Terminaison, Satisfiabilité, Puisance de calcul d'une clause de Horn Binaire

Thèse de Doctorat en Informatique

Jean–**C**hristophe Routier

LIFL – UA CNRS 369
Université des Sciences et Technologies de Lille

1er February 1994.

Je remercie Max Dauchet de l'honneur qu'il me fait en présidant le jury de cette thèse qui constitue un aboutissement aux recherches dont il avait été l'initiateur il y dix ans.

Je remercie Leszek Pacholski et Manfred Schmidt–Schauß d'avoir accepté d'être rapporteurs de cette thèse. Je les remercie pour leurs remarques, critiques et suggestions. N'étant pas francophones, ils ont dû se baser sur la traduction anglaise que j'avais établie à leur intention. Mon anglais n'étant pas toujours très performant, leur tâche n'en a été que plus méritoire, je les remercie pour leur compréhension. [1]

Mes remerciements vont également à Hélène Kirchner, Sophie Tison et Bruno Courcelle qui me font l'honneur de participer au jury.

Je tiens à remercier également Jean–Paul Delahaye qui a assuré la responsabilité administrative de cette thèse. Son rôle ne s'est cependant pas limité à cela. Il s'est toujours montré disponible afin de me faire bénéficier de ses grandes compétences. Il est à l'origine d'une des preuves présentées ici.

Philippe Devienne et Patrick Lebègue ont assuré l'encadrement scientifique de cette thèse. Toute ma gratitude et mes remerciements à eux deux, leur disponibilité, leur compétence, leur soutien et leur sympathie ont été d'une aide inestimable. Ce travail constitue une conclusion aux travaux qu'ils avaient débutés avec Max Dauchet, ils ne peuvent en être dissociés. Encore merci à tous les deux !

Enfin, merci à Jörg Würtz de l'université de Saarbrücken qui a pris part à une partie de ce travail, aux membres de l'équipe Méthéol de Lille pour leur disponibilité et à Henri Glanc qui a assuré la repoduction de cette thèse.

[1] I thank Leszek and Manfred Schmidt–Schauß who have accepted to be the referees of this thesis. I thank them for their remarks, criticisms and suggestions. They are not French–speaking and have worked on the english translation I had established for them. My english is not always very efficient, therefore their task is even more commendable. I thank them for their understanding.

Plan de Lecture

L'*Introduction* permet de présenter des travaux en liaison avec notre sujet et propose un résumé. Celui–ci, après un premier résumé très succinct, décrit linéairement le contenu de cette thèse. Il présente dans leurs grandes lignes les résultats et leurs idées de preuve. Il convient pour un premier coup d'œil rapide. A partir de celui–ci, le lecteur peut directement se reporter à la section correspondante pour plus de détails puisque les titres de parties et chapitres sont repérés respectivement par des caractères **gras** et des PETITES MAJUSCULES. La bibliographie est brièvement annotée pour tenter de guider le lecteur.

Les deux premières parties présentent des notions générales en calculabilité et sur les langages à Clauses de Horn. Le contenu de certains chapitres peut être déjà connu et ils peuvent donc être passés par certains lecteurs. Puisque j'ai considéré que la lecture pouvait ne pas être linéaire, certains passages peuvent se répéter par endroit. En outre, le lecteur devrait pouvoir se référer à l'index situé à la fin de cet ouvrage (j'espère qu'il se révèlera suffisamment complet). Cependant, je pense que *le chapitre 3 ne doit pas être délaissé* dans la mesure où il présente des éléments incontournables.

La dernière partie est constituée des résultats et de leurs preuves. Le chapitre 9 est fortement lié au chapitre 3 et est également incontournable. Les chapitres 10, 11 et 12 présentent les résultats des articles [18], [19] et [20] respectivement et justifient le titre. Les sous–cas présentés aux chapitres 10 et 11 peuvent être ignorés dans un premier temps.

There exists an english version of this thesis that can be transmitted to every interested person[2].

Bon Amusement...

[2] "Il existe une version anglaise de cette thèse qui peut être communiquée à toute personne intéressée."

Table des matières

Introduction

L'étude des schémas minimaux dans les langages de programmation permet souvent

- de dégager des propriétés utiles pour l'amélioration de programmes de tailles plus conséquentes (amélioration des techniques de compilation par exemple),
- de mettre en valeur la puissance du langage.

La famille des langages à clauses de Horn, dont Prolog est le représentant le plus connu, est basée sur le principe de la résolution de J.A. Robinson [42]. Les travaux de A. Colmerauer et R. K. Kowalski sont à la base de ces langages qui s'appuient sur la logique du premier ordre[3]. Ils constituent avec les dialectes LISP les principaux langages de l'intelligence artificielle.

Dans cette famille, le schéma non trivial le plus simple est constitué d'un seul fait, d'une règle récursive binaire et d'un but :

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

L'intérêt de l'étude de cette structure est multiple. D'une part, tout programme à clauses de Horn peut être vu comme le recouvrement de plusieurs de ces programmes. En conséquence, extraire des propriétés intéressantes sur ces programmes élémentaires devrait faciliter la manipulation et la gestion des plus "gros". D'autre part, le schéma de clause "$gauche \leftarrow droit$" apparaît fréquemment dans les programmes logiques mais aussi dans les bases de données déductives. Enfin, la récursivité est à la base de cette structure, on peut donc en espérer un meilleur contrôle.

Après un tour d'horizon qui permettra la présentation de domaines en relation avec notre sujet de préoccupations, je présenterai brièvement le contenu des chapitres à venir.

[3]Voir la seconde partie intitulée "Les Langages à Clauses de Horn".

Tour d'Horizon

Dans les systèmes de réécriture, M. Dauchet a établi qu'il était possible de simuler les machines de Turing par une seule règle de réécriture linéaire gauche [11]. Dans le cadre des clauses de Horn, S.Å. Tärnlund [47] a montré que l'on peut simuler les machines de Turing à l'aide d'un programme bâti à partir uniquement de clauses de Horn binaires (le corps est réduit à un littéral) et unaires (des faits). H. Blair [1] propose une codification du même ordre n'utilisant que des clauses binaires récursives, c'est–à–dire dont les symboles de prédicats de la tête et du corps sont identiques. Dans le même ordre d'idée, A. Parrain, P. Devienne et P. Lebègue [40] ont écrit un méta–interpréteur n'utilisant que deux règles récursives binaires, un fait et un but. Aucun résultat équivalent n'utilisant qu'une clause binaire n'existait dans la littérature jusqu'à maintenant.

Mais concernant des petits programmes à une seule clause binaire, certains résultats ont pu être établis. D'abord à l'aide du problème de l'implication de clauses qui est assez proche, et ensuite grâce aux résultats obtenus à l'aide des graphes orientés pondérés. Je présenterai également d'autres travaux pouvant être rattachés, plus ou moins directement, à l'étude de ces programmes.

Implication de Clauses

L'étude de l'implication de clauses, en logique du premier ordre, est motivée en intelligence artificielle et en déduction automatique par l'élimination de redondances dans les systèmes de démonstration automatique, les programmes logiques ou les bases de connaissances ; il s'agit alors dans ces deux derniers cas de clauses de Horn. La suppression de clauses par implication est en effet plus forte que celle par subsomption et permet d'éviter d'avantage de redondances.

L'implication $A \Rightarrow B$ de deux clauses A et B est équivalente à la non–satisfiabilité de l'ensemble de clauses constitué de la clause A et des clauses unitaires closes[4] obtenues à partir de la négation de B. Dans le cas particulier où A et B sont en fait les clauses de Horn $A = \alpha \leftarrow \beta_1, \cdots, \beta_n$ et $B = \gamma \leftarrow \delta_1, \cdots, \delta_p$, le problème de l'implication $A \Rightarrow B$ est donc équivalent à la non–satisfiabilité de l'ensemble $\{\alpha \leftarrow \beta_1, \cdots, \beta_n; \leftarrow \gamma; \delta_1 \leftarrow; \cdots; \delta_p \leftarrow\}$, c'est–à–dire qu'il revient à étudier le problème de l'existence de solutions pour le programme :

$$\begin{cases} \delta_1 \leftarrow \ . \\ \ \vdots \\ \delta_p \leftarrow \ . \\ \alpha \leftarrow \beta_1, \cdots, \beta_n \ . \\ \leftarrow \gamma \ . \end{cases}$$

[4]Un terme est *clos* si il ne contient aucune variable. Une clause est dite close si tous les termes qui la composent le sont.

où γ et les δ_i sont des termes clos. Si ce programme n'a pas de solution, alors A implique B. Le problème de l'implication de clauses est donc équivalent à celui de la non–satisfiabilité de la classe de tels ensembles de clauses.

Remarquons que dans le cas où $n = p = 1$, nous sommes ramenés à notre schéma de programmes.

M. Schmidt–Schauß a montré que dans le cas où A est une clause à deux littéraux, le problème $A \Rightarrow B$ était décidable alors qu'au contraire, il devenait indécidable si A avait quatre littéraux ou plus [46]. Dans le même article il établit également l'indécidabilité de la satisfiabilité d'un ensemble de deux clauses de Horn binaires et de deux clauses unitaires closes. Un peu plus tard, le cas des clauses de Horn à trois littéraux a était montré également indécidable, y compris pour une clause A particulière, par J. Marcinkowski et L. Pacholski [36, 35].

Revenons sur le résultat de décidabilité obtenu par M. Schmidt–Schauß dans le cas où A est constituée de deux littéraux. En se plaçant dans le cadre des clauses de Horn, il entraine d'après ce qui précède que pour notre classe de programmes l'existence de solutions et la terminaison, qui sont deux des problèmes que nous étudierons, sont décidables lorsque le fait et le but sont clos. Je vais décrire le principe de la preuve : le seul cas intéressant est celui où α et β_1 sont unifiables, les autres sont en effet triviaux. On peut donc considérer que la clause est récursive. A partir du but γ, on applique récursivement la clause A un certain nombre de fois. Appelons γ^n le but courant après n itérations. Deux cas sont possibles, soit à partir d'un certain n_0 les γ_n ne sont que des instances de buts déjà obtenus, le problème est alors résolu, soit la taille des termes de γ^n croit infiniment en fonction de n. Dans ce cas, à partir du moment où la taille des termes de γ^n est supérieure à celle des termes correspondant de δ_1, aucune nouvelle solution ne sera possible, l'ensemble de clauses n'est donc satisfiable que si l'on a obtenu des solutions auparavant.

Le cas à quatre littéraux est montré indécidable grâce à l'insatisfiabilité d'un ensemble de deux clauses à deux littéraux et un nombre quelconque de clauses unitaires closes. La preuve pour trois littéraux est basée sur une analyse des arbres de dérivation à l'aide des systèmes de Thue.

Le corollaire du cas décidable est le premier résultat concernant nos programmes : la terminaison et la satisfiabilité (l'existence de solutions) sont décidables lorsque le fait et le but sont clos. Dans le même temps, M. Dauchet, P. Devienne et P. Lebègue avaient développé le formalisme des graphes orientés pondérés et étudié le cas linéaire.

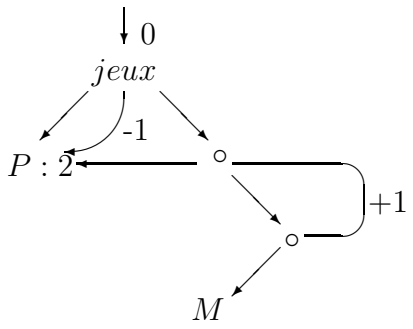
Graphes Orientés Pondérés

Informellement, un graphe orienté pondéré (ou GOP) [12, 15, 32] est un graphe orienté où :

- la racine est pondérée par un entier relatif,

- les arcs (orientés – ou flèches) sont pondérés par des entiers relatifs,
- les variables peuvent être périodiques

Exemple.



C'est un graphe orienté : il y a cinq noeuds étiquetés par des symboles de fonctions $(jeux, \circ)$ ou des variables (P, M), les arcs sont orientés, et ce graphe contient une boucle. Cependant, la racine $(jeux)$ est pondérée par 0, deux flèches sont pondérés par -1 et +1, et la variable P a pour période 2.

□

Le dépliage d'un GOP se fait en additionnant les poids au fur et à mesure du parcours des branches. Ceux–ci servent à l'indexation des variables dans la limite de l'intervalle d'interprétation. L'interprété d'un GOP est donc un arbre non–rationnel du fait du nombre infini des variables (indicées).

Les liens entre les GOPs et les clauses de Horn de la forme

$$gauche \leftarrow droite$$

sont importants. En effet, on peut en associer un à chacune de ces règles, il représente le plus petit terme constant (au renommage près) par rapport à la réécriture de cette clause. Pour rendre compte de n applications récursives, il suffit de considérer le dépliage pour un intervalle d'interprétation égal à $[1, n]$. De plus l'examen du GOP permet de déduire des propriétés pour la règle. En particulier, l'existence de boucles positives (resp. négatives) dans un GOP caractérise une décroissance (resp. croissance) globale des termes lors de la résolution.

A partir d'une analyse de propriétés liées à l'unification de GOPs et à leur interprétation, M. Dauchet, P. Devienne et P. Lebègue ont pu établir la décidabilité de la terminaison et de l'existence de solutions pour nos programmes dans le cas où le but et le fait sont linéaires[5] [12, 16]. Ils montrent en effet que l'on peut déterminer en fonction de la taille maximum des termes du fait et du but des conditions assurant la (non–)terminaison ou la (non–)satisfiabilité à partir d'un nombre calculable de réécriture récursive de la règle binaire.

Les GOPs ont été plus tard étendus aux systèmes pondérés d'équations [13, 17], plus puissants et plus simples d'utilisation, ils ont permis de retrouver ce résultat plus simplement. Les deux formalismes seront présentés plus précisément au chapitre 8.

[5]Un terme est dit *linéaire* quand aucune variable n'y apparaît deux fois.

Autres travaux

Je vais introduire maintenant rapidement plusieurs travaux en connection plus ou moins étroites avec notre sujet d'intérêt. Il s'agira d'abord de l'unification cyclique, puis des ρ–termes et enfin du problème de la satisfiabilité de formules.

L'Unification Cyclique.

Ce terme a été créé par W. Bibel, S. Hölldobler et J. Würtz dans [3]. Il fait référence au problème de l'existence de solutions pour les programmes à un fait, une clause récursive binaire et un but. C'est–à–dire l'unification du but – qui commence le cycle – et du fait – qui le termine – à travers la clause – qui l'engendre. Ils se sont en fait plus précisément intéressés aux cas où la clause $G \leftarrow D$ est telle qu'il existe une substitution σ pour laquelle on ait $\sigma G = D$ ou $G = \sigma D$. C'est ce qu'ils appellent respectivement des cycles gauche et droit. Pour certains cas particuliers de ces cycles, assez restrictifs d'après moi, ils prouvent la décidabilité de l'unification cyclique.

G. Salzer [45] a également abordé ce problème et l'a prouvé décidable pour une classe assez limitée de programmes. Il utilise une extension des ρ–termes présentés dans la section suivante.

