson (qui est exponentiel) et le rendent efficace en faisant du partage. Paterson et Wegman donnent un algorithme qui est linéaire si le problème de départ

Trivial
$(\exists \vec{y})\ s = s\ \wedge\ P \quad \rightarrow (\exists \vec{y})\ P$

Decompose
$(\exists \vec{y}) f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \wedge P \ \rightarrow\ (\exists \vec{y}) s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge P$

Clash
$(\exists \vec{y})\ f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)\ \wedge\ P \quad \rightarrow\ \perp$
si $f \neq g$

Merge
$(\exists \vec{y})\ x = s\ \wedge\ x = t\ \wedge\ P \quad \rightarrow (\exists \vec{y})\ x = s\ \wedge\ s = t\ \wedge\ P$
si $s, t \notin \mathcal{X}$ et $|s| \leq |t|$

Coalesce (Remplacement de variable)
$(\exists \vec{y})\ x = y\ \wedge\ P \quad \rightarrow (\exists \vec{y})\ x = y\ \wedge\ P\{x \mapsto y\}$
si $x, y \in V(P)$ et x est libre ou y est liée

Eliminate (Remplacement)
$(\exists \vec{y})\ x = t\ \wedge\ P \quad \rightarrow (\exists \vec{y})\ x = t\ \wedge\ P\{x \mapsto t\}$
si $x \in \mathcal{X}$ et $t \notin \mathcal{X}$ et $x \in V(P) \setminus V(t)$

EQE (Elimination des quantificateurs existensiels)
$(\exists x, \vec{y})\ x = s\ \wedge\ P \quad \rightarrow (\exists \vec{y})\ P$
si $x \notin V(s) \cup V(P)$

$(\exists x, \vec{y})\ P \quad \rightarrow (\exists \vec{y})\ P$
si $x \notin V(P)$

Check* (Test d'occurrence)
$(\exists x, \vec{y})\ x_1 = s_1[x_2]\ \wedge\ x_2 = s_2[x_3]\ \wedge\ \cdots\ \wedge\ x_n = s_n[x_1]\ \wedge\ P \quad \rightarrow\ \perp$
si un $s_i \notin \mathcal{X}$

FIG. 3.2 - *Les règles pour l'unification standard dans les arbres finis.*

a déjà ses variables partagées (deux occurrences d'une même variable sont le même objet physique).

L'ensemble de règles de la figure 3.2 permet de résoudre des problèmes d'unification standard. Volontairement, nous avons donné un ensemble de règles qui n'est pas minimal, c'est-à-dire dont un sous-ensemble strict peut toujours être un algorithme d'unification.

Théorème 3.6.5 *Les règles de la figure 3.2 préservent les ensembles de solutions et terminent pour tout problème d'unification. Les problèmes irreductibles sont des formes résolues.*

La correction des règles est laissée en exercice. On vérifie aisément qu'un problème d'unification n'est pas en forme résolue si et seulement si une règle peut s'appliquer. Pour prouver la terminaison, nous définissons une mesure sur les problèmes d'unification (à laquelle est associé un ordre bien fondé) et nous montrons que chaque application d'une règle fait décroître la mesure.

Definition 3.6.6 *Soit* $P \equiv (\exists \vec{y}) \;\; s_t = t_1 \;\wedge\; \cdots \;\wedge\; s_n = t_n$ *un problème d'unification. Une variable* x *est* résolue *dans* P *si elle n'a qu'une occurrence dans une équation* $x = s$ *de* P. $VNR(P)$ *est le nombre de variables non résolues dans* P.
$Tmax(P)$ *est le multiensemble* $\{max(|s_1|, |t_1|), \ldots, max(|s_n|, |t_n|)\}$.
Une équation $s_i = t_i$ *est quasi-résolue dans* P *si* s_i *ou* t_i *est une variable.*
$EQR(P)$ *est le nombre d'équations quasi-résolues de* P.
$NBV(P)$ *est le nombre de variables de* P.
Le poids *de* P *est le quadruplet* $(VNR(P), Tmax(P), EQR(P), NBV(P))$.
L'ordre associé est $\geq_W = (\geq, \geq_{mul}, \geq, \geq)_{lex}$, *où* $\geq$ *est l'ordre usuel sur les naturels.*

L'ordre $\geq_W$ est bien fondé par construction, étant obtenu en utilisant les extensions multi-ensemble et lexicographique à partir de l'ordre $\geq$.

La preuve de terminaison est résumée dans le tableau suivant qui donne la variation des composantes de la mesure $W(P)$ par l'application des règles.

	$VNR(P)$	$Tmax(P)$	$EQR(P)$	$NBV(P)$
Trivial	$\downarrow$			
Decompose	$=\downarrow$	$\downarrow$		
Clash	$\downarrow$			
Merge	$=\downarrow$	$=$	$\downarrow$	
Coalesce	$\downarrow$			
Eliminate	$\downarrow$			
EQE	$\downarrow$ $=$	 $=$	 $=$	 $\downarrow$
Check*	$\downarrow$			

3.6.3 Formes séquentiellement résolues

Une forme résolue (représentant un unificateur principal) peut être de taille exponentielle par rapport au problème de départ. La notion de *forme séquentiellement résolue* s'impose naturellement, si l'on veut éviter cet inconvénient en faisant du partage.

Definition 3.6.7 (Formes séquentiellement résolues) *Un problème d'unification P est en* forme résolue *si $P \equiv \top$ ou $P \equiv \bot$ ou*

$$P \equiv (\exists y_1, \dots, y_p)\ x_1 = t_1 \ \wedge\ \cdots\ \wedge\ x_n = t_n$$

avec

- $\forall i, j \in \{1, \dots, n\} \quad i \neq j \Rightarrow x_i \neq x_j$
- $i \leq j \Rightarrow x_i \notin V(t_j)$
- $x_i \in \{y_1, \dots, y_p\} \Rightarrow \exists j < i$ *tel que* $x_i \in V(t_j)$
- $t_i \in \mathcal{X} \Rightarrow x_i \notin \{y_1, \dots, y_p\}$ *et x_i n'apparaît nulle part ailleurs dans P.*

Théorème 3.6.8 *Les règles de la figure 3.2 privées de la règle **Eliminate** terminent pour tout problème d'unification. Les problèmes irréductibles sont des formes séquentiellement résolues.*

Il est évident de calculer une forme résolue à partir d'une forme séquentiellement résolue. Il suffit de faire du remplacement, et d'enlever les variables quantifiées existensiellement lorsqu'elles deviennent inutiles :

Lemme 3.6.9 *Soit P un problème en forme séquentiellement résolue auquel ni la règle **Eliminate**, ni la règle **EQE** ne s'appliquent. Alors, P est en forme résolue.*

3.6.4 Arbres rationnels

Rappelons que par définition, les arbres rationnels sont des arbres, peut-être infinis, mais qui n'admettent qu'un nombre fini de sous-arbres distincts. Les arbres rationnels infinis sont de la forme $t[t|_p]_{p.q}$. Mais une équation de la forme $s = t[t|_p]_{p.q}$ peut toujours se transformer en un problème équivalent $(\exists x)s = t[x]_{p.q} \ \wedge \ x = t|_p[x]_q$. On peut donc écrire les problèmes d'unification dans les arbres rationnels avec la même syntaxe que les problèmes d'unification standard.

Seuls vont différer la notion de solution et de forme résolue.

Definition 3.6.10 (Unification dans $\mathcal{RT}(\mathcal{F},\mathcal{X})$, sémantique) *Une $\mathcal{RT}$-solution de $s = t$ est une substitution sigma de $\mathcal{RT}(\mathcal{F},\mathcal{X})$ telle que $\mathcal{RT}(\mathcal{F},\mathcal{X}) \models s\sigma = t\sigma$. Cette définition s'étend aux problèmes d'unification de manière standard.*

Definition 3.6.11 ($\mathcal{RT}$-formes résolues) *Un problème d'unification P est en $\mathcal{RT}$-forme résolue si*

$P \equiv \top$ ou $P \equiv \bot$ ou

$$P \equiv (\exists y_1, \ldots, y_p) \ \ x_1 = t_1 \ \wedge \ \cdots \ \wedge \ x_n = t_n$$

avec

- $\forall i, j \in \{1, \ldots, n\} \quad i \neq j \Rightarrow x_i \neq x_j$
- $x_i \notin \{y_1, \ldots, y_p\}$ *pour* $1 \leq i \leq n$.

Lemme 3.6.12 *Tout problème d'unification en $\mathcal{RT}$-forme résolue autre que $\bot$ admet une solution dans $\mathcal{RT}(\mathcal{F},\mathcal{X})$.*

Théorème 3.6.13 *Les règles de la figure 3.2 privées des règles **Check*** terminent pour tout problème d'unification. Les problèmes irréductibles sont des $\mathcal{RT}$-formes résolues.*

3.7 Unification dans les théories équationnelles

En présence d'axiomes tels que la commutativité, la réécriture ne termine pas. Au lieu de considérer des règles de réécriture comme $x + y \to y + x$, on considère les classes d'équilvalence de la théorie équationnelle engendrée, (c'est-à-dire que l'on ne distinguera pas $a+b$ de $b+a$), au prix de tenir compte des propriétés modulo lesquelles on travaille au cours de la déduction. D'où la notion d'*unification modulo une théorie équationnelle* [105].

L'unification modulo une théorie équationnelle est indécidable en général, mais il y a des résultats de décidabilité pour de nombreuses théories. Ce que l'on aimerait faire (dans cette partie) c'est d'appliquer une méthode générale pour l'unification modulo E qui ne considère que les axiomes d'une présentation de E sans se soucier des structures algébriques sous-jacentes. C'est le problème de la *E-unification générale.* Bien sur, on ne dispose dans le cas général que d'une procédure de semi-décision.

Un autre problème est que la notion d'unificateur principal doit être adaptée. En effet, il n'y a pas, en général de solution unique qui permette d'obtenir toutes les autres par instanciation. Il faut donc recourir à des *ensembles complets de E-unificateurs.*

3.7.1 Definitions

On rappelle que la théorie équationnelle $=_E$ engendrée (présentée) par un ensemble d'équations E est la plus petite congruence sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ contenant toutes les instances des équations de E. Nous utiliserons la définition équivalente suivante:

Definition 3.7.1 (Théorie équationnelle) *Soit $E = \{l_1 = r_1, \ldots, l_n = r_n\}$ un ensemble d'équations (axiomes), avec $l_i, r_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. La théorie équationnelle $=_E$ est la cloture réflexive, symétrique transitive $\overset{*}{\leftrightarrow}_E$ de la relation $\leftrightarrow_E$ définie par $s \leftrightarrow_E t$ si il existe un axiome $l = r$ de E et une substitution θ tels que $s|_p \equiv l\theta$ et $t \equiv s[r\theta]_p$. E est appelé une* présentation *de la théorie $=_E$.*

E sera dite cohérente *si $\mathcal{T}(\mathcal{F}, \mathcal{X}) / =_E$ n'est pas un singleton.*

Par abus de langage, on parlera souvent de la théorie E pour désigner la théorie présentée par un ensemble E d'axiomes[2].

Théorème 3.7.2 (Birkhoff [10]) *Soit $E = \{l_1 = r_1, \ldots, l_n = r_n\}$ un ensemble d'équations avec $l_i, r_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. La théorie équationnelle $=_E$ est l'ensemble des identités qui sont valides dans la classe des modèles de E.*

Comme $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{T}(\mathcal{F}, \mathcal{X})/{=_E}$ est muni d'un préordre de subsomption :

Definition 3.7.3 *Un terme s est* plus général modulo E *qu'un terme t, ce que l'on notera $s \leq\!\!\bullet_E t$ si il existe une substitution θ telle que $s\theta =_E t$. t est appelé une* E-instance *ou* instance modulo E *de t. Cette définition s'étend de manière évidente aux substitutions. L'équivalence associée à $\leq\!\!\bullet_E$ est la* E-similarité, *notée $\stackrel{\bullet}{=}_E$.*

Contrairement à $\leq\!\!\bullet$, $\leq\!\!\bullet_E$ n'est pas toujours bien fondé:

Exemple 3.7.4 Soit $E = \{f(a(a(x)), s(y)) = f(a(x), y)\}$. On a la chaîne infinie d'inégalités:

$$f(a(x), 0) \bullet\!\!> f(a(x), s(0)) \bullet\!\!> f(a(x), s(s(0))) \cdots$$

où les termes sont deux à deux différents modulo $=_E$, mais où chaque terme est E-instance du précédent par la substitution $\{x \mapsto a(x)\}$.

La syntaxe des problèmes d'unifications modulo E est la même que pour l'unification standard (définition 3.6.1), mais les solutions ne sont pas les mêmes :

Definition 3.7.5 (Problèmes d'unification modulo E, sémantique) *Un E-unificateur d'un problème d'unification P est une substitution σ telle que $P\sigma$ est valide dans la classe des modèles de E. De manière équivalente (par le théorème de Birkhoff), un E-unificateur de $s = t$ est une substitution σ telle que $s\sigma =_E t\sigma$, définition qui s'étend de manière standard aux problèmes d'unification.*

Contrairement à l'unification syntaxique, il n'y a pas une unique substitution qui engendre toutes les autres par instanciation.

Exemple 3.7.6 Considérons la commutativité $C = \{x + y = y + x\}$. Les deux seules solutions de l'équation $x + y = a + b$ (où a et b sont des constantes libres) sont $\{x \mapsto a, y \mapsto b\}$ et $\{x \mapsto b, y \mapsto a\}$.

2. Ce qui n'est pas correct, car en général la relation $=_{E_1 \cup E_2}$ est différente de $=_{E_1} \cup =_{E_2}$.

Definition 3.7.7 *Un* ensemble complet de E-unificateurs, *que l'on note souvent $CSU_E(P)$ pour* complete set of unifiers, *d'un problème d'unification P est un ensemble de substitutions Σ tel que*

- *Tout $\sigma \in \Sigma$ est un E-unificateur de P,*
- *pour tout E-unificateur θ de P, il existe $\sigma \in \Sigma$ tel que $\sigma \mathrel{\leq\!\bullet}_E \theta$.*

Si de plus

- *Σ ne contient pas deux substitutions comparables pour $\mathrel{\leq\!\bullet}_E$,*

alors Σ est appelé un ensemble minimal complet *de E-unificateurs. Si $CSU_E(P)$ est un singleton, son unique élément σ est appelé E-*unificateur principal *de P (noté $mgu_E(P)$). Une théorie E où tous les problèmes d'unification résolubles ont un E-unificateur principal est dite* unitaire*. Si les ensembles complets d'unificateurs existent toujours et sont toujours finis (resp. peuvent être infinis), E est dite* finitaire *(resp.* infinitaire*). Une théorie qui n'admet pas toujours d'ensemble complet d'unificateurs est dite* nullaire.

Théorème 3.7.8 *Lorsqu'il existe, l'ensemble minimal complet de E-unificateurs d'un problème d'unification est unique modulo $\stackrel{\bullet}{=}_E$.*

<u>Preuve:</u>
Supposons qu'un problème d'unification ait deux ensembles minimaux, complets de E-unificateurs Σ_1 et Σ_2. La relation $\mathrel{\leq\!\bullet}_E$ établit deux surjections ϕ_{12} et ϕ_{21} de Σ_1 dans Σ_2 et de Σ_2 dans Σ_1 respectivement, car ce sont des ensembles *complets* de E-unificateurs. Supposons que pour $\sigma_1 \in \Sigma_1$ et $\sigma_2 \in \Sigma 2$, l'on ait $\phi_{12}(\sigma_1) = \sigma_2$ et $\phi_{21}(\sigma_2) = \sigma_1' \neq \sigma_1$. Alors, on a $\sigma_1 \mathrel{\leq\!\bullet}_E \sigma_1'$, ce qui contredit la minimalité de Σ_1. Donc $\sigma_1 \stackrel{\bullet}{=}_E \sigma_2$. □

Comme d'habitude, on cherche des formes résolues:

Definition 3.7.9 *Un* ensemble complet de formes résolues *de P est un ensemble $\{P_1, \ldots, P_n\}$ de formes résolues tel que P et $P_1 \ \vee \ \cdots \ \vee \ P_n$ ont le même ensemble de solutions.*

Cette définition est justifiée par le lemme suivant:

Lemme 3.7.10 *Soit E une théorie équationnelle cohérente et*

$$P \equiv (\exists y_1, \ldots, y_p) \ x_1 = t_1 \ \wedge \ \cdots \ \wedge \ x_n = t_n$$

un problème en forme résolue (au sens de l'unification standard). Alors la substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ *est un* E*-unificateur principal de* P*.*

Preuve:
Appliquer σ à P rend les deux membres de chaque équation identiques. L'existence d'un terme pour chaque variable existensiellement quantifiée est vérifiée en prenant cette même variable. Donc σ est un E-unificateur. Soit θ un E-unificateur de P. Pour $1 \leq i \leq n$, on a $x_i\sigma\theta \equiv t_i\theta =_E x_i\theta$, et pour chaque variable libre z des t_i, on a $z\sigma\theta \equiv z\theta$. Donc θ est E-instance de σ par elle-même. □

3.7.2 E-unification générale : le narrowing

Le problème de décider l'existence d'un E-unificateur est trivialement semi-décidable. En effet, $=_E$ étant semi-décidable, il suffira d'énumérer les substitutions (modulo la similarité) tout en faisant marcher en parallèle l'algorithme de semi-décision jusqu'à trouver (si elle existe) une E-solution. Cette approche n'est évidemment pas satisfaisante, et nous allons donner dans ce paragraphe des méthodes moins brutales.

La *surréduction* ou *narrowing* est une méthode générale de semi-décision de la E-unification dans une théorie E présentée par un système de réécriture convergent R. Le narrowing a été défini par Slagle [114], puis étudié, entre autres par Fay et Hullot [55, 70].

La règle de narrowing sécrit de la manière suivante:

Narrow
$s = t \quad \rightsquigarrow \ (s[r]_p = t)\theta$
si $l \rightarrow r \in R$ et θ est l'unificateur principal de $s|_p$ et l.

la substitution θ est appelée *substitution surréductrice*, et on pourra noter $s = t \overset{\theta}{\rightsquigarrow} s' = t'$. Une séquence de surréductions est appelée *surdérivation.*

Théorème 3.7.11 (Hullot [70]) *Soit* E *une théorie équationnelle admettant un système de réécriture convergent* R*. Pour tout* E*-unificateur* σ *de* $s = t$ *il existe une surdérivation*

$$s = t \overset{\theta_1}{\rightsquigarrow} s_1 = t_1 \cdots s_{n-1} = t_{n-1} \overset{\theta_n}{\rightsquigarrow} s_n = t_n$$

avec s_n *et* t_n *unifiables par* θ *et telle que* $\theta_1\theta_2 \cdots \theta_n\theta \trianglelefteq_E \sigma$*.*

Hullot prouve un résultat plus fort, à savoir qu'il n'est pas nécessaire d'appliquer la règle de surréduction à des positions dans les termes qui ont été introduites par les substitutions surréductrices. Il définit la *surréduction basique* en se restreignant aux *surdérivations basées sur un ensemble d'occurrences*, et obtient le même résultat de complétude. Les concepts manipulés sont intuitivement faciles, mais les preuves sont techniques, en particulier parce qu'il faut gérér les ensembles de positions auxquelles on s'autorise une étape de surréduction.

Nous allons introduire un formalisme adapté au narrowing, c'est-à-dire considérer des équations modulo E contraintes par des problèmes d'unification standard. Avec ce formalisme, on obtient une preuve simple d'un résultat plus fort : la complétude du narrowing basique pour les stratégies ascendantes.

Definition 3.7.12 (Problème de narrowing, syntaxe) *Un* problème de narrowing *est une formule contrainte $s =_E t \mid C$ où $s =_E t$ est une équation, et C un problème d'unification.*

On utilise $=_E$ pour mettre en avant qu'il s'agit d'une équation (contrainte) à résoudre dans la théorie E (contrairement à la contrainte C).

Definition 3.7.13 (Problème de narrowing, sémantique) *L'ensemble des solutions de $s =_E t \mid C$ est l'ensemble des substitutions $\theta\sigma$ où θ est un unificateur de C et σ est un E-unificateur de $s\theta =_E t\theta$.*

Pour résoudre une équation $s =_E t$, on peut partir du problème de narrowing $s =_E t \mid \top$ qui a par définition les mêmes solutions que $s =_E t$. Notre but est de montrer que pour chaque E-solution σ de $s =_E t$, il existe une réduction par les règles de la figure 3.3 aboutissant à $\top \mid C$ où σ est un unificateur de C.

On montre la complétude des règles pour le narrowing basique, pour les dérivations ascendantes gauches.

Théorème 3.7.14 (Complétude du narrowing basique) *Soit E une théorie équationnelle qui admet un système de réécriture convergent R. Soit σ un E-unificateur (en forme normale pour R) de $s = t$. Alors il existe une dérivation ascendante gauche*

$$s =_E t \mid \top \overset{*}{\rightsquigarrow} \top \mid C$$

Narrow
$s =_E t \mid C \quad \rightsquigarrow \quad s[r]_p = t \mid C \ \wedge \ s|_p = l$
si $l \to r \in R$ et p est une position non variable de s.

Blocking
$s =_E t \mid C \quad \rightsquigarrow \quad \top \mid C \ \wedge \ s = t$

Fail
$s =_E t \mid C \quad \rightsquigarrow \quad \bot$
si C est insatisfaisable.

FIG. 3.3 - *Les règles de la surréduction basique*

avec les règles de la figure 3.3 telle que σ soit un unificateur de

$$(\exists \mathcal{X} \setminus V(s = t)) \ \ C$$

Preuve:
Soit σ un E-unificateur de $s = t$ en forme normale pour R. Considérons une preuve par réécriture ascendante gauche

$$s\sigma \xrightarrow{*}_R u \xleftarrow{*}_R t\sigma$$

(qui existe, par hypothèse car R est convergent et $s\sigma =_E t\sigma$).
Montrons, par récurrence sur la longueur de la preuve que

$$s =_E t \mid \top \overset{*}{\rightsquigarrow} \top \mid C$$

où σ est un unificateur de $(\exists \mathcal{X} \setminus V(s = t)) \ \ C$.
Cas de base. Si la preuve est de longueur 0, en appliquant la règle **Blocking** à $s =_E t \mid \top$, on obtient $\top \mid s = t$, dont σ est un unificateur (car $s\sigma = t\sigma$).
Cas général. Supposons, par exemple, que $s\sigma$ se réécrive à la position p, par règle $l \to r$ avec la substitution θ. Les variables de s et l étant disjointes, on peut écrire

$$s\sigma \to s[r]_p\sigma\theta$$

où p est une position non-variable de s, σ est normalisée, et $l\theta$ est normalisé car la stratégie est ascendante. On a donc une preuve par réécriture

$$s[r]_p\sigma\theta \xrightarrow{*}_R u \xleftarrow{*}_R t\sigma = t\sigma\theta$$

Par l'hypothèse de récurrence, on a une surdérivation

$$s[r]_p = t \mid \top \overset{*}{\rightsquigarrow} \top \mid C_1$$

où $\sigma\theta$ est un unificateur de $(\exists \mathcal{X} \setminus (V(s[r]_p = t)) \quad C_1$. Mais par la règle, **Narrow**, on a $s = t \mid \top \rightsquigarrow s[r]_p = t \mid s|_p = l$, et donc

$$s = t \mid \top \overset{*}{\rightsquigarrow} \top \mid s|_p = l \ \wedge \ C_1$$

avec $\sigma\theta$ unificateur de $(\exists \mathcal{X} \setminus V(s[r]_p = t)) \quad s|_p = l \ \wedge \ C_1$ (car $l\theta = s|_p$ et $Dom(\theta) = V(l)$.) Mais alors σ est solution de $(\exists \mathcal{X} \setminus V(s = t)) \quad s|_p = l \wedge C_1$, car les variables du domaine de θ sont toutes quantifiées existensiellement. $\square$

3.7.3 Théories syntaxiques

Les *théories syntaxiques* ont été définies en 1985 par Kirchner comme les théories équationnelles sans effondrement[3] qui admettent une *présentation résolvante finie*, c'est-à-dire une présentation finie avec laquelle toutes les preuves peuvent se faire avec au plus une application d'un axiome à la racine [77, 78]. A cette époque, la communauté de réécriture n'était pas consciente du fait qu'il existe des quantités de théories syntaxiques autres que la commutativité et ses avatars.

L'avantage majeur des théories syntaxiques est que si l'on connaît une présentation résolvante finie, on peut deviner (de manière non déterministe) quel axiome pourra être appliqué à la racine. Cela fournit des stratégies descendantes pour résoudre le problème du mot ou des problèmes d'unification qui sont complètes, mais qui ne terminent pas en général. Un cas simple de théorie syntaxique est la commutativité $C(+)$ présentée par $\{x+y = y+x\}$. Il suffira d'appliquer la règle suivante en place de **Decompose** lorsque le

3. Une théorie est sans effondrement si elle n'admet pas de théorème équationnel de la forme $x = s$ avec $x \in \mathcal{X}$ et $s \neq x$.

symbole de tête est commutatif:

C-Mutate
$s_1 + s_2 = t_1 + t_2 \quad \rightarrow \ (s_1 = t_1 \ \wedge \ s_2 = t_2) \ \vee \ (s_1 = t_2 \ \wedge \ s_2 = t_1)$
si $+$ est un symbole commutatif.

La règle est évidemment correcte, et l'algorithme obtenu termine comme pour l'unification standard. Il suffit, pour une disjonction de prendre comme mesure le multiensemble des mesures (pour l'unification standard) des problèmes de la disjonction.

En 1990, il est apparu que de nombreuses théories intéressantes sont syntaxiques. Kirchner et Klay ont remarqué qu'une théorie sans effondrement E sur une signature $\mathcal{F}$ est syntaxique si et seulement si toutes les *équations générales* de la forme $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)$ avec $f, g \in \mathcal{F}$ ont un ensemble complet de E-unificateurs fini $\Sigma_{f,g}$ [80]. Dans ce cas, les équations $f(x_1, \ldots, x_n)\sigma = g(y_1, \ldots, y_m)\sigma$ où $\sigma \in \Sigma_{f,g}$ forment une présentation résolvante de E. En particulier, les théories associatives-commutatives étant à unification finitaire, ce sont des théories syntaxiques. Ce fait est établi indépendamment par Nipkow qui montre la syntaxicité des théories associatives-commutatives de manière élégante par une interprétation géométrique.

Ceci a suscité un intérêt croissant pour les théories syntaxiques, mais Klay a montré qu'il est indécidable si une théorie est syntaxique, et que le problème du mot est indécidable en général dans les théories syntaxiques [81].

Une classe importante de théories pour lesquelles l'unification peut être décidée par des moyens syntaxiques a été caractérisée par Comon, Jouannaud et Haberstrau [33]. Ils abandonnent l'hypothèse que les théories sont sans effondrement et introduisent la notion de théorie *cycle-syntaxique* pour résoudre les cycles $x = u[x]$. Les *théories minces*, c'est-à-dire les théories qui ont une présentation où toutes les variables sont à profondeur au plus 1 sont syntaxiques et cycle-syntaxiques. Le quotient de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ par une théorie mince a une théorie du premier ordre décidable.

La méthode syntaxique a été utilisée par Arnborg et Tidén pour l'unification modulo la distributivité gauche [121]. Dans ce travail les théories syntaxiques ne sont pas mentionnées, mais il est en fait prouvé que la présentation réduite à l'axiome de distributivité est résolvante, et (résultat plus difficile) que le processus d'unification termine dans ce cas particulier.

L'associativité-commutativité est certainement la théorie pour laquelle l'unification a été le plus extensivement étudiée [90, 115, 77, 54, 56, 67, 23, 19, 2, 76, 51], mais dans ce cas, la méthode syntaxique standard ne termine pas. Boudet et Contejean donnent cependant un critère syntaxique permettant de couper l'espace de recherche tout en préservant la complétude et en assurant la terminaison [18]. Cependant, il est fait appel à des arguments sémantiques, utilisant des propriétés des solutions minimales d'une équation diophantienne linéaire pour établir ce résultat.

Definition 3.7.15 *[77, 78] Soit $\approx$ une théorie équationnelle sans effondrement (c'est-à-dire une théorie qui ne contient pas $x \approx u[x]_p$ avec $x \in \mathcal{X}$ et $p \neq \Lambda$). La théorie $\approx$ est* syntaxique *si elle admet une* présentation résolvante finie *E, c'est-à-dire une présentation où toute égalité $s =_E t$ a une preuve*

$$s \xleftrightarrow{p_1}_E s_1 \xleftrightarrow{p_2}_E \cdots \longleftrightarrow_E s_n \xleftrightarrow{p_n}_E t$$

où au plus un p_i est Λ.

Il se trouve que AC, dans $\mathcal{T}(\{+\}, \mathcal{X})$, présentée par les deux axiomes

$$\begin{array}{rcll} (x+y)+z & = & x+(y+z) & \quad \text{A} \\ x+y & = & y+x & \quad \text{C} \end{array}$$

est syntaxique, mais la présentation ci-dessus n'est pas résolvante. En effet, deux applications d'un axiome à la racine sont nécessaires pour prouver l'égalité $(x+y)+(z+t) =_{AC} (x+z)+(y+t)$. Les cinq axiomes suivants forment une présentation résolvante de AC, qui peut être obtenue à partir des AC-unificateurs principaux de $x+y = u+v$ [80, 57], ou par une analyse des échanges possibles au cours d'une preuve entre les sous-termes gauche et droit de la racine [103] :

$$\begin{array}{rcll} (x+y)+z & = & x+(y+z) & \quad \text{A} \\ x+y & = & y+x & \quad \text{C} \\ (x+y)+z & = & (x+z)+y & \quad \text{RC} \\ x+(y+z) & = & y+(x+z) & \quad \text{LC} \\ (x+y)+(z+t) & = & (x+z)+(y+t) & \quad \text{MC} \end{array}$$

Comme deux termes de $\mathcal{T}(\{+\}, \mathcal{X})$ sont égaux modulo AC si et seulement si les multiensembles de leurs feuilles (variables) sont égaux, on peut donner l'interprétation intuitive suivante de ces axiomes, dans l'esprit de Nipkow [103].

Pour que $s_1 + s_2$ et $t_1 + t_2$ soient égaux modulo AC, un des cas suivants doit se présenter:

- $s_1 =_{AC} t_1$ et $s_2 =_{AC} t_2$. Dans ce cas, il n'y a pas besoin d'appliquer un axiome à la racine.

- Le multiensemble des feuilles de t_1 est strictement plus grand que celui de s_1. Dans ce cas, le seul axiome que l'on aura à appliquer à la racine est l'*associativité* (A), de gauche à droite. Dans le cas symétrique, l'axiome sera utilisé de droite à gauche.

- Les multiensembles des feuilles de s_1 et t_2 sont égaux ainsi que ceux de s_2 et t_1. Dans ce cas, il suffira d'appliquer une fois la *commutativité* (C) à la racine.

- Les multiensembles des feuilles de s_2 et t_2 sont disjoints. Dans ce cas, s_2 pourra être échangé avec un sous-terme de s_1 en utilisant la *commutativité droite* (RC).

- Dans le cas symétrique (les multiensembles des feuilles de s_1 et t_1 sont disjoints), il suffira d'appliquer une fois à la racine la *commutativité gauche* (LC).

- Finalement, si les intersections des multiensembles des feuilles de s_i et t_j sont tous incomparables pour $1 \leq i, j \leq 2$, la preuve pourra se faire avec une seule étape à la racine avec l'axiome de *commutativité médiane* (MC).

En fait, toutes les théories sans effondrement finitaires pour l'unification sont syntaxiques. Mieux, il suffit que toutes les *équations générales* de la forme $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)$ aient des ensembles complets finis de E-unificateurs. Par exemple, l'associativité est infinitaire pour l'unification, mais la présentation réduite à l'axiome d'associativité est résolvante et l'équation $x + y = z + u$ a les trois A-unificateurs principaux:

$$\{x \mapsto v_1, y \mapsto v_2, z \mapsto v_1, u \mapsto v_2\}$$

$$\{x \mapsto v_1 + v_2, y \mapsto v_3, z \mapsto v_1, u \mapsto v_2 + v_3\}$$

$$\{x \mapsto v_1, y \mapsto v_2 + v_3, z \mapsto v_1 + v_2, u \mapsto v_3\}$$

Théorème 3.7.16 ([80]) *Soit E une théorie équationnelle sans effondrement sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ telle que toutes les équations $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)$, pour $f, g \in \mathcal{F}$ ont un ensemble complet fini de E-unificateurs Σ_{fg}. Les axiomes de la forme $f(x_1, \ldots, x_n)\sigma = g(y_1, \ldots, y_m)\sigma$ où $\sigma \in \Sigma_{fg}$ forment une présentation résolvante de E.*

Nous allons donner une règle générale de *mutation* pour les théories syntaxiques (tout en prouvant le théorème ci-dessus).

Considerons l'équation

$$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$$

à résoudre modulo E. Supposons que E ait des ensembles finis de E-unificateurs Σ_{fg} pour toutes les équations $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)$ avec $f, g \in \mathcal{F}$. Si l'on abstrait les sous-termes immédiatement sous la racine par des variables dans les deux membres de léquation, on obtient le problème (évidemment équivalent):

$$(\exists \vec{x}, \vec{y}) f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m) \bigwedge_{i=1}^{n} x_i = s_i \bigwedge_{j=1}^{m} y_j = t_j$$

Où $\vec{x}, \vec{y}$ représentent les nouvelles variables introduites par l'abstraction. on peut résoudre l'équation générale $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)$ et l'on obtient la disjonction finie :

$$\bigvee_{\sigma \in \Sigma_{f,g}} ((\exists \vec{x}, \vec{y}, \vec{z}) \bigwedge_{i=1}^{n} x_i = x_i\sigma \bigwedge_{j=1}^{m} y_j = y_j\sigma \bigwedge_{i=1}^{n} x_i = s_i \bigwedge_{j=1}^{m} y_j = t_j)$$

où $\vec{z}$ représente les variables introduites par σ. Après des applications répétées de la règle **Eliminate**, on obtient :

$$\bigvee_{\sigma \in \Sigma_{f,g}} ((\exists \vec{x}, \vec{y}, \vec{z}) \bigwedge_{i=1}^{n} x_i = x_i\sigma \bigwedge_{j=1}^{m} y_j = y_j\sigma \bigwedge_{i=1}^{n} x_i\sigma = s_i \bigwedge_{j=1}^{m} y_j\sigma = t_j)$$

Maintenant, les variables quantifiées existensiellement qui n'apparaissent que comme membre gauche d'une équation sont inutiles, et la règle **EQE** s'applique, jusqu'à obtenir le problème suivant qui est E-équivalent au problème de départ :

$$\bigvee_{\sigma \in \Sigma_{f,g}} ((\exists \vec{z}) \bigwedge_{i=1}^{n} x_i\sigma = s_i \bigwedge_{j=1}^{m} y_j\sigma = t_j)$$

D'où la règle de mutation pour les théories qui ont des ensembles complets d'unificateurs pour les équations générales :

Mutate

$$f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m) \wedge P \to \bigvee_{\sigma\in\Sigma_{f,g}} ((\exists\vec{z}) \bigwedge_{i=1}^{n} x_i\sigma = s_i \bigwedge_{j=1}^{m} y_j\sigma = t_j \wedge P)$$

où $\Sigma_{f,g}$ est un ensemble complet de E-unificateurs de l'équation générale $f(x_1,\ldots,x_n) = g(y_1,\ldots,y_m)$, et les variables de $\vec{z}$ sont les nouvelles variables introduites par σ.

Dans le cas de l'unification modulo AC, cette règle permet de transformer l'équation $s_1 + s_2 = t_1 + t_2$ en la disjonction équivalente des 7 problèmes :

$(\exists v_1, v_2)$	$s_1 = v_1 \wedge s_2 = v_2 \wedge t_1 = v_1 \wedge t_2 = v_2$	(Id)
$(\exists v_1, v_2)$	$s_1 = v_1 \wedge s_2 = v_2 \wedge t_1 = v_2 \wedge t_2 = v_1$	(C)
$(\exists v_1, v_2, v_3)$	$s_1 = v_1 + v_2 \wedge s_2 = v_3 \wedge t_1 = v_1 \wedge t_2 = v_2 + v_3$	($A_\rightarrow$)
$(\exists v_1, v_2, v_3)$	$s_1 = v_1 \wedge s_2 = v_2 + v_3 \wedge t_1 = v_1 + v_2 \wedge t_2 = v_3$	($A_\leftarrow$)
$(\exists v_1, v_2, v_3)$	$s_1 = v_1 + v_2 \wedge s_2 = v_3 \wedge t_1 = v_1 + v_3 \wedge t_2 = v_2$	(RC)
$(\exists v_1, v_2, v_3)$	$s_1 = v_1 \wedge s_2 = v_2 + v_3 \wedge t_1 = v_2 \wedge t_2 = v_1 + v_3$	(LC)
$(\exists v_1, v_2, v_3, v_4)$	$s_1 = v_1 + v_2 \wedge s_2 = v_3 + v_4 \wedge t_1 = v_1 + v_3 \wedge t_2 = v_2 + v_4$	(MC)

On aura reconnu l'usage des 5 axiomes de la présentation résolvante de AC, pour des paramodulations immédiatement suivies de décompositions (licites grâce à la syntaxicité). Le premier problème correspond au cas où il n'y a pas besoin d'appliquer un axiome à la racine, et l'associativité peut s'appliquer dans les deux sens.

Bien sur, **Mutate** n'est pas suffisante pour calculer une (disjonction de) forme(s) résolue(s). L'ensemble de règles suivant est complet pour les théories syntaxiques simples c'est-à-dire n'admettant pas d'égalité $u =_E s[u]_p$ avec $p \neq \Lambda$.

Lemme 3.7.17 *Les règles de la figure 3.4 préservent les ensembles de solutions pour l'unification dans les théories syntaxiques simples. Les problèmes irréductibles sont des formes résolues.*

Malheuresement, ces règles ne terminent pas en général.

Exemple 3.7.18 Considérons le problème de AC-unification $x+x = y+y$. **Mutate** retourne (parmi d'autres) le problème obtenu par application de la

Mutate

$$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m) \wedge P \rightarrow \bigvee_{\sigma \in \Sigma_{f,g}} ((\exists \vec{z}) \bigwedge_{i=1}^{n} x_i\sigma = s_i \bigwedge_{j=1}^{m} y_j\sigma = t_j \wedge P)$$

où $\Sigma_{f,g}$ est un ensemble complet de E-unificateurs de l'équation générale $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)$, et les variables de $\vec{z}$ sont les nouvelles variables introduites par σ.

Merge

$x = s \ \wedge \ x = t \quad \rightarrow \ x = s \ \wedge \ s = t$

si $x \in \mathcal{X}$ et $s, t \notin \mathcal{X}$

Coalesce

$(\exists z_1, \ldots, z_n) \quad x = y \ \wedge \ P \quad \rightarrow \ (\exists z_1, \ldots, z_n) \quad x = y \ \wedge \ P\{x \mapsto y\}$

if $x, y \in V(P)$ et x est quantifiée existensiellement ou y est libre.

Eliminate

$(\exists y_1, \ldots, y_n) \ x = s \ \wedge \ P \quad \rightarrow \ (\exists y_1, \ldots, y_n) \ x = s \ \wedge \ P\{x \mapsto s\}$

si $x \in V(P)$, et $s \notin \mathcal{X}$, et $x \notin V(s)$ ou

si $s \in \mathcal{X}$, et $s \not\equiv x$ et $x, s \in V(P)$.

Check*

$x_1 = t_1[x_2]_{p_1} \ \wedge \ x_2 = t_2[x_3]_{p_2} \ \wedge \ \cdots \ \wedge \ x_n = t_n[x_1]_{p_n} \quad \rightarrow \ F$

si un $p_i \neq \Lambda$.

EQE

$(\exists x, y_1, \ldots, y_n) \ x = s \ \wedge \ P \quad \rightarrow \ (\exists y_1, \ldots, y_n) \ P$

si $x \notin V(s) \cup V(P)$.

FIG. 3.4 - *L'ensemble de règles pour l'unification dans les théories syntaxiques simples*

commutativité médiane :

$$(\exists v_1, v_2, v_3, v_4) \;\; x = v_1 + v_2 \;\wedge\; x = v_3 + v_4 \;\wedge\; y = v_1 + v_3 \;\wedge\; y = v_2 + v_4$$

Merge, appliqué à x fabrique l'équation $v_1 + v_2 = v_3 + v_4$. Une forme résolue de cette équation est $v_1 = v_3 \;\wedge\; v_2 = v_4$. Maintenant, **Coalesce** pourra fabriquer les équations $y = v_1 + v_1$ et $y = v_2 + v_2$. **Merge**, appliqué à ces deux équations fabrique un renommage de l'équation de départ.

D'un autre côté, la méthode syntaxique termine pour la théorie de la distributivité gauche. Bien que les auteurs ne parlent pas explicitement de théories syntaxiques, nous pouvons reformuler un résultat d'Arnborg et Tidén :

Théorème 3.7.19 (Arnborg-Tidén [121]) *La présentation réduite à l'axiome $x * (y + z) = (x * y) + (x * z)$ est résolvante. La méthode syntaxique termine pour l'unification modulo la distributivité gauche.*

Une classe de théories équationnelles pour lesquelles l'approche syntaxique marche toujours est celle des *théories minces*, c'est-à-dire des théories qui admettent une présentation où toutes les variables sont à profondeur au plus 1. Le théorème suivant à été établi récemment, en utilisant des méthodes syntaxiques :

Théorème 3.7.20 (Comon-Jouannaud-Haberstrau [34]) *La théorie du premier ordre du quotient $\mathcal{T}(\mathcal{F}, \mathcal{X})/ =_E$ où E est une théorie mince est décidable.*

Pour clore ce chapitre, nous montrons comment une approche syntaxique peut résoudre le problème de l'unification dans les anneaux booléens. Un anneau booléen est un anneau commutatif où l'addition est nilpotente ($x + x = 0$) et la multiplication idempotente ($x*x = x$). Chaque élément étant son propre inverse, on peut restreindre la signature à $\{+, *, 0, 1\}$. Par exemple, les ensembles, munis de l'intersection $(*)$ et de la différence symétrique $(+)$, forment un anneau booléen (0 est l'ensemble vide et 1 l'ensemble plein). Le calcul propositionnel est un autre exemple, en prenant le *et* pout $*$, le *ou exclusif* pour $+$ et les constantes *faux* et *vrai* pour 0 et 1.

Montrons comment résoudre des équations dans $\mathcal{T}(\{+, *, 0, 1\} \cup \mathcal{C}, \mathcal{X})$, où $\mathcal{C}$ est un ensemble de constantes libres, modulo la théorie des anneaux booléens. On illustrera notre propos en interprétant $*$ et $+$ comme l'intersection et la différence symétrique d'ensembles. Tout d'abord, on remarque

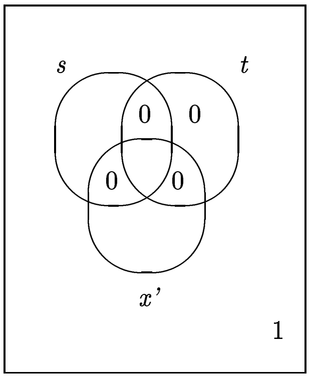
que résoudre $s = t$ revient à résoudre $s+t$=0. (Deux ensembles sont égaux si et seulement si leur différence symétrique est vide). On se restreint donc à la résolution d'équations de la forme $u = 0$. Il est facile de donner un système de réécriture convergent (modulo AC) pour les anneaux booléens pour lesquels les formes normales sont des sommes de monômes où chaque variable ou constante apparaît au plus une fois dans chaque monôme (idempotence de $*$) et où chaque monôme apparaît au plus une fois dans la somme (nilpotence de $+$). (0 est la somme vide, et 1 le monôme vide.) Notons $u\downarrow$ la forme normale de u.

Pour résoudre $u = 0$, on envisage trois cas:

1. Si $u\downarrow = 0$ alors l'équation est trivialement satisfaite.
2. Si $u\downarrow$ est un terme clos différent de 0, alors l'équation est insatisfaisable.
3. Sinon, $u\downarrow$ peut s'écrire sous la forme

$$(x * m_1) + (x + m_2) + \cdots + (x * m_i) + m'_1 + m'_2 + \cdots + m'_j$$

où x n'apparaît pas dans $m_1, \ldots, m_i, m'1, \ldots, m'_j$. Dans ce cas, notre problème revient à résoudre $(x * s) + t = 0$ où x n'apparaît pas dans s ni dans t, ou, de manière équivalente, $x * s = t$. Exprimons la valeur de x en fonction de s, t et d'une nouvelle variable x' :



Les 0 sur la figure sont nécéssaires pour que l'on ait bien $x * s = t$. Donc, une solution doit vérifier

$$(\exists x')x = x' + (x' * s) + (x' * t)$$

Trivial
$u = 0 \quad \rightarrow \top$
si $u \downarrow= 0$

Fail
$u = 0 \quad \rightarrow \bot$
si $u \downarrow= u'$ où u' est un terme clos différent de 0

Bool-mutate
$u = 0 \quad \rightarrow (\exists x') \ \ x = x' + (x' * s) + t \ \wedge \ (s * t) + t = 0$
si $u \downarrow= (x * s) + t$ avec $x \notin V(s) \cup V(t)$

FIG. 3.5 - *L'ensemble de règles pour l'unification booléenne*

En effet, les parties de x' qui ne sont pas égales à 0 correspondent au terme $x' + (x' * s) + (x' * t)$. Il faut également que t soit égal à $x' * t$. Donc, une solution doit vérifier

$$(\exists x')x = x' + (x' * s) + t$$

Remplaçons x par $x' + (x' * s) + t$ dans $(x * s) + t = 0$. On obtient

$$(x' * s) + (x' * s) + (t * s) + t = 0$$

qui est équivalent à

$$(s * t) + t = 0$$

Le problème de départ est donc équivalent à

$$(\exists x') \ \ x = x' + (x' * s) + t \ \wedge \ (s * t) + t = 0$$

D'où les règles pour l'unification booléenne de la figure 3.5.

Les règles de la figure 3.5 terminent évidemment, car à chaque application de **Bool-mutate**, il y a une variable non résolue de moins.

3.8 Arithmétique de Presburger

Les contraintes arithmétiques sont parmi les plus utilisées dans les applications. Malheureusement (ou heureusement ?) l'indécidabilité de l'arithmétique de Peano (un cas particulier des théorèmes d'incomplétude de Gödel) interdit l'utilisation de toute l'arithmétique dont nous avons l'habitude comme système de contraintes. L'important dans ce résultat d'indécidabilité n'est pas l'utilisation des quantificateurs, mais plutôt l'introduction de la multiplication puisque l'existence de solutions en nombre entiers à des systèmes d'équations polynomiales est indécidable (c'est le dixième problème de Hilbert, auquel Y. Matjasevic a apporté la contribution finale en 1970. Voir [45] pour un exposé de synthèse).

Le système arithmétique le plus populaire du point de vue des contraintes est ainsi l'arithmétique de Presburger, dans laquelle seule l'addition est autorisée. Cette arithmétique a été prouvée décidable il y a longtemps [106], mais des sous-problèmes comme les systèmes d'équations linéaires en nombres entiers ont été étudiés même bien avant. Ce sont les *systèmes d'équations diophantiennes linéaires.* L'exemple SEND + MORE = MONEY de l'introduction est un problème d'équations diophantiennes linéaires. Voici un autre exemple (d'après un numéro de *Scientific American*, exemple aussi repris dans [39]):

Exemple 3.8.1 Un bateau fait naufrage et seuls cinq marins et un singe parviennent à réchapper de la catastrophe en prenant pied sur une île déserte. La principale ressource de l'île semble être les cocotiers. Les naufragés décident d'effectuer une grande récolte de noix de coco dès leur arrivée sur l'île. Ils travaillent toute la journée et parviennent à rassembler un tas de noix de coco, puis épuisés, ils se couchent en convenant de procéder au partage le lendemain. Durant la nuit un premier marin se réveille en songeant que le partage pourrait bien poser des problèmes au matin et décide de prendre sa part immédiatement. Il partage le tas en 5 parts égales. Il reste une noix de coco qu'il donne au singe. Puis il cache sa part et reforme le tas avec les quatre parts restantes et va se recoucher. Un peu plus tard dans la nuit, un deuxième marin se lève et effectue exactement le même raisonnement que le premier marin. Il partage le tas en 5 parties égales et il reste à nouveau une noix de coco qu'il donne au singe. Il prend sa part, la cache, reforme le tas et retourne se coucher. En fait, successivement, les cinq marins procèdent de la même façon, à chaque fois il reste une noix pour le singe. Bien sûr, entre temps le tas diminue mais la nuit empêche les marins de s'en apercevoir. Au matin, les marins procèdent au partage. Ils constatent que le tas a dminué, mais, comme ils sont tous coupables, personne ne proteste. Le tas est partagé en cinq parties égales. Il reste une noix pour le singe. La question est de savoir combien de noix de coco il y avait au départ.

Si x_0 est le nombre de noix de coco ramassées, le problème est de résoudre la contrainte:

$$\exists x_1, x_2, x_2, x_4, x_5, x_6 \left\{ \begin{array}{rcl} x_0 & = & 5x_1 + 1 \\ 4x_1 & = & 5x_2 + 1 \\ 4x_2 & = & 5x_3 + 1 \\ 4x_3 & = & 5x_4 + 1 \\ 4x_4 & = & 5x_5 + 1 \\ 4x_5 & = & 5x_6 + 1 \end{array} \right.$$

De façon annexe le nombre de noix que récupère chaque marin est respectivement $x_1 + x_6, x_2 + x_6, x_3 + x_6, x_4 + x_6, x_5 + x_6$.

Il existe de très nombreuses méthodes pour la résolution des systèmes diophantiens linéaires. Par exemple, en utilisant les séries génératrices [53], par des méthodes d'automate (cf. chapitre 4), par des méthodes ad hoc comme [39] ou encore par des techniques de récurrence [58]. Enfin, il y a la méthode originale de Presburger (déconseillée en pratique), qui est décrite ci-après et qui utilise, pour le cas général des formules du premier ordre, une technique d'*élimination de quantificateurs*. Afin d'introduire cette méthode dans un cas plus simple, nous commencerons par considérer la *théorie des ordres discrets* dans le paragraphe 3.8.1 où l'on ne dispose même pas de l'addition, mais seulement de l'ajout d'une unité (ce qui est insuffisant pour exprimer les équations diophantiennes). Ensuite, nous verrons la méthode générale de Presburger dans le paragraphe 3.8.2.

3.8.1 Théorie des ordres discrets

Dans un premier temps, montrons la méthode d'élimination des quatificateurs pour la théorie des ordres discrets. Plus précisément, considérons les formules du premier-ordre construites sur les formules atomiques $s > t$ et $s = t$ où $s, t \in T(\{S, 0\}, X)$, S étant interprété comme le successeur et 0 comme 0.

On suppose que les formules sont maintenues en forme normale (prénexe) disjonctive. On applique alors les règles de la figure 3.6.

Lemme 3.8.2 *Les règles de la figure 3.6 sont correctes.*

Preuve:
Les seules règles pour lesquelles ce n'est pas évident sont les deux dernières règles de la figure.

$$\begin{array}{rcll}
S(t) = 0 & \rightarrow & \perp & \\
S(t) > 0 & \rightarrow & \top & \\
S(t) < 0 & \rightarrow & \perp & \\
t = t & \rightarrow & \top & \\
t = S^k(t) & \rightarrow & \perp & \text{Si } k \neq 0 \\
t > S^k(t) & \rightarrow & \perp & \\
t < S^k(t) & \rightarrow & \top & \text{Si } k \neq 0
\end{array}$$

$$\exists x. S^{m+n+1}(x) \sim t \wedge S^m(x) \approx u \quad \rightarrow$$
$$\exists x. S^{m+n+1}(x) \sim t \wedge S^{m+n+1}(x) \approx S^{n+1}(u)$$
Si $\sim$ et $\approx$ sont l'un des symboles $=, >, <$

$$\exists x. x = t \wedge \phi \quad \rightarrow \quad \phi\{x \mapsto t\}$$
Si $x \notin \mathcal{V}ar(t)$

$$\exists x. S^{n+1}(x) = t \wedge \phi \quad \rightarrow \quad t > S^n(0) \wedge \phi\{S^{n+1}(x) \mapsto t\}$$
Si les seules occurrences de x dans t sont dans des termes $S^{n+1}(x)$

$$\exists x. \phi \wedge S^n(x) > t_1 \wedge \ldots \wedge S^n(x) > t_m \wedge u_1 > S^n(x) \wedge \ldots \wedge u_k > S^n(x)$$
$$\rightarrow$$
$$\phi \wedge \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{k} u_j > S(t_i) \wedge u_j > S^n(0)$$
Si $x \notin \mathcal{V}ar(\phi)$

FIG. 3.6 - *Réduction des formules de la théorie des ordres discrets*

Considérons tout d'abord l'avant-dernière règle. Si σ est une solution du membre gauche de la règle, σ est aussi une solution du membre droit, par propriété de l'égalité. Si maintenant σ est une solution du membre droit, $t\sigma > n$. Soit donc $u = t\sigma - n$ et soit $\theta = \sigma\{x \mapsto u\}$. $\theta \models S^{n+1}(x) = t \wedge \phi$.

Considérons maintenant la dernière règle de la figure. À nouveau, toute solution du membre gauche est une solution du membre droit, par propriété de l'égalité. Inversement, si σ est une solution du membre droit, soit v le plus grand entier parmi $1 + t_1\sigma, \ldots, 1 + t_m\sigma, n$. $\theta = \sigma\{x \mapsto v - n\}$ satisfait $\phi \wedge \bigwedge_{i=1}^{m} S^n(x) > t_i \bigwedge_{j=1}^{k} u_j > S^n(x)$. □

Lemme 3.8.3 *Les règles de la figure 3.6 terminent.*

Preuve:
Soit Q une alternance de quantificateurs qui est soit vide soit contient $\forall$ comme quantificateur interne. On construit l'interprétation I de la façon suivante: $I(Q. \bigvee_{i \in J} c_i)$ est le multi-ensemble des $I(c_i)$; $I(\exists \vec{y}.a_1 \wedge \ldots \wedge a_m)$ est le triplet

$$(|\vec{x}|, I_x(a_1 \wedge \ldots \wedge a_n), |a_1 \wedge \ldots \wedge a_n|)$$

où $|\cdot|$ est la taille $x \in \vec{x}$ est le dernier élément du vecteur et $I_x(\phi)$ est le nombre de termes distincts dans lesquels apparait x et auquels sont appliqués l'un des symboles de prédicat. I décroit strictement par application de l'une des règles: ou bien une variable quantifiée existentiellement disparait, ou bien elle reste constante et la paire des deux dernières composantes décroit. □

Lemme 3.8.4 *Si ϕ est une conjonction de formules atomiques $\exists x.\phi$ est réductible par l'une des règles de la figure 3.6.*

Preuve:
Il suffit de remarquer que tous les cas sont couverts par l'une des règles de la figure 3.6. □

Théorème 3.8.5 *La théorie des ordres discrets est décidable.*

Preuve:
D'après les lemmes 3.8.3, 3.8.2 et 3.8.4, toute formule purement existentielle

peut être effectivement transformée en une formule sans quantificateur qui a mêmes solutions. On peut ainsi successivement éliminer tous les quantificateurs, transformant $\forall x.\phi$ en $\neg\exists x \neg\phi$ si la quantification interne est universelle. □

3.8.2 Arithmétique de Presburger proprement dite

On enrichit la syntaxe en ajoutant les prédicats $<_n$ (pour chaque entier naturel n) définis par:

$$x <_n y \stackrel{\text{def}}{=} \exists z.z > 0 \wedge y = x + nz$$

$<$ a ainsi la même signification que $<_1$.

Les formules sont maintenues en forme normale disjonctive et

1. les relations $u \not> v$ sont transformées en $u < v \vee u = v$
2. les relations $u \not>_n v$ sont transformées en

$$u < v + n \vee v <_n u + 1 \vee \ldots \vee v <_n u + n - 1$$

Si bien qu'on peut supposer que les formules ne contiennent pas de négation.

On considère maintenant une formule $\exists y.a_1 \wedge \ldots \wedge a_n$ où $a_1, \ldots, a_n$ sont des formules atomiques. Montrons qu'une telle formule est équivalente à une formule sans quantificateurs. Pour cela, on applique les règles de la figure 3.7.

** regles a simplifier **

** a completer **

3.9 Problèmes équationnels

[93, 84, 35, 94, 29]

3.10 Logique des traîts

3.11 Exercices

$$
\begin{array}{rcl}
t+u=t+v & \rightarrow & u=v \\
t+u <_n t+v & \rightarrow & u <_n v \\
\exists y. y=t \wedge \phi & \rightarrow & \phi\{y \mapsto t\} \\
 & & \text{Si } y \notin \mathcal{V}ar(t) \\
\exists y. ny+t=u \wedge my+t'=u' \wedge \phi & \rightarrow & \exists y. ny+t=u \\
 & & \quad \wedge mu+nt'=nu'+mt \wedge \phi \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,t',u') \\
\exists y. ny+t=u \wedge my+t' >_k u' \wedge \phi & \rightarrow & \exists y. ny+t=u \\
 & & \quad \wedge mu+nt' >_{kn} nu'+mt \wedge \phi \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,t',u') \\
\exists y. ny+t=u \wedge my+t' <_k u' \wedge \phi & \rightarrow & \exists y. ny+t=u \\
 & & \quad \wedge mu+nt' <_{kn} nu'+mt \wedge \phi \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,t',u') \\
\exists y. ny+t >_k u \wedge my+t' >_{k'} u' \wedge \phi & \rightarrow & \exists y. ny+t >_k u \wedge \phi \\
 & & \quad \wedge mu+nt' =_p nu'+mt \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,t',u') \text{ et } p=\mathrm{pgcd}(mk,nk') \\
\exists y. ny+t >_k u \wedge my+t' <_{k'} u' \wedge \phi & \rightarrow & \exists y. ny+t >_k u \wedge \phi \\
\multicolumn{3}{r}{\wedge\Big(\bigvee_{0\le i\le \frac{nk}{p}} \bigvee_{0\le j\le \frac{mk'}{p}} jmk+ink'+nt'+mu=mt+nu'} \\
 & & \vee mt+nu' <_q mu+nt'\Big) \\
\multicolumn{3}{r}{\text{Si } y \notin \mathcal{V}ar(t,u,t',u') \text{ et } p=\mathrm{pgcd}(mk,nk') \text{ et } q=\mathrm{ppcm}(mk,nk')} \\
\exists y. ny+t <_k u \wedge my+t' <_{k'} u' \wedge \phi & \rightarrow & \exists y. ny+t <_k u \wedge \phi \\
 & & \quad \wedge mu+nt' =_p nu'+mt \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,t',u') \text{ et } p=\mathrm{pgcd}(mk,nk') \\
\exists y. ny+t=u \wedge \phi & \rightarrow & (t=u \vee t <_n u) \wedge \phi \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,\phi) \\
\exists y. ny+t <_k u \wedge \phi & \rightarrow & \phi \wedge \\
\multicolumn{3}{r}{(t <_q u \vee \bigvee_{1\le j\le \frac{n}{p}} \bigvee_{0\le i\le \frac{k}{p}} ni+kj+t=u)} \\
\multicolumn{3}{r}{\text{Si } y \notin \mathcal{V}ar(t,u,\phi),\ p=\mathrm{pgcd}(n,k),\ q=ppcm(n,k)} \\
\exists y. ny+t >_k u \wedge \phi & \rightarrow & t =_p u\phi \\
 & & \text{Si } y \notin \mathcal{V}ar(t,u,\phi) \text{ et } p=\mathrm{pgcd}(k,n)
\end{array}
$$

Où $s =_p t$ est une abréviation pour $s=t \vee s <_p t \vee t <_p s$.

FIG. 3.7 - *Simplification des formules de l'arithmétique de Presburger*

Chapitre 4

Méthodes sémantiques

Les méthodes syntaxiques présentent la caractéristique de ne pas exploiter de représentation particulière des contraintes. Par opposition, ce que nous appelons *méthodes sémantiques* exploitent une (autre) représentation de l'ensemble des solutions. Un exemple typique de ces méthodes est l'utilisation des automates.

L'utilisation des automates pour résoudre des formules remonte à Büchi, au début des années 60. L'idée de base est d'associer à chaque formule un automate qui accepte l'ensemble des (tuples de) termes (ou de mots) qui satisfont la formule. Les connecteurs logiques correspondent alors aux opérations ensemblistes sur les langages reconnus par les automates. La satisfaisabilité d'une formule correspond à la décision du vide des langages de la classe considérée.

Obtenir une solution (comme il est requis par la définition 2.2.1) s'obtient en calculant une grammaire associée à l'automate. Cette grammaire (qui sera le plus souvent très facile à obtenir à partir de l'automate) permet d'engendrer autant de solutions qu'il est souhaité. Le caractère incrémental est lui aussi préservé. Le seul défaut de cette approche très séduisante est sans doute la présentation des solutions (qui se lisent mieux sur une forme résolue que sur un automate, surtout lorsque celui-ci a été calculé automatiquement).

Les résultats concernant les liens entre contraintes et automates présentés ici se fondent principalement sur les deux articles de synthèse de W. Thomas [120] et M. Dauchet [43]. Les résultats utilisés sur les automates sont rappelés ici sans preuves. Nous commençons par rappeler rapidement les résultats sur la logique monadique de second ordre à un successeur (le calcul *séquentiel*). Nous illustons ensuite la technique de Büchi sur l'exemple de l'arithmétique de Presburger. Puis nous introduisons les automates d'arbres d'états finis et énonçons succintement le théorème de Rabin. Nous étudions ensuite un certain nombre d'utilisations de ces techniques, introduisant quand c'est nécessaire de nouvelles classes d'automates: les contraintes de sorte (paragraphe 4.7), la théorie de l'entourage (paragraphe 4.8), les contraintes ensemblistes (paragraphe 4.9). Nous évoquons enfin une autre application des automates, cette fois à la réécriture, en liaison avec la recherche de stratégies optimales de réduction.

Dans l'intervalle, nous présentons d'autres méthodes sémantiques. En particulier, des méthodes de résolution d'équations diophantiennes linéaires. Il existe de très nombreuses méthodes de résolutions d'équations diophantiennes linéaires, presque toutes pouvant être classées dans les méthodes sémantiques (e.g. [39, 53, 52, 17]). Il faut néanmoins mettre à part la méthode originale de Presburger vue dans le chapitre précédent et ses prolongements [11, 40, 118]. Nous envisageons ici deux de ces méthodes sémantiques: l'unde par les automates, dans le prolongement du théorème de Büchi, l'autre directe, qui fait suite aux travaux de Fortenbacher [25]. Cette dernière tire parti cette fois de la représentation des solutions sous la forme de combinaisons linéaires d'une *base* de solutions (paragraphe 4.3). Dans le prolongement de la résolution d'équations diophantiennes linéaires, nous présentons aussi une méthode de résolution d'équations modulo l'associativité et la commutativité dans le paragraphe **??** (cas pur, le cas général étant considéré dans le chapitre 5)

4.1 Logique monadique du second ordre à un successeur

4.1.1 Syntaxe

Les formules de la logique monadique de second ordre à un successeur (en abrégé S1S) sont construites à partir de formules atomiques $t = u$ ou bien $t \in X$ où t, u sont des termes sur l'alphabet $\{0, S\}$ (0 est une constante et S un symbole unaire) et X est une variable *de second ordre*. On utilise ensuite les connecteurs logiques (par exemple) $\vee, \neg, \wedge, \bot, \top$ et les quantifications existentielles et universelle aussi bien sur les variables d'individu que sur les variables d'ensemble.

Si ϕ est une formule de S1S, l'ensemble $\mathcal{V}ar(\phi)$ des variables libres de

ϕ est défini de la même façon que pour la logique du premier ordre (voir paragraphe 2.1.5). Mais maintenant, à priori, $\mathcal{V}ar(\phi)$ contient deux sortes de variables: des variables d'individu et des variables d'ensemble. La notation $\phi(x_1, \ldots, x_n, X_1, \ldots, X_m)$ sera parfois utilisée pour indiquer explicitement quelles sont les variables libres de ϕ.