Enfin P. Hanschke et J. Würtz ont établi dans [25] que le problème était indécidable dans le cas général. La preuve est basée sur une codification du problème de Post dans un programme à un fait, une clause binaire et un but. Elle est présentée à la section 11.4 page 98.

Les ρ–Termes.

H. Chen et J. Hsiang ont défini dans [6] un nouveau formalisme permettant de représenter finiment des termes infinis ou plus exactement des ensembles infinis de termes : les ρ–termes. Ils ont établi un algorithme d'unification pour ces termes et défini une extension du langage Prolog (ρ–Prolog) qu'ils prouvent avoir la même sémantique dénotationnelle que Prolog. Tout programme en ρ–Prolog a son équivalent en Prolog. Les buts et avantages de ces termes sont de permettre de représenter finiment des ensembles infinis de solutions, de permettre des recherches infinis, d'éviter la non–terminaison de programmes Prolog classiques, etc.

A la notion classique de termes, ils ajoutent ceux bâtis à l'aide du symbole spécial $\diamond$ et de variables dites *variables degré* tels que :

$$\Phi(succ^2(\diamond), N, zero)$$

Ce (ρ–)terme représente l'ensemble des nombres pairs :

$$\left\{succ^{2p}(zero) \mid p \in \mathbb{N}\right\}.$$

N est une variable degré qui indique que l'on parcourt tous les entiers. On autorise en fait des variables degré de la forme $\alpha * N + k$ où α et k sont des entiers naturels, et N parcourt $\mathbb{N}$.

Si ce formalisme et les preuves qui en résultent sont complexes, sa puissance d'expression semble élevée. Il peut notamment être considéré comme un moyen de manipuler plus efficacement la programmation logique avec contraintes $CLP(X)$. Ils devraient permettre également de représenter facilement les points fixes de certaines structures de données.

Satisfiabilité de Formules.

L'étude de la satisfiabilité d'ensemble de formules logiques a été l'objet de nombreux travaux. Les cas des ensembles avec un petit nombre de sous–formules sont particulièrement intéressants puisqu'ils permettent de restreindre l'étude de schémas plus importants. Ainsi, la satisfiabilité du cas à cinq sous–formules a été établie comme indécidable dans [22] puisqu'elle y est montrée équivalente au problème de l'arrêt des machines à deux compteurs. Cependant pour trois et quatre sous–formules, le problème reste ouvert, il est soulevé dans [22] et [46].

Les formules étudiées pas H.R. Lewis et W. Goldfarb se placent dans le cadre de la logique quantifiée pure, où l'on ne s'autorise aucun symbole de fonctions mais un nombre infini possible de constantes. Ce sont des formules de la forme :

$$(\forall\exists)^*(\mathcal{A}_1 \wedge \mathcal{A}_2 \vee \cdots \wedge \mathcal{A}_n) \tag{1}$$

où les $\mathcal{A}_i$ sont des atomes positifs ou négatifs.

Il est possible d'élargir le problème aux formules du premier ordre, c'est–à–dire avec symboles de fonctions. Parmi les formules à quatre sous–formules, on peut extraire celles de la forme :

$$\forall X_i(p(t_1) \wedge (p(t_2) \vee \neg p(t_3)) \wedge \neg p(t_4)) \tag{2}$$

où les X_i sont les variables apparaissant dans t_1, t_2, t_3 et t_4.

Le problème de la non–satisfiabilité correspond alors à celui de l'existence de solutions pour le programme :

$$\begin{cases} p(t_1) \leftarrow \ . \\ p(t_2) \leftarrow p(t_3) \ . \\ \leftarrow p(t_4) \ . \end{cases}$$

qui correspond à notre structure !

Notons que, si il est possible d'éliminer les quantificateurs existentiels dans (1) en faisant apparaitre des symboles de fonctions, il existe des contraintes fortes sur les termes fonctionnels qui apparaissent alors. Il n'est donc pas possible d'adapter immédiatement les résultats obtenus pour les formules (2) aux formules de type (1).

En Bref...

Notre objectif est l'étude des schémas de programmes minimaux (non triviaux) dans les langages à clauses de Horn. C'est–à–dire la classe des programmes :

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où $fait$, $gauche$, $droit$ et but sont des termes quelconques.

Nous nous intéresserons particulièrement aux problèmes de la terminaison et de la satisfiabilité (existence de solutions), ainsi qu'au pouvoir calculatoire de cette classe. Les deux premiers problèmes seront montrés indécidables dans le cas général. Des sous–cas dépendant de la (non–)linéarité des termes $fait$, $gauche$, $droit$ ou but seront également examinés, certains sont décidables, d'autres non. Nous en profiterons pour résoudre un problème ouvert sur la satisfiabilité de formules en *logique du premier ordre*, en l'occurrence les formules à quatre sous–formules. En effet, ce problème est équivalent à celui de la satisfiabilité de nos programmes, il est donc indécidable. Les preuves de ces résultats sont basées sur des travaux du mathématicien J.H. Conway [8] qui propose une codification arithmétique des machines de Minsky [38]. Nous examinerons comment ces travaux peuvent être exprimés en termes de clauses de Horn binaires.

Il sera ensuite montré que la puissance de calcul de cette classe de programmes est complète, c'est–à–dire qu'elle est la même que celles des machines de Turing. En effet, nous construirons un méta–interpréteur Prolog constitué uniquement d'un fait, d'une clause binaire et d'un but, établissant ainsi le résultat.

Il sera tout d'abord nécessaire de présenter les **Outils Théoriques** utilisés. Nous verrons en premier lieu, des Notions de Base en calculabilité qui permettront de préciser les termes *fonctions récursives*, *décidabilité* ou *indécidabilité*. Plutôt que d'utiliser les machines de Turing, nous nous appuierons sur la notion équivalente des programmes à entrées et sorties entières.

Un autre formalisme leur est équivalent : Les Machines de Minsky [38]. Il s'agit de machines à registres et compteurs permettant une certaine arithmétisation des machines de Turing, deux registres étant suffisants pour les simuler complètement. Nous insisterons sur une classe particulière que nous appellerons *machines de Minsky nulles*. Elles sont caractérisées par une image réduite à $\{0\}$. Parmi elles, on peut dégager les *machines α–linéaires*, c'est–à–dire les machines qui pour toute entrée n calculent la sortie, lorsqu'elle existe, en moins de $\alpha.n$ transitions. On peut en déduire la notion d'*ensembles récursivement énumérables linéaires* qui correspondent à leurs domaines.

Un formalisme fondamental dans l'établissement de nos résultats est celui des Fonctions de Conway. Elles sont une traduction en terme de fonctions

des machines de Minsky et correspondent à une généralisation du problème de Collatz. Il s'agit de toutes les fonctions g périodiques de $\mathbb{N}$ dans $\mathbb{N}$ telle que $\frac{g(n)}{n}$ reste entier. Elles ont la structure suivante :

$$\forall 0 \leq k \leq d-1, \text{ si } n \pmod d = k, g(n) = a_k n$$

où les $a_0, \cdots, a_{d-1}$ sont tous des nombres rationnels tels que $g(n)$ soit entier. A partir de l'étude des itérés $g^{(k)}(n)$, J.H. Conway a établi le théorème, fondamental pour tous nos travaux :

Théorème(Conway) *Soit f une fonction récursive partielle quelconque, il existe une fonction g telle que :*

1. *$\frac{g(n)}{n}$ est périodique (mod d) pour un certain d et prend des valeurs rationnelles.*
2. *$\forall n \in \mathbb{N}, n \in Dom(f)$ ssi $\exists (k,j) \in \mathbb{N}^* \times \mathbb{N}, g^{(k)}(2^n) = 2^j$.*
3. *$g^{(k)}(2^n) = 2^{f(n)}$ pour le plus petit $k \geq 1$ tel que $g^{(k)}(2^n)$ est une puissance de 2.*

Ce résultat peut être établi grâce à une traduction des transitions des machines de Minsky en terme de facteurs. En s'intéresssant plus particulièrement aux *fonctions de Conway nulles*, correspondant aux machines de Minsky nulles, on remarque qu'à partir d'un élément n du domaine d'une fonction g, la seule puissance de 2 qui puisse être atteinte par applications (ou itérations) successives de g sur 2^n est 2^0. De plus par itérations "négatives" (ou réciproques) à partir de 2^0 on atteint comme puissance de 2 tous les éléments du domaine et uniquement eux. Cela permet de définir à partir de ces fonctions nulles des *relations* (notées $\leftrightarrow_g$), que nous appellerons "*de Conway*", qui caractérisent tout ensemble récursivement énumérable Σ contenant 0, grâce à une fonction de Conway nulle g, ainsi :

$$\Sigma = \left\{ n \in \mathbb{N} \mid 2^n \leftrightarrow_g 2^0 \right\}$$

Maintenant que ces résultats généraux sont précisés, il est bon de présenter **Les Langages à Clauses de Horn**. Après des Généralités sur la logique du premier ordre, permettant de définir les termes courants et de présenter la résolution SLD, nous parlerons plus précisément de Programmation Logique en présentant les stratégies de résolution et la structure des *listes–différence*. Enfin, il sera possible d'introduire les Clauses de Horn Binaires qui nous intéresseront plus particulièrement ainsi que leurs particularités.

Il est temps maintenant d'entrer dans le cœur du problème avec l'étude de ces **Clauses Binaires Récursives**. Une rapide étude de la Problématique

permet de poser les différentes questions qui vont nous concerner. Il s'agira de l'étude du comportement d'une clause binaire récursive telle que

$$p(gauche) \leftarrow p(droit).$$

et du plus petit schéma non–trivial de programme à clauses de Horn :

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où $fait$, *gauche*, *droit* et *but* sont des termes quelconques. Dans un premier temps, les problèmes qui nous intéressent sont :

- "*Etant donnés une clause binaire récursive et un but, ce programme s'arrête–t–il ?*"
- "*Etant donnés un fait, une règle récursive binaire et un but, ce programme a–t–il au moins une solution ?*"

Ils avaient été montrés tous deux décidables lorsque *but* et $fait$ sont clos [46] ou linéaires [13]. Pour ce dernier résultat, M. Dauchet, P. Devienne et P. Lebègue ont développé les formalismes des Graphes et Systèmes Pondérés.

Bien qu'ils puissent sembler très éloignés l'un de l'autre, les Clauses Binaires et Fonctions de Conway peuvent être mises en relation étroites. Il est en effet possible de coder toute fonction de Conway à l'aide d'une clause binaire récursive et d'un but. C'est–à–dire que l'on peut extraire d'un tel programme une famille de variables X_i (correspondant aux différents renommages de la variable X au cours de la résolution) telle que ces variables soit liées par des relations de la forme

$$X_{ai+b} = X_{a'i+b'}$$

ce qui peut correspondre, pour un bon choix des entiers a, b, a' et b', à des relations de la forme

$$X_i = X_{g(i)}$$

où g est une fonction de Conway. Grâce à la caractérisation des ensembles récursivement énumérables contenant 0 établie précédemment, il est possible de produire, pour chacun des ensembles Σ récursivement énumérables contenant 0, une règle binaire et un but tels que le premier argument du but initial soit une liste dont le $(2^n)^{\text{ème}}$ élément est marqué par un symbole particulier (par exemple $\sharp$) si et seulement si n appartient à Σ.

On peut alors très facilement résoudre le Problème de l'Arrêt et montrer[6] :

[6]Ce résultat a été initialement publié dans [18].

Résultat 1 *Le problème de l'arrêt d'une clause de Horn binaire linéaire droite pour un but donné est indécidable, que la résolution soit faite avec ou sans le test d'occurence.*

Ce résultat est basé sur la caractérisation des ensembles récursivement énumérables et l'indécidabilité de l'appartenance d'un élément à un ensemble donné. En fait, il est possible de produire une clause particulière explicitement constructible et un but pour lesquels l'arrêt est indécidable. Par contre, dans le cas d'une règle linéaire gauche, le problème est décidable.

S'attaquant au Problème du Vide, on en prouve[7] également l'indécidabilité !

Résultat 2
Pour un programme de la forme

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où $fait$ et $droit$ sont des termes linéaires, l'existence d'au moins une solution est indécidable.

La preuve est basée sur une propagation linéaire de la marque $\sharp$. On écrit un programme pour lequel une solution à la $(2^n)^{\text{ème}}$ itération signifie que n n'appartient pas à l'ensemble récursivement énumérable linéaire codé par la clause binaire. Qu'un ensemble linéaire soit total étant indécidable, le tour est joué. Une autre preuve basée sur une codification du problème de Post a été établie indépendamment dans [25].

Ce résultat a un corollaire important en logique du premier ordre. En effet, l'existence de solution pour le programmes

$$\begin{cases} p(t_1) \leftarrow \ . \\ p(t_2) \leftarrow p(t_3) \ . \\ \leftarrow p(t_4) \ . \end{cases}$$

est équivalente à la *non–satisfiabilité* de la formule du premier ordre

$$\forall X_i, (p(t_1) \wedge (p(t_2) \vee \neg p(t_3)) \wedge \neg p(t_4)).$$

où les X_i sont les variables apparaissant dans t_1, t_2, t_3 et t_4.

D'où le théorème[6]

Résultat 3 *La classe des formules du premier ordre avec quatre sous–formules est indécidable.*

[7] Ce résultat a été initialement publié dans [19].

Il faut distinguer ce problème de celui des formules quantifiées pures pour lesquelles les cas à trois ou quatre sous–formules restent ouverts.

Toujours à propos du problème de l'existence de solutions, étudiant des cas particuliers, il est possible d'établir la décidabilité lorsque trois sur quatre des termes $fait$, *gauche*, *droit* ou *but* sont linéaires ou lorsque l'un seulement est clos. Par contre, même lorsque la règle est linéaire (*gauche* et *droit* sont linéaires), le problème demeure indécidable.

Ces deux problèmes s'étant révélés indécidables, il semble justifié de s'intéresser à celui de la PUISSANCE DE CALCUL de cette classe de programmes. Partant du *méta–programme* à un but, deux règles binaires et un fait de A. Parrain, P. Devienne et P. Lebègue, lui appliquant certaines transformations non triviales, il est possible de prouver l'existence d'un *méta–interpréteur* Prolog ayant la structure

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

d'où le théorème[8]

Résultat 4 *La classe des programmes à une clause de Horn binaire récursive et deux clauses unaires a la même puissance d'expression que les machines de Turing.*

Ce méta–interpréteur est le plus petit que l'on puisse construire dans les langages à clauses de Horn. Tout schéma plus simple n'exprimant que des trivialités. Ce théorème peut être considéré comme l'équivalent dans les langages à clauses de Horn du théorème de Böhm et Jacopini [2] pour la programmation impérative.

En CONCLUSION, si les résultats d'indécidabilité ne permettent pas d'extraire des propriétés intéressantes pour les programmes à clauses de Horn en général, le dernier résultat permet de mettre en valeur la puissance d'expression de ces langages.