4.1.2 Sémantique

Soit ϕ est une formule de S1S telle que $\mathcal{V}ar(\phi) = \{x_1, \ldots, x_n, X_1, \ldots, X_m\}$. Soient $k_1, \ldots, k_n$ n entiers naturels et $E_1, \ldots, E_m$ m ensembles d'entiers naturels. Soit σ l'assignation $\{x_1 \mapsto k_1; \ldots; x_n \mapsto k_n; X_1 \mapsto E_1; \ldots; X_m \mapsto E_m\}$. Val($\phi$,$\sigma$) (évaluation de ϕ avec l'assignation σ) est définie par récurrence sur la structure des formules par:

- Val($\top$,σ) $= \top$
- Val($\bot$,σ) $= \bot$
- Val($t = u$,σ) $= \top$ si $t\sigma \equiv u\sigma$ et $\bot$ sinon
- Val($t \in X$, σ) $= \top$ si $t\sigma$ appartient à $X\sigma$ et $\bot$ sinon
- Les connecteurs logiques et quantifications s'interprètent de la même façon qu'en logique du premier ordre

σ *satisfait* ϕ (ce que l'on écrit $\sigma \models \phi$) si la formule ϕ s'évalue à $\top$ avec l'assignation σ. S'il n'y a pas d' ambiguïté sur l'association entre variable et valeur, on pourra écrire directement $k_1, \ldots, k_n, E_1, \ldots, E_m$ à la place de σ.

Exemple 4.1.1

$$\begin{aligned}\phi \stackrel{\text{def}}{=}\ & \exists Y, 0 \in X \wedge \forall x, (x \notin X \vee x \notin Y) \wedge (x \in X \vee x \in Y) \\ & \wedge (x \notin X \vee S(x) \in Y) \wedge (x \notin Y \vee S(x) \in X)\end{aligned}$$

possède pour seule variable libre X. Un seul ensemble d'entiers satisfait ϕ: l'ensemble des entiers naturels pairs.

Il est aussi possible d'enrichir la syntaxe par des constructions définissables en S1S. Par exemple, on pourra utiliser les connecteurs logiques $\Rightarrow$ et $\Leftrightarrow$, mais aussi:

la relation de sous-ensemble : $X \subseteq Y \stackrel{\text{def}}{=} \forall x, x \in X \Rightarrow x \in Y$

l'égalité d'ensembles : $X = Y \stackrel{\text{def}}{=} X \subseteq Y \wedge Y \subseteq X$

l'intersection d'ensembles : $Z = X \cap Y \stackrel{\text{def}}{=} \forall x.x \in Z \Leftrightarrow x \in X \wedge x \in Y$

l'union d'ensembles : $Z = X \cup Y \stackrel{\text{def}}{=} \forall x \in Z \Leftrightarrow x \in X \vee x \in Y$

n **ensembles forment une partition des entiers naturels** :

$$\text{partition}(X_1, \ldots, x_n) \stackrel{\text{def}}{=} (\forall x.x \in X_1 \vee \ldots \vee x \in X_n) \wedge \bigwedge_{i \neq j} x \notin X_i \cap X_j$$

l'ensemble vide : $X = \emptyset \stackrel{\text{def}}{=} \forall x.\neg x \in X$

être un singleton : $\text{Sing}(X) \stackrel{\text{def}}{=} \forall Y.(Y \subseteq X \Rightarrow (Y = \emptyset \vee Y = X)) \wedge \neg X = \emptyset$.

la relation d'ordre sur les entiers : la construction est moins triviale; $x \geq y$ si x appartient à tous les ensembles qui contiennent y et qui sont clos par successeur.

$$x \geq y \stackrel{\text{def}}{=} \quad \forall X.(y \in X \wedge \forall z.(z \in X \Rightarrow S(z) \in X)) \Rightarrow x \in X$$

Etant données deux formules ϕ_1, ϕ_2, on définit aussi la relation de *conséquence logique*: $\phi_1 \models \phi_2$ ssi, pour tout σ tel que $\sigma \models \phi_1$, alors $\sigma \models \phi_2$.

4.1.3 Logique monadique faible de second ordre à un successeur

La logique monadique *faible* du second ordre à un successeur, notée WS1S est définie comme S1S, excepté que les quantifications ne peuvent avoir lieu que sur les ensembles finis.

Les facilités syntaxiques du paragraphe précédent peuvent aussi être utilisées en WS1S. En fait, on utilise les mêmes formules, sauf pour partition (car il est maintenant exclu que n variables d'ensembles constituent une partition de l'ensemble de tous les entiers naturels).

En WS1S, il est par contre possible d'utiliser $\max(X_1, \ldots, X_n)$, l'entier le plus grand qui appartient à l'un des ensembles $X_1, \ldots, X_n$ défini par

$$\begin{aligned} x = \max(X_1, \ldots, X_n) \stackrel{\text{def}}{=} \quad & x \in \bigcup_{i=1}^{n} X_i \\ & \wedge \forall y.y > x \Rightarrow y \notin \bigcup_{i=1}^{n} X_i \end{aligned}$$

partition$(X_1, \ldots, X_n)$ est alors une formule établissant que $X_1, \ldots, X_n$ forment une partition d'un segment $[1, \max(X_1, \ldots, X_n)]$. partition s'exprime en WS1S:

$$\begin{aligned} \mathrm{partition}(X_1, \ldots, X_n) \stackrel{\mathrm{def}}{=} & \bigwedge_{i \neq j} X_i \cap X_j = \emptyset \\ & \wedge \forall x, y.(x \in \bigcup_{i=1}^{n} X_i \wedge y < x) \Rightarrow y \in \bigcup_{i=1}^{n} X_i \end{aligned}$$

4.1.4 Les mots comme modèles de (W)S1S

Un mot infini sur l'alphabet fini A (i.e. un arbre infini sur l'alphabet A où tous les symboles de fonction sont unaires) de cardinal n peut être vu comme une partition des entiers naturels en n ensembles disjoints E_a, pour $a \in A$. Il y a ainsi une correspondance que nous allons préciser ci-dessous entre mots et n-uples d'ensembles d'entiers. Si w est un mot infini, nous abrégerons $w(1^i)$ (conforme à la notation des arbres) en $w(i)$, qui dénote la ième lettre de w.

On peut associer à toute formule (non nécessairement close) de S1S un langage de mots infinis, construits sur un alphabet étendu. En effet, remarquons tout d'abord qu'on peut toujours supposer que les variables libres de ϕ sont des variables d'ensemble, en imposant à chaque variable d'individu d'être un singleton: $\phi(x)$ peut-être remplacée par $\phi(X) \stackrel{\mathrm{def}}{=} \mathrm{Sing}(X) \wedge \forall x.x \in X \Rightarrow \phi(x)$.

Ensuite, si $X_1, \ldots, X_n$ sont les variables libres de ϕ, ϕ définit un langage sur l'alphabet $\{0,1\}^n$: si w est un mot infini sur cet alphabet, notons $\pi_i(w)$ $(1 \leq i \leq n)$ l'ensemble d'entiers naturels k tels que la $k$ième lettre de w ait pour $i$ème composante 1. w *satisfait* ϕ ssi $\{X_1 \mapsto \pi_1(w); \ldots; X_n \mapsto \pi_n(w)\} \models \phi$

Exemple 4.1.2 Le mot infini 10101010... satisfait la formule de l'exemple 4.1.1.

Exemple 4.1.3 Prenons un autre exemple, à peine moins trivial. Si $\phi \stackrel{\mathrm{def}}{=} x \geq y$, les mots $w_1 \stackrel{\mathrm{def}}{=} \begin{smallmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{smallmatrix} \ldots$ et $w_2 \stackrel{\mathrm{def}}{=} \begin{smallmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{smallmatrix} \ldots$ satisfont ϕ: en les décryptant, w_1 code la paire d'ensembles $(\{4\}, \{2\})$ et w_2 code la paire $(\{0\}, \{0\})$.

Inversement, étant donné un alphabet A à n éléments, la ième lettre de A peut être identifiée à $(0, \ldots, 0, 1, 0, \ldots, 0) \in \{0,1\}^n$, le 1 se trouvant en position i. Un mot infini w_1 sur l'alphabet A est alors identifié à un mot w_2 sur l'alphabet $\{0,1\}^n$ tel que les ensembles $\pi_i(w_2)$ forment une partition des entiers naturels.

Les formules de S1S constituent donc un (autre) moyen de définir des langages de mots infinis.

Definition 4.1.4 *Le* langage *défini par une formule ϕ de S1S est l'ensemble des mots (infinis) qui satisfont ϕ.*

Cette définition peut être transposée au cas de WS1S: les mots sont alors des mots finis, ou, si l'on veut des mots infinis dont les composantes sont des tuples de 0 à partir d'un certain rang.

4.1.5 Théorème de Büchi

Le théorème de Büchi établit la décidabilité de S1S, en montrant qu'un langage de mots infinis est définissable en S1S si et seulement si il est accepté par un *automate de Büchi.* Les automates de Büchi généralisent les automates de mots finis et nous en donnerons une définition ci-dessous. La preuve s'appuie ensuite sur les propriétés des automates de Büchi, qu'il est hors de notre propos d'établir ici.

Dans le cas de la logique monadique *faible* de second ordre, on montre que les langages définissables sont ceux qui sont reconnus par un automate d'états finis, au sens habituel. Comme nous pouvons par contre supposer connus les résultats sur les automates d'états finis, nous montrons ci-dessous le théorème de Büchi dans le cas de WS1S. La preuve se généralisant aisément modulo les propriétés correspondantes des automates de Büchi.

Théorème 4.1.5 *Un langage est définissable en WS1S si et seulement si il est accepté par un automate d'états finis.*

Preuve:
Supposons tout d'abord que L est accepté par un automate d'états finis (c'est le sens facile de l'implication). Soit alors $Q = \{q_0, \ldots, q_m\}$ l'ensemble d'états, Q_f l'ensemble des états finaux, q_0 l'état initial, $A = \{a_1, \ldots, a_n\}$ l'alphabet et R l'ensemble de règles de production d'un automate (sans ϵ-

transitions) acceptant L. L est alors défini par la formule:

$$
\begin{array}{ll}
\phi_L(X_{a_1}, \ldots, X_{a_n}) \stackrel{\text{def}}{=} & \\
\exists X_{q_0}, X_{q_1}, \ldots, X_{q_m}. & \text{partition}(X_{q_0}, \ldots, X_{q_m}) \\
& \wedge \text{partition}(X_{a_1}, \ldots, X_{a_n}) \\
& \wedge 0 \in X_{q_0} \\
& \wedge \bigwedge_{q \in Q, a \in A} \forall x \in X_q . x \in X_a \Rightarrow (\bigvee_{qa \vdash q' \in R} S(x) \in X_{q'}) \\
& \wedge \bigvee_{q \in Q_f} S(\max(X_{a_1}, \ldots, X_{a_n})) \in X_q
\end{array}
$$

La deuxième ligne de cette formule exprime que les assignations qui satisfont ϕ_L sont bien des mots finis. Les lignes 1 et 3 construisent un *calcul* de l'automate en marquant chaque lettre par un état et en assurant que ce marquage est conforme aux transitions de l'automate. Enfin, la dernière ligne assure qu'on arrive dans un état final.

Pour l'implication inverse, on commence par se ramener à des formules plus simples: on applique d'abord le système de réécriture suivant: système de réécriture suivant (de simplification de formules):

$$
\begin{array}{rcl}
S(t) = S(u) & \rightarrow & t = u \\
S(t) = 0 & \rightarrow & \perp \\
x = S(t) & \rightarrow & \exists y . y = t \wedge x = S(y) \\
& & \qquad \text{Si } t \text{ n'est pas une variable et } y \notin \mathcal{V}ar(x = t) \\
t \in X & \rightarrow & \exists y . y = t \wedge y \in X \\
& & \qquad \text{Si } t \text{ n'est pas une variable et } y \notin \mathcal{V}ar(t)
\end{array}
$$

Ces règles sont évidemment correctes. De plus, elles terminent: considérer par exemple l'interprétation qui à chaque formule associe le multi-ensemble, pour toutes les formules atomiques a de $i(a)$ où $i(s = t)$ est la taille de l'équation (les variables étant ici comptées comme ayant une taille 0) et $i(s \in X)$ est la taille de s. Les formules irréductibles pour ce système de règles ne contiennent que des formules atomiques d'une des formes suivantes: $x = 0, x = S(y), x \in X, x = y$.

On ajoute ensuite la construction syntaxique $X = S(Y)$ qui doit être interprétée comme: X et Y sont des singletons $\{x\}$ et $\{y\}$ respectivement et $x = y + 1$, la construction $X = Y$, interprétée de façon similaire ainsi que la construction $X = 0$ qui est interprétée comme : X est le singleton 0. Toutes les variables d'individus sont alors remplacées par des variables d'ensemble:

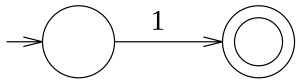


FIG. 4.1 - *Automate acceptant le langage défini par* $X = 0$

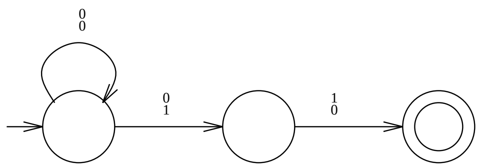


FIG. 4.2 - *Automate acceptant le langage défini par* $X = S(Y)$

$\exists x.\phi$ est remplacée par $\exists X.\phi\{x \mapsto X\}$ où $X \notin \mathcal{V}ar(\phi)$ et de même pour les quantifications universelles où les variables libres. Les expressions $X \in Y$ sont remplacées par $X \subseteq Y$. On obtient ainsi une formule équivalente dans laquelle les seules formules atomiques sont d'une des formes suivantes: $X = 0, X = S(Y), X = Y, X \subseteq Y$.

Montrons maintenant comment construire un automate (de paires) acceptant les ensembles d'ensembles finis satisfaisant chacune de ces formules atomiques. Ceux-ci sont donnés dans les figures 4.1, 4.2, 4.3 et 4.4 respectivement.

Ayant montré comment calculer les automates correspondant à chaque formule atomique, il nous reste à combiner ces formules atomiques. Tout d'abord, nous avons besoin d'une opération appelé *cylindrification* des automates. Etant donné un automate $\mathcal{A}$ sur l'alphabet $\{0,1\}^k$, le cylindrifié

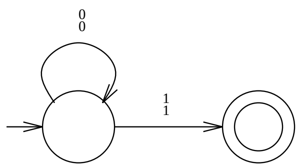


FIG. 4.3 - *Automate acceptant le langage défini par* $X = Y$

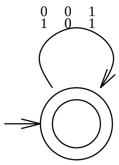


FIG. 4.4 - *Automate acceptant le langage défini par* $X \subseteq Y$

de $\mathcal{A}$ par rapport à $i \in \{1, \ldots, k+1\}$ est l'automate $\mathcal{A}'$ sur $\{0,1\}^{k+1}$ dont les états sont les mêmes que ceux de $\mathcal{A}$, les états finaux sont les mêmes que ceux de $\mathcal{A}$ et les transitions sont définies par:

$$q(a_1, \ldots, a_k) \vdash_{\mathcal{A}} q' \text{ ssi } \forall a \in \{0,1\},\ q(a_1, \ldots, a_{i-1}, a, a_i, \ldots, a_k) \vdash_{\mathcal{A}'} q'$$

Un mot w est alors accepté par $\mathcal{A}'$ ssi le mot w défini par $w(n) = (a_1, \ldots, a_k)$ ssi $w'(n) = (a_1, \ldots, a_{i-1}, a, a_i, \ldots, a_k)$ pour un certain $a \in \{0,1\}$ est accepté par $\mathcal{A}$.

Supposons maintenant que les formules $\phi_1(\vec{Z}, X_1, \ldots, X_n)$ et $\phi_2(\vec{Z}, Y_1, \ldots, Y_m)$ ($\vec{Z}$ est l'ensemble des variables libres communes aux deux formules, que nous supposons ici sans perte de généralité, être listées en premier pour chacune des formules) définissent des langages L_1 et L_2 reconnus par les automates $\mathcal{A}_1$ et $\mathcal{A}_2$ respectivement. Par cylindrifications, on construit les automates $\mathcal{A}'_1$ et $\mathcal{A}'_2$ reconnaissant les langages définis par les formules $\phi'_1(\vec{Z}, \vec{X}, \vec{Y})$ et $\phi'_2(\vec{Z}, \vec{X}, \vec{Y})$ respectivement, où ϕ_i et ϕ'_i sont identiques aux variables libres près. Par intersection (resp. union) des automates $\mathcal{A}'_1$ et $\mathcal{A}'_2$ on obtient alors un automate qui accepte le langage défini par $\phi_1 \wedge \phi_2$ (resp. $\phi_1 \vee \phi_2$). On utilise ici la clôture par intersection et union de la classe des langages réguliers.

Étant donné un automate acceptant le langage défini par ϕ, on peut construire un automate acceptant le langage défini par $\neg\phi$ par complémentation de l'automate. On utilise ici la clôture par complémentaire de la classe des langages réguliers.

Étant donné un automate acceptant le langage défini par $\phi(X_1, \ldots, X_n)$, l'automate acceptant le langage défini par $\exists X_i,\ \phi(X_1, \ldots, X_n)$ s'obtient par *projection* selon la ie'me composante: Il suffit de conserver les mêmes états, les mêmes états finaux, et de remplacer chaque transition $q(a_1, \ldots, a_n) \vdash q'$ par $q(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \vdash q'$.

Il reste enfin la quantification universelle, mais celle-ci s'obtient par double complémentation: $\forall X,\ \phi$ est logiquement équivalente à $\neg\exists X,\ \neg\phi$.

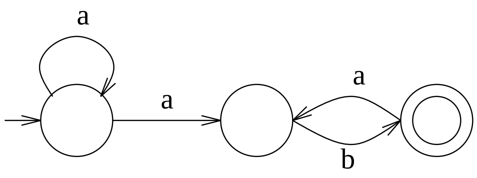


FIG. 4.5 - *Automate de Büchi acceptant $a^*(ab)^\omega$*

Ainsi, par récurrence sur la structure des formules, tout langage définissable est accepté par un automate d'états finis, ce qui complète la preuve. □

Dans le cas de S1S il faut définir une classe d'automates appropriée:

Definition 4.1.6 *Un* automate de Büchi *sur un alphabet A est un quadruplet (Q, q_0, Q_f, R) où Q est un ensemble fini d'*états, *$q_0 \in Q$ est l'*état initial, *Q_f est un sous-ensemble d'*états finaux *de Q et R est un ensemble fini de règles de production de la forme $qa \vdash q'$ où $q, q' \in Q$ et $a \in A$.*

Un mot infini w est accepté *par l'automate (Q, Q_f, R) s'il existe une application ρ des entiers naturels dans Q telle que*

- $\rho(0) = q_0$
- *pour tout i, $\rho(i) = q, w(i) = a, \rho(i+1) = q' \Rightarrow qa \vdash q' \in R$ (ces deux conditions assurent que ρ est un* calcul *de l'automate sur w)*
- *il existe un ensemble infini d'indices i tels que $w(i) \in Q_f$ (c'est la condition d'acceptance de Büchi: elle exprime que le calcul est un succès).*

Exemple 4.1.7 L'automate de la figure 4.5 accepte l'ensemble des mots infinis qui sont constitués d'une suite finie de a suivie d'une séquence infinie $ababababab\ldots$. Ce que l'on note $a^*(ab)^\omega$.

En effet, pour de tels mots, on constuit un calcul de l'automate par $\rho(i) = q_1$ si $w(i+1) = a$ et $(i = 0$ ou $\rho(i-1) = q_1)$, $\rho(i) = q_2$ si $w(i+1) = b$, $\rho(i) = q_3$ si $w(i) = b$.

Inversement, si w est accepté par l'automate, alors $\rho(i) = q_1 \Rightarrow \forall j < i, \rho(j) = q_1$ et $\rho(i) = q_2 \Rightarrow w(i) = a \wedge w(i+1) = b$, $\rho(i) = q_3 \Rightarrow w(i) = b \wedge w(i+1) = a$. Ce qu'on montre par récurrence sur i.

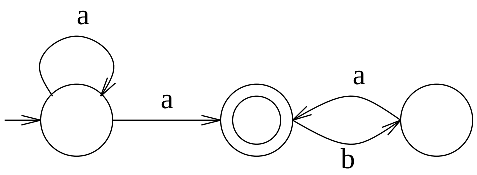


FIG. 4.6 - *Un autre automate acceptant* $a^*(ab)^\omega$

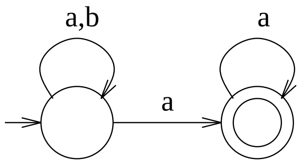


FIG. 4.7 - *Un autre automate acceptant* $(a+b)^*a^\omega$

L'automate de la figure 4.6 accepte aussi le même langage. Il est obtenu en prenant q_2 au lieu de q_3 pour seul état final. Comme automates de mots finis, il accepterait un langage distinct de l'automate de la figure 4.5.

Remarquons que les automates des figures 4.5 et 4.6 sont non-déterministes. Il est malgré tout possible ici de calculer un automate déterministe acceptant le même langage (ce qui est laissé en exercice). Mais il existe des langages reconnus par un automate de Büchi qui sont intrinsèquement non-déterministes, comme l'automate de la figure 4.7.

Mais si l'on excepte la déterminisation, les langages Büchi-reconnaissables possèdent les mêmes propriétés essentielles que les langages reconnaissables de mots finis: clôture par intersection, union, cylindrification, projection (ce qui est facile à montrer), clôture par complémentation (ce qui est assez difficile à montrer, du fait de l'impossibilité de déterminiser), ainsi que la décidabilité du vide.

Si l'on admet ces propriétés de clôture et de dćidabilité, le théorème 4.1.5, s'étend aux mots infinis (la preuve est la même, à ceci près que la condition d'acceptance doit être modifiée dans le passage automates $\to$ logique):

Théorème 4.1.8 *Un langage est définissable en S1S si et seulement si il est accepté par un automate de Büchi.*

Corollaire 4.1.9 *S1S est décidable.*

4.2 Arithmétique de Presburger

Revenons à l'arithmétique de Presburger dont nous avons déjà parlé dans les paragraphes ?? et 3.8. L'objectif est ici de montrer qu'en codant les entiers convenablement, les formules de l'arithmétique de Presburger sont en fait des formules de WS1S.

4.2.1 Les entiers vus comme ensembles finis d'entiers

On peut tout d'abord voir les entiers comme des mots sur l'alphabet $\{0,1\}$: si n est un entier, notons $\overline{n}$ le mot qui est la représentation binaire de n, écrite de droite à gauche, puis complétée à droite par une infinité de zéros. On obtient ainsi une bijection entre les entiers naturels et les mots finis sur $\{0,1\}$ (ou, si l'on veut, les mots infinis qui ne contiennent que des zéros à partir d'un certain rang). Mieux encore, on identifie ces mots à des sous-ensembles finis de $\mathbf{N}$ en considérant la projection π_1 déjà définie: Si n est un entier, on note $\hat{n}$ l'ensemble des entiers naturels i tels que $\overline{n}(i) = 1$. On obtient ainsi une bijection des entiers naturels dans les ensembles finis d'entiers naturels, la réciproque étant à nouveau notée $\hat{}$.

Exemple 4.2.1 $\hat{19} = \{0,1,4\}$ car $\overline{19} = 11001$. $\widehat{\{5,2\}} = 34 = 2^5 + 2^2$

Il reste maintenant à montrer que l'on peut associer à chaque formule ϕ de l'arithmétique de Presburger une formule $\hat{\phi}$ de WS1S de telle sorte que

$$\begin{array}{l}\{x_1 \mapsto n_1; \ldots x_k \mapsto n_k\} \models \phi(x_1,\ldots,x_k)\\ \text{ssi}\\ \{X_1 \mapsto \hat{n_1}; \ldots X_k \mapsto \hat{n_k}\} \models \hat{\phi}(X_1,\ldots,X_k)\end{array}$$

Pour cela, ramenons nous d'abord, par un système de réécriture de formules simple, à des formules particulières de l'arithmétique.

4.2.2 Simplification des formules de l'arithmétique

On utilise le système de règles suivant, qui suppose l'associativité-commutativité de $+$.

$$\begin{array}{rcll}
t = S(u) & \rightarrow & \exists x.\, t = S(x) \wedge x = u & \\
 & & \quad \text{si } u \text{ n'est pas une variable et } x \text{ n'est pas libre dans } t, u & \\
t = u + v & \rightarrow & \exists x.\, t = x + v \wedge x = u & \\
 & & \quad \text{si } u \text{ n'est pas une variable et } x \text{ n'est pas libre dans } t, u, v & \\
S(t) = S(u) & \rightarrow & t = u & \\
S(t) = u + v & \rightarrow & \exists x. x = S(t) \wedge x = u + v & \text{si } x \text{ n'est pas libre dans } t, u, v \\
t + u = v + w & \rightarrow & \exists x. x = t + u \wedge x = v + w & \text{si } x \text{ n'est pas libre dans } t, u, v, w
\end{array}$$

Lemme 4.2.2 *Ce système termine.*

Preuve:
Il suffit de considérer par exemple le multi-ensemble, pour toutes les équations $s = t$ de la formule, de la somme des tailles de s et t et du nombre d'occurrences de 0 dans s, t. Cette interprétation est compatible avec les propriétés des connecteurs logiques ainsi qu'avec l'associativité-commutativité de $+$. Elle est de plus strictement décroissante par application d'une des règles. □

D'autre pat, les règles transforment bien des formules en des formules équivalentes. On obtient ainsi des formules de l'arithmétique telles que toutes les formules atomiques sont d'une des formes suivantes: $x = 0, x = y, x = S(y), x = y + z$.

4.2.3 Traduction de l'arithmétique en WS1S

$\hat{\phi}$ est construit par récurrence sur la structure de ϕ et en supposant ϕ irréductible pour les règles du paragraphe précédent.

Au lieu de $\hat{\phi}$, il suffit en fait de calculer directement l'automate d'états finis qui accepte le langage défini par $\hat{\phi}$. Les connecteurs logiques s'interprétant comme les opérations habituelles sur les automates (cf. paragraphe 4.1.5). Il reste donc uniquement à construire les automates acceptant les langages définis par $x \hat{=} 0, x \hat{=} y, x \hat{=} S(y), x \hat{=} y + z$. Ces automates sont respectivement donnés dans les figures 4.8, 4.9, 4.10 et 4.11.

Il en résulte la décidabilité de l'arithmétique de Presburger, par propriétés de clôture des langages reconnaissables et par décidabilité du vide pour ces automates.

Cependant, la méthode élégante décrite ci-dessus est peu efficace si elle est appliquée telle quelle. Le principal facteur de complexité vient du grand

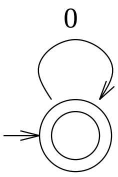


FIG. 4.8 - *Automate acceptant le langage défini par* $x \widehat{=} 0$

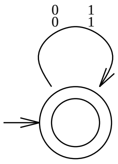


FIG. 4.9 - *Automate acceptant le langage défini par* $x \widehat{=} y$

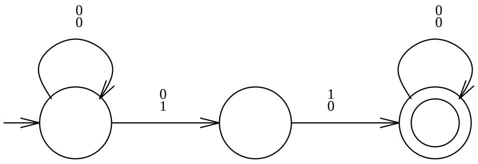


FIG. 4.10 - *Automate acceptant le langage défini par* $x \widehat{=} S(y)$

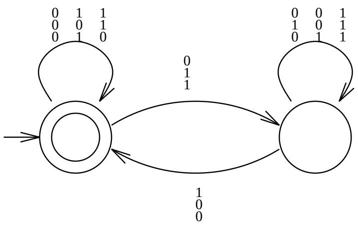


FIG. 4.11 - *Automate acceptant le langage défini par* $x \widehat{=} y + z$

nombre de variables utilisé. On peut mettre au point de nombreuses améliorations, en particulier éviter la transformation des formules du paragraphe précédent en construisant directement des automates pour chaque expression atomique "pure".

Nous allons montrer comment décider l'arithmétique de Presburger au moyen d'automates de manière plus efficace que celle suggérée ci-dessus. Nous allons d'abord montrer comment construire directement l'automate acceptant les (codages des) solutions d'une équation diophantienne linéaire — ce résultat s'étendant facilement aux diséquations et aux inéquations —, puis nous donnerons une méthode efficace pour le fragment existensiel. Enfin, nous montrerons que les automates finis donnent un algorithme quasiment optimal pour la théorie du premier ordre.

4.2.4 Construction de l'automate pour une équation

Nous montrons ici comment construire directement un automate qui reconnait les solutions d'une équation diophantienne linéaire. Plutôt qu'une définition formelle, nous donnons un exemple à partir duquel il sera facile de comprendre comment construire l'automate en général.

Considérons l'équation diophantienne linéaire

$$(e) \qquad\qquad x + 2y - 3z = 2$$

Une solution de (e) doit en paritculier être une solution modulo 2. Si l'on écrit une solution (c_1, c_2, c_3) sous la forme $(2c'_1 + b_1, 2c'_2 + b_2, 2c'_3 + b_3)$, alors $(b_1, b_2, b_3) \in \{0,1\}^3$ doit être solution de $x + 0y + z = 0$, (c'est-à-dire de $b_1 = b_3$ dans notre cas). Dans le cas contraire, les deux membres de l'équation n'auraient pas la même parité et (c_1, c_2, c_3) ne pourrait pas être solution.

Soit $S_2(e)$ l'ensemble des solutions de (E) modulo 2. Dans notre exemple, $S_2(e) = \{(0,0,0), (0,1,0), (1,0,1), (1,1,1)\}$. Pour chaque triplet $(b_1, b_2, b_3) \in S_2(e)$, il faut satisfaire l'équation

$$c_1 - b_1 + 2(c_2 - b_2) - 3(c_3 - b_3) = 2 - (b_1 + 2b_2 - 3b_3)$$

qui peut se diviser par 2. Donc (c'_1, c'_2, c'_3) doit être solution de

$$(e(b_1, b_2, b_3)) \qquad x + 2y - 3z = \frac{2 - (b_1 + 2b_2 - 3b_3)}{2}$$

On a ainsi transformé (e) en une disjonction équivalente de 4 nouvelles équations sur les quotients par 2 de x, y, z.

Reformulons cette transformation en termes de langages formels : soit $[\![e]\!]$ l'ensemble des mots sur $\{0,1\}^3$ qui codent les solutions de e.

On a montré que

$$[\![e]\!] = \bigcup_{(b_1,b_2,b_3)\in S_2(e)} \begin{smallmatrix} b_1 \\ b_2 \\ b_3 \end{smallmatrix} \cdot [\![e(b_1,b_2,b_3)]\!]$$

Ce qui donne pour notre exemple :

$$\begin{aligned}[\![x+2y-3z=2]\!] = \quad & \begin{smallmatrix} 0 \\ 0 \\ 0 \end{smallmatrix} \cdot [\![x+2y-3z=1]\!] \cup \begin{smallmatrix} 0 \\ 1 \\ 0 \end{smallmatrix} \cdot [\![x+2y-3z=0]\!] \\ \cup \quad & \begin{smallmatrix} 1 \\ 0 \\ 1 \end{smallmatrix} \cdot [\![x+2y-3z=2]\!] \cup \begin{smallmatrix} 1 \\ 1 \\ 1 \end{smallmatrix} \cdot [\![x+2y-3z=1]\!]\end{aligned}$$

On peut à présent procéder de manière analogue pour les nouvelles équations que l'on a calculées. On aura par exemple :

$$\begin{aligned}[\![x+2y-3z=1]\!] = \quad & \begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix} \cdot [\![x+2y-3z=2]\!] \cup \begin{smallmatrix} 0 \\ 1 \\ 1 \end{smallmatrix} \cdot [\![x+2y-3z=1]\!] \\ \cup \quad & \begin{smallmatrix} 1 \\ 0 \\ 0 \end{smallmatrix} \cdot [\![x+2y-3z=0]\!] \cup \begin{smallmatrix} 1 \\ 1 \\ 0 \end{smallmatrix} \cdot [\![x+2y-3z=-1]\!]\end{aligned}$$

En supposant que ce processus termine, ce que l'on prouvera par la suite, on obtient un système d'équations linéaires gauches sur Σ^* dont la solution est un langage rationnel. La figure 4.12 montre un automate qui reconnait les solutions de $x+2y-3z=2$.

L'état initial est $[e]$ et l'état final $[x+2y-3z=0]$: chaque codage d'une solution pouvant être suivi d'une suite de $\begin{smallmatrix} 0 \\ 0 \\ 0 \end{smallmatrix}$.

De manière générale, l'automate est construit de la manière suivante : on part d'un ensemble d'états qui ne contient que $[e]$ où e est l'équation à résoudre et l'"état poubelle" $[\perp]$. L'ensemble des transitions T ne contient initialement qu'une transition de $[\perp]$ par lui-même par chaque lettre de Σ.

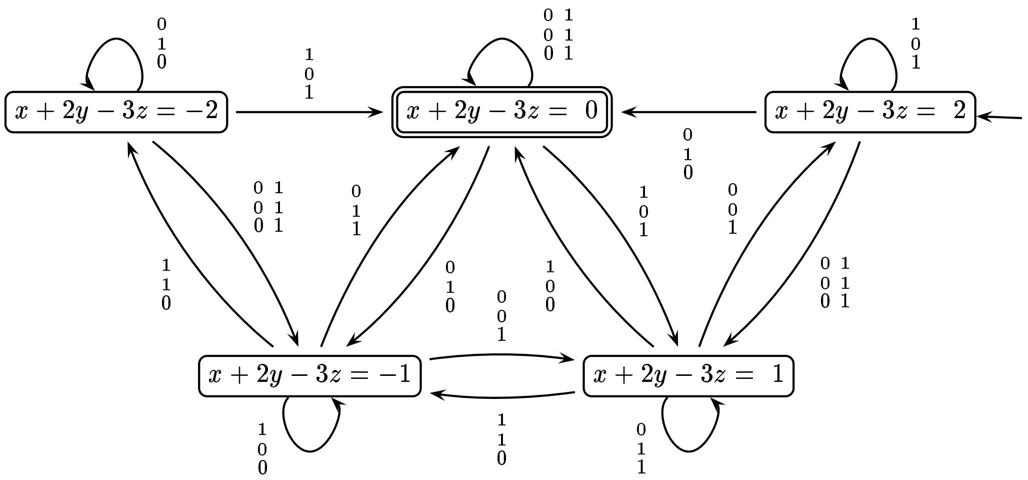


FIG. 4.12 - *L'automate reconnaissant les solutions de* $x + 2y - 3z = 2$

Ensuite, on sature l'ensemble Q des états et T en appliquant les règles :

Si $[a_1x_1 + \cdots + a_nx_n = k] \in Q$ et $\alpha = \begin{matrix} b_1 \\ \vdots \\ b_n \end{matrix} \in \Sigma$ alors

si $k - (a_1b_1 + \cdots + a_nb_n)$ **est pair, alors**

$$\left\{ \begin{array}{ll} [a_1x_1 + \cdots + a_nx_n = k] \xrightarrow{\alpha} [a_1x_1 + \cdots + a_nx_n = \frac{k-(b_1a_1+\cdots+b_na_n)}{2}] & \\ & \text{est ajouté à } T \\ [a_1x_1 + \cdots + a_nx_n = \frac{k-(b_1a_1+\cdots+b_na_n)}{2}] & \\ & \text{est ajouté à } Q \end{array} \right.$$

si $k - (a_1b_1 + \cdots + a_nb_n)$ **est impair, alors**

$$[a_1x_1 + \cdots + a_nx_n = k] \xrightarrow{\alpha} [\perp] \quad \text{est ajouté à } T$$

Le seul état initial est $[e]$ et le seul état final est $[a_1x_1 + \cdots a_nx_n = 0]$.

Il reste à montrer que l'ensemble des états (équations) que l'on peut engendrer à partir de n'importe quel état (équation) initial est fini. Cela découle du fait que si $|k| > \Sigma_{i=1}^{n}|a_i|$, alors, pour toute lettre $\alpha = b_1 \cdots b_n$, $|k| > |\frac{k-(b_1a_1+\cdots+b_na_n)}{2}|$. Donc, pour une équation $a_1x_1 + \cdots + a_nx_n = k$ le nombre d'états de l'automate (autres que $[\bot]$) est borné par $|k| + \Sigma_{i=1}^{n}|a_i|$.

Proposition 4.2.3 *Soit e l'équation diophantienne linéaire $a_1x_1 + \cdots + a_nx_n = k$. L'ensemble des solutions de e est reconnu par un automate fini, déterministe et complet A ayant au plus $|k| + \Sigma_{i=1}^{n}|a_i| + 1$ états et au plus 2^n transitions à partir de chaque état. Si l'on définit la* taille $|e|$ *de e comme la somme des longueurs de $|a_1|, \ldots, |a_n|, |k|$, écrites en binaire et du nombre n de variables, alors l'automate A peut être construit en temps $2^{|e|}$.*

Remarque La méthode que nous avons donnée ne construit pas nécessairement un automate minimal : si l'on engendre un état $[e]$ avec une équation E insatisfaisable, il faudrait que cet état soit identifié à $[\bot]$. Par exemple, si l'on prend pour état initial $[5x + y = 3]$, on va calculer une transition par $\begin{smallmatrix}0\\1\end{smallmatrix}$ *vers l'état $[5x + y = -1]$, étiqueté par une équation insatisfaisable. En revanche, deux états $[e]$ et $[e']$, où e et e' sont deux équations satisfaisables différentes, ne seront jamais identifiés. En effet, e et e' sont de la forme $a_1x_1 + \cdots + a_nx_n = k$ et $a_1x_1 + \cdots + a_nx_n = k'$, elles ne diffèrent que par les constantes des membres droits et ne peuvent donc pas avoir les mêmes ensembles de solutions. Donc, la minimalisation ne consiste dans notre cas qu'à identifier les états à partir desquels l'état final est inaccessible avec l'état $[\bot]$. Cette opération peut se faire en temps linéaire en la taille de A.*

Nous allons utiliser dans la suite la notation suivante :

Notation Soit ϕ une formule de l'arithmétique de Presburger dont les variables sont $x_1, \ldots, x_n$. Pour $\alpha = \begin{matrix} b_1 \\ \vdots \\ b_n \end{matrix} \in \{0,1\}^n$, $\phi \otimes \alpha$ est la formule $\phi\{x_1 \mapsto 2x_1 + b_1, \ldots, x_n \mapsto 2x_n + b_n\}$.

L'état atteint à partir d'un état étiqueté par une formule ϕ par la lettre α est étiqueté par une formule équivalente à $\phi \otimes \alpha$.

4.2.5 L'algorithme de l'automate

De la reconnaissabilité du langage des solutions des formules atomiques, découle la décidabilité des formules du premier ordre bâties sur ces formules atomiques. Etant donné un automate qui reconnaît les solutions d'une équation ϕ, on définit récursivement l'automate A_ψ qui accepte les solutions d'une formule arbitraire ψ. Rappelons la méthode générale:

- $\mathcal{A}_\phi$ est calculé comme au paragraphe précédent pour les formules atomiques (équations diophantiennes linéaires).

- $\mathcal{A}_{\phi_1 \wedge \phi_2} = \mathcal{A}_{\phi_1} \cap \mathcal{A}_{\phi_2}$ est l'automate qui reconnaît l'intersection des langages. Il se construit en temps quadratique.

- $\mathcal{A}_{\phi_1 \vee \phi_2} = \mathcal{A}_{\phi_1} \cup \mathcal{A}_{\phi_2}$ est l'automate qui reconnaît la réunion des langages. Il se construit en temps linéaire.

- $\mathcal{A}_{\neg\phi}$ est l'automate qui reconnaît le complément du langage accepté par $\mathcal{A}_\phi$. Son calcul peut nécessiter une déterminisation ce qui peut donner un automate exponentiellement plus grand dans le pire des cas.

- $\mathcal{A}_{\exists x.\phi}$ se calcule en temps linéaire par *projection* une autre opération standard (voir [120]) consistant à "oublier" une composante des lettres de l'alphabet. Cette opération peut rendre non déterministe un automate déterministe.

- $\mathcal{A}_{\forall x.\phi} = \mathcal{A}_{\neg\exists x.\neg\phi}$. Cette traduction peut être très coûteuse en général : l'automate peut être exponentiel par rapport à l'automate de départ.

Pour décider la validité d'une formule ϕ, il suffira donc de calculer $\mathcal{A}_\phi$ et de voir s'il contient un état final accessible. C'est ce que l'on appellera l'*algorithme de l'automate* pour l'arithmétique de Presburger.

4.2.6 Diséquations et inéquations

Comme on vient de le voir, il est facile de calculer un automate acceptant le langage des solutions d'un système diophantien linéaire comprenant des diséquations et des inéquations (en traduisant $a \geq b$ par $\exists x.a = b + x$). Cependant, la méthode générale n'est pas très efficace et peut produire des automates non-déterministes.

Le cas des diséquations est trivial : il suffit d'échanger les états acceptants et non acceptants de l'automate reconnaissant les solutions de l'équation correspondante. Pour le cas des inéquations, nous étendons la méthode donnée pour les équations afin d'éviter l'introduction de variables auxiliaires dont l'élimination par projection produit des automates non déterministes. Ensuite, nous montrerons comment on peut calculer efficacement l'automate pour un *système* d'équations, et nous donnerons un résultat de complexité pour le fragment existensiel dans le paragraphe 4.2.8.

Comme précedemment, les états de l'automate seront étiquetés par des formules (inéquations) et le langage $[\![i]\!]$ reconnu à partir d'un état $[i]$ sera l'ensemble des (codages des) solutions de I. Ecrivons une inéquation sous la forme

$$(i) \quad a_1x_1 + \cdots + a_nx_n \leq k$$

où $a_1, \ldots, a_n, k \in \mathbb{Z}$. Le seul cas où une telle inéquation n'a pas de solution, est celui où $k < 0$ et $a_1, \ldots, a_n \geq 0$. Dans ce cas, l'automate est trivial, avec un seul état non acceptant. Autrement, comme pour toute formule, le mot αw code une solution de i si et seulement si w est une solution de $i \otimes \alpha$. Le schéma des transitions est semblable à celui pour les équations à ceci près que les deux membres d'une inéquation n'ont pas nécessairement besoin d'avoir la même parité. Considérons par exemple l'inéquation (état) $[x+2y-3z \leq 1]$. Si on lit la lettre $\begin{smallmatrix}1\\0\\1\end{smallmatrix}$ (signifiant que x et y sont impairs et y pair), le reste du mot doit coder une solution de l'inéquation $2x+1+4y-6z-3 \leq 1$ qui est équivalente à $2(x+2y-3z) \leq 3$ ou encore à $x+2y-3z \leq \frac{3}{2}$. Mais cette dernière inéquation a les mêmes solutions dans $\mathbb{N}^3$ que $x+2y-3z \leq 1$. Donc, comme précédemment, l'automate reconnaissant les solutions de i est obtenu en partant de $Q = \{[i], [\perp]\}$ et t contenant uniquement les transitions de $[\perp]$ vers lui-même, en appliquant à saturation les règles suivantes ($\lfloor\cdot\rfloor$ désigne la partie entière d'un nombre rationnel) :

Si $[a_1x_1 + \cdots + a_nx_n \leq k] \in Q$ et $\alpha = \begin{array}{c} b_1 \\ \vdots \\ b_n \end{array} \in \Sigma$ alors

si $k \geq 0$ **ou** $a_i < 0$ **pour un** i

$$\left\{ \begin{array}{l} [a_1x_1 + \cdots + a_nx_n \leq k] \xrightarrow{\alpha} [a_1x_1 + \cdots + a_nx_n \leq \lfloor \frac{k-(b_1a_1+\cdots+b_na_n)}{2} \rfloor] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{est ajouté à } T \\ \\ [a_1x_1 + \cdots + a_nx_n \leq \lfloor \frac{k-(b_1a_1+\cdots+b_na_n)}{2} \rfloor] \text{ est ajouté à } Q \end{array} \right.$$

si $k < 0$ **et** $a_1 > 0, \ldots, a_n > 0$

$$[a_1x_1 + \cdots + a_nx_n \leq \lfloor \frac{k-(b_1a_1+\cdots+b_na_n)}{2} \rfloor] \xrightarrow{\alpha} [\perp] \quad \text{est ajouté à } T$$

Les états finals sont les inéquations pour lesquelles $(0, \ldots, 0)$ est solution, c'est-à-dire les états $[a_1x_1 + \cdots + a_nx_n \leq k]$ avec $k \geq 0$.

La preuve de terminaison est la même que pour les (dis)équations.

Proposition 4.2.4 *Soit i l'inéquation $a_1x_1 + \cdots + a_nx_n \leq k$. L'automate construit ci-dessus est fini, déterministe et complet. Il reconnaîtles solutions de l'inéquation i. il a au plus $|k| + \Sigma_{i=1}^n |a_i| + 1$ états et au plus 2^n transitions à partir de chaque état.*

Encore une fois, nous ne construisons pas nécessairement un automate minimal : l'inéquation $2x - 2y \leq 1$ a les mêmes solutions dans $\mathbf{N}^{\,2}$ que $x \leq y$. L'automate minimal reconnaissant les solutions de $x \leq y$ a deux états, mais le lecteur pourra vérifier que notre algorithme calcule un automate plus grand. La minimisation peut cependant se faire en temps polynomial.

4.2.7 Systèmes d'équations

Plutôt que de calculer l'automate reconnaissant l'intersection des langages (comme on peut toujours le faire pour une conjonction de formules atomiques), on peut calculer directement un automate pour une conjonction. La complexité au pire sera la même, mais en pratique, les performances seront bien meilleures.

Dans le cas d'une équation, on a vu que si on lit une lettre α dans un état $[e]$, on atteint un état autre que $[\perp]$ si et seulement si $e \otimes \alpha$ a ses deux membres de la même parité. Si l'équation $a_1x_1 + \cdots + a_nx_n = k$ a au moins un coefficient a_i impair, il y aura exactement 2^{n-1} transitions à partir de tout état autre que $[\perp]$ vers un état autre que $[\perp]$ et 2^{n-1} transitions vers $[\perp]$. On peut profiter du fait que pour une conjonction d'équations, les lettres pour lesquelles il y aura une transition ne vont pas coïncider en général.

Soit S le système d'équations :

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & +\cdots+ & a_{1n}x_n & = & k_1 \\ a_{21}x_1 & + & a_{22}x_2 & +\cdots+ & a_{2n}x_n & = & k_2 \\ \vdots & & & & & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & +\cdots+ & a_{mn}x_n & = & k_m \end{array}$$

Calculons l'automate reconnaissant les solutions de S. L'alphabet est $\{0,1\}^n$. Les états de l'automate (autres que $[\perp]$ seront étiquetés par des conjonctions d'équations de la forme

$$\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^{n} a_{ij}x_j = k_i$$

Si la lettre $\begin{matrix} b_1 \\ \vdots \\ b_n \end{matrix}$ est lue dans un tel état, le reste du mot doit coder une solution de

$$\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^{n} a_{ij}(2x_j + b_j) = k_i$$

ou, de manière équivalente de

$$\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^{n} 2(a_{ij}x_j) = k_i - \Sigma_{j=1}^{n} a_{ij}b_j$$

Cette formule est insatisfaisable si pour un i k_i et $\Sigma_{j=1}^{n} a_{ij}b_j$ n'ont pas la même parité. En revanche, si pour $1 \leq i \leq m$, k_i et $\Sigma_{j=1}^{n} a_{ij}b_j$ ont la même parité, la formule obtenue est équivalente à

$$\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^{n} a_{ij}x_j = \frac{k_i - \Sigma_{j=1}^{n} a_{ij}b_j}{2}$$

qui est le nouvel état de l'automate. Cet état a été obtenu en ajoutant un vecteur d'entiers au vecteur des constantes des membres droits avant de les diviser par 2. Ce vecteur d'entiers *ne dépend que de la lettre* α. Appelons *incrément* de la lettre $b_1 \cdots b_n$ le m-uplet

$$Incr(\alpha) = (\Sigma_{j=1}^n a_{1j} b_j, \ldots, \Sigma_{j=1}^n a_{mj} b_j)$$

Si l'on définit le *vecteur de parité* d'un état

$$\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^n a_{ij} x_j = k_i$$

comme $(k_1(\text{mod}2), \ldots, k_m(\text{mod}2)) \in \{0,1\}^i$, on pourra calculer à l'avance, pour chacune des 2^m parités possibles, les lettres de $\{0,1\}^n$ pour lesquelles il y aura une transition vers un état autre que $[\perp]$, ainsi que l'incrément correspondant. Ceci rendra plus efficace le processus de saturation par les règles suivantes :

Si $[a_1 x_1 + \cdots + a_n x_n = k] \in Q$ et $\alpha = \begin{matrix} b_1 \\ \vdots \\ b_n \end{matrix} \in \Sigma$ alors

si pour $1 \leq i \leq m$**,** $k_i - \Sigma_{j=1}^n a_{ij} b_j$ **est pair**

$$\left\{ \begin{array}{l} [\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^n a_{ij} x_j = k_i] \xrightarrow{\alpha} [\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^n a_{ij} x_j = \frac{k_i - \Sigma_{j=1}^n a_{ij} b_j}{2}] \\ \qquad\qquad\qquad\qquad \text{est ajouté à } T \\ \\ [\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^n a_{ij} x_j = \frac{k_i - \Sigma_{j=1}^n a_{ij} b_j}{2}] \\ \qquad\qquad\qquad\qquad \text{est ajouté à } Q \end{array} \right.$$

si $k_i - \Sigma_{j=1}^n a_{ij} b_j$ **est impair pour un** i

$$[\bigwedge_{1 \leq i \leq m} \Sigma_{j=1}^n a_{ij} x_j = k_i] \xrightarrow{\alpha} [\perp] \quad \text{est ajouté à } T$$

Exemple 4.2.5 Considérons le codage dans l'arithmétique de Presburger du problème du singe et des noix de coco :

$$\begin{array}{rrrrrrl}
-4x_1 & +5x_2 & & & & & = 1 \\
 & -4x_2 & +5x_3 & & & & = 1 \\
 & & -4x_3 & +5x_4 & & & = 1 \\
 & & & -4x_4 & +5x_5 & & = 1 \\
 & & & & -4x_5 & +5x_6 & = 1
\end{array}$$

Montrons comment le schéma de transitions peut être implanté efficacement. Les états de l'automate (autres que $[\perp]$) seront étiquetés par des formules

$$\bigwedge_{1 \le i \le 5} -4x_i + 5x_{i+1} = k_i$$

L'automate de chaque équation aura 32 transitions vers des états autres que $[\perp]$. La figure 4.13 donne une représentation compacte des transitions pour le problème du singe et des noix de coco. Il n'y a que 2 transitions vers des états autres que $[\perp]$ pour chaque vecteur de parité possible.

4.2.8 Complexité de la décision du fragment existensiel

Comme on l'a déjà dit, les opérations standard sur les automates permettent, pour toute formule bâtie avec des (in-,dis-)équations et les connecteurs logiques $\wedge$ et $\vee$, de calculer un automate qui accepte ses solutions. Le résultat suivant montre que cette méthode est quasi-optimale (le problème est NP-complet) :

Proposition 4.2.6 *Soit ϕ une formule de l'arithmétique de l'arithmétique de Presburger utilisant des équations, des diséquations, des inéquations et les connecteurs $\vee$ et $\wedge$. Soit $C(\phi)$ le nombre de connecteurs dans ϕ, et $|\phi| = C(\phi) + |\phi_1| + \cdots + |\phi_n|$, où $\phi_1, \ldots, \phi_n$ sont les sous-formules atomiques de ϕ et $|\phi_i|$ est le nombre de variables dans ϕ_i plus la somme des longueurs (y compris les constantes des membres droits), écrits en binaire.*

On peut construire un automate complet, non déterministe qui reconnaît les solutions de ϕ en temps $O(2^{|\phi|})$. La taille $|A|$ de A étant le nombre détats de A plus le nombre de transitions, le vide du langage reconnu par A peut se décider en temps $O(|A|)$.

Cela découle des propositions 4.2.3 et 4.2.4 et de la complexité de la construction des automates reconnaissant l'intersection et la réunion des langages[1].

1. Rappelons que si A_1 a Q_1 états et T_1 transitions, et A_2 a Q_2 états et T_2 transitions, l'automate reconnaissant l'union se construit en temps linéaire et celui reconnaissant l'in-

Parité de l'état	*Lettre* α	$Incr(\alpha)$	*Parité de l'état*	*Lettre* α	$Incr(\alpha)$
0 0 0 0 0	1 0 0 0 0 0	(-4, 0, 0, 0, 0)	1 0 0 0 0	1 1 0 0 0 0	(1, -4, 0, 0, 0)
	0 0 0 0 0 0	(0, 0, 0, 0, 0)		0 1 0 0 0 0	(5, -4, 0, 0, 0)
0 0 0 0 1	1 0 0 0 0 1	(-4, 0, 0, 0, 5)	1 0 0 0 1	1 1 0 0 0 1	(1, -4, 0, 0, 5)
	0 0 0 0 0 1	(0, 0, 0, 0, 5)		0 1 0 0 0 1	(5, -4, 0, 0, 5)
0 0 0 1 0	1 0 0 0 1 0	(-4, 0, 0, 5, -4)	1 0 0 1 0	1 1 0 0 1 0	(1, -4, 0, 5, -4)
	0 0 0 0 1 0	(0, 0, 0, 5, -4)		0 1 0 0 1 0	(5, -4, 0, 5, -4)
0 0 0 1 1	1 0 0 0 1 1	(-4, 0, 0, 5, 1)	1 0 0 1 1	1 1 0 0 1 1	(1, -4, 0, 5, 1)
	0 0 0 0 1 1	(0, 0, 0, 5, 1)		0 1 0 0 1 1	(5, -4, 0, 5, 1)
0 0 1 0 0	1 0 0 1 0 0	(-4, 0, 5, -4, 0)	1 0 1 0 0	1 1 0 1 0 0	(1, -4, 5, -4, 0)
	0 0 0 1 0 0	(0, 0, 5, -4, 0)		0 1 0 1 0 0	(5, -4, 5, -4, 0)
0 0 1 0 1	1 0 0 1 0 1	(-4, 0, 5, -4, 5)	1 0 1 0 1	1 1 0 1 0 1	(1, -4, 5, -4, 5)
	0 0 0 1 0 1	(0, 0, 5, -4, 5)		0 1 0 1 0 1	(5, -4, 5, -4, 5)
0 0 1 1 0	1 0 0 1 1 0	(-4, 0, 5, 1, -4)	1 0 1 1 0	1 1 0 1 1 0	(1, -4, 5, 1, -4)
	0 0 0 1 1 0	(0, 0, 5, 1, -4)		0 1 0 1 1 0	(5, -4, 5, 1, -4)
0 0 1 1 1	1 0 0 1 1 1	(-4, 0, 5, 1, 1)	1 0 1 1 1	1 1 0 1 1 1	(1, -4, 5, 1, 1)
	0 0 0 1 1 1	(0, 0, 5, 1, 1)		0 1 0 1 1 1	(5, -4, 5, 1, 1)
0 1 0 0 0	1 0 1 0 0 0	(-4, 5, -4, 0, 0)	1 1 0 0 0	1 1 1 0 0 0	(1, 1, -4, 0, 0)
	0 0 1 0 0 0	(0, 5, -4, 0, 0)		0 1 1 0 0 0	(5, 1, -4, 0, 0)
0 1 0 0 1	1 0 1 0 0 1	(-4, 5, -4, 0, 5)	1 1 0 0 1	1 1 1 0 0 1	(1, 1, -4, 0, 5)
	0 0 1 0 0 1	(0, 5, -4, 0, 5)		0 1 1 0 0 1	(5, 1, -4, 0, 5)
0 1 0 1 0	1 0 1 0 1 0	(-4, 5, -4, 5, -4)	1 1 0 1 0	1 1 1 0 1 0	(1, 1, -4, 5, -4)
	0 0 1 0 1 0	(0, 5, -4, 5, -4)		0 1 1 0 1 0	(5, 1, -4, 5, -4)
0 1 0 1 1	1 0 1 0 1 1	(-4, 5, -4, 5, 1)	1 1 0 1 1	1 1 1 0 1 1	(1, 1, -4, 5, 1)
	0 0 1 0 1 1	(0, 5, -4, 5, 1)		0 1 1 0 1 1	(5, 1, -4, 5, 1)
0 1 1 0 0	1 0 1 1 0 0	(-4, 5, 1, -4, 0)	1 1 1 0 0	1 1 1 1 0 0	(1, 1, 1, -4, 0)
	0 0 1 1 0 0	(0, 5, 1, -4, 0)		0 1 1 1 0 0	(5, 1, 1, -4, 0)
0 1 1 0 1	1 0 1 1 0 1	(-4, 5, 1, -4, 5)	1 1 1 0 1	1 1 1 1 0 1	(1, 1, 1, -4, 5)
	0 0 1 1 0 1	(0, 5, 1, -4, 5)		0 1 1 1 0 1	(5, 1, 1, -4, 5)
0 1 1 1 0	1 0 1 1 1 0	(-4, 5, 1, 1, -4)	1 1 1 1 0	1 1 1 1 1 0	(1, 1, 1, 1, -4)
	0 0 1 1 1 0	(0, 5, 1, 1, -4)		0 1 1 1 1 0	(5, 1, 1, 1, -4)
0 1 1 1 1	1 0 1 1 1 1	(-4, 5, 1, 1, 1)	1 1 1 1 1	1 1 1 1 1 1	(1, 1, 1, 1, 1)
	0 0 1 1 1 1	(0, 5, 1, 1, 1)		0 1 1 1 1 1	(5, 1, 1, 1, 1)

FIG. *4.13* - La table de transitions "schématique" pour le problème du singe et des noix de coco. L'état initial a le vecteur de parité 11111. Dénotons un état par le vecteur $(k_1, \ldots, k_5)$ des constantes des membres droits. L'état atteint à partir de $(k_1, \ldots, k_5)$ en lisant une lettre α de la parité correspondante est $\frac{(k_1,\ldots,k_5)+incr(\alpha)}{2}$. Par exemple, l'état initial $(1, 1, 1, 1, 1)$ a une transition par 111111 vers lui-même et une transition par 011111 vers $(3, 1, 1, 1, 1)$. L'automate a 9524 états dont $(0, 0, 0, 0, 0)$ (le problème est satisfaisable). L'ensemble des états se calcule en 1 seconde en CAMLlight.

4.2.9 Théorie du premier ordre

Comme on l'a déjà dit, la décidabilité de l'arithmétique de Presburger découle du théorème de Büchi. Cependant, en principe l'algorithme pourrait nécessiter un temps non-ELEMENTAIRE. Comme pour le théorème de Büchi, l'élimination des quantificateurs existensiels peut se faire en temps linéaire par projection. Cependant, chaque quantificateur universel peut nécessiter une déterminisation, opération qui construit un automate de taille exponentielle. (Comme, par exemple, dans S1S, ou dans le problème de non-équivalence d'expressions régulières, voir [117, 96]). Mais dans notre cas nos automates n'ont pas toute la complexité des automates finis en général : il y a une borne triplement exponentielle pour le nombre d'états d'un automate déterministe minimal acceptant une formule :

Lemme 4.2.7 *Soit $\phi \equiv Q_n x_n, \ldots, Q_1 x_1 \quad \psi$ une formule de l'arithmétique de Presburger où Q_i est soit $\exists$ soit $\forall$ et ψ est une formule sans quantificateur avec les variables $x_1, \ldots, x_n, y_1, \ldots, y_m$. Il existe un automate déterministe complet reconnaissant les solutions de ϕ aayant au plus $O(2^{2^{2^{|\psi|}}})$ états, où $|\psi|$ est défini comme dans la proposition précédente.*

<u>Preuve:</u>
Comme précédemment, les étets seront (étiquetés par) des formules. Ces formules seront toujours de la forme $B(\cdots, Q_n x_n \cdots Q_1 x_1 \psi_i, \cdots)$ où ψ_i est un état de l'automate acceptant les solutions de ψ (tel qu'on l'a calculé au paragraphe précédent) et $B(\cdot, \ldots, \cdot)$ est une combinaison booléenne utilisant les seuls connecteurs $\vee$ et $\wedge$.

l'état initial est $[\phi] = [Q_n x_n \cdots Q_1 x_1 \psi]$ (où la combinaison Booléenne est triviale).

Si les quantificateurs portent sur les entiers naturels, les deux règles suivantes sont trivialement correctes :

$\forall$-bit
$$\forall x\ \phi \quad \to \quad \forall x \ \ \phi\{x \mapsto 2x\} \ \wedge \ \forall x \ \ \phi\{x \mapsto 2x+1\}$$

$\exists$-bit
$$\exists x\ \phi \quad \to \quad \exists x \ \ \phi\{x \mapsto 2x\} \ \vee \ \exists x \ \ \phi\{x \mapsto 2x+1\}$$

tersection en temps $O(|Q_1| \times |Q_2|) + O(|T1| \times |T_2|)$.

Si la lettre $\alpha = b_1 \cdots b_m \in \{0,1\}^m$ est lue dans un état

$$[B(\cdots, Q_n x_n \cdots Q_1 x_1 \;\; \psi_i, \cdots)]$$

alors le reste du mot doit coder une solution de

$$B(\cdots, Q_n x_n \cdots Q_1 x_1 \;\; \psi_i\{y_1 \mapsto 2y_1 + b_1, \ldots, y_m \mapsto 2y_m + b_m\}, \cdots)$$

Si l'on applique la règle $\forall$-**bit** ou $\exists$-**bit** successivement à $Q_1, \ldots, Q_n$ dans la formule précédente, on obtient une formule

$$B(\cdots, C(\cdots, Q_n x_n \cdots Q_1 x_1 \;\; \psi_i \otimes \beta_j, \cdots), \cdots)$$

où C est une combinaison booléenne de $\wedge$ et $\vee$, $\beta_j \in \{0,1\}^{n+m}$ et $[\psi_i \otimes \beta_j]$ est un état de l'automate acceptant les solutions de ψ.

En résumé, si on lit la lettre $\alpha \in \{0,1\}^m$ dans un état

$$[B(\cdots, Q_n x_n \cdots Q_1 x_1 \;\; \psi_i, \cdots)]$$

on atteint un état étiqueté par une formule

$$B'(\cdots, Q_n x_n \cdots Q_1 x_1 \;\; \psi_i', \cdots)$$

où B' est une combinaison de $\wedge$ et $\vee$ et les ψ_i' sont des états de l'automate acceptant les solutions de ψ. Par la proposition 4.2.6, il y a au plus $2^{|\psi|}$ états dans un automate minimal reconnaissant les solutions de ψ. Donc, il y a au plus $2^{2^{2^{|\psi|}}}$ formules non équivalentes qui peuvent étiqueter les états d'un automate minimal acceptant les solutions de ϕ, les états étiquetés par des formules équivalentes étant identifiés. D'où la borne annoncée. Les états finals sont les formules Ψ telles que $0 \cdots 0 \models \Psi$. Remarquons qu'on n'a pas besoin de les calculer effectivement. $\square$

Ce lemme donne une borne pour tous les automates intermédiaires que l'on calcule en appliquant la méthode de Büchi à l'arithmétique de Presburger :

Théorème 4.2.8 *L'algorithme de l'automate de la section 4.2.5 décide de l'arithmétique de Presburger en temps triplement exponentiel.*

Preuve:
Si l'on utilise l'algorithme de la section 4.2.5 et que l'on minimise l'automate après chaque élimination d'un quantificateur, chaque automate intermédiaire est plus petit que tout automate déterministe acceptant le même langage et a, par le lemme précédent, une taille au plus triplement exponentielle. Le temps nécessaire pour son calcul est polynomial en cette taille. □

4.3 Résolution d'équations linéaires diophantiennes

Nous nous intéressons ici à un fragment de l'arithmétique de Presburger, déjà étudiée par des méthodes d'automates dans le paragraphe précédent: les systèmes d'équations diophantiennes linéaires.

Le problème consiste à trouver les solutions entières, positives d'équations de la forme

$$a_1x_1 + \cdots + a_nx_n = k$$

où $a_1, \ldots, a_n, k \in \mathbb{Z}$. Pour cela, il faut calculer d'une part les solutions minimales de l'équation, et d'autre part l'ensemble S des solutions non-nulles, positives et minimales (pour $>^n$) de l'*équation homogène associée*

$$a_1x_1 + \cdots + a_nx_n = 0$$

Toute solution est alors somme d'une solution minimale et d'une combinaison linéaire de solutions de S.