[8]Ce résultat a été initialement publié dans [20].

Part I

Outils Théoriques

Dans ce travail, j'utiliserai les termes "décidabilité", "récursivement énumérable", et autres de la théorie de la calculabilité. Il est donc utile, et même indispensable, de les définir auparavant. Cependant loin de moi l'idée de faire une présentation détaillée de ces notions, un lecteur plus curieux que les autres ou moins familiers avec celles–ci, pourra consulter les différentes références indiquées. Ainsi pour une présentation informelle et abordable par la plupart se référer à [27]. Alors que pour une présentation théorique plus approfondie et plus mathématique consulter plutôt [28], [44] et [43], par exemple.

Je présenterai aussi le formalisme des machines de Minsky, peut-être moins connues que leurs grandes sœurs les machines de Turing. Puis nous parlerons de leurs cousines dans la théorie des nombres, les fonctions de Conway. Celles-ci sont à la base des preuves des principaux résultats qui seront présentés par la suite, je m'attarderai donc un peu plus sur leur présentation.

Chapitre 1

Notions de Base

Les définitions suivantes sont fondamentales pour l'informatique en général. Elles précisent ce que l'informatique (c'est–à–dire les programmes) peut, ou ne peut pas, accomplir.

A la base de ces notions se trouvent la récursivité et les machines de Turing. Celles–ci se composent d'états, de transitions permettant de passer d'un état à un autre sous certaines conditions et d'une bande sur laquelle on peut lire ou écrire des informations dépendantes de l'état courant. Leur formalisme est un peu lourd à présenter. Comme il n'est pas au centre de nos intérêts, je le remplacerai par un autre équivalent : celui des programmes à i entrées entières et j sorties entières. Suivront ensuite des définitions et résultats classiques en théorie de la calculabilité.

1.1 Programmes à i Entrées et j Sorties Entières

Je reprends la présentation faite dans [14]. Elle permet de présenter plus intuitivement, bien que moins rigoureusement, les différents résultats qui nous intéressent.

Definition 1 Les *programmes à i entrées entières et j sorties entières* sont définis par :

- Un programme à i entrées entières est un programme ayant ses i premières instructions telles que : "lire un entier n", n'ayant aucun autre ordre de lecture et sans retour possible à une de ces instructions.
- Un programme à j sorties entières est un programme qui se termine par exactement j instructions telles que : "écrire n", suivies de "stop" et n'ayant aucun autre ordre d'écriture ni d'arrêt.

Proposition 1 *L'ensemble Π_i^j des programmes à i entrées et j sorties entières est dénombrable.*

Preuve. On note Σ l'alphabet fini du langage utilisé pour écrire les programmes de Π_i^j. Σ^* désigne alors l'ensemble de toutes les suites finies d'éléments de Σ. Il s'agit en fait de tous les textes que l'on peut construire à partir de cet alphabet. Parmi ceux–ci, un certain nombre correspond à des programmes de Π_i^j. Nous allons les numéroter et créer ainsi une bijection entre Π_i^j et $\mathbb{N}$, nous aurons alors prouvé le résultat.

Pour réaliser cette numérotation, nous allons d'abord examiner un à un les textes de longueur 1 de Σ^* (ils sont en nombre fini puisqu'il s'agit en fait de Σ lui–même), si il y en a parmi eux qui se trouve dans Π_i^j on les numérote de un en un à partir de 0 (en fait nos programmes finissant nécessairement par *stop* – de longueur 4 – il y a peu de chance d'en trouver). Puis on examine les textes de longueur 2 de Π_i^j et on les numérote de la même manière en commençant la numérotation là où s'était arrêtée la précédente. Et on continue le processus avec les textes de longueur 3, 4, ..., n, etc. □

Nous avons ainsi défini une numérotation de Π_i^j. D'autres sont bien sûr possibles. Nous supposerons que nous en avons fixé une pour Π_1^1 une fois pour toute. L'idée d'une telle numérotation est dûe au mathématicien Kurt Gödel. On parlera donc par la suite de *nombre de Gödel* d'un programme pour se référer à l'entier qui le caractérise dans la numérotation choisie. Je noterai π_n le programme de Π_1^1 dont le nombre de Gödel est n.

Nous pouvons maintenant établir l'existence de *programmes universels*, c'est–à–dire de programmes qui prenant pour paramètres un programme π et une entrée e, produisent comme sortie la valeur de π pour l'entrée e.

Proposition 2 *Il existe un programme U de Π_2^1 tel que $U(n,e) = s$ si $\pi_n(e) = s$ et $U(n,e)$ ne termine pas si $\pi_n(e)$ ne termine pas.*

Esquisse de Preuve. Le programme U pourrait être de la forme suivante :

- lire n
- lire e
- retrouver π_n
- calculer $\pi_n(e)$
- écrire $\pi_n(e)$
- stop

L'étape "retrouver π_n" est nécessairement finie (et donc possible), il suffit de reconstituer la procédure de numérotation choisie et de s'arrêter sur le texte numéro n. On vérifie donc bien qu'un tel programme U vérifie la proposition. □

L'existence de programmes universels est importante. Ils traduisent la possibilité de décrire un langage à l'aide de lui–même, cela s'appelle la méta–programmation. On peut ainsi considérer la puissance d'un langage à sa capacité de méta–programmation. Dans les langages interprétés, tels que Prolog ou Lisp, un programme universel est souvent appelé meta–interpréteur. Dans un chapitre ultérieur, nous nous attacherons à montrer l'existence d'un méta–interpréteur Prolog de très petite taille.

1.2 Fonction Récursives

La notion de fonctions récursives est cruciale, celles–ci désignent en effet ce qui est programmable. Elles traduisent donc à la fois la puissance et les limites des langages de programmation. Cette affirmation est appelée *Thèse de Church*.

Definition 2 Une fonction f est *récursive* (ou *calculable*), si il existe un programme π de Π_1^1 tel que pour tout entrée entièré n, π s'arrête et écrit la valeur de $f(n)$ si n appartient au domaine de définition de f, et ne s'arrête pas sinon.

Definition 3 Une fonction est dite *récursive totale* si elle est définie sur tout $\mathbb{N}$ et *récursive partielle* sinon.

Remarque 1 Les fonctions partielles correspondent à ce que l'on nomme généralement en mathématique effectivement fonctions alors que les fonctions totales sont appelées habituellement applications.

Il existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ qui ne sont pas récursives (donc non calculables par un programme). Le pouvoir d'expression des langages de programmation est donc limité. Quel que soit le langage, dès qu'il est assez puissant (qu'il a suffisamment de primitives de base), il permet d'exprimer toutes les fonctions récursives, mais quoiqu'on puisse lui ajouter il ne calculera rien de plus. Les langages de programmation sont donc tous potentiellement équivalents, il reste évidemment que leurs capacités à résoudre plus ou moins facilement tel ou tel problème diffèrent. Il est intéressant d'étudier à quoi correspond le "suffisamment" dans "suffisamment de primitives de base". Pour les langages impératifs, il a été établi qu'une seule boucle `tant–que` était suffisante. Ce résultat est attribué un peu abusivement à C. Böhm et G. Jacopini [2], l'historique détaillé de ce théorème est présentée dans [26].

A partir de la définition précédente, on peut établir celles d'ensembles récursifs et récursivement énumérables.

Definition 4 Un ensemble Σ est *récursif* si il existe une fonction récursive totale qui pour tout n écrit en un temps fini 1 si $n \in \Sigma$ et 0 si $n \notin \Sigma$.

Tout ensemble fini est évidemment récursif, l'ensemble des puissances de 2 est lui aussi récursif, etc.

Definition 5 Un ensemble Σ est *récursivement énumérable* si il existe une fonction récursive partielle dont Σ est le domaine (de définition).

1.3 Décidabilité

Definition 6 On dit qu'une propriété est *décidable* si il existe un programme qui, pour toute entrée paramétrant cette propriété, écrit en un temps fini 1, si la propriété est vraie pour cette entrée, et 0 sinon. C'est–à–dire si l'ensemble des valeurs qui vérifient cette propriété est récursif.

Il existe de très nombreux exemples de propriétés décidables : déterminer si un nombre est premier, si il est pair, si la somme de ses diviseurs vaut 2985, etc.

Definition 7 Une propriété est dite *semi–décidable* si il existe un programme qui, pour toute entrée paramétrant cette propriété, écrit en un temps fini 1 si la propriété est vraie pour cette entrée et qui ne s'arrête pas sinon. C'est–à–dire si l'ensemble des valeurs qui vérifient cette propriété est récursivement énumérable.

Le terme "*indécidable*" est souvent utilisé à la place de "semi–décidable". Je m'autoriserai cet abus. Voici quelques exemples de propriétés indécidables, pardon semi–décidables :

- déterminer si, étant donnés un programme et une entrée, ce programme s'arrête pour cette entrée.
- décider si, étant donnés une fonction et un entier, cet entier appartient au domaine de la fonction.
- déterminer si une fonction donnée est totale.

Précisons un peu le sens exact du terme *indécidable*, il désigne les propriétés pour lesquelles il n'existe pas de programme qui réponde 1 en temps fini si la propriété est vraie. Par exemple, la propriété de non–arrêt d'un programme est (réellement) indécidable (il en est en fait de même pour toute négation de propriété semi–décidable).

Pour finir ce court paragraphe, signalons qu'il existe bien sûr des propriétés pour lesquelles on ne sait pas encore si elles sont ou non décidables. Par exemple savoir si en appliquant à un nombre donné le programme de Collatz (Cf. page 33) on atteint 1. Cette propriété est de manière évidente semi–décidable : le programme est une procédure de semi–décision ; mais on ne sait pas par contre décider si un nombre n'a pas cette propriété (l'existence d'un tel nombre est encore à établir).

Chapitre 2

Les Machines de Minsky

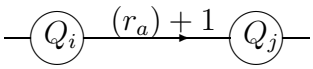
Dans son livre "*Finite and Infinite Machines*" [38], M. Minsky présente des machines qu'il prouve avoir la même puissance de calcul que les machines de Turing. Elles correspondent en quelque sorte à une arithméthisation de ces dernières. La bande est remplacée par des compteurs (ou registres) contenant des valeurs entières, la notion d'état est conservée et les transitions, caractérisant toujours des passages d'un état à un autre, entrainent cette fois au lieu d'une modification de la bande celle d'un registre. Ces machines sont appelées de ce fait "*machines à registres et états*". M. Minsky prouve qu'il suffit de deux registres pour simuler les machines de Turing.

Je ne vais pas présenter dans le détail ces machines, elles sont définies dans le Chapitre XI du livre de M. Minsky. Je préfère adopter la présentation faite par J.H. Conway [8]. Celle–ci me parait plus simple et une présentation plus formelle n'apporterai rien ici. Je présenterai également une classe particulière de machines, dites *nulles*, à partir de laquelle nous travaillerons par la suite.

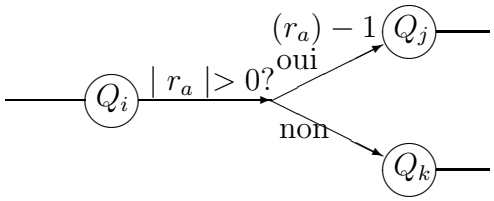
2.1 Présentation

Definition 8 Les *machines de Minsky* sont des machines comportant un nombre fini d'états Q_i – parmi lesquels un état d'entrée et un ou plusieurs états de sortie – et de registres r_j. Le contenu des registres, noté $\mid r_j \mid$, est un entier naturel. Deux types de transitions, représentables sous forme d'automates, sont possibles :

- " dans l'état Q_i, ajouter 1 au registre r_a et aller dans l'état Q_j."



- " dans l'état Q_i, si $\mid r_a \mid > 0$ alors soustraire 1 au registre r_a et passer à l'état Q_j, sinon passer simplement dans l'état Q_k."



Proposition 3 *Ces machines ont la même puissance de calcul que les machines de Turing et donc que les programmes de* Π_1^1.

Cette équivalence est traduite en exprimant en terme de transformations sur les registres les modifications de la bande de la machine de Turing. En effet, pour toute fonction récursive f, il existe une machine de Minsky qui démarrant avec des contenus de registres égaux à $n, 0, 0, ...$ finit[1] avec des contenus de registres égaux à $f(n), 0, 0, ...$ si n est dans le domaine de f et ne termine pas sinon. Minsky a établi qu'il suffit simplement de deux registres pour obtenir l'équivalence (Cf. [38] Chapitre XIV, pp. 255–258).

Du fait de ce résultat, on peut étendre aux machines de Minsky les définitions et les résultats obtenus dans le chapitre précédent :

- Le *domaine* d'une machine de Minsky $\mathcal{M}$ est l'ensemble des n pour lesquels le calcul de $\mathcal{M}(n)$ est fini, on le notera $Dom(\mathcal{M})$.
- Une machine de Minsky est dite *totale* si son domaine est égal à $\mathbb{N}$.
- L'arrêt d'une machine de Minsky est indécidable.
- Il est indécidable de déterminer si une machine de Minsky donnée est totale ou non.

2.2 Une Classe Particulière de Machines

Pour le besoin de certaines preuves qui apparaîtront plus loin, nous allons définir ici une famille de machines de Minsky particulières que nous appellerons *machines de Minsky nulles* puisqu'elles sont caractérisées par une image réduite à $\{0\}$.

Definition 9 Une machine de Minsky $\mathcal{M}$ est dite *nulle* si :

- $0 \in Dom(\mathcal{M})$
- tous les registres sont nuls à la fin du calcul, c'est–à–dire que la fonction récursive f associée à $\mathcal{M}$ vérifie :

$$\forall n \in Dom(f), f(n) = 0.$$

[1] C'est–à–dire "atteint un des ses états finaux".

Nous nous intéresserons plus précisément à des notions de complexité de calcul, en nombre de transitions, pour les machines de Minsky nulles. En particulier nous définirons ainsi les machines de Minsky linéaires :

Definition 10 Une machine de Minsky $\mathcal{M}$ est dite *(α–)linéaire* si il existe un entier naturel α tel que pour toute entrée n du domaine de de $\mathcal{M}$, $\mathcal{M}(n)$ est calculée en moins de $\alpha.n$ transitions.

L'existence de telles machines est évidente. Considérons simplement, la machine nulle à un seul registre et qui le décrémente jusqu'à atteindre 0. Elle est évidemment 1–linéaire. Il est tout aussi facile d'imaginer des machines α–linéaires pour tout α. Le théorème suivant établit le résultat qui nous intéresse les concernant.

Théorème 1 *Déterminer si une machine de Minsky nulle linéaire est totale ou non, est indécidable.*

Preuve.[2] A partir de toute machine de Minsky $\mathcal{M}_0$, on construit une machine nulle linéaire $\mathcal{M}_\alpha$, où α est un entier, de la manière suivante (n est le paramètre d'entrée de $\mathcal{M}_\alpha$) :

1. Calculer $\alpha.n$ et le ranger dans un nouveau registre r_α n'apparaissant pas dans $\mathcal{M}_0$,
2. Si $\mid r_\alpha \mid = 0$ aller à l'étape 5 sinon soustraire 1 à r_α et continuer,
3. Exécuter une transition élémentaire du calcul de $\mathcal{M}_0(0)$,
4. Si $\mathcal{M}_0(0)$ a atteint un de ses états finaux alors entrer dans une boucle infinie sinon retourner au pas 2
5. Mettre à 0 tous les registres et arrêter (aller dans un état final).

$\mathcal{M}_\alpha$ se construit aisément à partir de $\mathcal{M}_0$, seul le registre r_α est à ajouter. De plus, il faut insérer entre chaque transition de $\mathcal{M}_0$ une transition correspondant à l'étape 2, puis ajouter après chaque état final de $\mathcal{M}_0$ une boucle infinie (chacun peut en imaginer une sans difficulté) et enfin ajouter les transitions exécutant l'étape 5. Si l'on représente la machine $\mathcal{M}_0$ sous forme d'automate, on se convainc encore plus facilement de la faisabilité de cette transformation.

$\mathcal{M}_\alpha$ est une machine de Minsky nulle, en effet, il est facilement vérifiable que 0 appartient au domaine de $\mathcal{M}_\alpha$ et de plus, si pour un entier n le calcul s'arrête, par construction les registres sont nécessairement tous nuls (étape 5).

Vérifions maintenant que $\mathcal{M}_\alpha$ est linéaire. Soit n un élément du domaine de $\mathcal{M}_\alpha$, l'étape 1 du calcul peut être accomplie en $\alpha.n$ transitions. Puis on atteint

[2] L'idée de base de cette preuve est dûe à Jean–Paul Delahaye.

l'étape 5 (puisque n est dans le domaine de $\mathcal{M}_\alpha$) après $2.\alpha.n$ transitions. En effet, on va effectuer n fois (tant que $\mid r_\alpha \mid \neq 0$), une soustraction à r_α et une étape du calcul de $\mathcal{M}_0(0)$. Nous en sommes donc maintenant au point 5, il faut remettre à 0 tous les registres. La somme des k registres de $\mathcal{M}_0$ est au plus égale à $\alpha.n$ puisque l'on a effectué $\alpha.n$ transitions dans le calcul de $\mathcal{M}_0(0)$ et donc au plus $\alpha.n$ incrémentations des registres. En conséquence, on peut construire les transitions effectuant l'étape 5 de telle sorte qu'il faille au plus $\alpha.n + k$ transitions pour mettre à 0 tous les registres. Il en résulte que la complexité de $\mathcal{M}_\alpha$ en nombre d'itérations est $4 \times \alpha \times n + k$. $\mathcal{M}_\alpha$ est donc linéaire.

Nous avons donc vérifié ensemble que $\mathcal{M}_\alpha$ est nulle et linéaire. Examinons à quelle condition elle sera totale. Un entier n sera dans le domaine de $\mathcal{M}_\alpha$ si et seulement si la boucle infinie de l'étape 4 n'est pas atteinte, c'est–à–dire, si et seulement si $\mathcal{M}_0(0)$ est calculée en plus de $\alpha.n$ transitions et ce quelque soit n. Or le "temps" nécessaire au calcul de $\mathcal{M}_0(0)$ étant constant, la seule solution est que celui–ci soit infini, autrement dit que 0 ne soit pas dans le domaine de $\mathcal{M}_0$. Or, nous avons vu que l'appartenance d'un élément à un domaine était indécidable, nous en déduisons donc qu'il est indécidable de savoir si la machine nulle linéaire $\mathcal{M}_\alpha$ est totale. □

A partir de ces machines nulles et linéaires, je vais caractériser certains ensembles récursifs :

Definition 11 Un *ensemble récursif* Σ est dit *linéaire* si il existe une machine de Minsky nulle linéaire dont Σ est le domaine.

Corollaire 2 *Déterminer si un ensemble récursif linéaire est égal à* $\mathbb{N}$ *est indécidable.*

Preuve. Par application immédiate du théorème 1. □

Chapitre 3

Les Fonctions de Conway

Dans la section précédente, nous avons présenté les machines de Minsky qui peuvent être vues comme une arithmétisation des machines de Turing puisque la notion de bande y est remplacée par celle de registres à valeurs entières. Ici, nous allons nous intéresser à un des (nombreux) travaux du mathématicien J.H. Conway (connu entre autres pour le célèbre "jeu de la vie"). Celui a poussé encore plus loin cette arithmétisation puisqu'il propose une traduction en terme de fonctions numériques des machines de Minsky. Elle résulte de l'étude d'une généralisation de la conjecture de Collatz. Il est possible d'étendre à ces fonctions les résultats obtenus pour les machines de Minsky, y compris la caractérisation des ensembles récursivement énumérables. Nous définirons à cette occasion des relations que nous appellerons "de Conway".

3.1 La Conjecture de Collatz

Celle–ci affirme que le programme ci–dessous termine pour tout entier n.

TantQue $n > 1$ faire
  si n est pair
    alors $n \leftarrow \frac{n}{2}$
    sinon $n \leftarrow 3n + 1$
finTantQue

Appelée également "conjecture de Syracuse" ou "problème $3x + 1$", son origine exacte n'est pas clairement établie [30]. Elle est attribuée à Lothar Collatz de l'Université de Hambourg dans les années 30. Ce problème a circulé de bouche à oreille au sein de la communauté mathématique particulièrement dans les années 50–60. Dès qu'il atteignait une université, il monopolisait l'attention de tous au détriment des autres travaux de recherches. Une plaisanterie fut faite affirmant que ce problème faisait partie d'une conspiration visant à ralentir de la recherche en mathématiques aux Etats–Unis. Il a continué à soulever l'intérêt

depuis ce temps. J.C. Lagarias, passionné par ce sujet, a établi une bibliographie très complète et impressionnante par le nombre de publications sur le sujet [31]. Malgré tous ces efforts, cette conjecture reste ouverte.

Nabuo Yoneda de l'Université de Tokyo l'aurait vérifiée pour tout n inférieur à 2^{40}. Le comportement de la suite de Collatz d'un nombre, c'est–à–dire la suite des nombres obtenus successivement lors de l'exécution du programme précédent, semble totalement aléatoire et chaotique. Par exemple, à partir de $2^{500}-1$, des entiers supérieurs à $2^{500}\times 10^{88}$ sont atteints. Un autre exemple, si à partir de 26, il ne faut que 10 étapes pour atteindre 1, il en faut en revanche 111 à partir de 27.

$$26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$\underbrace{27 \to 82 \to 41 \to ... \to 4 \to 2 \to 1}_{111 \text{ étapes}}$$

La conjecture peut être formulée ainsi :

Conjecture 4 (Collatz) *Soit g la fonction définie par*

$$g(n) = \begin{cases} \frac{1}{2}n & (n \equiv 0 \pmod 2) \\ 3n+1 & (n \equiv 1 \pmod 2) \end{cases}$$

pour tout entier n, il existe $k \in \mathbb{N}$ tel que $g^{(k)}(n) = 1$.

J.H. Conway a étudié les fonctions, que j'appelerai par la suite "*fonctions de Conway*"[1], plus générales suivantes :

$$g(n) = \begin{cases} a_0 n + b_0 & (n \equiv 0 \pmod p) \\ \cdots & \cdots \\ a_i n + b_i & (n \equiv i \pmod p) \\ \cdots & \cdots \\ a_{p-1} n + b_{p-1} & (n \equiv p-1 \pmod p) \end{cases}$$

où les a_i et les b_i sont des nombres rationnels tels que $g(n)$ reste toujours un entier. Il a étudié le comportement des itérations $g^{(k)}(n)$. Comme nous allons le voir, il a prouvé que même lorsque les b_i sont nuls, le comportement de ces fonctions reste imprévisible. Cela a été réalisé par une interprétation des machines de Minsky par ces fonctions. Nous examinerons ensuite par analogie aux machines nulles des fonctions de Conway dites *nulles* qui me permettront de définir certaines relations.

[1] Par la suite la lettre g désignera une fonction de Conway.

3.2 Présentation

J.H. Conway [8] considère la classe des fonctions linéaires périodiques par parties de $\mathbb{N}$ dans $\mathbb{N}$ ayant la structure suivante :

$$\forall k,\ 0 \leq k \leq d-1,\ g(n) = a_k n\ (n \equiv k\ (\text{mod}\ d)).$$

où les $a_0, ..., a_{d-1}$ sont tous des nombres rationnels tels que $g(n) \in \mathbb{N}$. Ces fonctions correspondent exactement aux fonctions entières telles que $\frac{g(n)}{n}$ est périodique. Il a étudié le comportement des itérés $g^{(k)}(n)$ et a établi le théorème suivant (où $\mathbb{N}^*$ désigne $\mathbb{N} \setminus \{0\}$) :

Théorème 3 (Conway) *Soit f une fonction récursive partielle quelconque, il existe une fonction g telle que :*

1. *$\frac{g(n)}{n}$ est périodique (mod d) pour un certain d et prend des valeurs rationnelles.*
2. *$\forall n \in \mathbb{N}, n \in Dom(f)$ ssi $\exists (k,j) \in \mathbb{N}^* \times \mathbb{N}$, $g^{(k)}(2^n) = 2^j$.*
3. *$g^{(k)}(2^n) = 2^{f(n)}$ pour le plus petit $k \geq 1$ tel que $g^{(k)}(2^n)$ est une puissance de 2.*

Le premier point exprime le fait que g est une fonction de Conway. Le second indique comment à partir de cette fonction, on peut caractériser l'appartenance au domaine de f fonction partielle récursive. Le troisième traduit comment par itération de g on peut calculer la valeur de f en n. Ce dernier point permet de considérer que la puissance d'expression des fonctions de Conway est aussi élevée que celle des programmes de Π^1_1, des machines de Turing ou de Minsky. Ceci est normal dans la mesure où, comme nous allons le voir, elles sont une traduction directe de ces dernières.

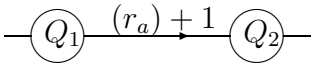
Principe de la Preuve. J.H. Conway montre qu'à toute machine de Minsky il est possible d'associer une fonction g définie comme ci–dessus et qui simule pas à pas le comportement de cette machine. En fait, il explique comment construire cette fonction à partir de la machine :

- au registre r_i on associe un nombre premier p_i et l'on exprime la valeur, k, du registre par $(p_i)^k = (p_i)^{|r_i|}$.
- à chaque état Q_j, on associe également un nombre premier P_j (les nombres premiers choisis sont bien sûr tous différents).
- une configuration de la machine, traduite par la valeur k_i des registres r_i et par l'état courant Q_j, est exprimée par un entier de la forme :

$$(p_1)^{k_1} * (p_2)^{k_2} * \cdots * (p_n)^{k_n} * P_j$$

Maintenant étudions comment exprimer les transitions. Cela se fait de manière naturelle si l'on a bien compris la représentation de la situation courante par un nombre entier décrite ci–dessus :

- pour les transitions "positives" :

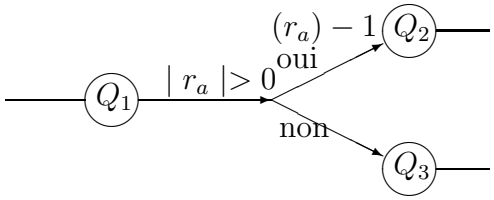


  Si l'état Q_1 est caractérisé par le nombre premier P_1, l'état Q_2 par P_2 et le registre r_a par p_a, cette transition peut alors être traduite par la multiplication de la situation courante de la machine par le facteur :

$$\frac{P_2}{P_1} \times p_a$$

  qui signifie "Je quitte l'état Q_1 ($\times\frac{1}{P_1}$), vais dans l'état Q_2 ($\times P_2$) et ajoute 1 au registre r_a ($\times p_a$)".

- pour les transitions "négatives" :



  En reprenant les conventions précédentes et en attribuant à Q_3 le nombre premier P_3, on peut exprimer cette transition à l'aide des facteurs :

$$\frac{P_2}{P_1} \times \frac{1}{p_a} \qquad \text{ou} \qquad \frac{P_3}{P_1}$$

  Le choix entre ces facteurs, correspondant respectivement aux cas $|\ r_a\ |>0$ et $|\ r_a\ |=0$, sera réalisé par le "modulo d" de la définition des fonctions g. Il en est de même pour la détection de "Suis–je dans l'état P_i ?".

Pour chaque transition de la machine de Minsky à coder il faut créer les facteurs correspondants, puis à partir de ces facteurs déterminer la période d de telle manière que $g(n)$ reste entier et calculer tous les a_i. Cela est un peu fastidieux mais réalisable.

On comprend qu'à chaque itération de la fonction g correspond une transition élémentaire de la machine de Minsky concernée. □

Je parlerai donc de fonction de Conway *associée* à une machine de Minsky et réciproquement.

Remarque 2 Par construction, le nombre d'itérations nécessaires à partir de $g(2^n)$ pour atteindre $2^{f(n)}$ est égal aux nombres de transitions utilisées par la machine de Minsky associée $\mathcal{M}$ pour produire $f(n)$ à partir de n.

Exemple 1 J.H. Conway et R.K. Guy [23] se sont appliqués à construire un générateur de nombres premiers à l'aide de ce principe. Considérons la suite de rationnels suivante *:*

$\frac{17}{7.13}$	$\frac{2.3.13}{5.17}$	$\frac{19}{3.17}$	$\frac{23}{2.19}$	$\frac{29}{3.11}$	$\frac{7.11}{29}$	$\frac{5.19}{23}$	$\frac{7.11}{19}$	$\frac{1}{17}$	$\frac{11}{13}$	$\frac{13}{11}$	$\frac{3.5}{2.7}$	$\frac{3.5}{2}$	$\frac{5.11}{1}$
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)

Cette séquence ne correspond pas exactement à une fonction de Conway, elle traduit plutôt la transcription, décrite précédemment, d'une machine de Minsky en facteurs. Pour obtenir la "véritable" fonction de Conway, il faudrait, comme déjà expliqué, calculer la période et les coefficients qui en découlent à partir de ces facteurs.

Considérons que notre donnée est 2^n où n est un nombre premier. Une étape de calcul est de multiplier le nombre courant par le premier rationnel de la suite ci–dessus tel que l'on obtienne un entier. On applique ce processus jusqu'à obtenir une puissance de 2. Cette puissance est alors le plus petit nombre premier plus grand que n.