Huet a proposé une borne sur les solutions minimales [68], améliorée par Lambert [85]. Disposant d'une telle borne, il suffit d'énumérer $\mathbb{N}^n$ jusqu'à la borne, et de retenir les solutions minimales.

Nous allons ici décrire une solution moins coûteuse en pratique pour calculer S. La méthode consiste également à énumérer $\mathbb{N}^n$, mais d'une façon astucieuse qui permet de réduire considérablement l'espace de recherche tout en préservant la complétude.

Nous commençons par présenter un algorithme pour une équation qui a été proposé par Clausen et Fortenbacher [25], puis, nous verrons comment Contejean et Devie ont étendu cette méthode au cas de systèmes d'équations [39]. Les deux algorithmes ont en outre l'avantage de s'adapter facilement au cas inhomogène.

4.3.1 Algorithme de Clausen-Fortenbacher

Le cas homogène

Notation On notera e_i le $i$ème vecteur de la base canonique de $\mathbb{N}^n$. Etant donné un vecteur $v \in \mathbb{N}^n$, $v(i)$ désigne la $i$ème composante de v.

Definition 4.3.1 *Le* défaut *du vecteur* $c \equiv (c_1, \ldots, c_n) \in \mathbb{N}^n$ *par rapport à l'équation diophantienne linéaire* $e \equiv a_1x_1 + \cdots + a_nx_n = 0$ $(a_1, \ldots, a_n \in \mathbb{Z})$ *est l'entier relatif* $a_1c_1 + \cdots + a_nc_n$. *On le note* $D(c, e)$ *ou plus simplement* $D(c)$, *s'il n'y a pas d'ambigüité.*

D'une certaine façon, le défaut mesure combien un vecteur *n'est pas* solution d'une équation. En effet, $(c_1, \ldots, c_n) \in \mathbb{N}^n$ est solution d'une équation e si et seulement si son défaut par rapport à e est égal à zéro.

La clef de la méthode réside dans le lemme trivial suivant :

Lemme 4.3.2 *Soit e l'équation diophantienne linéaire* $e \equiv a_1x_1 + \cdots + a_nx_n = 0$. *Si* $c = (c_1, \ldots, c_n) \in \mathbb{N}^n$ *n'est pas solution de e, alors, pour toute solution d, plus grande que c pour l'ordre produit cartésien* $>^n$, *il existe un vecteur* e_i *de la base canonique de* $\mathbb{N}^n$ *tel que* $d \geq^n c+e_i$ *avec* $D(e_i) \times D(c) < 0$.

<u>Preuve:</u>

Supposons que $b = (b_1, \ldots, b_n) \geq^n C$ soit solution et que les seules composantes pour lesquelles b est plus grand que c correspondent à des coefficients de même signe que $D(c)$. On a alors $|D(b)| \geq |D(c)|$ et b ne peut pas être une solution.□

La complétude de l'algorithme de la figure 4.14 découle du lemme précédent.

Proposition 4.3.3 *Etant donné une équation diophantienne linéaire*

$$e \equiv a_1x_1 + \cdots + a_nx_n = 0$$

l'algorithme de la figure 4.14 termine toujours et calcule l'ensemble S des solutions non-nulles, positives, minimales pour $\geq^n$ *de e.*

<u>Preuve:</u>

La complétude découle du lemme 4.3.2. La terminaison est garantie par le

Entrée : l'équation $e \equiv a_1x_1 + \cdots + a_nx_n = 0$

- Soient
  - $S_0 = \emptyset$
  - $M_0 = \{e_1, \ldots, e_n\}$.
- On définit
  - $S_{i+1} = S_i \cup \{v \in M_i \mid D(v, e) = 0\}$
  - $M_{i+1} = \{v + e_i \mid \nexists s \in S_{i+1} \text{ t.q. } v \geq^n s, \text{ et } D(v, e) \times D(e_i, e) < 0\}$.
- On retourne le premier S_k tel que $M_k = \emptyset$.

FIG. 4.14 - *L'algorithme de Clausen-Fortenbacher, pour une équation homogène, première version.*

fait que comme chaque vecteur engendré est obtenu à partir du précédent par l'ajout d'un vecteur e_i ayant un défaut de signe opposé, la valeur absolue des défauts des vecteurs engendrés est toujours inférieure ou égale à celle des coefficients. Soit k la plus grande valeur absolue des coefficients. Si, partant d'un vecteur de base e_i, on ajoute $2k$ fois un vecteur de base ayant un défaut de signe opposé à celui du vecteur courant, par le principe des tiroirs, on engendrera deux vecteurs v et w tels que $v <^n w$ et $D(v) = D(w)$. Mais alors, $w - v$ est solution. Comme l'algorithme engendre les vecteurs en ordre non décroissant pour $>^n$, et qu'il est complet, $w - v$ appartient à l'ensemble des solutions déjà calculées avant d'engendrer w. Donc on n'incrémentera jamais w, et l'algorithme termine après au plus $2k$ itérations. □

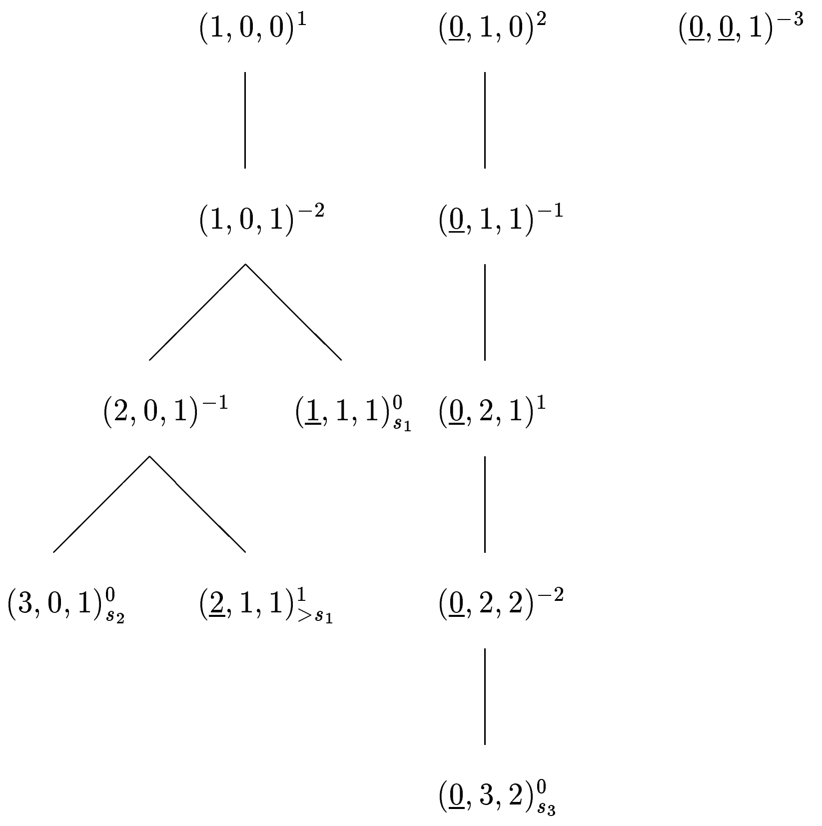
On remarquera, en appliquant l'algorithme de la figure 4.14 à l'équation $x + 2y - 3z = 0$ que la solution $(1, 1, 1)$ est engendrée 4 fois. Pour éviter d'engendrer plusieurs fois une solution, on peut, lorsque un vecteur v peut être incrémenté par e_i et e_j avec $i < j$, engendrer les vecteurs $v + e_i$ et $v + e_j$ où la $i$ème composante de $v + e_j$ est *gelée*, c'est-à-dire qu'on s'interdit de jamais l'incrémenter. L'algorithme ainsi modifié est évidemment plus efficace, et il reste complet pour la raison suivante. Soit $v \in M_i$ un vecteur engendré par l'algorithme, non plus grand qu'une solution minimale, et $S_{>v}$ l'ensemble des solutions minimales plus grandes que v. Soient $e_{l_1}, \ldots, e_{l_k}$ les vecteurs de base ayant un défaut de signe opposé. Pour $1 \leq h \leq k$, soit S_h l'ensemble des solutions supérieures ou égales à $v + e_{i_h}$. Alors les ensembles $S_1, S_v \setminus S_1, S_v \setminus (S_1 \cup S_2), \ldots, S_v \setminus (S_1 \cup \cdots \cup S_{k-1})$ forment une partition de $S_{>v}$. Donc, il n'est pas nécessaire de considérer les solutions plus grandes que $v + e_i$ issues de $v + e_j$, car elles sont déjà engendrées à partir de $v + e_i$. On obtient ainsi l'algorithme de la figure 4.15 où un vecteur n'est jamais engendré deux fois.

Entrée : l'équation $e \equiv a_1x_1 + \cdots + a_nx_n = 0$

- Soient
  - $S_0 = \emptyset$
  - $M_0 = \{e_1, \ldots, e_n\}$
    où les composantes $e_i(1), \ldots, e_i(i-1)$ de e_i sont gelées pour $i > 1$.
- On définit
  - $S_{i+1} = S_i \cup \{v \in M_i \mid D(v, e) = 0\}$
  - $M_{i+1} = \{v + e_i \mid \nexists s \in S_{i+1} \text{ t.q. } v \geq^n s,$
    $\text{et } D(v, e) \times D(e_i, e) < 0,$
    $\text{et } v(i) \text{ n'est pas gelée}\}$
    où, si l'on a engendré $v + e_i$ et $v + e_j$ avec $i < j$ à partir de v, la $i$ème composante de $v(e_j)$ est gelée.
- On retourne le premier S_k tel que $M_k = \emptyset$.

FIG. 4.15 - *L'algorithme de Clausen-Fortenbacher, pour une équation homogène, deuxième version.*

Proposition 4.3.4 *Etant donné une équation diophantienne linéaire $e \equiv a_1x_1 + \cdots + a_nx_n = 0$, l'algorithme de la figure 4.15 termine toujours et calcule l'ensemble S des solutions non-nulles, positives, minimales pour $\geq^n$ de e. Chaque vecteur engendré au cours du calcul ne l'est qu'une fois.*



Calcul de l'algorithme de Clausen-Fortenbacher sur l'équation $x + 2y - 3z = 0$. Les défauts des vecteurs figurent en exposant et les composantes gelées sont soulignées.

FIG. 4.16 - *Exemple d'exécution de l'algorithme de la figure 4.15.*

La figure 4.16 montre le fonctionnement de l'algorithme sur l'équation $x + 2y - 3x = 0$.

Extension au cas inhomogène

L'algorithme de Clausen-Fortenbacher s'étend facilement au cas inhomogène. En effet, toute solution positive de l'équation

$$e \equiv a_1x_1 + \cdots + a_nx_n = k$$

est la somme d'une solution positive, minimale, non nulle de e et d'une combinaison linéaire des solutions positives, minimales, non nulles de l'équation homogène associée

$$a_1x_1 + \cdots + a_nx_n = 0$$

Si l'on applique l'algorithme à l'équation homogène

$$e' \equiv a_1x_1 + \cdots + a_nx_n - kx' = 0$$

les solutions où x' vaut 1, sont les solutions positives, minimales, non nulles de e, et celles oú x' vaut 0 sont les solutions positives, minimales, non nulles de l'équation homogène associée. On pourra donc utiliser l'algorithme précédent en ajoutant une nouvelle variable x' et en gelant la composante associée dès qu'elle atteint 1.

4.3.2 Algorithme de Contejean-Devie

Contejean et Devie ont adapté l'algorithme précédent à la résolution directe de systèmes d'équations, à partir d'une interprétation géometrique simple de l'algorithme de Clausen-Fortenbacher [38, 39].

La notion de défaut s'étend naturellement au cas de systèmes d'équations :

Definition 4.3.5 *Le* défaut *du vecteur $c \equiv (c_1, \ldots, c_n) \in \mathbb{N}^n$ par rapport au système $\mathcal{S}$ d'équations diophantiennes linéaires:*

$$\begin{array}{ccccccc} a_{11}x_1 & + & \cdots & + & a_{1n}x_n & = & 0 \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ a_{m1}x_1 & + & \cdots & + & a_{mn}x_n & = & 0 \end{array}$$

où les a_{ij} sont dans $\mathbb{Z}$ est ve vecteur $D(v) \in \mathbb{Z}^m$

$$(\Sigma_{i=1}^{n} a_{1i}c_i, \ldots, \Sigma_{i=1}^{n} a_{mi}c_i)$$

Un vecteur $c \equiv (c_1, \ldots, c_n) \in \mathbf{N}^n$ est alors solution de $\mathcal{S}$ si et seulement si son défaut est $(0, \ldots, 0)$.

La figure 4.17 montre comment on peut traduire la restriction de Clausen-Fortenbacher sur les défauts des variables dans le cas de plusieurs équations. Dans le cas d'une équation, la condition que le défaut du vecteur de base que l'on rajoute ait un signe contraire à celui du défaut du vecteur courant impose de n'incrémenter que les composantes dont le défaut est tel que l'on *revient vers l'origine*. Cela revient, dans le cas d'un système à n'incrémenter que les composantes dont le défaut est dans le demi-espace ouvert délimité par l'hyperplan orthogonal au défaut courant et contenant l'origine. Cette restriction préserve la complétude pour la même raison que dans le cas d'une équation : si l'on incrémente un vecteur v tel que $D(v) \neq 0$ uniquement en ses composantes qui ne vérifient pas cette restriction, on ne reviendra pas dans le demi-espace contenant l'origine. On peut donc adapter l'algorithme de Clausen-Fortenbacher à la résolution de systèmes en n'incrémentant le vecteur courant v de e_i que si le produit scalaire $D(v) \cdot D(e_i)$ est strictement négatif.

Lemme 4.3.6 *Soit $\mathcal{S}$ le système d'équations diophantiennes linéaires*

$$\begin{array}{ccccccc} a_{11}x_1 & + & \cdots & + & a_{1n}x_n & = & 0 \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ a_{m1}x_1 & + & \cdots & + & a_{mn}x_n & = & 0 \end{array}$$

Si $c = (c_1, \ldots, c_n) \in \mathbb{N}^n$ n'est pas solution de $\mathcal{S}$, alors, pour toute solution d, plus grande que c pour l'ordre produit cartésien $>^n$, il existe un vecteur e_i de la base canonique de $\mathbb{N}^n$ tel que $d \geq^n c + e_i$ avec $D(e_i) \cdot D(c) < 0$.

On obtient l'algorithme de la figure 4.18 qui est évidemment complet par le lemme 4.3.6, mais dont la terminaison n'est pas triviale,contrairement au cas d'une équation. On pourra se reporter à [39] pour la preuve de terminaison.

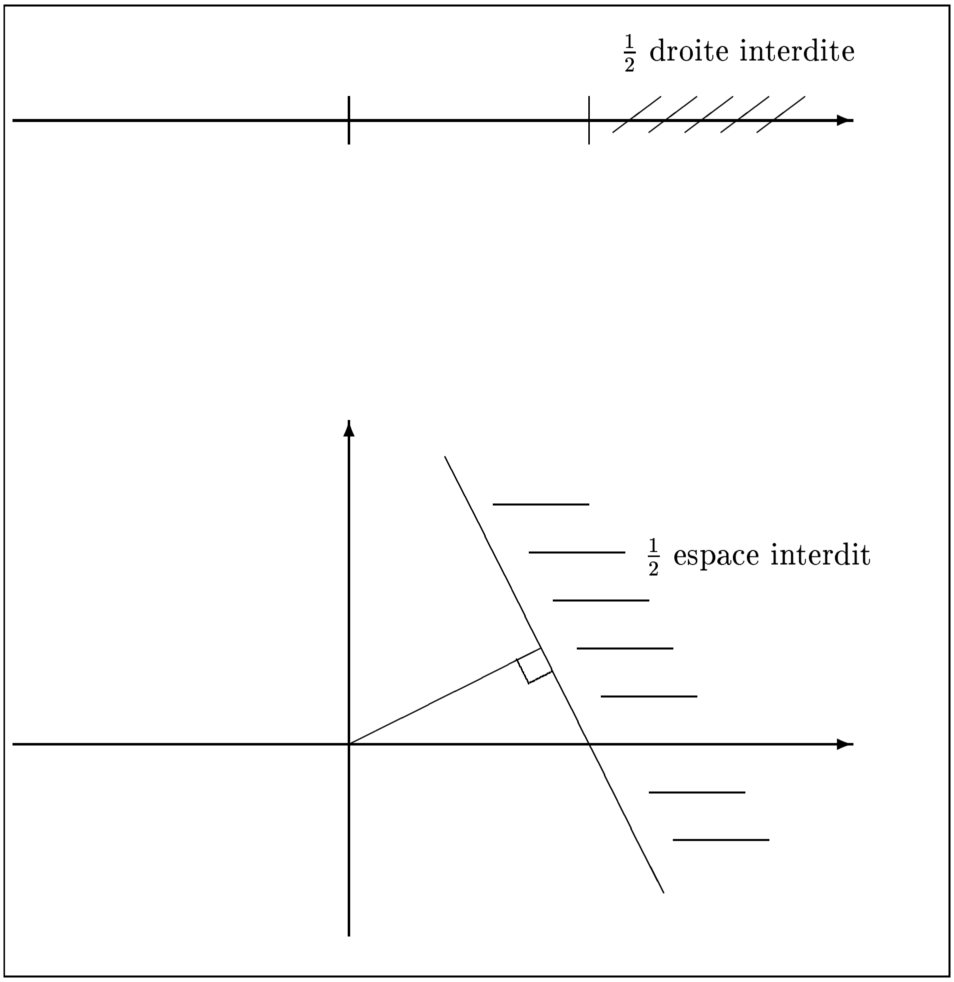


FIG. 4.17 - *La restriction géométrique imposée sur les défauts.*

Entrée : le système d'équations diophantiennes linéaires S

$$\begin{array}{ccccccc} a_{11}x_1 & + & \cdots & + & a_{1n}x_n & = & 0 \\ \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & \cdots & + & a_{mn}x_n & = & 0 \end{array}$$

- Soient
  - $S_0 = \emptyset$
  - $M_0 = \{e_1, \ldots, e_n\}$
    où les composantes $e_i(1), \ldots, e_i(i-1)$ de e_i sont gelées pour $i > 1$.
- On définit
  - $S_{i+1} = S_i \cup \{v \in M_i \,|D(v,e) = (0, \ldots, 0)\}$
  - $M_{i+1} = \{v + e_i \mid \nexists s \in S_{i+1}$ t.q. $v \geq^n s$,
    et $D(v) \cdot D(e_i) < 0$,
    et $v(i)$ n'est pas gelée$\}$
    où, si l'on a engendré $v + e_i$ et $v + e_j$ avec $i < j$ à partir de v, la $i$ème composante de $v(e_j)$ est gelée.
- On retourne le premier S_k tel que $M_k = \emptyset$.

FIG. 4.18 - *L'algorithme de Contejean-Devie pour un système d'équations diophantiennes linéaires*

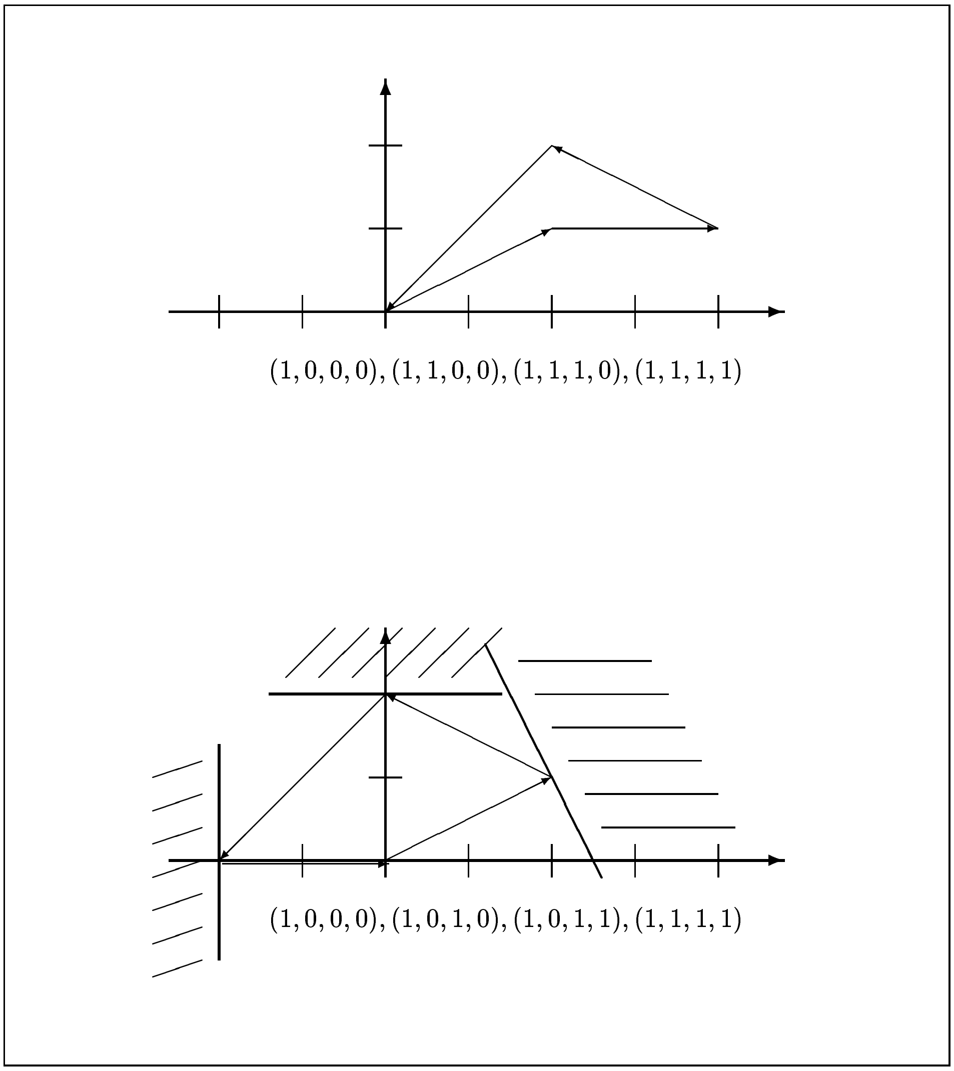


FIG. 4.19 - *Deux séquences de défauts correspondant à la solution* $(1, 1, 1, 1)$ *du système*

$$\begin{array}{rcrcrcrcl} 2x_1 & + & 2x_2 & - & 2x_3 & - & 2x_4 & = & 0 \\ x_1 & & & + & x_3 & - & 2x_4 & = & 0 \end{array}$$

Seule la deuxième sera construite par l'algorithme de Contejean-Devie, car la première viole la restriction géometrique.

4.4 Unification associative-commutative élémentaire

La théorie de l'associativié-commutativité est celle qui a été le plus étudiée pour l'unification [90, 115, 77, 54, 56, 67, 23, 19, 2, 76, 51].

Nous décrivons ici le cas *élémentaire*, c'est-à-dire où l'on restreint l'alphabet à un symbole associatif-commutatif $+$, et l'on résoud des équations dans $\mathcal{T}(\{+\}, \mathcal{X})$ modulo la théorie AC présentée par les axiomes

$$\begin{array}{rcll} (x+y)+z & = & x+(y+z) & \quad \text{A} \\ x+y & = & y+x & \quad \text{C} \end{array}$$

On a vu dans la section 3.7 que AC est une théorie syntaxique, mais que la méthode syntaxique ne termine pas. Nous allons donner une méthode sémantique pour l'unification AC-élémentaire, mais auparavant, nous introduisons une notation pour les substitutions (formes résolues) qui est adaptée à ce problème.

Definition 4.4.1 *Soient* $v_1, \ldots, v_n$ *des variables,* $t_1, \ldots, t_m$ *des termes, et pour* $i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$, $a_{ij} \in I\!N$ *. Le tableau suivant*

$$T \equiv \quad \begin{array}{c|ccc} & v_1 & \cdots & v_n \\ \hline t_1 & a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \vdots \\ t_m & a_{m1} & \cdots & a_{mn} \end{array}$$

où, pour tout $j \in \{1, \ldots, n\}$ $\Sigma_{i=1}^{m} a_{ij} \neq 0$ *désigne la substitution qui à chaque* v_i *associe le terme*

$$\underbrace{t_1 + \cdots + t_1}_{a_{1i} \text{ fois}} + \cdots + \underbrace{t_m + \cdots + t_m}_{a_{mi} \text{ fois}}$$

Les sous-tableaux admissibles *de T sont les sous-tableaux (au sens où l'on a enlevé certaines lignes) dont la somme des coefficients d'aucune colonne n'est nulle. (Cette condition est nécessaire pour qu'un sous-tableau représente une substitution bien formée).*

Dans la définition ci-dessus, on a — sans le dire — déjà quitté l'approche syntaxique, car on se permet de considérer que $+$ a une arité variable, ce qui se justifie pour un symbole associatif, mais pas en général. Une autre considération d'ordre sémantique nous permet de nous restreindre aux problèmes de la forme

$$P \equiv \underbrace{x_1 + \cdots + x_1}_{a_1 \text{ fois}} + \cdots + \underbrace{x_n + \cdots + x_n}_{a_n \text{ fois}} = \underbrace{y_1 + \cdots + y_1}_{b_1 \text{ fois}} + \cdots + \underbrace{y_m + \cdots + y_m}_{b_m \text{ fois}}$$

où $\{x_1, \ldots, x_n\} \cap \{y_1, \ldots, y_m\} = \emptyset$. En effet, l'équation $x + s = x + t$ a les mêmes solutions que $s = t$ modulo AC.

Au problème P ci-dessus, on associe léquation diophantienne linéaire

$$(e) \equiv a_1 v_1 + \cdots + a_n v_n = b_1 w_1 + \cdots + b_m w_m$$

Le lemme suivant établit un lien entre P et (e) :

Lemme 4.4.2 *Soit σ une solution de P. Soit z une variable introduite par σ. Soient $c_1, \ldots, c_n, d_1, \ldots, d_m$ les nombres respectifs d'occurrences de z dans $x_1\sigma, \ldots, x_n\sigma, y_1\sigma, \ldots, y_m\sigma$. Alors $(c_1, \ldots, c_n, d_1, \ldots, d_m)$ est une solution de (e).*

Preuve:

Supposons le contraire pour une variable z introduite par σ. Alors, il n'y a pas le même nombre d'occurrences de z dans $s\sigma$ et $t\sigma$ ce qui contredit le fait que σ est solution. $\square$

La donnée de l'ensemble des solutions minimales, positives, non nulles de (e) (calculé par exemple avec l'algorithme de Clausen-Fortenbacher) nous permet de calculer un ensemble complet de AC-unificateurs de P. Le résultat suivant a été trouvé indépendamment par Stickel et par Livesey et Siekmann [116, 90, 115] :

Théorème 4.4.3 (Unification AC élémentaire) *Soit P le problème d'unification*

$$\underbrace{x_1 + \cdots + x_1}_{a_1 \text{ fois}} + \cdots + \underbrace{x_n + \cdots + x_n}_{a_n \text{ fois}} = \underbrace{y_1 + \cdots + y_1}_{b_1 \text{ fois}} + \cdots + \underbrace{y_m + \cdots + y_m}_{b_m \text{ fois}}$$

où $\{x_1, \ldots, x_n\} \cap \{y_1, \ldots, y_m\} = \emptyset$. Soit

$$S = \{(c_{1,1}, \ldots, c_{1,n+m}), \ldots, (c_{k,1}, \ldots, c_{k,n+m})\}$$

l'ensemble des solutions mininales, positives, non nulles de l'équation diophantienne

$$a_1 v_1 + \cdots + a_n v_n = b_1 w_1 + \cdots + b_m w_m$$

Alors, les sous-tableaux admissibles de

$$T \equiv \begin{array}{c|cccccc} & x_1 & \cdots & x_n & y_1 & \cdots & y_m \\ \hline z_1 & c_{1,1} & \cdots & c_{1,n} & c_{1,n+1} & \cdots & c_{1,n+m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ z_k & c_{k,1} & \cdots & c_{k,n} & c_{k,n+1} & \cdots & c_{k,n+m} \end{array}$$

où $z_1, \dots, z_k$ sont des nouvelles variables, forment un ensemble complet de AC-unificateurs de P.

<u>Preuve:</u>

Correction : Tout sous-tableau admissible est une substutution bien formée. Cette substitution est une solution car par construction $(x_1 + \cdots + x_n)\sigma$ et $(y_1 + \cdots + y_m)\sigma$ ont le même nombre d'occurrences de chaque z_i.
Complétude : Soit θ une solution de P. Soit $W = \{w_1, \dots, w_l\}$ l'ensemble des variables introduites par θ. Par le lemme 4.4.2, pour chaque $w \in W$, il existe $(k_1, \dots, k_{n+m})$, solution de (e), telle que k_i est le nombre d'occurrences de w dans x_i si $i \leq n$ ou y_{i-n} si $i > n$. Comme S est un ensemble générateur des solutions de (e),

$$(k_1, \dots, k_{n+m}) = \alpha_1 s_1 + \cdots + \alpha_k s_k$$

où $s_i = (c_{i,1}, \dots, c_{i,n+m}) \in S$. On définit $t(z_i, w) = \underbrace{w + \cdots + w}_{\alpha_i \text{ fois}}$ si $\alpha_i \neq 0$.
Soit σ le sous-tableau de T où l'on a enlevé les lignes correspondant aux z_i pour lesquels $t(z_i, w)$ n'est défini pour aucun $w \in W$. Par construction, σ est un sous-tableau admissible et

$$\theta =_{AC} \sigma\{z_i \mapsto t(z_i, w_1) + \cdots + t(z_i, w_l)\}$$

□

Exemple 4.4.4 Soit $P \equiv x + x = y + z + t$. Les sous-tableaux admissibles de

	x	y	z	t
u_1	1	2	0	0
u_2	1	0	2	0
u_3	1	0	0	2
u_4	1	1	1	0
u_5	1	1	0	1
u_6	1	0	1	1

sont au nombre de 45. Ils forment un ensemble minimal complet d'AC-unificateurs de P. Par exemple, le sous-tableau où l'on a gardé les lignes correspondant à v_1, v_2, v_5 représente la solution

$$\{x \mapsto v_1 + v_2 + v_5, y \mapsto v_1 + v_1 + v_5, z \mapsto v_2 + v_2, t \mapsto v_5\}$$

4.5 Automates d'arbres

Les automates d'arbres généralisent les automates de mots à des situations où les symboles ne sont pas nécessairement tous unaires. Ils possèdent par contre les mêmes propriétés que les automates de mots. On peut ainsi (et nous en verrons des exemples) tirer parti de ce pouvoir d'expression accru pour résoudre de nouveaux systèmes de contraintes avec leur aide.

Il y a peu d'ouvrages de synthèses sur les automates d'arbres. En fait [60] est, à notre connaissance, le seul livre publié qui leur est entièrement consacré (et qui ne soit pas une collection d'articles). On pourra ausi consulter [42, 120].

4.5.1 Le point de vue algébrique

La définition la plus simple d'un ensemble réconnaissable d'arbres est la généralisation immédiate de la définition des ensembles reconnaissables de mots qui sont les images homomorphes inverses d'un sous-ensemble d'un monoide fini:

Definition 4.5.1 *Etant donné un alphabet fini de symboles $\mathcal{F}$, un sous-ensemble L de $T(\mathcal{F})$ est* reconnaissable *s'il existe une $\mathcal{F}$-algèbre finie $\mathcal{A}$ de domaine Q, contenant un sous-ensemble fini Q_f tel que $L = h_{\mathcal{A}}^{-1}(Q_f)$ où $h_{\mathcal{A}}$ est l'unique morphisme de $T(\mathcal{F})$ dans $\mathcal{A}$.*

Exemple 4.5.2 Si $\mathcal{F}$ contient un symbole binaire f et une constante a, l'ensemble des arbres finis ayant un nombre pair de nœuds est reconnaissable car c'est l'ensemble des arbres t tels que $h_{\mathcal{A}}(t) = q_p$ où $\mathcal{A}$ est l'algèbre à deux éléments $\{q_i, q_p\}$ vérifiant: $a =_{\mathcal{A}} q_i$, $f(q_p, q_p) =_{\mathcal{A}} f(q_i, q_i) =_{\mathcal{A}} q_i$, $f(q_i, q_p) =_{\mathcal{A}} f(q_p, q_i) = q_p$.

Cependant, cette définition est peu opérationnelle et ne donne pas une bonne intuition.