En partant de 2^1 et en itérant infiniment le processus, on générera donc tous les nombres premiers. On pourra les reconnaître en considérant toutes les puissances de 2. Voici les 19 étapes requises de 2^1 à 2^2 :

$$2^1 \xrightarrow{(m)} 15 \xrightarrow{(n)} 825(=25\times 33) \xrightarrow{(e)} 725(=25\times 29) \xrightarrow{(f)} 1925 \xrightarrow{(k)} 2275 \downarrow$$
$$170 \xleftarrow{(a)} 910 \xleftarrow{(k)} 770 \xleftarrow{(f)} 290 \xleftarrow{(e)} 330 \xleftarrow{(j)} 390 \xleftarrow{(b)} 425 \xleftarrow{(a)} \swarrow$$
$$\downarrow$$
$$\searrow \xrightarrow{(b)} 156 \xrightarrow{(j)} 132 \xrightarrow{(e)} 116 \xrightarrow{(f)} 308 \xrightarrow{(k)} 364 \xrightarrow{(a)} 68 \xrightarrow{(i)} 2^2$$

□

Cette présentation des fonctions de Conway devrait être suffisante pour nous avoir permis de comprendre l'essentiel sur cette notion. Dans la prochaine section, nous allons examiner un aspect plus spécifique et essentiel pour la compréhension des travaux qui suivront.

3.3 Relations de Conway

Nous venons de voir que les fonctions de Conway et les machines de Minsky sont en relation étroite. Il est donc naturel d'étendre certaines définitions propres à ces dernières aux premières.

Definition 12 Soit g une fonction de Conway, son *domaine*, noté $Dom(g)$, est :

$$Dom(g) = \left\{ n \in \mathbb{N} \mid \exists (k,p) \in \mathbb{N}^* \times \mathbb{N}, g^{(k)}(2^n) = 2^p \right\}$$

Une fonction de Conway sera dite *totale* si son domaine est $\mathbb{N}$.

Considérant les fonctions de Conway associées aux machines de Minsky nulles, nous définissons les fonctions de Conway nulles ainsi :

Definition 13 Une fonction de Conway g est dite *nulle* si

- 0 appartient à son domaine
- pour tout n du domaine de g, la première puissance de 2 obtenue en itérant $g(2^n)$ est 2^0.

De la même manière nous parlerons de *fonctions de Conway linéaires nulles*, elles correspondront de manière évidente aux machines de Minsky linéaires nulles définies précédemment. Dans la mesure où nous avons vu qu'à chaque transition de la machine correspond une itération (Cf. Remarque 2), il est naturel de mesurer la "complexité" des fonctions en nombre d'itérations comme nous l'avons fait en nombre de transitions pour les machines.

Nous allons particulièrement étudier ces fonctions nulles et dégager des propriétés intéressantes permettant de définir ce que j'appellerai les *relations de Conway*. Pour cela je vais analyser le comportement des *itérations "négatives"* des fonctions de Conway. Je les définis ainsi :

Definition 14 Soit g une fonction de Conway :

$$\forall k \in \mathbb{N}, g^{(-k)}(n) = \left\{ m \in \mathbb{N} \mid g^{(k)}(m) = n \right\}.$$

En effet nos fonctions de Conway n'étant pas injectives, l'image réciproque d'un nombre par une de ces fonctions sera a priori un ensemble.

Examinons maintenant le comportement de ces itérations négatives pour les puissances de 2, puisque ce sont elles qui nous concernent directement.

Proposition 5 *Soient g une fonction de Conway nulle et n un entier quelconque, la seule puissance de 2 que l'on peut obtenir en appliquant itérativement g à 2^n est 2^0. Et de plus, par itérations négatives de g à partir de 2^0 on atteint tout $Dom(g)$ et aucune autre puissance de 2.*

Preuve. Etant donné que g est nulle, 0 appartient à son domaine. Il existe donc $k > 0$ tel que $g^{(k)}(2^0) = 2^0$. Et k étant le plus petit nombre tel que $g^{(k)}(2^0)$ est une puissance de 2, on en déduit qu'aucune autre puissance de 2 ne peut être obtenue par applications successives de g à partir de 2^0.

Maintenant, il suffit de considérer la définition des fonctions nulles. Pour un n donné, si celui–ci n'appartient pas au domaine, aucune puissance de 2 ne sera atteinte. Par contre, si il y appartient, puisque la première puissance de 2 obtenue après itérations successives de g sur 2^n est $2^{f(n)} = 2^0$, on déduit de ce qui précède que c'est la seule possible.

La seconde partie de la proposition est maintenant immédiate : il était acquis par définition qu'à partir de 2^0 on atteignait, par itérations négatives, tous les éléments de $Dom(g)$, que ce soient les seules puissances de 2 atteintes découle de ce que nous venons de voir. □

Il résulte de cette proposition que, si un entier n appartient au domaine d'une fonction de Conway nulle g, il n'existe alors qu'un seul chemin entre 2^n et 2^0 et ce, que l'on considère des itérations positives ou négatives de g (nous ignorerons les boucles sur 2^0 puisqu'elles ne contiennent aucune autre puissance de 2). L'existence d'un tel chemin est absolument conditionné par l'appartenance de n au domaine de g. De plus, deux chemins ne peuvent se "couper". Ainsi, les transitions de Conway "$2^n \rightarrow g(2^n)$" peuvent être étendues à des relations "$2^n \leftrightarrow_g g(2^n)$".

Definition 15 Une *relation de Conway* est définie à partir d'une fonction de Conway nulle et de sa relation de base :

$$\forall n \in \mathbb{N}, 2^n \leftrightarrow_g g(2^n).$$

On peut alors caractériser les ensembles récursivement énumérables contenant 0, c'est–à–dire les domaines de fonctions nulles, à l'aide de ces relations.

Proposition 6 *Pour tout ensemble récursivement énumérable Σ contenant 0, il existe une relation de Conway $\leftrightarrow_g$ telle que :*

$$\Sigma = \left\{ n \in \mathbb{N} \mid 2^n \leftrightarrow_g 2^0 \right\}.$$

Preuve. Les ensembles récursivement énumérables contenant 0 sont les domaines des machines de Minsky nulles, et donc des fonctions de Conway nulles. Si Σ est le domaine de la fonction nulle g, alors $\leftrightarrow_g$ satisfait la proposition. □

Cette proposition est fondamentale pour la suite. En effet, dans plusieurs preuves, nous créerons des ensembles récursivement énumérables en partant de 2^0 et en utilisant les itérations négatives d'une fonction de Conway nulle afin d'énumérer les éléments de cet ensemble. Nous utiliserons alors les propriétés d'indécidabilité sur les ensembles.

Part II

Les Langages à Clauses de Horn

La famille des langages à clauses de Horn, dont Prolog est le représentant le plus connu, est basée sur le principe de la résolution de J.A. Robinson [42]. Les travaux de A. Colmerauer et R. K. Kowalski sont à la base de ces langages qui s'appuient sur la logique du premier ordre. Ils constituent avec les dialectes LISP les principaux langages de l'intelligence artificielle.

L'importance de ce qui va suivre justifie que nous lui consacrions une place à part. En effet les résultats qui seront présentés en troisième partie ont pour cadre les langages à clauses de Horn et plus précisément une classe particulière de ces programmes.

Après avoir introduit rapidement les termes de la logique du premier ordre ainsi que des notions telles que la résolution, nous parlerons un peu plus particulièrement de programmation logique avant de présenter quelques particularités liées à notre classe de programmes.

Chapitre 4

Généralités

Nous nous appuyons ici sur les chapitres 1 et 2 de [34] (voir aussi [14]).

4.1 Beaucoup de Définitions

Nous allons étudier maintenant l'essentiel de la terminologie utilisée en théorie du premier ordre. Considérons donné un alphabet $\mathcal{A}$, divisé en plusieurs classes de symboles que voici :

- les *variables*, elles seront normalement représentées par les lettres majuscules U, V, X, $\cdots$, éventuellement indicées, nous noterons *Var* l'ensemble des variables utilisées,
- les *constantes* , représentées par les minuscules a, b, c, $\cdots$,
- les *symboles de fonctions* d'*arité*[1] strictement positives (ceux d'arité nulle étant les constantes), dénotés par les minuscules f, g, $\cdots$, nous noterons $\mathcal{F}$ l'ensemble des symboles de fonction,
- les *symboles de prédicats*, dénotés le plus souvent par les lettres p, q,$\cdots$,
- des *connecteurs*, ce sont les symboles $\neg$, $\wedge$, $\vee$, $\leftarrow$,$\leftrightarrow$ représentant respectivement la négation, la conjonction, la disjonction, l'implication et l'équivalence,
- les *quantificateurs* universels et existentiels, $\forall$ et $\exists$.

Il arrivera que ces conventions ne soient pas toujours très fidèlement respectées, essentiellement afin de donner des noms un peu plus explicites aux

[1] l'*arité* d'une fonction ou d'un prédicat est son nombre d'arguments. On parle d'arité binaire quand l'arité vaut 2, ternaire quand elle vaut 3, $\cdots$, n-aire quand elle vaut n.

différents symboles utilisés. Néanmoins le contexte devrait lever toute ambiguité. Cependant, les variables commenceront toujours par une majuscule et les constantes et symboles de fonction et de prédicat par une minuscule.

Nous pouvons maintenant présenter la terminologie traditionnelle.

Un *terme* est défini ainsi :

- une variable est un terme,
- une constante est un terme,
- si f est un symbole de fontion n–aire et si $t_1, \cdots, t_n$ sont des termes alors $f(t_1, \cdots t_n)$ est un terme.

Un terme est dit *clos*, si aucune variable n'y apparaît. Il est dit *linéaire*, si aucune variable n'y apparaît deux fois.

Une *formule* (bien–formée) est définie inductivement ainsi :

- si p est un symbole de prédicat n–aire et si $t_1, \cdots, t_n$ sont des termes alors $p(t_1, \cdots t_n)$ est une formule (on parle dans ce cas de *formule atomique* ou d'*atome*),
- si F et G sont des formules, alors $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$ et $F \leftrightarrow G$ en sont aussi,
- si F est une formule et X une variable, alors $\forall X, F$ et $\exists X, F$ sont des formules.

Un *langage du premier ordre* d'un alphabet donné est l'ensemble de toutes les formules construites à partir des symboles de cet alphabet. Un *littéral* est un atome, on dit alors qu'il est *positif*, ou la négation d'un atome, il est alors qualifié de *négatif*.

Definition 16 Une *clause* est une formule de la forme

$$\forall X_1 \cdots \forall X_n, L_1 \vee \cdots \vee L_m$$

où chaque L_i est un littéral et X_1, $\cdots$, X_n sont les seules variables apparaissant dans $L_1 \vee \cdots \vee L_m$.

On utilise habituellement une notation plus particulières pour les clauses, ainsi la clause

$$\forall X_1 \cdots \forall X_n, A_1 \vee \cdots \vee A_m \vee \neg B_1 \vee \cdots \vee \neg B_p$$

où les A_i et les B_i sont des atomes, sera notée

$$A_1, \cdots, A_m \leftarrow B_1, \cdots, B_p$$

Une *clause de Horn* est une clause de la forme

$$A \leftarrow B_1, \cdots, B_n$$

c'est–à–dire une clause avec un seul littéral positif. A est appelé la *tête* de clause et $B_1, \cdots, B_n$ le *corps* de la clause. Une *clause unité* (ou *fait*) est une clause de Horn dont le corps est vide, c'est–à–dire ne contenant aucun littéral négatif. C'est donc une clause de la forme "$fait \leftarrow$.". Un *but* est une clause de Horn dont la tête est vide, c'est–à–dire ne contenant aucun littéral positif. C'est donc une clause de la forme $\leftarrow but$.

Definition 17 Un *programme* à clauses de Horn ou *programme défini* est un ensemble fini de clauses de Horn.

Exemple 2 Ainsi :

$$\left\{ \begin{array}{ll} element(X, [X|\_]) \leftarrow \ . &] \text{ fait} \\ element(X, [\_|L]) \leftarrow element(X, L) \ . &] \text{ clause binaire} \\ \leftarrow element(c, [b, c, a, d, e]) \ . &] \text{ but} \end{array} \right] \text{ programme défini}$$

□

4.2 Interprétation, Satisfiabilité et Implication

Intuitivement, une *interprétation* d'un langage du premier ordre revient à donner une signification (concrète) à ses différents constituants. Elle est caractérisée par un ensemble non–vide $\mathcal{S}$, puis à l'attribution à chaque constante d'un élément de $\mathcal{S}$, à chaque symbole de fonction n–aire d'une application de $\mathcal{S}^n$ dans $\mathcal{S}$ et, enfin, à chaque symbole de prédicat n–aire d'une application de $\mathcal{S}^n$ dans $\{Vrai, Faux\}$. On attribue également à chaque variable du langage un élément de $\mathcal{S}$.

On peut ainsi déterminer une *valeur de vérité*, relativement à une interprétation, pour toute formule du langage en interprétant de manière habituelle les connecteurs et quantificateurs.

Exemple 3 [2]Constituons une petite base de données sur des relations de parenté.

Etant donné l'alphabet suivant :

- trois constantes : $\{a, b, c\}$
- un symbole de fonction unaire : $\mathcal{F}$=$\{père\_de\}$
- un ensemble de variables : $Var = \{X, Y, \cdots\}$

[2]tiré de [14] : exemple 2, pp. 103–104

- deux symboles de prédicat : $est\_père$ et $est\_père\_de$

Une interprétation revient donc à donner une situation concrète. C'est–à–dire que l'on va affecter à chaque constante un personnage et indiquer ce qui est vrai ou faux pour ceux–ci. Pour chaque symbole s du langage, je noterai $\overline{s}$ son interprétation.

Prenons $\mathcal{S}$=$\{Alain, Bernard, Christian, ancêtre\}$, puis les attributions suivantes :

- $\overline{a} = Alain$, $\overline{b} = Bernard$, $\overline{c} = Christian$
- $\overline{père\_de} : \overline{b} \mapsto \overline{a}$, $\overline{c} \mapsto \overline{a}$, $\overline{a} \mapsto ancêtre$, $ancêtre \mapsto ancêtre$
- $\overline{est\_père}(\overline{a}) = Vrai$, $\overline{est\_père}(\overline{b}) = Faux$, $\overline{est\_père}(\overline{c}) = Faux$, $\overline{est\_père}(ancêtre) = Vrai$
- $\overline{est\_père\_de}(X, Y) = Vrai$ si et seulement si ($X = \overline{a}$ et $Y = \overline{b}$) ou ($X = \overline{a}$ et $Y = \overline{c}$) ou ($X = ancêtre$ et $Y = \overline{a}$) ou ($X = ancêtre$ et $Y = ancêtre$).

La valeur de vérité de la formule

$$\exists X, est\_père\_de(X, b)$$

est donc $Vrai$ en effet, $\overline{est\_père\_de}(Alain, Bernard)$ est vraie. □

Dans le cas où une formule[3], donc une clause, est vraie relativement à une interprétation, on dit que cette interprétation est un *modèle* pour cette formule. Ainsi dans l'exemple précédent, l'interprétation donnée est un modèle pour la formule $\exists X, est\_père\_de(X, b)$. On peut alors définir ce qu'est une formule (et donc une clause) satisfiable :

Definition 18 Une formule est *satisfiable* si il existe une interprétation du langage qui en soit un modèle. Un ensemble de formules est satisfiable si chacune de ses formules est satisfiable.

En programmation logique, on travaille en fait dans ce qui est appelé l'*univers de Herbrand*. Il s'agit de l'ensemble de tous les termes clos qui peuvent être formés à partir des constantes et des symboles de fonction du langage. On définit alors une *interprétation de Herbrand*, en prenant pour ensemble $\mathcal{S}$ l'univers de Herbrand, et la notion de *modèle de Herbrand*. Il est alors possible de prouver que si un ensemble de clauses (et non de formules ici) a un modèle, et est donc satisfiable, il a un modèle de Herbrand.

[3]Pour être exact, toutes les variables apparaissant dans cette formule doivent alors être liées par un quantificateur.

Definition 19 Etant donnés un ensemble de clauses Σ, et une clause $\mathcal{C}$, on dit que $\mathcal{C}$ est une *conséquence logique* de Σ, ou que Σ *implique* $\mathcal{C}$, si pour toute interprétation $\mathcal{I}$ du langage, si $\mathcal{I}$ est un modèle pour Σ alors $\mathcal{I}$ est un modèle pour $\mathcal{C}$.

L'*implication de clauses* a donné lieu à de nombreux travaux. Ce problème est équivalent, étant données deux clauses de Horn

$$\mathcal{C}_1 = \alpha \leftarrow \beta_1, \cdots, \beta_n \text{ et } \mathcal{C}_2 = \gamma \leftarrow \delta_1, \cdots, \delta_p.$$

à celui de la non–satisfiabilité (de l'existence de solutions) pour l'ensemble de clauses (le programme) :

$$\left\{ \begin{array}{l} \delta_1 \leftarrow \quad . \\ \qquad \vdots \\ \delta_p \leftarrow \quad . \\ \alpha \leftarrow \beta_1, \cdots, \beta_n \ . \\ \leftarrow \gamma \ . \end{array} \right.$$

où γ et les δ_i sont des termes clos.

Il a été établi que le problème $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ est décidable si $\mathcal{C}_1$ est binaire (le corps est réduit à un seul atome) [46], et indécidable si $\mathcal{C}_1$ est ternaire et de Horn [36, 35] ou quaternaire [46].

La notion de satisfiabilité est proche de celle d'existence de solutions pour les programmes à clauses de Horn. En effet, étant donnés un programme défini $\mathcal{P}$ et un but $\mathcal{B}$, on dira que $\mathcal{B}$ est une conséquence logique de $\mathcal{P}$, c'est–à–dire que $\mathcal{P}$ aura au moins une solution avec $\mathcal{B}$, si $\mathcal{P} \cup \{\neg\mathcal{B}\}$ est insatisfiable. Il faut comprendre le terme de "satisfaction" dans le sens de la "satisfaction d'un but".

4.3 Unification

Avant d'approfondir cette notion de solution d'un programme logique en parlant de résolution, il convient de présenter celles de *substitution* et d'*unification*.

Une *substitution* σ est un ensemble fini de la forme $\{X_1/t_1, \cdots, X_n/t_n\}$ où les X_i sont des variables distinctes et les t_i sont des termes différents des X_i. L'ensemble des X_i est appelé le *domaine* de σ et celui des t_i sa *portée*.

Etant données $\sigma = \{X_1/s_1, \cdots, X_n/s_n\}$ et $\theta = \{Y_1/t_1, \cdots, Y_m/t_m\}$ deux substitutions, la *composition* $\sigma\theta$ de σ et θ est la substitution obtenue de l'ensemble

$$\{X_1/s_1\theta, \cdots, X_n/s_n\theta, Y_1/t_1, \cdots, Y_m/t_m\}$$

en supprimant les $X_i/s_i\theta$ tels que $X_i = s_i\theta$ et les Y_j/t_j tels que Y_j appartienne à $\{X_1, \cdots, X_n\}$.

Une *substitution de renommage* (ou simplement *renommage*) pour une expression $\mathcal{E}$ est une substitution où les X_i sont toutes les variables de $\mathcal{E}$ et où tous les t_i sont des variables n'apparaissant pas dans $\mathcal{E} \setminus \{X_1, \cdots, X_n\}$. Si $\mathcal{E}$ est une expression et σ une substitution, l'*instance* de $\mathcal{E}$ par σ, notée $\mathcal{E}\sigma$, est l'expression obtenue de $\mathcal{E}$ en remplaçant simultanément toute occurence des variables X_i par le terme t_i associé.

Exemple 4 Soient $\mathcal{E}$ l'expression $p(f(X,Y), b, g(Y))$

- si $\sigma = \{X/a, Y/g(b)\}$ alors $\mathcal{E}\sigma = p(f(a, g(b)), b, g(g(b)))$.
- si $\sigma = \{X/U, Y/V\}$, σ est un renommage et $\mathcal{E}\sigma = p(f(U,V), b, g(V))$. □

Les substitutions sont intéressantes dans la mesure où elles permettent d'unifier un ensemble d'expressions, c'est–à–dire de rendre égales leurs instances par cette substitution.

Soit $\mathcal{S}$ un ensemble d'expressions. Une substitution σ est appelée *unificateur* de $\mathcal{S}$ si $\mathcal{S}\sigma$ est un singleton. σ est appelé *unificateur le plus général* (*upg*) si pour chaque unificateur θ de $\mathcal{S}$, il existe une substitution γ telle que $\theta = \sigma\gamma$.

Exemple 5 $\mathcal{S}$=$\{p(f(X,Y), b, X), p(f(a,X), Z, X)\}$ est unifiable, en effet $\sigma = \{X/a, Y/a, Z/b\}$ est un unificateur pour S. Il en est même l'unificateur le plus général, et $\mathcal{S}\sigma = \{p(f(a,a), b, a)\}$ □

L'opération d'unification de deux expressions $\mathcal{E}_1$ et $\mathcal{E}_2$ est notée $\mathcal{E}_1 \vee \mathcal{E}_2$.

Il existe un algorithme bien connu d'unification, qui, étant donné un ensemble d'expressions $\mathcal{S}$, termine si $\mathcal{S}$ est unifiable et renvoie un upg pour $\mathcal{S}$. Sinon, il termine et indique que $\mathcal{S}$ n'est pas unifiable. Une des étapes de cet algorithme est appelée *test d'occurrence*. Il consiste à vérifier, avant d'ajouter le substituant X/t à l'unificateur en cours de calcul, si la variable X n'apparaît pas dans le terme t ; si c'est le cas, on ne peut ajouter ce substituant. En effet on ne peut demander à faire la substitution $X/f(X)$ car il faudrait alors également substituer $f(X)$ par $f(f(X))$ etc. Dans la pratique, ce test est très coûteux en temps et par souci d'efficacité il est généralement omis, aux programmeurs de faire attention. Mais notons que cette omission ne respecte plus la logique !

4.4 Résolution

La procédure de résolution originale est dûe à J.A. Robinson [42]. Celle utilisée en programmation logique a été décrite par R.A. Kowalski dans [29]. Elle est connue sous le nom de *résolution SLD*. Le terme *SLD* signifie que cette résolution est *Linéaire*, dirigée par une règle de *Sélection* et s'applique sur des programmes *Définis*.

La règle de sélection d'atome est la règle qui choisit à chaque étape de résolution l'atome à dériver.

Soit $\mathcal{B}$ le but $\leftarrow A_1, \cdots, A_{m,} \cdots A_n$ et $\mathcal{C}$ la clause $A \leftarrow B_1, \cdots B_p$. On dit que $\mathcal{B}'$ est *dérivé* de $\mathcal{B}$ et $\mathcal{C}$ en utilisant l'upg σ si, A_m étant l'atome choisi par la règle de sélection, σ est l'upg de A_m et de A et $\mathcal{B}'$ est le but

$$\leftarrow (A_1, \cdots, A_{m-1,} B_1, \cdots, B_p, A_{m+1}, \cdots, A_n)\sigma.$$

On dit que $\mathcal{B}$' est le *résolvant* de $\mathcal{B}$ et de $\mathcal{C}$.

Definition 20 Soient $\mathcal{P}$ et $\mathcal{B}$ un programme et un but définis. Une *dérivation SLD* de $\mathcal{P} \cup \{\mathcal{B}\}$ est une séquence $\mathcal{B}_0 = \mathcal{B}$, $\mathcal{B}_1, \cdots$ de buts, une séquence $\mathcal{C}_1, \mathcal{C}_2, \cdots$ d'instances de clauses de $\mathcal{P}$ et une séquence $\sigma_1, \sigma_2, \cdots$ d'upg telles que chaque $\mathcal{B}_{i+1}$ est dérivé de $\mathcal{B}_i$ et $\mathcal{C}_{i+1}$ en utilisant σ_{i+1}.

Une dérivation SLD peut être finie ou infinie. Lors de la résolution, chaque $\mathcal{C}_i$ est telle qu'elle ne contient aucune des variables apparaissant dans les dérivations précédant $\mathcal{B}_{i-1}$. Ceci est accompli par un *renommage des variables* avant chaque dérivation. Cela peut être accompli en indiçant les variables de la $i^{\text{ème}}$ dérivation par i. Cette technique d'indexation nous sera très utile par la suite.

Une *réfutation SLD* de $\mathcal{P} \cup \{\mathcal{B}\}$ est un dérivation SLD finie de $\mathcal{P} \cup \{\mathcal{B}\}$ qui a pour dernier but la clause vide, généralement notée $\Box$. Si $\mathcal{P} \cup \{\mathcal{B}\}$ admet une réfutation alors on dit que le programme admet une solution. Si $\mathcal{P}$ et $\mathcal{B}$ représentent respectivement un programme et un but définis, une *réponse* (ou *substitution réponse*) σ de $\mathcal{P} \cup \{\mathcal{B}\}$ est la substitution obtenue en restreignant la composition $\sigma_1 \cdots \sigma_n$ aux variables de $\mathcal{B}$. $\sigma_1 \cdots \sigma_n$ étant la séquence des upg de la réfutation SLD de $\mathcal{P} \cup \{\mathcal{B}\}$.

Chapitre 5

Programmation Logique

Ce chapitre sera l'occasion de présenter rapidement les stratégies des résolutions en profondeur et en largeur d'abord ainsi que le mécanisme des listes–différence. Il est bien entendu hors de propos ici d'expliquer comment programmer en Prolog (ou tout autre langage de la même famille), d'autant que de nombreux ouvrages le font très bien (Cf. [7], [4] par exemple).

5.1 Stratégies de Résolution

Plusieurs stratégies sont possibles en programmation logique afin d'accomplir la recherche d'une réfutation. Elles sont toutes équivalentes dans la mesure où tout ce qui est programmable dans une stratégie l'est dans une autre (bien entendu, les programmes sont alors syntaxiquement différents). Par convention, lors d'une dérivation on choisit toujours l'atome le plus à gauche du but courant et l'on examine les clauses du programme dans l'ordre où elles apparaissent. Une dérivation peut être représentée par un arbre (appelé avec originalité *arbre de dérivation* ou *arbre SLD*). La plus couramment utilisée est celle où l'on effectue un parcours en *profondeur d'abord avec retours en arrière* de l'arbre de dérivation mais on peut également utiliser une stratégie en *largeur d'abord*.

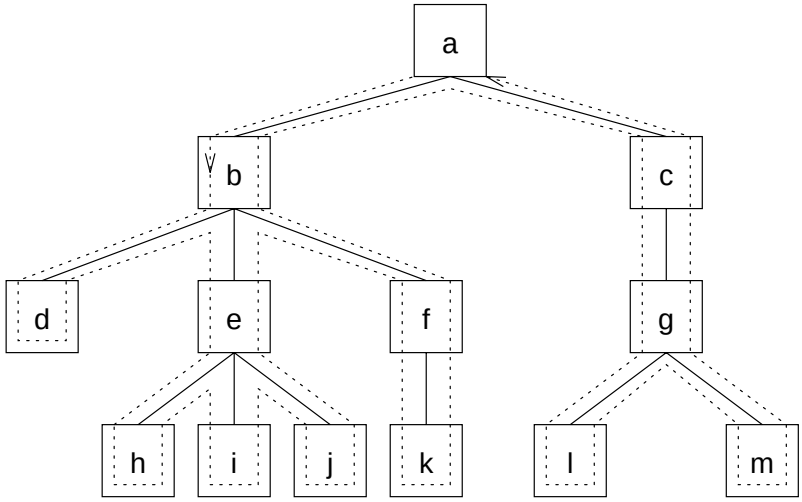
5.1.1 Profondeur d'abord

La stratégie en profondeur d'abord avec retours en arrière (ou *backtracking*) revient à sélectionner, à chaque étape de la résolution, la première clause du programme (rappelons qu'il s'agit d'un ensemble de clauses) dont la tête s'unifie avec le but courant, d'en déduire un nouveau but et de continuer la résolution avec celui–ci. Dès que l'on a atteint une feuille de l'arbre de résolution, c'est–à–dire soit aucune clause ne s'unifie avec le but courant (échec), soit le but courant est la clause vide (succès), on recommence avec le but immédiatement précédent (c'est le retour en arrière ou backtracking) en essayant de lui appliquer une autre

clause du programme, et ainsi de suite jusqu'à avoir tout essayé.

Ce mécanisme est repris dans le schéma du parcours de l'arbre ci–dessous.

Exemple 6 Voici un exemple de parcours en profondeur d'abord d'un arbre :



Les noeuds sont examinés dans l'ordre suivant : a,b,d,e,h,i,j,f,k,c,g, l,m. Les feuilles d,h,i,j,k,l,m représentent soit des échecs, soit des succès.

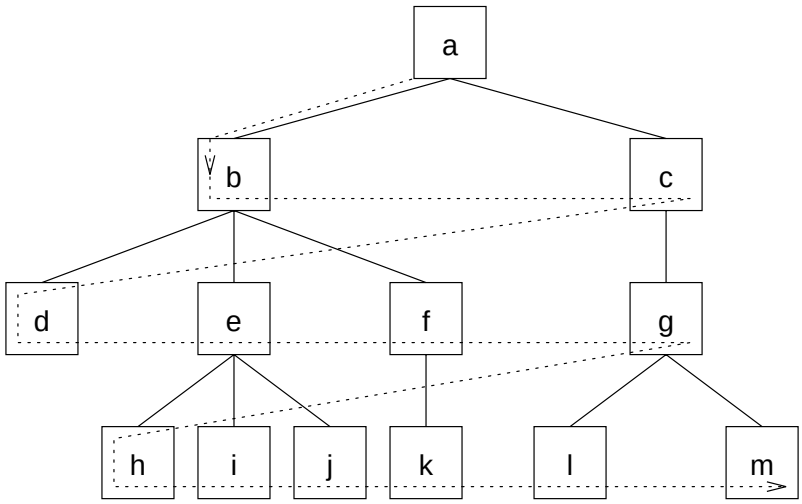
Chaque branche correspond à l'application d'une règle sur le but courant qui constitue le noeud. □

L'avantage de cette stratégie est que lorsque l'on cherche simplement à obtenir une solution mais que l'arbre est assez profond, on l'obtient assez rapidement. De plus, sa mise en place est assez facile et peu coûteuse en espace mémoire. Par contre, elle se perd dans la première branche infinie rencontrée. Elle ne parcourera donc complètement que les arbres finis, on dit alors de cette stratégie qu'elle n'est pas *équitable* .