4.5.2 Les aaa

$\mathcal{F}$ est un alphabet fini de symboles de fonction.

Definition 4.5.3 *Un* automate ascendant d'arbres *(aaa en abrégé) est un triplet formé d'un ensemble fini d'états Q, d'un sous-ensemble Q_f d'états finaux et d'un ensemble fini de règles de réécriture $f(q_1, \ldots, q_n) \to q$ où $q_1, \ldots, q_n, q \in Q$ ou bien $q \to q'$ où $q, q' \in Q$. (Ces dernières règles sont aussi appelées ϵ-transitions).*

$t \in T(\mathcal{F})$ est accepté *par l'automate $\mathcal{A} = (Q, Q_f, R)$ ssi il existe $q \in Q_f$ tel que $t \xrightarrow[R]{*} q$.*

Le langage $L(\mathcal{A})$ *accepté par un aaa $\mathcal{A}$ est l'ensemble des arbres acceptés par $\mathcal{A}$.*

Remarquons ici qu'un aaa peut être *non déterministe*, c'est à dire qu'il se peut que $t \xrightarrow[R]{*} q_1$ et $t \xrightarrow[R]{*} q_2$ sans que $q_1 = q_2$. Il se peut aussi que certaines expressions $f(q_1, \ldots, q_n)$ soient irréductibles. Autrement dit, et pour faire le lien avec la notion de reconnaissabilité déjà définie, les aaa correspondent en général à des algèbres finies dans lesquelles les fonctions peuvent être partiellement définies et/ou "multi-valuées".

Definition 4.5.4 *Un aaa est* déterministe *quand chaque terme de la forme $f(q_1, \ldots, q_{a(f)})$ ($f \in \mathcal{F}, q_1, \ldots, q_{a(f)} \in Q$) peut se réduire d'au plus une façon par R.*

Un aaa est complètement spécifié *lorsque, pour tout terme $t \equiv f(q_1, \ldots, q_{a(f)})$, ($f \in \mathcal{F}, q_1, \ldots, q_{a(f)} \in Q$) il existe un état $q \in Q$ tel que $t \xrightarrow[R]{*} q$.*

On peut alors faire le lien avec la reconnaissabilité:

Proposition 4.5.5 *Un langage est reconnaissable ssi il est accepté par un aaa déterministe et complètement spécifié.*

Preuve:
Si L est reconnaissable, $L = h^{-1}(E)$ où E est un sous-ensemble fini d'une $\mathcal{F}$-algèbre $\mathcal{A}$. On choisit alors pour ensemble d'états le domaine de $\mathcal{A}$ et E pour états finaux, R étant défini par $f(q_1, \ldots, q_{a(f)}) \to q$ ssi $f(q_1, \ldots, q_n) =_{\mathcal{A}} q$. L'automate obtenu est déterministe et complètement spécifié. Il accepte L.

Inversement, si L est accepté par l'automate $\mathcal{A} = (Q, Q_f, R)$, on considère la $\mathcal{F}$-algèbre de domaine $Q/_\sim$ où $\sim$ est la plus petite relation d'équivalence sur Q contenant les ϵ-transitions. Les opérations sont définies par $f(q_1, \ldots, q_{a(f)}) =_{\mathcal{A}} q$ ssi $f(q_1, \ldots, q_n) \xrightarrow[R]{*} q'$ et $q' \sim q$. L est alors l'image inverse de Q_f par le morphisme canonique. □

Cette proposition justifie aussi la confusion de notations entre algèbres finies et automates.

4.5.3 Réduction du non-déterminisme

Il s'avère en fait que tout langage accepté par un aaa est aussi accepté par un aaa déterministe et complètement spécifié. Il y a donc coincidence entre reconnaissabilité et acceptance par un aaa.

Proposition 4.5.6 *Pour tout aaa, il existe un aaa déterministe et complètement spécifié qui accepte le même langage.*

Preuve:
La preuve est semblable à celle des langages de mots. On commence par éliminer les ϵ-transitions. Puis on construit un automate déterministe $\mathcal{A}' = (Q', Q'_f, R')$à partir de $\mathcal{A} = (Q, Q_f, R)$ de la façon à ce que:

$$\left\{ \begin{array}{l} Q' \subseteq \mathcal{P}(Q) \\ Q'_f = \{S \in Q' \mid S \cap Q_f \neq \emptyset\} \\ \forall f \in \mathcal{F}.S_1, \ldots, S_{a(f)} \in Q' \Rightarrow \\ \qquad\qquad \{q' \mid \exists q_1 \in S_1, \ldots \exists q_{a(f)} \in S_{a(f)}, f(q_1, \ldots, q_{a(f)}) \xrightarrow[R]{} q'\} \in Q' \\ f(S_1, \ldots, S_{a(f)}) \xrightarrow[R']{} S' \Leftrightarrow \\ \qquad\qquad S' = \{q' \mid \exists q_1 \in S_1, \ldots, \exists q_{a(f)} \in S_{a(f)}, f(q_1, \ldots, q_{a(f)}) \xrightarrow[R]{} q'\} \end{array} \right.$$

$\mathcal{A}'$ accepte le même langage que $\mathcal{A}$, ce que l'on montre par récurrence sur la longueur de la réduction.

Enfin, pour obtenir un automate complètement spécifié (à partir d'un automate déterministe ou non), il suffit d'ajouter un état puits p et les règles

$f(q_1, \ldots, q_{a(f)}) \to p$ pour tous $q_1, \ldots, q_{a(f)} \in Q \cup \{p\}$ tels que $f(q_1, \ldots, q_{a(f)})$ est irréductible par $\underset{R}{\to}$. □

Corollaire 4.5.7 *L est reconnaissable ssi il est accepté par un aaa.*

Nous emploierons donc désormais indifféremment les deux terminologies.

4.5.4 Propriétés de clôture

Les ensembles reconnaissables d'arbres possède les mêmes propriétés de clôture que les ensembles reconnaissables de mots (et les preuves sont identiques et donc omises):

Proposition 4.5.8 *La classe des langages d'arbres reconnaissable est close par union, intersection et complémentaire*

Pour la côture par substitution (comme pour les clôtures par concaténation et étoile, que nous ne traiterons pas ici), il faut expliquer comment la substitution dans les mots se généralise aux arbres:

Definition 4.5.9 *Si A et B sont deux sous-ensembles de $T(\mathcal{F})$ et a est une constante de $\mathcal{F}$, l'ensemble $A[B/a]$ obtenu* par substitution de a par B dans A *est l'ensemble des arbres obtenu en remplaçant dans un arbre de A toutes les occurrences de a par un arbre de B (pas nécessairement le même à chaque occurrence).*

Proposition 4.5.10 *si A et B sont deux sous-ensembles reconnaissables de $T(\mathcal{F})$ et $a \in \mathcal{F}$ est une constante, alors l'ensemble d'arbres $A[B/a]$ est lui aussi reconnaissable.*

Preuve:

Etant donnés des aaa déterministes et complètement spécifiés $\mathcal{A}_1 = (Q_1, Q_f^1, R_1)$ et $\mathcal{A}_2 = (Q_2, Q_f^2, R_2)$ acceptant respectivement A et B, on construit un automate $\mathcal{A}$ acceptant $A[B/a]$ de la façon suivante: si $a \underset{R_1}{\longrightarrow} q_a$, alors $Q = Q_1 \uplus Q_2$, Q_f est Q_f^1 si $q_a \notin Q_f^1$ et $Q_f = Q_f^1 \uplus Q_f^2$ si $q_a \in Q_f^1$. Enfin, $f(q_1, \ldots, q_n) \underset{R}{\to} q$ ssi

- ou bien $q_1, \ldots, q_n, q \in Q_1$, $f \neq a$ et $f(q_1, \ldots, q_n) \underset{R_1}{\longrightarrow} q$

– ou bien $q_1, \ldots, q_n, q \in Q_2$ et $f(q_1, \ldots, q_n) \xrightarrow[R_2]{} q$

– ou bien $\{q_1, \ldots, q_n\} \subseteq Q_f^2 \uplus \{q_a\}$ et $f(q'_1, \ldots, q'_n) \xrightarrow[R_1]{} q$ où $q'_i = q_i$ si $q_i \in Q_1$ et $q'_i = q_a$ sinon.

Par construction, $b \in B$ ssi $b \xrightarrow[R]{*} q_f$ avec $q_f \in Q_f^2$. De plus, par définition, $u \in A[B/a]$ ssi il existe $v \in A$ et $b_1, \ldots, b_n \in B$ tels que $u = v[b_1]_{p_1} \ldots [b_n]_{p_n}$ où $p_1, \ldots, p_n$ sont les positions (disjointes) de a dans u. Alors $u \xrightarrow[R]{*} v[q_a]_{p_1} \ldots [q_a]_{p_n} \xrightarrow[R]{*} q_f$ avec $q_f \in Q_f$. Réciproquement, si $t \xrightarrow[R]{*} q_f \in Q_f$, soient $p_1, \ldots, p_n$ les positions maximales de t telles que $t|_{p_i} \xrightarrow[R]{*} q_a$ et $t[q_a]_{p_1} \ldots [q_a]_{p_n} \xrightarrow[R]{*} q_f$. Alors, $t[a]_{p_1} \ldots [a]_{p_n} \in A$ et $t|_{p_i} \in B$. □

4.5.5 Résultats de décidabilité

Proposition 4.5.11 *Soit $\mathcal{A}$ un aaa. On peut décider en temps polynômial (dans le nombre de transitions de $\mathcal{A}$) si $\mathcal{A}$ reconnait au moins un arbre (resp. reconnait un langage fini).*

<u>Preuve:</u>
Concernant la décision du vide, il s'agit de décider si l'on peut atteindre un état final. Pour cela, parcourant la liste des transitions, on marque les états q tels que $a \to q$ pour une constante a quelconque. Puis on itère: “marquer les états q tels qu'il existe une règle $f(q_1, \ldots, q_n) \to q$ où $q_1, \ldots, q_n$ sont les états marqués”. Lorsqu'un itération ne produit plus détats marqués, on arrête le calcul. Le langage est non vide si, à l'issue de ce calcul un état final au moins est marqué. Comme un état au moins est marqué à chaque itération, il y a au plus $|Q|$ itérations. L'algorithme est donc $O(|Q| \times |R|)$.

En ce qui concerne la finitude, il suffit pour chaque état atteignable et productif q, (i.e. marqué dans l'algorithme précédent) de tester s'il existe une réduction $t[q]_p \xrightarrow[R]{+} q$ où t ne contient que des états atteingables. Le langage est infini ssi il existe au moins un état (atteignable et productif) ayant cette propriété. Ce test s'effectue en temps quadratique en associant à chaque état la liste des états que l'on peut atteindre à partir de q. □

Nous énonçons le résultat suivant sans preuve (se reporter à [113] pour une preuve):

Théorème 4.5.12 *La décision de l'inclusion de deux langages d'arbres donnés par des aaa non déterministes est EXPTIME-complète.*

4.5.6 Pompage

On ajoute à l'alphabet $\mathcal{F}$ un symbole supplémentaire $\bullet$.

Pour $t \in T(\mathcal{F} \cup \{\bullet\})$ et $u \in T(\mathcal{F})$ (resp. $u \in T(\mathcal{F} \cup \{\bullet\})$) , on note $t \circ_p u$ le terme $t[u]_p \in T(\mathcal{F})$ (resp. $t[u]_p \in T(\mathcal{F} \cup \{\bullet\})$) lorsque $p \in \mathrm{Pos}(t)$ et $t|_p \equiv \bullet$. $\circ_p$ est indéfinie sinon.

Si $u \in T(\mathcal{F} \cup \{\bullet\})$ et $p \in \mathrm{Pos}(u)$, $u_p^k \in T(\mathcal{F} \cup \{\bullet\})$ est le terme défini par $u_p^0 \stackrel{\text{def}}{=} \bullet$ et $u_p^{k+1} \stackrel{\text{def}}{=} u[u_p^k]_p$.

Lemme 4.5.13 ((de pompage)) *Si A est un langage reconnaissable, alors il existe une constante K telle que, pour tout arbre t de profondeur $n > K$, il existe $t_1, u \in T(\mathcal{F} \cup \{\bullet\})$, $p \in Pos(u)$, $q \in Pos(t_1)$ et $t_2 \in T(\mathcal{F})$ tels que*

- *$u \not\equiv \bullet$, $u|_p \equiv \bullet$ et $t_1|_q \equiv \bullet$*
- *$t \equiv t_1 \circ_q u \circ_p t_2$*
- *Pour tout m, $t_1 \circ_q u_p^m \circ_p t_2 \in A$*

Preuve:
On choisit pour K le nombre d'états d'un automate $\mathcal{A}$ acceptant A. Si t est de profondeur strictement supérieure à K, soit p une position de t telle que $|p| = K$. Soit aussi ρ un calcul de l'automate sur t, c'est-à-dire un arbre $\rho(t)$ tel que $\mathrm{Pos}(t) = \mathrm{Pos}(\rho(t))$ et étiqueté par Q tel que $\rho(\Lambda) \in Q_f$ et, pour toute position $p' \in \mathrm{Pos}(t)$, si $t(p') = f$ a pour arité n, et si $\rho(t)(p') = q$, alors il existe une règle $f(q_1, \ldots, q_n) \to q$ de $\mathcal{A}$ telle que $\rho(t)(p \cdot i) = q_i$ pour tout i. Un tel calcul existe si et seulement si l'automate accepte t (voir aussi le paragraphe qui suit). Comme $|p| \geq K$, il existe deux préfixes distincts de p: p_1 et $p_1 \cdot p_2$ tels que $\rho(t)(p_1)$ et $\rho(p_1 \cdot p_2)$ sont étiquetés par le même état q. On choisit alors $u \equiv t|_{p_1}[\bullet]_{p_2}$, $t_1 = t[\bullet]_{p_1}$ et $t_2 = t|_{p_1 \cdot p_2}$. On a $t \equiv t_1 \circ_{p_1} u \circ_{p_2} t_2$ par construction. Comme $p_1 \neq p_1 \cdot p_2$, on a aussi $u \not\equiv \bullet$. Le terme $s_m \stackrel{\text{def}}{=} t_1 \circ_{p_1} u_{p_2}^m \circ_{p_2} t_2$ est accepté par $\mathcal{A}$ car on peut construire le calcul de $\mathcal{A}$ sur s_m de la façon suivante: pour toute position p'' de la forme $p_1 \cdot p_2^m \cdot p_3$, $\rho_m(s_m)(p'') \stackrel{\text{def}}{=} \rho(t)(p_1 \cdot p_2 \cdot p_3)$, pour toute position p'' de la forme $p_1 \cdot p_2^k \cdot p_3$

où $k < m$ et p_2 n'est pas un préfixe de p_3, $\rho_m(s_m)(p'') \stackrel{\text{def}}{=} \rho(t)(p_1 \cdot p_3)$; pour toute autre position de p'' de s_m, $\rho_m(s_m)(p'') \stackrel{\text{def}}{=} \rho(t)(p'')$. Les seuls problèmes qui peuvent survenir dans la vérification que ρ_m est un calcul sur s_m sont les positions d'interface. Mais $\rho(t)(p_1) = \rho(t)(p_1 \cdot p_2) = q$ implique que $\rho_m(s_m)(p_1 \cdot p_2^k) = q$. □

Comme d'habitude, le lemme de pompage sert d'abord à montrer que des ensembles d'arbres ne sont pas reconnaissables.

Exemple 4.5.14 Soit $\mathcal{F} = \{f, c\}$ (c est une constante et f est binaire). Soit A l'ensemble des arbres ayant deux sous-arbres identiques, i.e. l'ensemble $\{f(t,t) \mid t \in T(\mathcal{F})\}$. A n'est pas reconnaissable. En effet, pour toute constante K, pour tout $n > K$, on peut trouver un terme t_n de profondeur n tel que $t \in A$ et, pour tous $t_1, u \in T(\mathcal{F} \cup \{\bullet\}), p \in \mathrm{Pos}(u), q \in \mathrm{Pos}(t_1), t_2 \in T(\mathcal{F})$ tels que $u \not\equiv \bullet$, $u|_p \equiv \bullet$, $t_1|_q \equiv \bullet$, $t \equiv t_1 \circ_q u \circ_p t_2$, alors $t_1 \circ_q u \circ_p u \circ_p t_2 \notin A$.

Pour cela, on choisit pour t_n l'arbre $f(P_n, P_n)$ où P_n est le peigne gauche de profondeur $n-1$: $P_1 \stackrel{\text{def}}{=} c$ et $P_{n+1} \stackrel{\text{def}}{=} f(P_n, c)$. $t_n \in A$. Maintenant, $q \cdot p \neq \Lambda$. La première lettre de $q \cdot p$ est 1 ou 2. On peut supposer par symétrie que c'est 1. Alors, $t_1 \circ_q u \circ_p u \circ_p t_2|_2 \equiv P_n$ alors que la taille de $t_1 \circ_q u \circ_p u \circ_p t_2|_1$ est au moins $|P_n| + |u| - 1$. Ce qui montre bien que $t_1 \circ_q u \circ_p u \circ_p t_2 \notin A$ (les deux fils immédiats n'ont pas même taille).

4.5.7 Les ada

Les automates d'arbres étudiés jusqu' à présent sont des automates *ascendants*, c'est à dire qui synthétisent des états jusquà obtenir un état final. De façon symétrique, on peut faire calculer les automates de manière descendante en “faisant dégouliner” els états de la racine aux feuilles. Ces automates jouent d'ailleurs un rôle important dans deux cas: celui des automates d'arbres infinis pour lesquels il serait impossible d'utiliser des aaa et celui de l'évaluation paresseuse en programmation fonctionnelle (cf paracgraphe **??** car dans ce cas les sous-arbres peuvent être inconnus.

Definition 4.5.15 *Un* automate descendant d'arbres *(ada en abrégé) est un triplet formé d'un ensemble fini d'états, d'un sous-ensemble d'états initiaux et d'un ensemble fini de règles de transition de la forme:*

$$q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$$

si f est un symbole de fonction d'arité n et $x_1, \ldots, x_n$ sont des variables. Notons que l'ensemble de règles est ici un système de réécriture sur $T(\mathcal{F} \cup Q, \mathcal{X})$, les états étant des symboles unaires.

L'ada $\mathcal{A}$ accepte *t s'il existe un état initial q_i tel que $q_i(t) \xrightarrow[R]{*} t$ où R est l'ensemble des règles de transition.*

Le langage accepté par $\mathcal{A}$ est l'ensemble des termes acceptés par $\mathcal{A}$.

Nous ne nous étendrons pas sur ces automates car:

Proposition 4.5.16 *Un ensemble A est accepté par un ada ssi il est accepté par un aaa.*

Preuve:
Pour passer d'un aaa à un ada ou inversement, il suffit en fait de remplacer états finaux par états initiaux (ou inversement) et d'inverser le sens des règles de transition, en ajoutant des variables (ou en les retirant). □

Mais les ada possèdent un mauvais comportement vis-à-vis de la réduction du non-déterminisme et c'est pourquoi nous avons choisi de développer les aaa plutôt que les ada. En effet, l'exemple suivant très simple montre un langage qui ne peut pas être reconnu par un ada déterministe. (NB: un ada est déterministe si, pour tout terme t et tout état q il existe au plus une règle applicable à $q(t)$ et il y a un seul état initial).

Exemple 4.5.17 Soit $L = \{f(a,a), f(b,b)\}$. On suppose que l'alphabet ne contient que deux constantes a et b et un symbole binaire f. L est reconnu par l'aaa déterministe dont les règles de transition sont $a \to q_a$, $b \to q_b$, $f(q_a, q_a) \to q_f$, $f(q_b, q_b) \to q_f$ où q_f est le seul état final. En inversant le sens des règles, on obtient en particulier les deux règles $q_f(f(x_1, x_2)) \to f(q_a(x_1), q_a(x_2))$ et $q_f(f(x_1, x_2)) \to f(q_b(x_1), q_b(x_2))$. Deux règles sont ainsi applicables aux termes de la forme $f(t_1, t_2)$.

En fait, c'est irréparable car pour tout automate descendant déterministe acceptant $f(a,a)$ et $f(b,b)$, on doit avoir une unique règle de l'automate de la forme $q_i(f(x_1, x_2)) \to f(q_1(x_1), q_2(x_2))$ où q_i est un état initial. De plus il faut que $q_1(a) \to a$ et $q_2(a) \to a$ pour accepter $f(a,a)$ et il faut que $q_1(b) \to b$ et $q_2(b) \to b$ pour que $f(b,b)$ soit accepté. Mais alors, l'automate accepte aussi $f(a,b)$ et $f(b,a)$.

4.5.8 Automates avec tests entre frères

Pour avoir une classe de langages contenant le langage de l'exemple 4.5.14 et possédant encore toutes les bonnes propriétés des langages reconnaissables, on peut augmenter un peu l'expressivité des aaa.

Définissons tout d'abord une classe d'automates générale (automates avec tests d'égalité et de diségalité) qui est, elle, trop expressive, comme nous allons le voir, mais qui donne un cadre agréable pour plusieurs généralisations intéressantes des aaa.

Definition 4.5.18 *Un* automate ascendant contraint *(aac) est un triplet (Q, Q_f, R) où Q est un ensemble fini d'états, Q_f est un esous-ensemble d'états finaux et R est un ensemble fini de règles de la forme $f(q_1, \ldots, q_n) \xrightarrow{c} q$, où $q_1, \ldots, q_n, q \in Q$ et c est une conjonction d'égalités et de diségalités*

$$p_1 = q_1 \wedge \ldots \wedge p_n = q_n \wedge p'_1 \neq q'_1 \wedge \ldots \wedge p'_m \neq q'_m$$

où $p_1, \ldots, p_n, q_1, \ldots, q_n, p'_1, \ldots, p'_m, q'_1, \ldots, q'_m \in \mathbb{N}^+$ sont des mots non vides d'entiers.

Pour pouvoir définir un calcul de l'aac sur un terme, nous avons d'abord besoin de définir l'interprétation des contraintes intervenant dans les règles de l'automate.

Definition 4.5.19 *Les contraintes $p = q$ et $p \neq q$ intervenant dans les règles de l'automate sont des prédicats unaires sur $T(\mathcal{F})$ définis comme suit:*

- $t \models p = q$ *ssi* $p \in Pos(t)$ *et* $q \in Pos(t)$ *et* $t|_p \equiv t|_q$.
- $t \models p \neq q$ *ssi* $p \in Pos(t)$ *et* $q \in Pos(t)$ *et* $t|_p \not\equiv t|_q$

Les conjonctions sont interprétées comme d'habitude.

Remarquons que $p \neq q$ n'est pas la négation de $p = q$ car un terme peut ne satisfaire aucun de ces deux prédicats lorsque p ou q ne font pas partie de l'ensemble de ses positions.

Exemple 4.5.20 Soit $\mathcal{F} = \{a, f, g\}$ où a est une constante, f est unaire et g est binaire. La contrainte $12 \neq 2 \wedge 21 = 11$ est satisfaite par $t \equiv g(g(a, a), g(a, a))$ car $t|_{12} \equiv a \not\equiv g(a, a) \equiv t|_2$ et $t|_{21} \equiv a \equiv t|_{11}$. Les contraintes $12 \neq 2$ et $12 = 2$ ne sont ni l'une ni l'autre satisfaites par $g(f(a), a)$ car 12 n'est pas une position de $g(f(a), a)$.

Definition 4.5.21 *Un* calcul *de l'automate* $\mathcal{A} = (Q, Q_f, R)$ *sur le terme* $t \in T(\mathcal{F})$ *est une application* ρ *de* $Pos(t)$ *dans* Q *telle que, pour toute position* $p \in Pos(t)$, *si* $t|_p \equiv f(t_1, \ldots, t_n)$ *et* $\rho(p) = q$, *alors il existe une règle* $f(q_1, \ldots, q_n) \xrightarrow{c} \in R$ *telle que*

- *pour tout* $1 \leq i \leq n$, $\rho(p \cdot i) = q_i$
- $t \models c$

Remarquons qu'il peut n'exister aucun calcul de $\mathcal{A}$ sur t.

Exemple 4.5.22 Soit $\mathcal{F} = \{0, s, f\}$ où 0 est une constante, s est unaire et f est binaire. Soit $\mathcal{A}$ l'automate dont le seul état final est q_f et dont les règles sont $0 \to q_0$, $s(q_0) \to q_s$, $s(q_s) \to q_s$, $f(q_s, q_s) \xrightarrow[q]{1 \neq 2} f$.

Cet automate est déterministe (il existe au plus un calcul de $\mathcal{A}$ sur tout terme). Il n'existe aucun calcul de l'automate sur $s(f(0,0))$ et le calcul de $\mathcal{A}$ sur $f(s(0), 0)$ est $q_f(q_s(q_0), q_0)$.

Definition 4.5.23 *Un calcul* ρ *de* $\mathcal{A} = (Q, Q_f, R)$ *sur* t *est* réussi *lorsque* $\rho(\Lambda) \in Q_f$.

t est accepté *par* $\mathcal{A}$ *s'il existe un calcul réussi de* $\mathcal{A}$ *sur* t.

Le langage reconnu *apr* $\mathcal{A}$ *est l'ensemble des arbres acceptés par* $\mathcal{A}$.

Remarquons que cette définition coincide avec la définition du langage accepté par un aa lorsque $\mathcal{A}$ ne contient aucune contrainte.

Exemple 4.5.24 L'ensemble des arbres binaires complets sur $\mathcal{F} = \{f, a\}$ est accepté par l'aac dont les règles sont: $a \to q$ et $f(q,q) \xrightarrow[q]{1=2}$ où q est le seul état final.

Exemple 4.5.25 Un exemple très simple de langage reconnu par un aac (et qui sera réutilisé ensuite) est celui des instances d'un terme non linéaire. Par exemple, soit $t \equiv f(x, f(y, x))$. Supposons de plus que l'alhpabet se compose d'une constante a et du symbole binaire f. L'ensemble des termes $t\sigma \in T(\mathcal{F})$ est accepté par l'automate (non déterministe) dont le seul état final est q_f et les règles sont

$$\begin{array}{rcl@{\qquad}rcl}
a & \to & q & f(q,q) & \to & q_1 \\
f(q,q_1) & \to & q_1 & f(q_1,q) & \to & q_1 \\
f(q_1,q_1) & \to & q_1 & f(q,q_1) & \xrightarrow{1=22} & q_f \\
f(q_1,q_1) & \xrightarrow{1=22} & q_f & & &
\end{array}$$

Exemple 4.5.26 L'auomate sur $\mathcal{F} = \{e, a, b, \times\}$ (e, a, b sont des constantes et $\times$ est binaire) dont les règles sont $a \to q$, $b \to q$, $q \times q \xrightarrow{1 \neq 2} q_\times$, $q_\times \times q \xrightarrow{1 \neq 2 \wedge 12 \neq 2} q_\times$, $e \to q_e$ et dont tous les états sont finaux accepte l'ensemble des termes de $T(\mathcal{F})$ qui ne peuvent être réduits par le système de réécriture

$$\mathcal{R} = \left\{ \begin{array}{rclrcl} x \times x & \to & e; & x \times e & \to & x; \\ e \times x & \to & x; & x \times (y \times z) & \to & (x \times y) \times z; \\ (x \times y) \times y & \to & x & & & \end{array} \right.$$

Donnons rapidement les bonnes propriétés de ces automates:

Théorème 4.5.27 *la classe des langages reconnus par des aac est close par union, intersection, complémentaire. Tout langage de la classe est reconnu par un automate complètement spécifié et déterministe.*

Preuve:
Concernant la clôture par union et par intersection, les preuves sont les mêmes que pour les aaa (il suffit de prendre pour contrainte la conjonction des contraintes de chaque règles, dans le cas de l'intersection).

Pour la spécification complète, on ajoute un état puits. Pour tout symbole f d'arité n, et tous états $q_1, \ldots, q_n$ soient $c_1, \ldots, c_k$ les contraintes des règles dont le membre gauche est $f(q_1, \ldots, q_n)$. On ajoute alors la règle (ou les règles s'il y a des disjonctions): $f(q_1, \ldots, q_n) \xrightarrow{\neg c_1 \wedge \ldots \wedge \neg c_n} p$.

** a completer ** □

Mais malheureusement:

Théorème 4.5.28 *Le problème de savoir si un acc accepte au moins un arbre est indécidable.*

Preuve:
On réduit le problème de correspondance de Post (PCP). Si $w_1, \ldots, w_n$ et $w'_1, \ldots, w'_n$ sont les séquences de mots du PCP sur l'alphabet $\{a, b\}$, on considère un alphabet de symboles de fonction $\mathcal{F}$ contenant h (ternaire), a, b (unaires) et 0 (constante). Chaque mot $a_1 \ldots a_k$ du PCP est identifié à $a_1(\ldots(a_k(\bullet)\ldots)$, mot de $T(\mathcal{F} \cup \{\bullet\})$. Comme ces mots ne contiennent qu'une occurrence de $\bullet$, on pourra utiliser $\circ$ sans préciser la position.

Nous construisons maintenant l'aac $\mathcal{A} = (Q, Q_f, R)$ de la façon suivante:

- Q contient un état par préfixe d'un des mots w_i, w_j, ainsi que trois autres états: q_0, q et q_f. $Q_f = \{q_f\}$.
- R contient les règles suivantes:

$$\begin{array}{rcl@{\qquad}rcl} a(q_0) & \rightarrow & q_a & a(q_{w_i}) & \rightarrow & q_a \\ b(q_0) & \rightarrow & q_b & b(q_{w_i}) & \rightarrow & q_b \\ b(q_{w'_i}) & \rightarrow & q_b & a(q_{w'_i}) & \rightarrow & q_a \\ b(q_p) & \rightarrow & q_{b\cdot p} & a(q_p) & \rightarrow & q_{a\cdot p} \end{array}$$

pour tous les mots w_i, w'_i et pour tous les préfixes p d'un des mots w_i, w'_i tels que $a \cdot p$ (resp. $b \cdot p$) est aussi un préfixe de l'un des w_i ou de l'un des w'_i. R contient aussi les règles:

$$\begin{array}{rcl} h(q_{w_i}, q, q_{w'_i}) & \xrightarrow{1\cdot 1^{|w_i|}=2\cdot 1 \wedge 3\cdot 1^{|w'_i|}=2\cdot 3} & q \\ h(q_0, q_0, q_0) & \rightarrow & q \\ h(q_{w_i}, q, q_{w'_i}) & \xrightarrow{1\cdot 1^{|w_i|}=2\cdot 1 \wedge 3\cdot 1^{|w'_i|} \wedge 1=3} & q_f \\ 0 & \rightarrow & q_0 \end{array}$$

La règle de membre gauche $h(q_0, q_0, q_0)$ assure le début de la construction d'une séquence de Post. Les règles de membre gauche $h(q_{w_i}, q.q_{w'_i})$ assurent que l'on construit bien un successeur dans le problème de correspondance: la contrainte assure que le sous-terme à la position 1 est obtenu par concaténation de w_i au terme à la position $2 \cdot 1$ et que le sous terme à la position 3 est obtenu par concaténation de w'_i au sous-terme à la position $2 \cdot 3$. Enfin, on ajoute à cette condition la contrainte $1 = 3$ pour aller dans l'état final; cette contrainte exprime que l'on a réussi à construire deux mots identiques en concaténant successivement les paires de Post: on a une soultion de PCP. (Voir figure 4.20).

Les détails de la preuve que $\mathcal{A}$ accepte un arbre si et seulement si PCP a une solution sont laissés au lecteur. □

Cette indécidabilité nous oblige à restreindre la classe des automates à contraintes considérées. La classe qui suit est appropriée. La plupart des résultats la concernant peuvent être trouvés dans [12].

Definition 4.5.29 *Un* automate à contraintes entre frères *est un acc dans lequel les positions testées par les règles sont toutes de profondeur 1.*

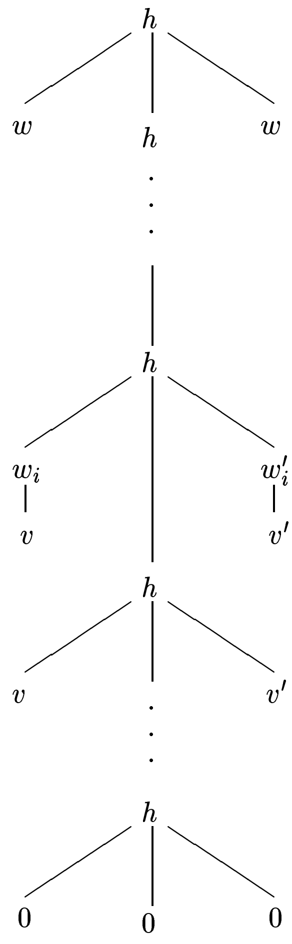


FIG. 4.20 - *Un acc reconnaissant les solution de PCP*

Les arbres binaires complets (cf. exemple 4.5.24) sont un exemple de langage accepté par un automate à contraintes ente frères alors que l'exemple 4.5.25 donne un langage accepté par un aac mais pas par un automate à contraintes entre frères.