5.1.2 Largeur d'abord

Une étape de résolution consiste à considérer toutes les clauses dont la tête s'unifie avec le but courant, d'en déduire autant de sous–buts possibles dont on examine successivement pour chacun une seule étape de résolution. Un but vide ou un échec est évidemment définitivement abandonné. Ce mécanisme est repris dans le schéma ci–dessous.

Exemple 7 Voici un exemple de parcours en largeur d'abord d'un arbre :



Les noeuds sont examinés dans l'ordre suivant : a,b,c,d,e,f,g,h,i,j,k, l,m. □

L'avantage est que le parcours de l'arbre est cette fois complet. Cette stratégie est *équitable*. Même pour un arbre infini, toutes les solutions seront trouvées. Mais, cette stratégie est très coûteuse en mémoire, il faut mémoriser l'arbre entier qui devient vite gigantesque pour peu que chaque noeud ait plusieurs fils.

5.2 Les Listes–Différence

Les listes se représentent en programmation logique par une suite d'éléments ordonnés. On les note généralement ainsi : $[a, b, c, d]$. On accède à la tête (premier élément) et au corps (le reste de la liste) d'une liste par unification avec le schéma $[Tête \mid Corps]$ où $Tête$ et $Corps$ sont des variables. Ainsi $[a, b \mid L]$ représente en fait une liste commençant par a et b.

Les *listes–différence* ([10],[24],[4],[37]) sont une structure de données permettant de manipuler plus efficacement les opérations sur les listes. Elles ne sont pas indispensables en programmation logique mais elles nous seront très utiles par la suite, il est donc préférable d'en avoir compris le fonctionnement.

La concaténation de deux listes (lorsque rien ne sera précisé, il s'agira de listes "classiques") se fait à l'aide du petit programme suivant :

$$\begin{cases} append([], L, L) \leftarrow \ . \\ append([X|L1], L2, [X|L3]) \leftarrow append(L1, L2, L3) \ . \end{cases}$$

Ainsi pour concaténer les listes $[a, b, c]$ et $[d, e]$, il faut prendre pour but :

$$\leftarrow p([a, b, c], [d, e], L) \ .$$

Mais, on peut effectuer cela plus facilement. Il suffit d'avoir un point de vue légèrement différent sur les listes. En effet, la liste $[a, b, c]$ peut être vue comme la

liste $[a, b, c, x, y, z]$ dans laquelle on ne considère pas la liste $[x, y, z]$. Une liste L est donc représentée par un couple de listes (L_1, L_2) que je noterai en introduisant un symbole de fonction spécial $L_1 - L_2$. Cette structure de représentation s'appelle une *liste–différence*. Ainsi une représentation de la liste $[a, b, c]$ pourrait être :

$$[a, b, c, x, y, z] - [x, y, z]$$

Il faut bien s'apercevoir que ce symbole de fonction n'est pas indispensable, partout on peut le remplacer par la virgule qui sépare habituellement deux arguments. Remarquons également que la représentation d'une liste n'est plus unique, j'aurai pu écrire la liste ci-dessus de nombreuses manières :

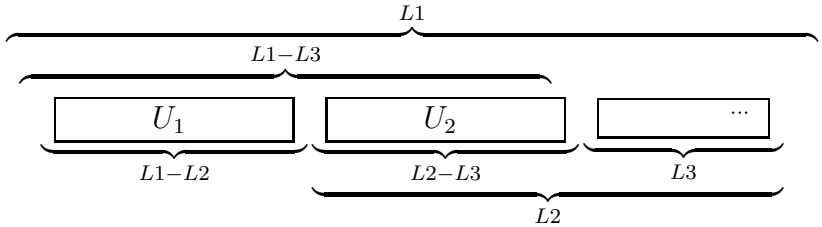
$$[a, b, c] - [] \,, \quad [a, b, c \mid L] - L, \quad [a, b, c, d, e] - [d, e] \,, \text{ etc.}$$

$L - L$ représente la liste vide. La manipulation de ces listes–différence doit se faire avec précaution. La présence du test d'occurence est indispensable lors de leur utilisation, de plus il faut avoir conscience que l'unification des listes–différence n'est pas nécessairement cohérente avec l'unification des termes qu'elles représentent ($[] - []$ et $[a] - [a]$ représentent toutes les deux la liste vide mais ne sont pas unifiables !).

On peut maintenant écrire l'opération de concaténation à l'aide de cette structure de données. Il suffit d'un simple fait :

$$append(L1 - L2, L2 - L3, L1 - L3).$$

Examinons comment cette clause unitaire permet la concaténation de deux listes U_1 et U_2 :



$U_1 = V_1 - T_1$ est unifiée à $L1 - L2$ et $U_2 = V_2 - T_2$ à $L2 - L3$, la clause impose donc que l'on ait $T_1 = V_2$, et il s'ensuit que la liste résultat vaut $U = V_1 - T_2 = L1 - L3$. Ce qui donne pour l'exemple précédent, le but

$$\leftarrow append([a, b, c \mid L] - L, [d, e \mid LL] - LL, LLL).$$

pour lequel la résolution provoque les instanciations

$$T_1 = V_2 = L = [d, e \mid LL] \text{ et } LLL = [a, b, c, d, e \mid LL] - LL.$$

Cette méthode d'ajout d'éléments à une liste sera utilisée dans des preuves à venir. Il est donc important d'en bien comprendre le mécanisme, qui n'est somme toute pas très compliqué pour peu que l'on prenne la peine, une fois, de bien examiner ce qui se produit.

Chapitre 6

Clauses de Horn Binaires

6.1 Définition

Definition 21 Une *clause de Horn* est dite *binaire* si son corps est réduit à un seul atome. Elle est dite *binaire récursive* si les symboles des prédicats de la tête et du corps sont les mêmes.

On dira qu'elle est linéaire–droite (resp. linéaire–gauche) si l'atome du corps (resp. de la tête) est linéaire.

Voici un exemple, pris presque tout à fait au hasard parmi une foultitude possible, de clause récursive binaire linéaire–droite :

$$append([X \mid L], LL, [X \mid R]) \leftarrow append(L, LL, R).$$

Nombreux sont ceux qui déclarent être plus que lassés de cet exemple trop souvent utilisé à leu goût. Ils souhaiteraient plus de diversité. Je ne le leur reprocherai pas, sauf qu'ici il me semble approprié. En effet, il est justement particulièrement bien connu et chacun pense sûrement qu'il n'y a plus grand chose de nouveau à dire à son sujet. Or nous verrons justement que certains de ses cousins proches sont d'une complexité inattendue.

Seules les clauses récursives étant d'un intérêt réel, j'utiliserai par la suite souvent le terme *binaire* à la place de *récursive binaire* qui est un peu long, et celui de *règle* à la place de *clause de Horn* (ou *clause* en plus court), pour varier les plaisirs.

Exemple 8 Voici d'autres exemples de clauses (binaires) (récursives) :

$fille(jolie) \leftarrow fille(francaise).$	"les filles françaises sont jolies"
$entier(succ(X)) \leftarrow entier(X).$	"*succ(X)* est un entier si X en est un"
$ami(X, Y) \leftarrow ami(Y, X).$	"X est un ami de Y si Y est un ami de X"

□

Pour de tels exemples simples, il semble possible de se faire une bonne intuition de ce qui se passe lors de la résolution. Mais déjà on dégage des comportements différents : nombre de solutions fini ou infini pour une résolution qui se termine ou non. Ces deux problèmes, terminaison et existence de solutions, occuperont une bonne part des travaux qui suivront.

6.2 Indexation des Variables

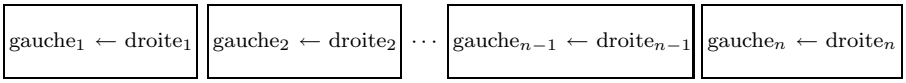
Nous avons vu que durant la résolution, avant d'appliquer une clause, ses variables formelles sont renommées en de nouvelles variables qui n'apparaissent pas ailleurs, cela se fait généralement par l'ajout d'un indice à ces variables formelles. Dans le cas d'une seule règle binaire, elle seule pouvant s'appliquer, la valeur de cet indice peut correspondre, par exemple, au numéro de l'inférence de la règle.

$$\begin{gathered} \text{i}^{\text{ème}} \text{ inférence :} \\ append([X_i \mid L_i], LL_i, [X_i \mid R_i]) \leftarrow append(L_i, LL_i, R_i). \end{gathered}$$

Maintenant, la séquence d'inférences utilisant la clause

$$gauche \leftarrow droit$$

peut être représentée sous la forme d'une suite de dominos :



Comme dans le jeu des dominos, le i$^{\text{ème}}$ peut être suivi du (i+1)$^{\text{ème}}$ si les deux arêtes contigües sont les mêmes. C'est–à–dire ici, si les termes $gauche_{i+1}$ et $droit_i$ sont unifiables et si la contrainte qui découle de cette unification est compatible avec les précédentes. Ainsi, appliquer n fois la clause binaire revient à résoudre le système suivant :

$$\{gauche_{i+1} = droite_i \mid i \in [1, n-1]\}$$

Pour poursuivre avec notre exemple original de la clause "*append*", appliquer cette clause n fois revient à résoudre le système :

$$\begin{gathered} \forall i \in [1, n-1], \\ append([X_{i+1} \mid L_{i+1}], LL_{i+1}, [X_{i+1} \mid R_{i+1}]) = append(L_i, LL_i, R_i) \end{gathered}$$

c'est-à-dire, plus lisiblement :

$$\forall i \in [1, n-1] \begin{cases} L_i & = & [X_{i+1} \mid L_{i+1}] \\ LL_i & = & LL_{i+1} \\ R_i & = & [X_{i+1} \mid R_{i+1}] \end{cases}$$

Exemple 9 Pour la clause suivante

$$p(s(x), s(s(Y))) \leftarrow p(X, Y).$$

la série de dominos est

$$\cdots \boxed{p(s(X_i), s(s(Y_i))) \leftarrow p(X_i, Y_i)} \boxed{p(s(X_{i+1}), s(s(Y_{i+1}))) \leftarrow p(X_{i+1}, Y_{i+1})} \cdots$$

En conséquence, à chaque itération il faut résoudre les égalités :

$$X_i = s(X_{i+1}) \text{ et } Y_i = s(s(Y_{i+1}))$$

□

Il est clair que pour rendre compte de toute la résolution, il faut considérer le but et le fait (j'écris "*le* fait", puisque nous ne nous intéresserons qu'au cas avec une seule clause unaire). On peut également formaliser sous forme d'équations ces deux éléments. Ainsi, appliquer n fois la clause "$gauche \leftarrow droite.$" au but "$\leftarrow but.$" et vérifier si il y a solution à la $n^{\text{ème}}$ inférence, c'est–à–dire si l'on a unification avec le fait "$fait \leftarrow .$", revient à résoudre :

$$\left\{ \begin{array}{rcll} but & = & gauche_1 & \\ gauche_{i+1} & = & droite_i & \forall i \in [1, n-1] \\ droite_n & = & fait & \end{array} \right.$$

Cette possibilité d'indexation des variables et de mise en équations de la résolution va être à la base des travaux qui suivront. Nous montrerons par exemple comment il est possible d'exprimer les itérations imprévisibles des fonctions de Conway grâce à ces outils. Ils sont également à l'origine du formalisme des systèmes pondérés d'équations que nous examinerons plus loin.

Part III

Clauses Binaires Récursives

Entrons dans le cœur du sujet. Nous allons étudier maintenant le comportement des programmes construits à partir d'une clause de Horn récursive binaire, d'un fait et d'un but :

$$\begin{cases} p(fait) \leftarrow \quad . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

Ils constituent le plus petit schéma non trivial possible en clauses de Horn. Leur étude devrait permettre d'essayer de mettre en valeur la complexité et la puissance de ces langages à travers ce plus petit noyau. Tout programme à clauses de Horn pouvant être vu comme un recouvrement de plusieurs programmes de cette forme, dégager certaines propriétés intéressantes les concernant devrait permettre d'améliorer la compréhension des langages à clauses de Horn en général.

En outre, cette étude est proche de celle des formules logiques du premier ordre de la forme

$$\forall X_i, (p(t_1) \vee (p(t_2) \wedge \neg p(t_3)) \vee \neg p(t_4))$$

où les X_i sont les variables apparaissant dans t_1, t_2, t_3 et t_4.

Nous verrons que les travaux qui vont suivre permettront de résoudre le problème de la satisfiabilité de ces formules.

Après une rapide introduction, je présenterai les formalismes des graphes orientés pondérés et des systèmes pondérés d'équations. Puis je montrerai comment les fonctions de Conway peuvent être exprimées par des programmes ayant la structure précédente. Enfin j'étudierai les problèmes de la terminaison, de l'existence d'au moins une solution et de la puissance d'expression pour ces programmes.

Chapitre 7

Problématique

Après avoir posé les bases requises, théoriques, générales et présenté les clauses de Horn, nous pouvons enfin aborder l'étude des programmes à une clause de Horn binaire récursive.

La classe de programmes qui nous concerne ici est donc celle ayant le schéma

$$\begin{cases} p(fait) \leftarrow \quad . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où $fait$, $gauche$, $droit$ et but sont des termes quelconques, que j'appellerai *termes caractéristiques* du programme.

Ces programmes peuvent sembler au premier regard d'une extrême simplicité et d'un intérêt limité. En effet, une instance de cette classe est le, plus que connu dans la communauté de la programmation logique, programme "*append*" :

$$\begin{cases} append([], L, L) \leftarrow \quad . \\ append([T|L], LL, [T|LLL]) \leftarrow append(L, LL, LLL) \ . \\ \leftarrow append(?, ?, ?) \ . \end{cases}$$

Heureusement, nous irons ici plus loin que cette première impression pour vérifier que bien qu'étant d'une très haute simplicité structurelle, ils sont d'une complexité d'expression élevée. Ainsi, bien que le programme *append* semble maîtrisé, certains de ses cousins proches nous réservent des surprises.

Deux problèmes semblent particulièrement pertinents à étudier concernant cette famille de programmes (et il n'y a là rien d'original) :

- Le premier est le *problème de l'arrêt* :

  "*Etant donnés une clause de Horn binaire récursive et un but, ce programme s'arrête–t–il ?*"

- Le second est celui de l'existence d'au moins une solution, que nous appellerons également *problème du vide* :

"*Etant donnés un fait, une clause de Horn binaire récursive et un but, ce programme a–t–il au moins une solution ?*"

Ce problème a été également défini comme étant le problème de l'*unification cyclique* par W. Bibel, S. Hölldobler et J. Würtz dans [3]. C'est–à–dire l'unification du but – qui commence le cycle – et du fait – qui le termine – à travers la clause de Horn binaire – qui engendre le cycle.