Cette classe de langages possède maintenant les propriétés souhaitées:

Théorème 4.5.30 *La classe des langages reconnus par des automates avec tests entre frères est effectivement close par toutes les opérations Booléennes. Il est possible de réduire le non-déterminisme. Le vide des automates déterministes est décidable en temps polynômial.*

Preuve:
Les propriétés de clôture résultent de celles des aac: il suffit de remarquer que, dans les constructions effectuées pour les aac, si à l'origine les contraintes ne portent que sur des mots de longueur 1, il en est de même des automates résultant.

** a completer ** □

4.6 Théorème de Rabin

Un des plus beaux théorèmes de l'informatique théorique est la généralisation par M. O Rabin du théorème de Büchi à la logique monadique de second ordre à plusieurs successeurs. Les conséquences de ce résultat sont nombreuses et touchent plusieurs domaines de l'informatique théorique (comme les modèles du parallélisme, la théorie des modèles finis, la théorie de langages. Voir [120]). Ce résultat a aussi des conséquences sur la résolution de contraintes. Sa preuve est en fait une généralisation de celle du théorème de Büchi: on associe à chaque formule de S2S un automate d'arbres infinis (*automate de Rabin*). La décidabilité de S2S se ramène alors à prouver des propriétés de clôture des automates de Rabin, ainsi que la décidabilité du vide. Mai l'une des propriété de clôture (par complémentaire) est très difficile à établir. Bien que de nombreuses preuves aient été proposées après celle de Rabin, elles demandent toutes un investissement non négligeable et/ou des connaissnces en informatique théorique (par exemple sur la théorie des jeux de Gurevich-Harrington). Nous admettrons donc ce résultat ici. On pourra se faire une bonne idée du problème en consultant l'article de W. Thomas [120].

4.6.1 La logique monadique de second ordre à 2 successeurs: S2S

La syntaxe des formules de la logique est identique à celle de S1S, excepté que l'on peut désormais utiliser deux symboles de de successeurs S_1, S_2 au lieu du seul symbole S. Traditionnellement, au lieu de noter $S_i(x)$ on écrit $x \cdot i$. Autrement dit, les termes sont des mots $x \cdot w$ ou bien w où $w \in \{1,2\}^*$.

Si $\phi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ est une formule de S2S, $w_1, \ldots, w_m \in \{1,2\}^*$, $E_1, \ldots, E_n \subseteq \{1,2\}^*$ et σ est l'assignation $\{x_1 \mapsto w_1; \ldots; X_n \mapsto E_n\}$, $\sigma \models \phi$ est défini comme pour $S1S$; l'égalité est celle des mots de $\{1,2\}^*$.

On définit en S2S les relations de sous-ensemble, d'égalité d'ensembles, d'intersection, d'union, de partition, l'ensemble vide et le singleton comme en S1S. On peut également définir l'ordre préfixe des mots de façon analogue à S1S:

$$x \geq y \stackrel{\text{def}}{=} \quad \forall X.(y \in X \wedge \forall z \in X.z \cdot 1 \in X \wedge z \cdot 2 \in X) \Rightarrow x \in X$$

LA *logique monadique faible à deux successeurs WS2S* est obtenue en ne quantifiant que sur des ensembles finis.

Il est aussi possible de généraliser les définitions précédentes au cas de k successeurs: on obtient les logiques SkS et WSkS. Aucune difficulté supplémentaire n'apparait.

4.6.2 Les arbres comme modèles de (W)SkS

Comme pour S1S, étant donné un alphabet $\mathcal{F} = \{f_1, \ldots, f_n\}$, on peut associer à chaque arbre (fini ou infini) t construit sur $\mathcal{F}$ un tuples de variables $(X_1, \ldots, X_n)$ qui représentent les ensembles de noeuds de t étiquetés respectivement par $f_1, \ldots, f_n$. Ce qui permet, de voir un arbre sur $\mathcal{F}$ comme un modèle d'une formule $\phi(X_1, \ldots, X_n)$ de SkS, k étant l'arité maximale d'un symbole de $\mathcal{F}$.

Inversement, toute formule $\phi(X_1, \ldots, X_n)$ peut être interprétée dans l'ensemble des arbres étiquetés par $\{1, \ldots, k\}^n$ (ceci est semblable au cas des mots).

Definition 4.6.1 *Si $\mathcal{F}$ est un alphabet de symboles de fonction ayant n éléments, un sous-ensemble $\mathcal{S}$ de $IT(\mathcal{F})$ (resp $T(\mathcal{F})$) est* définissable *en SkS s'il existe une formule $\phi(X_1, \ldots, X_n)$ telle que $t \in \mathcal{S}$ ssi $t \models \phi(X_1, \ldots, X_n)$.*

4.6.3 Le théorème de Thatcher et Wright

Ce théorème établit l'équivalence entre langage d'arbres définissables en WS2S et aaa. On en déduit donc la décidabilité de WS2S.

Théorème 4.6.2 *$\mathcal{S}$ est définissable en WSkS ssi $\mathcal{S}$ est accepté par un aaa.*

La preuve de ce résultat est semblable à celle du théorème de Büchi.

Par propriétés des aaa, il en résulte:

Corollaire 4.6.3 *WSkS est décidable.*

4.6.4 Automates de Rabin et décidabilité de SkS

Dans le cas de S2S, la classe d'automates adéquate est celle des *automates de Rabin.* Un tel automate est défini par un ensemble fini d'états Q, un état initial, un ensemble de transitions (comme dans les ada) et un ensemble $\{(L_1, U_1), \ldots, (L_m, U_m)\}$ de paires de sous-ensembles de Q: les *paires acceptantes.* C'est ce mode d'acceptance qui caractérise les automates de Rabin.

Etant donné un arbre binaire complet infini t, un *calcul* de l'automate de Rabin $\mathcal{A}$ sur t est une application ρ de $\{1,2\}^*$ dans Q telle que, pour toute position p de t, $t(p) = f$ et $\rho(p) = q, \rho(p \cdot 1) = q_1, \rho(p \cdot 2) = q_2$ implique l'existence d'une règle $q(f(x,y)) \to f(q_1(x), q_2(y))$.

Un calcul est *réussi* si, pour tout mot infini $w \in \{1,2\}^\omega$, il existe une paire (L_i, U_i) tel qu'un ensemble fini seulement de préfixes de w soit étiqueté par un état de L_i, alors qu'un ensemble infini de préfixes de w sont étiquetés par un état de U_i.

t est *accepté* par l'automate de Rabin s'il existe un calcul réussi de l'automate sur t. Un ensemble d'arbres est Rabin-reconnaissable s'il existe un automate de Rabin qui accepte exactement les arbres de cet ensemble.

Exemple 4.6.4 ** a completer **

La plus grosse difficulté de S2S tient en la propriété de clôture suivante:

Théorème 4.6.5 (Rabin) *La classe des langages Rabin-reconnaissables est close par complémentaire.*

Il existe plusieurs preuves de ce résultat difficile qui sont soit des preuves directes, soit utilisant les jeux de Gurevich-Harrington.

La généralisation du théorème de Thatcher et Wright aux arbres infinis est moins difficile, mais pas triviale.

Théorème 4.6.6 (Rabin) *Un ensemble d'arbres est Rabin-reconnaissable ssi il est définissable en S2S.*

Il résulte des deux théorèmes précédants et des autres propriétésde clôture (plus faciles à établir):

Corollaire 4.6.7 *S2S est décidable.*

Cependant, la complexité de cette théorie est très élevée: pour toute machine de Turing M qui permet de décider S2S et pour toute fonction primitive récursive f, il existe une formule ϕ de S2S telle que le nombre d'étapes de calcul de la machine M sur ϕ est supérieur à $f(|\phi|)$ ($|\phi|$ est la taille de la formule).

4.7 Contraintes de sorte

Les contraintes de sorte permettent de spécifier des fonctions partielles dont le domaine de définition est un ensemble reconnaissable. Il existe différentes versions des contraintes de sorte que nous allons passer en revue. Mais pour commencer, établissons le lien avec les signatures avec sortes ordonnées.

4.7.1 Signatures avec sortes ordonnées

Les automates d'arbres permettent d'exprimer certaines propriétés très simples de typage, interprétées de façon ensembliste. Un exemple typique est celui des *sortes ordonnées* (voir par exemple définitions et propriétés comparées de deux sémantiques pour les algèbres avec sortes ordonnées dans [125]).

Definition 4.7.1 *Une* signature avec sortes ordonnées *est un couple formé d'un alphabet de symboles de fonction et d'un aaa.*

Pourquoi utiliser un autre nom et ne pas utiliser simplement des automates d'arbres? Cela a en fait des raisons historiques: les signatures avec sortes ordonnées n'ayant pas été imm'ediatement reconnues comme des automates d'arbres.

Exemple 4.7.2 Considérons la signature avec sortes ordonnées:

$$\begin{array}{rcl@{\qquad}rcl}
0 & \to & \text{nat} & s(\text{nat}) & \to & \text{nat} \\
\text{nat} & \to & \text{int} & p(\text{int}) & \to & \text{int} \\
s(\text{int}) & \to & \text{int} & eq(\text{nat},\text{nat}) & \to & \text{bool} \\
eq(\text{int},\text{int}) & \to & \text{bool} & +(\text{nat},\text{nat}) & \to & \text{nat} \\
+(\text{int},\text{int}) & \to & \text{int} & 0 & \to & \text{bool} \\
not(\text{bool}) & \to & \text{bool} & +(\text{bool},\text{bool}) & \to & \text{bool}
\end{array}$$

Tous les états sont ici finaux. Grâce à cet automate on peut par exemple tester si un terme est bien formé ou à quel(les) sorte(s) il appartient. Par exemple: $0 + eq(0, p(0) + s(0))$ est accepté dans la sorte bool.

Le calcul de l'automate sur un terme se comporte comme une interprétation abstraite dans le domaine des sortes.

4.7.2 Un système simple de contraintes de sortes: syntaxe

Les signatures avec sortes ordonnées peuvent alors être utilisées comme système de contraintes. Donnons-en tout d'abord la syntaxe.

Etant donné un ensemble fini S (symboles de sorte) et un ensemble fini de symboles de fonction $\mathcal{F}$, l'ensemble $\mathcal{ES}$ des *expressions de sortes* est le plus petit ensemble contenant $S, \bot_S, \top_S$ et tel que:

- si ζ_1, ζ_2sont des expressions de sortes, alors $\zeta_1 \wedge \zeta_2$, $\zeta_1 \vee \zeta_2$, $\neg\zeta_1$ sont des expressions de sorte.
- si $\zeta_1, \ldots, \zeta_n$ sont des expressions de sorte et f est un symbole de fonction d'arité n, alors $f(\zeta_1, \ldots, \zeta_{a(f)})$ est une expression de sorte

Les formules atomiques du système de contraintes sont des expressions $t \in \zeta$ où $t \in T(\mathcal{F}, \mathcal{X})$ et ζ est une expression de sorte.

Les formules du système de contraintes sont de la forme $\exists x_1, \ldots, x_n. \phi$ où ϕ est une formule propositionnelle sur un ensemble (fini) de formules atomiques.

4.7.3 Un système simple de contraintes de sorte: interprétation

Le domaine d'interprétation des contraintes est l'ensemble des termes $T(\mathcal{F})$. Il reste à donner l'interprétation des prédicats unaires $\in \zeta$.

Etant donné un aaa $\mathcal{A}$ dont les états contiennent l'ensemble S des symboles de sorte, on définit l'interprétation $[\![\cdot]\!]_{\mathcal{A}}$ des expressions de sorte de la façon suivante:

- lorsque $\zeta \in S$, $[\![\zeta]\!]_{\mathcal{A}}$ est l'ensemble des arbres accepté par $\mathcal{A}$, en remplaçant les états finaux de $\mathcal{A}$ par le seul état final ζ.
- $[\![\top_S]\!]_{\mathcal{A}} = T(\mathcal{F})$ et $[\![\bot_S]\!]_{\mathcal{A}} = \emptyset$
- $[\![\zeta_1 \wedge \zeta_2]\!]_{\mathcal{A}} \stackrel{\text{def}}{=} [\![\zeta_1]\!]_{\mathcal{A}} \cap [\![\zeta_2]\!]_{\mathcal{A}}$. $[\![\zeta_1 \vee \zeta_2]\!]_{\mathcal{A}} \stackrel{\text{def}}{=} [\![\zeta_1]\!]_{\mathcal{A}} \cup [\![\zeta_2]\!]_{\mathcal{A}}$, $[\![\neg\zeta]\!]_{\mathcal{A}} \stackrel{\text{def}}{=} T(\mathcal{F}) \setminus [\![\zeta]\!]_{\mathcal{A}}$
-
$$[\![f(\zeta_1, \ldots, \zeta_n)]\!]_{\mathcal{A}} \stackrel{\text{def}}{=} \{f(t_1, \ldots, t_n) \mid \forall i = 1, \ldots, n, t_i \in [\![\zeta_i]\!]_{\mathcal{A}}\}$$

Les formules atomiques sont alors interprétées de la façon suivante: σ est une solution de $t \in \zeta$ ssi $t\sigma \in [\![\zeta]\!]_{\mathcal{A}}$

4.7.4 Un système simple de contraintes de sortes: simplification des contraintes

Il reste à montrer comment simplifier ces contraintes et montrer, en particulier que l'ensemble des contraintes ayant une solution est récursif.

Pour cela, montrons tout d'abord un petit lemme sur la décomposition des contraintes de sorte.

Lemme 4.7.3 *Pour tout $f \in \mathcal{F}$ d'arité n, il existe une fonction f^{-1} de $\mathcal{ES}$ dans les sous-ensembles finis de $\mathcal{ES}^n$ telle que, pour tous $t_1, \ldots, t_n) \in T(\mathcal{F})$,*

$$f(t_1, \ldots, t_n) \in [\![\zeta]\!]_{\mathcal{A}} \;\; ssi \;\; \exists (\zeta_1, \ldots, \zeta_n) \in f^{-1}(\zeta),\; \forall i \in \{1, \ldots, n\},\; t_i \in [\![\zeta_i]\!]_{\mathcal{A}}$$

<u>Preuve:</u>
La preuve s'effectue par récurrence sur la structure des expressions de sorte. Lorsque ζ est un symbole de sorte, on utilise l'automate:

$$f^{-1}(\zeta) = \{(\zeta_1, \ldots, \zeta_n) \mid f(\zeta_1, \ldots, \zeta_n) \xrightarrow[R]{+} \zeta\}$$

où R est l'ensemble des règles de l'automate.

Il reste à définir systématiquement f^{-1} pour chaque construction possible:

- $f^{-1}(\top_S) = \{(\top_S, \ldots, \top_S)\}$ et $f^{-1}(\bot_S) = \{(\bot_S, \ldots, \bot_S)\}$

- $f^{-1}(g(\zeta_1, \ldots, \zeta_m)) = \emptyset$ si $f \neq g$
- $f^{-1}(f(\zeta_1, \ldots, \zeta_m)) = \{(\zeta_1, \ldots, \zeta_m)\}$
- $f^{-1}(\zeta \wedge \zeta') = \{(\zeta_1 \wedge \zeta'_1, \ldots, \zeta_n \wedge \zeta'_n) \mid (\zeta_1, \ldots, \zeta_n) \in f^{-1}(\zeta), (\zeta'_1, \ldots, \zeta'_n) \in f^{-1}(\zeta')\}$
- $f^{-1}(\zeta \vee \zeta') = f^{-1}(\zeta) \cup f^{-1}(\zeta')$

Pour la négation, il nous faut regarder un pas plus profond. si $\zeta \in S$, $f^{-1}(\neg\zeta)$ est construit de la façon suivante: on considère un l'automate déterministe $\mathcal{A}_d$ construit à partir de $\mathcal{A}$ comme en 4.5.6. Chaque état Q de $\mathcal{A}_d$ (i.e. ensemble fini d'états de $\mathcal{A}$) est identifié à l'expression de sorte

$$\bigwedge_{q \in Q} q \wedge \bigwedge_{q \notin Q} \neq Q$$

alors $f^{-1}(\neg\zeta)$ est l'ensemble des tuples $(\zeta_1, \ldots, \zeta_n)$ tels que $f(\zeta_1, \ldots, \zeta_n) \xrightarrow[R_d]{+} \zeta$ où R_d est l'ensemble des règles de $\mathcal{A}_d$. Maintenant, lorsque l'expression ζ n'est pas dans S, on définit $f^{-1}(\neg\zeta)$ pour chaque cas:

- $f^{-1}(\neg\neg\zeta) = f^{-1}(\zeta)$
- $f^{-1}(\neg(\zeta_1 \vee \zeta_2)) = f^{-1}((\neg\zeta_1) \wedge (\neg\zeta_2))$ (et on applique la définition ci-dessus)
- $f^{-1}(\neg(\zeta_1 \wedge \zeta_2)) = f^{-1}((\neg\zeta_1) \vee (\neg\zeta_2))$
- $f^{-1}(\neg\top_S) = f^{-1}(\bot_S)$ et $f^{-1}(\neg\bot_S) = f^{-1}(\top_S)$
- $f^{-1}(\neg g(\zeta_1, \ldots, \zeta_n)) = f^{-1}(\top_S)$ si $f \neq g$
- $f^{-1}(\neg f(\zeta_1, \ldots, \zeta_n)) = \{(\neg\zeta_1, \top_S, \ldots, \top_S), \ldots, (\top_S, \ldots, \top_S, \neg\zeta_n)\}$

La vérification du fait que cette construction vérifie les propriétés énoncées est fastidieuse et sans surprise. Nous renvoyons à [32] (lemme 1 page 175) pour les détails. □

Les règles de simplification de contraintes s'en déduisent. Elles sont reportées dans la figure 4.21. Nous supposons aussi désormais que les contraintes de sorte sont mainenues en forme normale disjonctive. Autrement dit, on suppose que ce sont des disjonctions de conjonctions de littéraux qui sont

$$\begin{array}{rcll}
t \in \zeta \wedge t \in \zeta' & \rightarrow & t \in (\zeta \wedge \zeta') & \\
t \in \zeta \vee t \in \zeta' & \rightarrow & t \in (\zeta \vee \zeta') & \\
\neg(t \in \zeta) & \rightarrow & t \in (\neg\zeta) & \\
f(t_1, \ldots, t_n) \in \zeta & \rightarrow & \displaystyle\bigvee_{(\zeta_1,\ldots,\zeta_n) \in f^{-1}(\zeta)} \bigwedge_{i=1}^{n} t_i \in \zeta_i & \\
t \in \zeta & \rightarrow & \bot & \text{Si } [\![\zeta]\!]_{\mathcal{A}} = \emptyset \\
\exists x. x \in \zeta \wedge \phi & \rightarrow & \phi & \text{Si } [\![\zeta]\!]_{\mathcal{A}} \neq \emptyset \text{ et } x \notin \mathcal{V}ar(\phi)
\end{array}$$

FIG. 4.21 - *Règles de simplification des contraintes de sorte*

des formules atomiques ou des négations de formules atomiques. Les conjonctions de littéraux étant le cas échéant précédées de quantifications existentielles.

Remarquons que ces règles de simplification sont correctes et que leur applicabiblité est décidable en vertu de la proposition 4.5.11. Il reste à montrer qu'elles termninent et inspecter les formules irréductibles.

Lemme 4.7.4 *Les règles de la figure 4.21 terminent.*

Preuve:
Nous ordonnons les formules dans lesquelles tous les prédicats d'appartenance ainsi que les quantifications ont été effacés par $\geq_{apo}$ étendant la préséance $f \succ \wedge \succ \vee$, pour tout symbole $f \in \mathcal{F}$.

La relation de réduction est contenue dans $>_{apo}$, ce qui se vérifie aisément. Par exemple,

$$f(t_1, \ldots, t_n) \succ_{apo} \bigvee_{(\zeta_1,\ldots,\zeta_n) \in f^{-1}(\zeta)} \bigwedge_{i=1}^{n} t_i$$

car $f \succ \wedge$, $f \succ \vee$ et $f(t_1, \ldots, t_n) >_{apo} t_i$ par propriété de sous-terme. □

Definition 4.7.5 *Les* formes résolues *pour le système de contraintes de sortes, sont $\bot$, $\top$ ou bien des disjonctions de formules*

$$x_1 \in \zeta_1 \wedge \ldots \wedge x_n \in \zeta_n$$

où $x_1, \ldots, x_n$ sont des variables distinctes et $[\![\zeta_1]\!]_{\mathcal{A}}, \ldots [\![\zeta_n]\!]_{\mathcal{A}}$ sont tous non vides.

Lemme 4.7.6 *Les formules irréductibles pour les règles de la figure 4.21 sont des formes résolues.*

Preuve:
Les contraintes irréductibles ne peuvent pas contenir de négation à cause de la règle $\neg(t \in \zeta) \rightarrow t \in \neg\zeta$. Elles ne peuvent pas contenir non plus de formule atomique $t \in \zeta$ où t n'est pas une variable, à cause de la règle de décomposition. Ensuite, elles ne peuvent pas contenir une conjonction $x \in \zeta \wedge x \in \zeta'$ car elle serait transformée en $x \in (\zeta \wedge \zeta')$. Enfin, $[\![\zeta]\!]$ doit être non vide pour tout ζ apparaissant dans une forme irréductible, à cause de la règle du vide et les contraintes irréductibles ne peuvent ainsi pas contenir de quantification existentielle à cause de la règle d'élimination de ces quantificateurs. $\square$

Lemme 4.7.7 *Une forme résolue (qui n'est pas $\perp$) a toujours au moins une solution.*

Preuve:
Immédiat: il suffit (pour une conjonction donnée) d'associer à chaque variable x un élément de $[\![\zeta]\!]$ si $x \in \zeta$ est dans la contrainte. $\square$

Il résulte des lemmes précédents:

Corollaire 4.7.8 *Les contraintes de sorte sont décidables.*

Il suffit en effet de réduire la formule en une forme résolue. Si la contrainte ne possède pas de variable libre, la forme résolue ne peut être que $\top$ ou $\perp$.

4.7.5 Un système simple de contraintes de sortes: exemple d'utilisation

Revenons à notre introduction: les automates d'arbres sont des signatures avec sortes ordonnées. L'utilisation de ces sortes ordonnées présente des avantages en programmation logique ou en réécriture ou en démonstration automatique ou encore dans les spécifications algébriques. Ces avantages sont de plusieurs nature: tout d'abord elles permettent plus d'expressivité, ou, dans le cas de la programmation, elles permettent d'exprimer plus simplement certaines propriétés. En second lieu, elles permettent d'isoler certains fragments simples de la logique qu'on considère et de les repousser dans des contraintes de sorte afin d'utiliser des techniques plus efficaces que les techniques générales. (Voir par exemple [59]). Ainsi, les variables intervenant dans les programmes, les spécifications ou les règles de réécriture sont contraintes à appartenir à un langage d'arbres reconnaissable. Il faut néanmoins pouvoir effectuer les opérations habituelles, i.e. ne perdre d'un côté ce qu'on gagne de l'autre par l'ajout de contraintes. Un exemple typique est l'unification, qui doit pouvoir s'étendre au cas des sortes ordonnées. Ce problème de l'unification avec sortes ordonnées a été étudié dans des dizaines d'articles. Mais le point de vue que nous adoptons permet d'en donner une version simple; les contraintes sont des conjonctions d'équations et de contraintes d'appartenance (ces dernières étant définies comme dans les paragraphes précédents). Le mélange des contraintes d'unification et des contraintes d'appartenance ne pose en fait aucun problème (on peut même généraliser encore comme nous le verrons dans le paragraphe suivant). Mais il faut reconnaitre les défauts de la méthode: de façon générale et pour un aaa quelconque, l'ensemble de formes résolues peut être exponentiel par rapport à la formule de départ, essentiellement à cause de la décomposition des contraintes de sorte. Lorsque la décomposition est "unique" ($f^{-1}(\zeta)$ a toujours au plus un élément), alors la complexité est la même que celle de l'unification sans sorte.

4.7.6 Extensions diverses du système de contraintes simple

On l'a vu, une première extension consiste à ajouter des équations. Mais on peut même en fait considére les formules du premier ordre arbitraires sur les formules atomiques $s = t$ et $s \in \zeta$. On peut aussi étendre les interprétations possibles aux langages d'arbres acceptés par des automates entre frères. En fait, presque rien ne change; il faut remplacer la règle de décomposition par une règle de la forme:

$$f(t_1, \ldots, t_n) \in \zeta \to \bigvee_{(A,B,(\zeta_1,\ldots,\zeta_n)) \in f^{-1}(\zeta)} \bigwedge_{i=1}^{n} t_i \in \zeta_i \bigwedge_{(i,j)\in A} t_i = t_j \bigwedge_{(i,j)\in B} t_i \neq t_j$$

Ici, bien sûr, $f^{-1}(\zeta)$ ne consiste plus seulement en un ensemble fini de tuples d'expressions de sortes, mais comprend en outre deux ensembles finis de paires d'indices.

La terminaison des transformations ne posera pas plus de problèmes que précedemment dans le cas de formules sans quantificateurs car tous les termes apparaissant dans le membre droit sont des sous-termes du terme de gauche.

On obtient ainsi la décidabilité d'une théorie des arbres finis avec contraintes d'appartenance, celles-ci étant interprétées comme des langages d'arbres acceptés par des automates à tests entre frères [32].

4.7.7 Contraintes de sortes sur les arbres infinis

De nombreux langages de programmation logique utilisent les contraintes pour des raisons que nous avons déjà expliquées. Ces langages considèrent aussi des interprétations dans les arbres infinis pour plusieurs raisons: ces interprétations permettent d'éviter le test d'occurrence dans l'unification (opération coûteuse et répétée). Elle permettent aussi de considérer des structures cycliques (un exemple typique est la logique des traits). Le langage PROLOG II [26] utilise depuis longtemps une interprétation dans les arbres infinis.

L'utilisation des contraintes de sorte permet d'ajouter une discipline de types simple à cet ensemble, certaines définitions de prédicats unaire pouvant être traduites dans cette structure de sorte.

** a completer [97] **

4.8 Théorie de l'entourage

Le problème à l'origine de l'étude de cette théorie (de ces contraintes devrions nous dire) est celui de la *réductibilité close*, lui même issu des preuves par récurrence dans les théories équationnelles.

La question centrale dans ce type de preuves (cf. [4, 74, 31]) est le suivant: étant donné un système de réécriture R et un terme t est-ce que tous les termes $t\sigma \in T(\mathcal{F})$ sont réductibles par R? On dit dans ce cas que t est *inductivement réductible* par R.

Exemple 4.8.1 Si $\mathcal{F} = \{s, 0\}$, 0 étant une constante et s un symbole unaire et $R = \{s(s(0)) \to 0\}$, alors le terme $s(s(x))$ n'est pas réductible par R mais il est inductivement réductible car toutes ses instances contiennent un sous-terme $s(s(0))$.

Cette propriété a été montrée décidable par D. Plaisted en 1985 [104]. Mais la preuve originale est très difficile. De nombreuses autres preuves ont ensuite été proposées. Une seule nous intéresse ici car elle s'obtient simplement comme cas particulier de résolution de contraintes d'entourage.

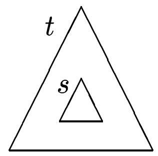


FIG. 4.22 - t *entoure* s

4.8.1 Les contraintes d'entourage: syntaxe et sémantique

Etant donnés n termes $l_1, \ldots, l_n \in T(\mathcal{F}, \mathcal{X})$, les formules atomiques de ce système de contraintes sont $\text{entoure}_{l_i}(t)$ où $t \in T(\mathcal{F}, \mathcal{X})$. Autrement dit, on dispose de n prédicats unaires sur les termes. Les contraintes sont alors des formules arbitraires du premier ordre sur ces formules atomiques.

Les formules atomiques sont interprétées à partir du prédicat binaire d'entourage défini sur $T(\mathcal{F}, \mathcal{X})$: $s \trianglelefteq t$ (t *entoure* s) s'il existe une substitution σ et une position $p \in \text{Pos}(t)$ telles que $t|_p \equiv s\sigma$ (cf figure 4.22).

Par exemple, $f(f(g(f(a,a), f(a,a)), a), a)$ entoure $f(g(f(x,a), y), x)$.

Si $t_1, \ldots, t_n \in T(\mathcal{F})$, $\sigma = \{x_1 \to t_1; \ldots; x_n \to t_n\} \models \text{entoure}_l(s)$ ssi $l \trianglelefteq s\sigma$. On interprète ainsi les contraintes d'entourage dans $T(\mathcal{F})$.

Il se trouve alors que le problème de réductibilité close s'exprime comme une contrainte particulière: t est inductivement réductible par $R = \{l_1 \to r_1; \ldots; l_n \to r_n\}$ ssi

$$\models \forall x.\text{entoure}_t(x) \Rightarrow (\text{entoure}_{l_1}(x) \vee \ldots \vee \text{entoure}_{l_n}(x))$$

4.8.2 Résolution des contraintes d'entourage

Les contraintes d'entourage sont décidables dans le cas général. La preuve consiste à associer à chaque formule un automate, puis à utiliser les propriétés de clôture et de décision de cette classe. Lorsque t est linéaire, le prédicat entoure_t est en fait un ensemble reconnaissable (par un aaa) d'arbres. Mais lorsque ce terme n'est pas linéaire, il peut être nécessaire de faire appel à des aac. Or nous avons vu (théorème **??**) que le vide est en général indécidable pour les aac. Mais il s'agit ici d'aac particuliers, pour lesquels *un nombre borné d'égalités doit être testé sur chaque branche.* Dans ce cas particulier, le vide est décidable, mais difficile à prouver (cf. [21, 20]). Nous nous contenterons donc ici d'étudier le cas où les termes sont linéaires et de définir la classe d'automates pour le cas général, en en admettant les propriétés essentielles.

** completer **

4.8.3 Langages de formes normales et automates à contraintes diségalitaires

.
** a introduire **

Théorème 4.8.2 *Le langage des termes qui n'entourent aucun des termes $l_1, \ldots, l_n$ est accepté par un acd déterministe dont le nombre des états (et le nombre de règles) est $O(2^{|l_1|+\ldots+|l_n|})$.*

Preuve:

** a completer **

4.9 Contraintes ensemblistes

Les contraintes ensemblistes ont été introduites comme approximation de programmes pour l'inférence et le contrôle de types ([**?**]). Plus récemment, des systèmes plus perfectionnés permettent de faire de l'analyse de programmes, notamment en programmation logique [64]. (Voir aussi la thèse de M. Tommasi [122]).

Montrons comment, en programmation logique, on peut utiliser des contraintes d'ensemble pour calculer des types. L'idée de base est que les types sont des ensembles (prédicats). Le type τ_P d'un prédicat P est alors une approximation (par le haut) de P: si $t \notin \tau_P$, alors le programme définissant P échouera sur t; si $t \in \tau_P$, le programme peut réussir sur t. Ceci correspond à l'esprit du calcul de type statique dans les langages de programmation: éviter autant que possible les erreurs à l'execution.

Exemple 4.9.1 [122] Soit le programme logique

$$\begin{array}{lcl} PRED(S(0),0) & & \\ PRED(S(S(X)),S(Y)) & \leftarrow & PRED(S(X),Y) \end{array}$$

Le type τ_{PRED} est le produit de deux ensembles de termes T_1 et T_2 tels que $S(0) \in T_1$, $S(t) \in T_1 \Rightarrow S(S(t)) \in T_1$, $0 \in T_2$, $t \in T_2 \Rightarrow S(t) \in T_2$. En fait, T_1 et T_2 peuvent être choisis comme les plus petits ensembles ayant cette propriété, ce qui nous donne pour T_1 l'ensemble des termes $\{S^{n+1}(0) \mid n \geq 0\}$ et pour T_2 l'ensemble des termes $\{S^n(0) \mid n \geq 0\}$.

L'analyse de programme procède d'une manière semblable en calculant des approximations des valeurs que peut peut prendre chacune des variables.

Exemple 4.9.2 Soit le morceau de programme impératif

```
While x ≠ 0 do
      x:= P(x);
      y:= S(S(x))
```

Où p est le prédecesseur et s le successeur.

On note X_0 et Y_0 les ensembles de valeurs que peuvent prendre x et y en entrée de ce morceau de programme et X_1, Y_1 les ensembles de valeurs que peuvent prendre x et y après chaque exécution du corps de boucle et enfin X_2, Y_2 les valeurs que peuvent avoir x et y après exécution du morceau de programme.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
X_0 \cap \overline{0} & \subseteq & X_1 & Y_0 & \subseteq & Y_1 \\
P(X_1) \cap \overline{0} & \subseteq & X_1 & S(S(Y_1)) & \subseteq & Y_1 \\
X_0 \cap 0 & \subseteq & X_2 & Y_0 \subseteq & Y_2 & \\
X_1 \cap 0 & \subseteq & X_2 & Y_1 \subseteq & Y_2 &
\end{array}$$

X_0 et Y_0 étant donnés, les valeurs que peuvent prendre x et y après execution de ce morceau de programme sont ainsi toujours contenues dans la plus petite solution du système de contraintes ci-dessus, en l'occurrence $X_2 = \{0\}$ et $Y_2\{s^{2n}(y) \mid y \in Y_0, n \geq 0\}$.

Dans le cas de programmes logiques, on calcule une approximation des valeurs que peuvent prendre les variables, ainsi que des ensembles de succès (plus petits point fixes) des prédicats.

Exemple 4.9.3 [122]

$$\begin{array}{rcl}
P(x) & \leftarrow & Q(x), R(x) \\
Q(a) & & \\
Q(f(y)) & \leftarrow & Q(y) \\
R(f(z)) & &
\end{array}$$

Si S_P, S_Q, S_R sont les approximations des ensembles de succès de P, Q, R et X, Y, Z les approximations des ensembles de valeurs que peuvent prendre x, y, z, on aura:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
S_P & \supseteq & X & S_Q & \supseteq & \{a\} \cup f(Y) \\
S_R & \supseteq & f(Z) & X & \supseteq & S_Q \cap S_R \\
Y & \supseteq & S_Q & Z \supseteq & \top &
\end{array}$$

On trouvera plus de détails ainsi que d'autres exemples d'application dans, par exemple, [64, 122].

4.9.1 Expressions ensemblistes

Etant donné un alphabet de symboles de fonction $\mathcal{F}$, et un ensemble de variables d'ensemble, les expressions ensemblistes sont formées de la manière suivante:

- les variables sont des expressions ensemblistes
- $\top$ et $\perp$ sont des expressions ensemblistes
- si η_1 et η_2 sont des expressions ensemblistes, alors $\eta_1 \cap \eta_2$, $\eta_1 \cup \eta_2$ et $\neg\eta_1$ (aussi écrit quelquefois $\overline{\eta_1}$) sont des expressions ensemblistes.
- si $\eta_1, \ldots, \eta_n$ sont des expressions ensemblistes et si f est un symbole de fonction d'arité n, alors $f(\eta_1, \ldots, \eta_n)$ est une expression ensembliste.

A ces constructions de base peuvent s'ajouter diverses constructions, comme la *projection*: si f est un symbole d'arité n, η est une expression ensembliste et $1 \leq i \leq n$, alors $f_i^{-1}(\eta)$ est une contrainte ensembliste.

Cependant, nous mettons à part ces extensions car nous ne montrerons l'algorithme de décision que pour le système de base.

4.9.2 Contraintes ensemblistes: syntaxe

Les formules atomiques que l'on considère sont des inclusions $\eta_1 \subseteq \eta_2$ où η_1 et η_2 sont des expressions ensemblistes.

Là encore, on peut aussi considérer certaines extensions comme la possibilité d'avoir la négation de telles formules atomiques. Les contraintes restent alors décidables, mais l'algorithme de décision est beaucoup plus compliqué.

Les contraintes ensemblistes sont des conjonctions de formules atomiques.

4.9.3 Interprétation des contraintes ensemblistes

Le domaine d'interprétation est l'ensemble des parties de $T(\mathcal{F})$. Chaque variable X est ainsi assignée à un sous-ensemble de $T(\mathcal{F})$.

$\top$ est interprété comme $T(\mathcal{F})$ lui-même alors que $\perp$ s'interprète comme l'ensemble vide.

Les différents connecteurs ainsi que les inclusions s'interprètent comme les opérations habituelles sur les ensembles.

Exemple 4.9.4 Soit $\mathcal{F} = \{f, a\}$ où f est unaire et a est une constante. Considérons la contrainte suivante:

$$\begin{cases} X & \subseteq & Y \\ f(Y) & \subseteq & \overline{Y} \\ f(\overline{Y}) & \subseteq & Y \end{cases}$$

$X = \emptyset$ et $Y = \{f^{2i}(a) \mid i \geq 0\}$ est une solution. Mais il y en a d'autres, en particulier en donnant d'autres valeurs pour X ou en choisissant pour Y l'ensemble des puissances impaires de f.

En fait, lorsque $\mathcal{F}$ contient seulement des symboles unaires et une constante, les contraintes d'ensemble s'expriment en SkS (k est le nombre de symboles unaires). Dans l'exemple 4.9.4 on peut ainsi associer à la contrainte la formule de S1S suivante dont les solutions sont celles de la contrainte:

$$\phi(X, Y) \stackrel{\text{def}}{=} \forall x.(x \in X \rightarrow x \in Y) \wedge (x \in Y \rightarrow S(x) \notin Y)) \wedge x \notin Y \rightarrow S(x) \in Y$$

Mais, d'une façon générale, il faut construire une nouvelle classe d'automates [61].

4.9.4 Automates avec variables libres

Definition 4.9.5 *Un* automate avec variables libres *(aavl) est un triplet (Q, Q_f, R) où Q est un ensemble fini d'états, Q_f est un tuple $(F_1, \ldots, F_k)$ d'ensembles d'états et R est un ensemble de règles de la forme*

$$f(q_1, \ldots, q_n) \rightarrow q$$

où $f \in \mathcal{F}$ est d'arité n.

k est appelé rang *de l'automate.*

Un calcul d'un aavl sur un terme $t \in T(\mathcal{F})$ est une application de $T(\mathcal{F})$ dans Q qui est compatible avec les règles de l'automate (comme précédemment).

Definition 4.9.6 *Si $\mathcal{A}$ est un aavl de rang n, le* langage $\mathcal{L}(\mathcal{A})$ accepté *par $\mathcal{A}$ est l'ensemble des n-uples d'ensembles $(L_1, \ldots, L_n)$ tels qu'il existe un calcul ρ et un élément $(F_1, \ldots, F_n) \in Q_f$ tels que, pour tout i, $L_i = \{t \in T(\mathcal{F}) \mid \rho(t) \in F_i\}$.*

Remarquons que, lorsque l'automate est déterministe, il existe au plus un calcul ρ. $\mathcal{L}(\mathcal{A})$ est alors vide ou bien réduit à un singleton, qui est un tuple de langages d'arbres reconnaissables par un aaa.

4.9.5 Contraintes d'ensemble et aavl

On peut maintenant établir la correspondance entre les aavl et les contraintes d'ensemble. Si C est une contrainte d'ensemble soit $E(C)$ l'ensemble des expressions $e \equiv f(e_1, \ldots, e_n)$ ($f \in \mathcal{F}$) et des variables apparaissant dans C. A chaque sous-ensemble S de $E(C)$, on associe un état q_S de l'automate. Intuitivement, en arrivant dans cet état, on aura un terme qui appartient à tous les ensembles de S (et pas aux autres).

Parmi les états ainsi construits, on ne garde que ceux qui correspondent à la contrainte: les contraintes d'ensemble peuvent être vues comme des formules propositionnelles sur les variables $E(C)$ (l'intersection étant identifiée à la conjonction, l'inclusion à l'implication etc...). Chaque sous-ensemble S de $E(C)$ correspond à une valuation ϕ_S de l'ensemble des formules propositionnelles sur $E(C)$ dans $\{\top, \bot\}$; $\phi_S(e)$ est $\top$ ssi $e \in S$. Q est alors l'ensemble des q_S tels que $\phi_S(C) = \top$.

** detailler un peu? **

Exemple 4.9.7 Supposons que C est la contrainte $f(X \cap Y, Y) \subseteq \bot \wedge \overline{X} \subseteq Y$. Q est l'ensemble $\{q_{\{X\}}, q_{\{Y\}}, q_{\{X,Y\}}\}$. $q_{\{f(X \cap Y, Y)\}}$ n'appartient pas à l'ensemble Q car la contrainte ne s'évalue pas à $\top$ si l'on remplace $f(X \cap Y, Y)$ par $\top$. De même, $q_\emptyset$ n'est pas dans Q car, si l'on remplace X par $\bot$ et Y par $\bot$, la contrainte $\overline{X} \subseteq Y$ n'est pas satisfaite.

Exemple 4.9.8 Reprenant l'exemple 4.9.4, l'ensemble Q est constitué de $q_{\{X,Y\}}, q_{\{X,Y,f(\overline{Y})\}}, q_{\{Y\}}, q_{\{Y,f(\overline{Y})\}}, q_{\{f(Y)\}}, q_\emptyset$.

Les règles de transition de l'automate sont alors les règles $f(q_{S_1}, \ldots, q_{S_n}) \rightarrow q_S$ telles que S est une union de l'ensemble des $f(e_1, \ldots, e_n)$ tels que $\phi_{S_i}(e_i) = \top$ pour tout i et d'un ensemble de variables.

Exemple 4.9.9 Continuons l'exemple 4.9.7 en supposant $\mathcal{F} = \{a, f\}$. Les règles de l'automate sont:

$$\begin{array}{ccc}
a \to q_{\{X\}} & a \to q_{\{Y\}} & a \to q_{\{X,Y\}} \\
f(q_{\{X\}}, q_{\{X,Y\}}) \to q_{\{X,Y\}} & f(q_{\{X\}}, q_{\{X,Y\}}) \to q_{\{X\}} & f(q_{\{X\}}, q_{\{X,Y\}}) \to q_{\{Y\}} \\
f(q_{\{X\}}, q_{\{X\}}) \to q_{\{X,Y\}} & f(q_{\{X\}}, q_{\{X\}}) \to q_{\{X\}} & f(q_{\{X\}}, q_{\{X\}}) \to q_{\{Y\}} \\
f(q_{\{X\}}, q_{\{Y\}}) \to q_{\{X,Y\}} & f(q_{\{X\}}, q_{\{Y\}}) \to q_{\{X\}} & f(q_{\{X\}}, q_{\{Y\}}) \to q_{\{Y\}} \\
f(q_{\{Y\}}, q_{\{X,Y\}}) \to q_{\{X,Y\}} & f(q_{\{Y\}}, q_{\{X,Y\}}) \to q_{\{X\}} & f(q_{\{Y\}}, q_{\{X,Y\}}) \to q_{\{Y\}} \\
f(q_{\{Y\}}, q_{\{X\}}) \to q_{\{X,Y\}} & f(q_{\{Y\}}, q_{\{X\}}) \to q_{\{X\}} & f(q_{\{Y\}}, q_{\{X\}}) \to q_{\{Y\}} \\
f(q_{\{Y\}}, q_{\{Y\}}) \to q_{\{X,Y\}} & f(q_{\{Y\}}, q_{\{Y\}}) \to q_{\{X\}} & f(q_{\{Y\}}, q_{\{Y\}}) \to q_{\{Y\}} \\
f(q_{\{X,Y\}}, q_{\{X\}}) \to q_{\{X,Y\}} & f(q_{\{X,Y\}}, q_{\{X\}}) \to q_{\{X\}} & f(q_{\{X,Y\}}, q_{\{X\}}) \to q_{\{Y\}}
\end{array}$$

Exemple 4.9.10 Continuons l'exemple 4.9.8. On obtient les règles suivantes:

$$\begin{array}{ccc}
a \to q_{\emptyset} & a \to q_{\{Y\}} & a \to q_{\{X,Y\}} \\
f(q_{\{X,Y\}}) \to q_{\{f(Y)\}} & f(q_{\{X,Y,f(\overline{Y})\}}) \to q_{\{f(Y)\}} & f(q_{\{Y\}}) \to q_{\{f(Y)\}} \\
f(q_{\{Y,f(\overline{Y})\}}) \to q_{\{f(Y)\}} & f(q_{\emptyset}) \to q_{\{X,Y,f(\overline{Y})\}} & f(q_{\emptyset}) \to q_{\{Y,f(\overline{Y})\}} \\
f(q_{\{f(Y)\}}) \to q_{\{X,Y,f(\overline{Y})\}} & f(q_{\{f(Y)\}}) \to q_{\{Y,f(\overline{Y})\}} &
\end{array}$$

Il reste à préciser les états finaux. On choisit $Q_f = (F_1, \ldots, F_n)$ où $F_i = \{S \in Q \mid X_i \in S\}$

Théorème 4.9.11 ([61]) *Pour tout système de contraintes d'ensemble, il existe un aavl qui accepte l'ensemble de ses solutions.*

** a completer **

4.9.6 Propriétés de clôture et de décision

Théorème 4.9.12 ([122]) *Le vide est décidable pour les aavls.*

En fait, ce résultat est démontré dans un cas plus général dans [122].

Théorème 4.9.13 ([122]) *La classe des langages reconnus par un aavl est effectivement close par union, intersection, cylindrification et projection. Elle n'est pas close par complémentaire*

Exemple 4.9.14 Pour montrer la non-clôture par complémentaire, il suffit de considérer l'exemple suivant: L'ensemble des langages L sur l'alphabet $\{a\}$ tels que $\exists k, \forall n > k, a^n \in L$ est reconnu par un aavl. Mais son complémentaire ne l'est pas. En effet, si L_1 est accepté par un aavl, et L_1 est infini, alors pour k suffisamment grand, il existe un $a^k \in L_1$ et, par un argument de pompage, il existe un langage L_2, également accepté par l'automate, et contenant a^{k+mt} où m est un entier arbitraire.

4.9.7 Solutions régulières

Théorème 4.9.15 *Soit $\mathcal{A}$ un aavl, alors*

1. *Si $(L_1, \ldots, L_n)$ est un tuple de langages réguliers, alors l'appartenance de $(L_1, \ldots, L_n)$ à $L(\mathcal{A})$ estd écidable*

2. *Si $L(\mathcal{A})$ est non vide, alors il contient un tuple de langages réguliers.*

[61, 22, 6]

4.10 Problèmes de compléments AC

** a completer ** formules avec AC [91]

4.11 Séquentialité de la réécriture

** a completer ** [69, ?]

4.12 Exercices

1. Les opérations de projection et de cylindrification sont elles inverses l'une de l'autres ? i.e. si l'on cylindrifie par rapport à i, puis on projette selon la ième composante, retrouve-t-on toujours le langage de départ?

2. Donner un automate de Büchi déterministe acceptant $a^*(ab)^\omega$ (cf. figure 4.5 pour un automate non-déterministe acceptant ce langage).

3. Montrer qu'il n'existe aucun automate de Büchi déterministe acceptant le langage $(a+b)^*a^\omega$ (cf. figure 4.7).

4. Le langage de l'exemple 4.5.26 est-il reconnu par un aaa? par un automate à tests entre frères? Justifier sa réponse.

5. Donner un aac déterministe qui accepte le langage de l'exemple 4.5.25.

6. Donner une formule de S2S $\phi(X)$ qui est satisfaite ssi X est un ensemble fini.

7. Donner une signature avec sortes ordonnées et une règle de réécriture contrainte $f(x) \rightarrow a \mid x \in \zeta$ où ζ est une sorte, f est un symbole unaire et a une constante tels que la relation de réduction associée n'est pas confluente.

8. Montrer que toute classe de langages d'arbres qui possède la propriété de décomposabilité du lemme 4.7.3 et pour laquelle le vide est décidable est contenue dans la classe des langages reconnaissables. (Autrement dit, les contraintes de sortes envisagées sont les seules que nous puissions envisager!)

9. Montrer comment réduire la décision des contraintes ensemblistes construites sur un alphabet de k symboles unaires (plus une constante) à SkS.

Chapitre 5

Combinaison de contraintes

Pour des systèmes de contraintes complexes (ou composés) il est naturel de chercher à décomposer la résolution en sous-problèmes plus simples qui peuvent être résolus séparément.

** a completer **

5.1 Purification et choix non-déterministes

5.2 Unification dans des mélanges de théories

[13, 15, 111]

5.3 Unification AC

[76, 16]
Unification OS [14]

5.4 Contraintes d'ordre

[37, 98, 28, 36]

Chapitre 6

Méthodes incomplètes ou hybrides

6.1 Problématique: l'exemple des domaines finis

[9]
Ca doit aller qque part:
[39, 87, 123, 8, 108] [88, 89, 86, 65, 3, 59] [63, **?**, 44, 49, 50] [83, 110]

Bibliographie

[1] *Tenth Annual IEEE Symposium on Logic in Computer Science*, San Diego, CA, June 1995.

[2] M. Adi and C. Kirchner. AC-unification race: the system solving approach. In *Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems, LNCS 429*, 1990.

[3] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, CA, 1992.

[4] L. Bachmair. *Canonical Equational Proofs.* Birkhäuser, Boston, 1991.

[5] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction, Saratoga Springs, NY, LNAI 607.* Springer-Verlag, June 1992.

[6] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th IEEE Symp. Logic in Computer Science, Montréal*, 1993.

[7] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1(4):329–349, Dec. 1985.

[8] R. Backofen and G. Smolka. A complete and recursive feature theory. Research Report RR–92–30, DFKI, Saarbrücken, Sept. 1992.

[9] R. Backofen and R. Treinen. How to win a game with features. In J.-P. Jouannaud, editor, *Proc. Int. Conf. on Constraints in Computational*

Logics, volume 845 of *Lecture Notes in Computer Science*, pages 320–335. Springer-Verlag, Sept. 1994.

[10] G. Birkhoff. On the structure of abstract algebras. In *Proc. Cambridge Phil. Society, 31*, 1935.

[11] W. Bledsoe. A new methode for proving presburger formulas, 1975.

[12] B. Bogaert and S. Tison. Equality and disequality constraints on brother terms in tree automata. In A. Finkel, editor, *Proc. 9th Symp. on Theoretical Aspects of Computer Science*, Paris, 1992. Springer-Verlag.

[13] A. Boudet. *Unification dans les Mélanges de Théories équationnelles.* Thèse de doctorat, Université Paris-Sud, Orsay, France, Feb. 1990.

[14] A. Boudet. Unification in order-sorted algebras with overloading. In Kapur [75].

[15] A. Boudet. Combining unification algorithms. *Journal of Symbolic Computation*, 16:597–626, 1993.

[16] A. Boudet. Competing for the AC-unification race. *Journal of Automated Reasoning*, 11:185–212, 1993.

[17] A. Boudet and H. Comon. Diophantine equations, presburger arithmetic and finite automata. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP'96)*, Lecture Notes in Computer Science, 1996.

[18] A. Boudet and E. Contejean. "Syntactic" AC-unification. In Jouannaud [73], pages 136–151.

[19] A. Boudet, E. Contejean, and H. Devie. A new AC-unification algorithm with a new algorithm for solving diophantine equations. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, pages 289–299. IEEE Computer Society Press, June 1990.

[20] A.-C. Caron, H. Comon, J.-L. Coquidé, M. Dauchet, and F. Jacquemard. Pumping, cleaning and symbolic constraints solving. In *Proc. Int. Conference on Algorithms, Languages and Programming*, Jerusalem, July 1994.

[21] A.-C. Caron, J.-L. Coquidé, and M. Dauchet. Encompassment properties and automata with constraints. In C. Kirchner, editor, *Proc. 5th. Int. Conf. on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 690, Montreal, Canada, 1993. Springer-Verlag.

[22] W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Proceedings of 9th IEEE Symposium on Logic in Computer Science*, Paris, 1994. in press.

[23] J. Christian and P. Lincoln. Adventures in associative-commutative unification. Technical Report ACA-ST-275-87, MCC, AI Program Austin, Austin, Oct. 1987.

[24] V. Chvàtal. *Linear Programming.* W.H. Freeman and Company, 1983.

[25] M. Clausen and A. Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8(1&2):201–216, 1989.

[26] A. Colmerauer. Equations and inequations on finite and infinite trees. In *FGCS'84 Proceedings*, pages 85–99, Nov. 1984.

[27] A. Colmerauer. Une introduction à PROLOG III. Unpublished draft, 1988.

[28] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[29] H. Comon. Complete axiomatizations of some quotient term algebras. *Theoretical Comput. Sci.*, 118(2), Sept. 1993.

[30] H. Comon. Constraints in term algebras (short survey). In T. R. M. Nivat, C. Rattray and G. Scollo, editors, *Proc. Conf. on Algebraic Methodology and Software Technology*, Univ. of Twente, 1993. Springer Verlag, series Workshop in Computing. Invited talk.

[31] H. Comon. Inductionless induction. In R. David, editor, *2nd International Conference in Logic For Computer Science: Automated Deduction. Lecture notes.*, Chambéry, July 1994. Univ. de Savoie.

[32] H. Comon. Sequentiality, second-order monadic logic and tree automata. In *Tenth Annual IEEE Symposium on Logic in Computer Science* [1].

[33] H. Comon and C. Delor. Equational formulae with membership constraints. *Information and Computation*, 112(2):167–216, Aug. 1994.

[34] H. Comon, M. Haberstrau, and J.-P. Jouannaud. Decidable properties of shallow equational theories. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 1992.

[35] H. Comon, M. Haberstrau, and J.-P. Jouannaud. Syntacticness, cycle-syntacticness and shallow theories. *Information and Computation*, 111(1):154–191, May 1994.

[36] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[37] H. Comon, R. Nieuwenhuis, and A. Rubio. Orderings, AC-theories and symbolic constraint solving. In *Tenth Annual IEEE Symposium on Logic in Computer Science* [1], pages 375–385.

[38] H. Comon and R. Treinen. Ordering constraints on trees. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, 11-13 Apr. 1994. Springer-Verlag. (Invited Lecture).

[39] E. Contejean and H. Devie. Résolution de systèmes linéaires d'équations diophantiennes. *Comptes-Rendus de l'Académie des Sciences de Paris*, 313:115–120, 1991. Série I.

[40] E. Contejean and H. Devie. An efficient algorithm for solving systems of diophantine equations. *Information and Computation*, 113(1):143–172, Aug. 1994.

[41] D. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7. 1972.

[42] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. *Information Processing*, 1983.

[43] M. Dauchet. *Cours d'informatique théorique.* Université de Lille, 1984.

[44] M. Dauchet. Rewriting and tree automata. In H. Comon and J.-P. Jouannaud, editors, *Proc. Spring School on Theoretical Computer*

Science: Rewriting, Lecture Notes in Computer Science, to appear, Odeillo, France, 1994. Springer-Verlag.

[45] M. Dauchet. Symbolic constraints and tree automata. In Jouannaud [73], pages 217–218.

[46] M. Davis. Unsolvable problems. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 567–594. North-Holland, 1977.

[47] C. Delor and L. Puel. Extension of the associative path ordering to a chain of associative-commutative symbols. In *Proc. 5th Rewriting Techniques and Applications, Montréal, LNCS 690*, 1993.

[48] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, Mar. 1982.

[49] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[50] M. Dincbas. Constraints, logic programming and deductive databases. In *Proc. of France-Japan Artificial Intelligence and Computer Science Symposium*, ICOT, Tokyo, Japan, Oct. 1986.

[51] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F.Berthier. The Constraint Logic Programming Language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, 1988.

[52] E. Domenjoud. AC unification through order-sorted AC1 unification. *Journal of Symbolic Computation*, 14(6):537–556, Dec. 1992.

[53] E. Domenjoud and A. P. Tomás. From elliot- mac mahon to an algorithm for general linear constraints on naturals. In U. Montanari, editor, *Principles and Practice of Constraint Programming*, Cassis, France, Sept. 1995. To appear.

[54] Elliot. On linear homogeneous diophantine equations. *Quartely J. of Pure and Apllied Maths*, 136, 1903.

[55] F. Fages. Associative-commutative unification. *Journal of Symbolic Computation*, 3(3), June 1987.

[56] M. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction, Austin, Texas*, pages 161–167, 1979.

[57] A. Fortenbacher. An algebraic approach to unification under associativity and commutativity. In *Proc. Rewriting Techniques and Applications 85, Dijon, LNCS 202*. Springer-Verlag, May 1985.

[58] M. Franzen and L. Henschen. A new approach to universal unification and its application to AC-unification. In *Lusk and Overbeek eds., Proc. 9th int. Conf. on Automated Deduction*. Springer-Verlag, May 1988.

[59] L. Fribourg. A new presburger arithmetic decision procedure based on extended prolog execution. Unpublished Draft, 1990.

[60] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 300–309, 1991.

[61] M. Gécseg and M. Steinby. *Tree Automata*. Akademia Kiadó, Budapest, 1984.

[62] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *Proc. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS*, 1993.

[63] M. Haberstrau. *ECOLOG: un Environnement pour la programmation en LOGique COntrainte*. Thèse de doctorat, Université Paris-Sud, Orsay, France, Dec. 1993.

[64] N. Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, 1992.

[65] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[66] J. Herbrand. Recherches sur la théorie de la démonstration. Thèse d'Etat, Univ. Paris, 1930. Also in: Ecrits logiques de Jacques Herbrand, PUF, Paris, 1968.

[67] A. Herold and J. H. Siekmann. Unification in abelian semi-groups. *Journal of Automated Reasoning*, 3(3):247–283, 1987.

[68] G. Huet. An algorithm to generate the basis of solutions to homogeneous linear diophantine equations. *Inf. Process. Lett.*, 7(3), Apr. 1978.

[69] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 415–443. MIT Press, 1991. This paper was written in 1979.

[70] J. M. Hullot. *Compilation de formes canoniques dans des théories équation–nelles.* Thèse de Doctorat, Univ. Paris Sud, 1980.

[71] J. Jaffar, J.-L. Lassez, and M. J. Maher. A logic programming language scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 441–467. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[72] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. Yap. the clp(r) language and system. I.B.M. Thomas J. Watson, Apr. 1988.

[73] J.-P. Jouannaud, editor. *First International Conference on Constraints in Computational Logics*, volume 845, München, Germany, Sept. 1994.

[74] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1), July 1989.

[75] D. Kapur, editor. *11th International Conference on Automated Deduction*, Lecture Notes in Computer Science vol. 607, Saratoga Springs, NY, June 1992.

[76] D. Kapur and P. Narendran. Double-exponential complexity of computing a complete set of *ac*-unifiers. In *Proc. 7th IEEE Symp. Logic in Computer Science, Santa Cruz*, June 1992.

[77] C. Kirchner. Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories equationnelles. Thèse d'Etat, Univ. Nancy, France, 1985.

[78] C. Kirchner. Computing unification algorithms. In *Proc. 1st IEEE Symp. Logic in Computer Science, Cambridge, Mass.*, pages 206–216, 1986.

[79] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on automatic deduction.

[80] C. Kirchner and F. Klay. Syntactic theories and unification. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[81] F. Klay. Undecidable properties of syntactic theories. In *Proc. 4th Rewriting Techniques and Applications, LNCS 488*, Como, Italy, 1991.

[82] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[83] D. Kozen. Logical aspects of set constraints. In *Proc. Conf. Computer Science Logic*, 1993.

[84] K. Kunen. Answer sets and negation as failure. In *Proc. 4th. Int. Conf. on Logic Programming, Melbourne*, pages 219–228, 1987.

[85] J. L. Lambert. Une borne pour les générateurs des solutions entières positives d'une équation diophantienne linéaire. *Comptes Rendus de l'Académie des Sciences de Paris*, 305:39,40, 1987. Série I.

[86] J.-L. Lassez and K. MacAloon. Independence of negative constraints. In *Proc. TAPSOFT 89 (Vol. 1), Barcelona, LNCS 351*. Springer-Verlag, Mar. 1989.

[87] J.-L. Lassez and K. MacAloon. A constraint sequent calculus. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[88] J.-L. Lassez, M. J. Maher, and K. G. Marriot. Unification revisited. In *Proc. Workshop on Found. of Logic and Functional Programming, Trento, LNCS 306*. Springer-Verlag, Dec. 1986.

[89] J.-L. Lassez and K. G. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):1–17, Sept. 1987.

[90] M. Livesey and J. H. Siekmann. Unification of bags and sets. Research report, Institut fur Informatik I, Universität Karlsruhe, West Germany, 1976.

[91] D. Lugiez and J.-L. Moysset. Complement problems and tree automata in AC-like theories. In *Proc. Symp. on Theoretical Aspects of Computer Science*, Würzburg, 1993. also available as techincal report CRIN 92-R-175.

[92] C. Lynch. Paramodulation without duplication. In *Tenth Annual IEEE Symposium on Logic in Computer Science* [1], pages 167–177.

[93] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, pages 348–357, July 1988.

[94] A. I. Mal'cev. Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems. Collected Papers. 1936-1967*, pages 262–289. North-Holland, 1971.

[95] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Prog. Lang. Syst.*, 4(2):258–282, 1982.

[96] A. R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. Manuscript, 1973.

[97] J. Niehren, A. Podelski, and R. Treinen. Equational and membership constraints for infinite trees. In *Proc. 5th Rewriting Techniques and Applications, Montréal, LNCS 690*, 1993.

[98] R. Nieuwenhuis. Simple LPO constraint solving methods. *Inf. Process. Lett.*, 47(2):65–69, Aug. 1993.

[99] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Bruckner, editor, *Proc. European Symp. on Programming, LNCS 582*, pages 371–389, Rennes, 1992. Springer-Verlag.

[100] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In Kapur [75], pages 477–491.

[101] R. Nieuwenhuis and A. Rubio. A precedence-based total AC-compatible ordering. In C. Kirchner, editor, *Proc. 5th Rewriting Techniques and Applications, Montréal, LNCS 690*. Springer-Verlag, June 1993.

[102] R. Nieuwenhuis and A. Rubio. AC-superposition with constraints: no AC unifier needed. In *Proc. 12th Int. Conf. on Automated Deduction*, LNAI, Nancy, June 1994. Springer-Verlag.

[103] T. Nipkow. Proof transformations for equational theories. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[104] D. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65:182–215, 1985.

[105] G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7, 1972.

[106] M. Presburger. Über die Vollständingen einer gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves*, Warszawa, 1929.

[107] M. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.

[108] C. Ringeissen. Combinaison de résolutions de contraintes. Thèse de Doctorat, Université de Nancy I, France, Dec. 1993.

[109] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[110] A. Rubio. Automated deduction with constrained clauses. PhD Thesis, Univ. de Catalunya, 1994.

[111] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computation*, 1990. Special issue on Unification.

[112] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[113] H. Seidl. Deciding equivalence of finite tree automata. *Siam Journal of Computing*, 19(3):424–437, 1990.

[114] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *J. of the Association for Computing Machinery*, 21(4):622–642, 1974.

[115] M. Stickel. A unification algorithm for associative-commutative functions. *J. ACM*, 28(3):423–434, 1981.

[116] M. E. Stickel. A complete unification algorithm for associative-commutative functions. In *Proceedings 4th International Joint Conference on Artificial Intelligence, Tbilissi (USSR)*, pages 71–76, 1975.

[117] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proc. 5th ACM Symp. on Theory of Computing*, pages 1–9, 1973.

[118] N. Suzuki and D. Jefferson. Verification decidability of presburger array programs. *J. ACM*, 1:191–205, Jan. 198027.

[119] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, CA, 1951.

[120] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 134–191. Elsevier, 1990.

[121] E. Tiden and S. Arnborg. Unification problems with one-sided distributivity. *Journal of Symbolic Computation*, 3:183–202, 1987.

[122] M. Tommasi. Automates et contraintes ensemblistes. Thèse de l'Univ. de Lille, Feb. 1994.

[123] R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–458, Nov. 1992.

[124] L. Vigneron. Associative commutative deduction with constraints. In *Proc. 12th Int. Conf. on Automated Deduction*, LNAI, Nancy, June 1994. Springer-Verlag.

[125] U. Waldmann. Semantics of order-sorted specifications. *Theoretical Comput. Sci.*, 94:1–35, 1992.