Ce problème a un équivalent en logique du premier ordre. Celui de la satisfiabilité des formules du premier ordre à quatre sous–formules telles que

$$\forall X_i, [p(t_1) \wedge (q(t_2) \vee r(t_3)) \wedge s(t_4)]$$

où P, Q, R et S sont des prédicats positifs ou négatifs et X_i les variables apparaissant dans t_1, t_2, t_3 et t_4. En effet la consistance d'une sous–classe de ces formules :

$$\forall X_i, [p(t_1) \wedge (p(t_2) \vee \neg p(t_3)) \wedge \neg p(t_4)]$$

correspond au problème de l'existence de solutions pour le programme

$$\begin{cases} p(t_1) \leftarrow \ . \\ p(t_2) \leftarrow p(t_3) \ . \\ \leftarrow p(t_4) \ . \end{cases}$$

Dans le cadre des formules quantifiées pures (c'est–à–dire sans symbole de fonction), la satisfiabilité des formules à cinq sous–formules atomiques a été établie indécidable il y a une vingtaine d'années dans [22], puisqu'elle y est montrée équivalente au problème de l'arrêt de machines à deux compteurs [38]. W. Goldfarb et H.R. Lewis donnent les problèmes à trois ou quatre sous–formules comme ouverts.

Ces deux problèmes ont été montrés décidables par M. Schmidt–Schauß[46] dans le cas où *but* et $fait$ sont clos. Ce résultat est un corollaire de son travail sur l'implication de clauses qui est équivalente au problème de décision d'un ensemble constitué d'une clause n–aire et de clauses unaires.

M. Dauchet, P. Devienne et P. Lebègue, [12, 16], ont étudié le cas linéaire et l'ont montré également décidable. Ils ont défini et utilisé à cette occasion une technique basée sur une extension des graphes pondérés : les graphes orientés pondérés. Je présenterai plus en détail ce formalisme dans le chapitre suivant.

Au cours de leur travail sur l'unification cyclique, W. Bibel, S. Hölldobler et J. Würtz ont examiné certains cas particuliers, également décidables : ceux où *gauche* et *droit* sont "faiblement" unifiables, c'est–à–dire qu'il existe une substitution σ telle que $\sigma gauche = droit$ ou $\sigma droit = gauche$. C'est ce qu'ils appellent des cycles gauche, respectivement droit.

Ici, nous montrerons que ces deux problèmes sont indécidables dans le cas général. Les preuves seront basées sur une codification des fonctions de Conway par des programmes ayant le schéma ci–dessus. Une autre preuve de l'indécidabilité du problème du vide a été établie par P. Hanschke et J. Würtz de l'Université de Saarbrück. Nous examinerons également certains cas particuliers en fonction de la (non–)linéarité de leurs termes caractéristiques.

Une fois ces résultats, déjà quelque peu surprenants, établis, il semble naturel d'essayer d'aller plus loin et de s'intéresser au pouvoir d'expression de cette classe de programmes. M. Dauchet [11] a montré qu'il est possible de simuler n'importe quelle machine de Turing à l'aide d'une seule règle de réécriture linéaire gauche. Dans [47], il est montré que toute fonction calculable est calculable par un programme construit uniquement à partir de clauses unaires ou binaires. Une autre codification, à partir de clauses unaires et binaires *récursives*, est proposée dans [1]. A. Parrain, P. Devienne et P. Lebègue [40] ont proposé un méta–interpréteur n'utilisant qu'un fait, un but et deux clauses binaires récursives.

Nous montrerons ici que tout calcul peut être accompli à l'aide d'un fait, d'un but et d'une seule règle récursive binaire, établissant ainsi l'équivalence entre la puissance d'expression de ces programmes et celle des machines de Turing. La preuve utilisera à nouveau la codification des fonctions de Conway ainsi que des transformations logiques sur des méta–programmes.

Ce résultat est équivalent pour les langages déclaratifs au théorème de Böhm–Jacopini. Celui–ci établit que dans le cadre des langages impératifs, tout calcul peut être accompli par un programme n'ayant qu'une seule boucle "tant que". En fait, bien qu'il soit connu sous le nom de "Théorème de Böhm–Jacopini", ce n'est pas cette forme qui leur est due ; la lecture de [26] fournit les détails de l'historique de ce théorème. Celui–ci est considéré comme une justification mathématique à la programmation structurée. Je montre que pour les langages à clauses de Horn, tout programme peut être transformé en un autre, composé de deux clauses unaires et une clause binaire, qui lui soit équivalent. Cette transformation conserve à la fois la terminaison et les solutions. Cela montre que la puissance d'expression d'une seule clause de Horn peut être utilisée comme un outil de décision (laissons de côté l'aspect raisonnable d'une telle démarche...).

Chapitre 8

Graphes et Systèmes Pondérés

Deux formalismes ont été développés par M. Dauchet, P. Devienne et P. Lebègue de l'Université de Lille afin de formaliser le comportement des règles récursives binaires. Il s'agit des *Graphes Orientés Pondérés* (GOPs) et des *Systèmes Pondérés d'Equations*.

Le premier [12, 15, 32, 16] est une extension des graphes orientés. Ils généralisent la notion d'arbres infinis (voir [9]). Les GOPs offrent une représentation graphique des relations entre les termes d'une clause récursive $G \leftarrow D$. Ils représentent le point fixe le plus général de ces règles. Une interprétation de ces graphes, correspondant à un dépliage, permet de rendre compte des itérations successives de la règle. Ils ont permis d'établir la décidabilité de la terminaison et de l'existence de solutions pour une règle binaire avec un but et un fait linéaires.

Le second est une extension des GOPs eux–mêmes [13]. Les systèmes pondérés d'équations fournissent une approche des GOPs en termes d'équations. Ils permettent de profiter des propriétés et des algorithmes connus des systèmes d'équations. Bien que leur pouvoir d'expression soit plus élevé que celui des GOPs, ils sont plus simples à comprendre et à utiliser. Ils ont été étendus dans [17] aux liens exceptions et linéaires.

Ces formalismes se sont révélés être des outils de preuves efficaces pour certains des résultats à venir et ils sont à l'origine des travaux présentés ici. Nous allons donc les présenter de manière assez détaillée.

8.1 Les GOPs

8.1.1 Présentation

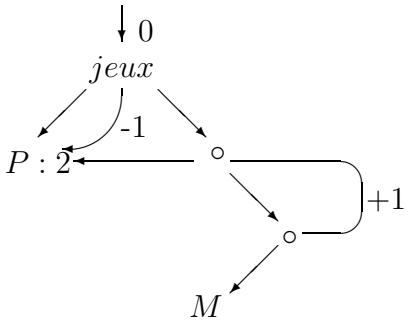
Définition.

Informellement, un *graphe orienté pondéré* (ou *GOP*) est un graphe orienté où :

- la racine est pondérée par un entier relatif,

- les arcs (orientés – ou flèches) sont pondérés par des entiers relatifs,
- les variables peuvent être périodiques

Exemple 10 Voici un exemple[1]

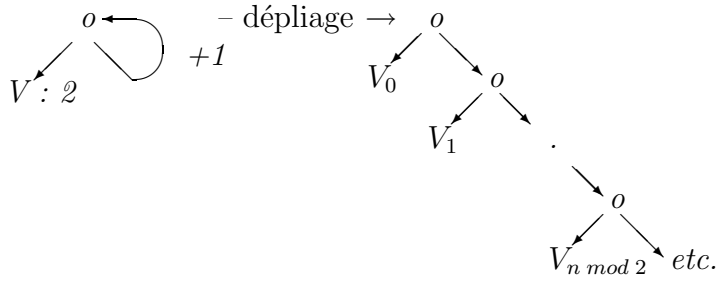


C'est un graphe orienté : il y a cinq noeuds étiquetés par des symboles de fonctions ($jeux, \circ$) ou des variables (P, M), les arcs sont orientés, et ce graphe contient une boucle. Cependant, la racine ($jeux$) est pondérée par 0, deux flèches sont pondérées par -1 et +1, et la variable P a pour période 2. □

Dépliage et Interprétation.

Le *dépliage* des graphes orientés génèrent un arbre régulier infini. Ici, il est généralisé par une pondération des arcs et l'affectation de période à certaines variables. Les poids le long des branches sont additionnés et fournissent les indices (modulo la période) des variables, le poids initial (pi) étant celui de la racine. En fait, cette interprétation se fait relativement à un intervalle $\mathcal{J}$ de $\mathbb{Z}$. Dès qu'un indice sort des limites de cet intervalle, le dépliage de la branche correspondante s'arrête. La feuille de cette branche est alors étiquetée par une variable spéciale indicée du poids courant.

Exemple 11 Voici un exemple de dépliage d'un GOP :



□

L'*interprétation* d'un GOP $\mathcal{G}$, est donc un arbre non–rationnel du fait du nombre infini de variables, on la note $\mathcal{I}^{pi}_{\mathcal{J}}(\mathcal{G})$.

[1] tiré de [16].

Unification.

Un algorithme d'unification des GOPs est présenté dans [16]. Deux GOPs, $\mathcal{G}$ et $\mathcal{G}'$, sont dits *unifiables* si, pour un poids initial donné, les arbres correspondant à leur dépliage sont unifiables.

Notant $[m, M]_{a\rightarrow\leftarrow b}$ (resp. $[m, M]_{a\leftarrow\rightarrow b}$), l'intervalle $[m + a, M - b]$ (resp. $[m - a, M + b]$), on établit le théorème suivant :

Théorème 4 [16] *Soient $\mathcal{G}$ et $\mathcal{G}$' deux graphes orientés pondérés finis unifiables et $\mathcal{J}$ un intervalle d'interprétation, alors il existe deux constantes, a et b, telles que :*

$$\mathcal{I}^{pi}_{\mathcal{J}_{a\rightarrow\leftarrow b}}(\mathcal{G} \vee \mathcal{G}') \leq \mathcal{I}^{pi}_{\mathcal{J}}(\mathcal{G}) \vee \mathcal{I}^{pi}_{\mathcal{J}}(\mathcal{G}') \leq \mathcal{I}^{pi}_{\mathcal{J}_{a\leftarrow\rightarrow b}}(\mathcal{G} \vee \mathcal{G}')$$

Preuve. Voir Théorème 4.10 pp. 192 de [16]. □

Ce théorème est fondamental dans l'utilisation des GOPs. Il signifie que les effets de bord liés à l'unification ont une taille constante égale à a à gauche et à b à droite. En conséquence, après avoir supprimé les effets de bord, la solution approchée ne dépend plus de l'intervalle d'interprétation $\mathcal{J}$.

8.1.2 GOP et Clause de Horn

Considérons la clause

$$G \leftarrow D.$$

Un GOP permet de calculer le plus petit terme, non rationnel en général, qui est constant (au renommage près) par rapport à la réécriture dûe à cette règle binaire. En fait, ce terme est *semi–rationnel*, c'est–à–dire que la non–rationnalité n'est dûe qu'à un nombre infini de variables distinctes, en fait ici de variables d'indices distincts. Plus précisément, ce terme est égal, une fois les indices supprimés, à l'arbre rationnel représentant le terme unifié $G \vee D$.

J'ai décrit précédemment, l'interprétation d'un GOP. Pour rendre compte de n applications de la règle, il suffit de ne tenir compte que du dépliage obtenu sans rencontrer de poids courant supérieur à n, c'est–à–dire de choisir $\mathcal{J}$=$[1, n]$.

8.1.3 Propriétés

Dans un GOP, un *boucle positive* (resp. *négative*) caractérise un terme dont la taille décroit (resp. croit) pendant les inférences. C'est–à–dire une relation de la forme $X_i = f(X_{i+k})$ (resp. $X_{i+k} = f(X_i)$).

Notation 1 J'utiliserai les conventions suivantes

- $G_i \leftarrow D_i$ est la clause où G_i (resp. D_i) est le terme G (resp. D) dont les variables ont été indexées par i.

- S_i est l'unificateur le plus général de $G_i = D_{i-1}$ (c'est–à–dire une solution de l'équation). Remarquons que S_i et S_j sont les mêmes aux indices près.
- σ_n est l'unificateur le plus général de $\{G_i = D_{i-1} \mid i \in [2, n]\}$, c'est–à–dire que $\sigma_n = S_n \circ \cdots \circ S_2$.
- $G^n \leftarrow D^n$ est la clause dont une application équivaut à n applications de la clause $G \leftarrow D$. Ces deux clauses sont équivalentes par rapport au problème de l'arrêt.
- G^∞ désigne la limite de la suite $(G^n)_{n \in \mathbf{N}}$, $G^\infty = \sigma_\infty(G_1)$.

Proposition 7 *Si G^∞ est linéaire, alors pour tout symbole de fonction f, l'ensemble des chemins de G^∞ qui mènent à un noeud étiqueté par f, noté $Dom_f(G^\infty)$, est un langage rationnel.*

Preuve. A partir du Théorème 4. □

Proposition 8 *Si une clause récursive, $G \leftarrow D$, est linéaire gauche (G est linéaire) alors G^∞ est linéaire.*

Preuve. G_1 est linéaire. Nous allons étudier dans un premier temps, la linéarité de $S_2(G_1)$. Soit X une variable appartenant à $Var(G_1)$ et $Var(D_1)$[2] (si X n'appartenait pas à $Var(D_1)$, elle ne serait pas affectée par la substitution). Puisque ce sont les termes de la forme $\sigma_n(G_1)$ qui nous concernent, le seul cas intéressant est celui où X est dans le domaine de la substitution S_2.

Supposons que nous ayons à unifier X avec un terme t de G_2. Nécessairement t est linéaire. Il est facile de vérifier qu'aucune variable de t n'apparaît dans le domaine de S_2. En effet, G_2 est linéaire, donc une variable de t n'apparaît nulle part ailleurs dans G_2 et donc ne peut pas avoir été substituée par un terme de D_1. Ainsi dans S_2, nous trouvons la substitution $\{X/t_2\}$, où t ne contient que des variables de G_2.

Considérons maintenant S_3. Puisque S_3 et S_2 sont égales (aux indices près), ils partagent les mêmes propriétés. Ainsi, toute variable de $Var(G_2) \cap Var(D_2)$ qui apparaît dans le somaine de S_3 est substituée par un terme linéaire qui ne contient que des variables de G_3. Ceci est vrai en particulier pour les variables de t. En conséquence, nous pouvons affirmer que $\sigma_3(X) = S_3 \circ S_2(X)$ est un terme linéaire. Et donc $\sigma_3(G_1)$ est linéaire.

Supposons que cela est vérifié pour tout $\sigma_i(G_1)$, avec $i \leq n$. Nous étudions S_{n+1}, le même raisonnement que précédemment est possible puisqu'il suffit de changer les indices. Donc $\sigma_{n+1}(X) = S_{n+1} \circ \sigma_n(X)$ est linéaire, d'où $\sigma_{n+1}(G_1)$ est linéaire. Ainsi, nous pouvons affirmer que pour tout n, $\sigma_n(G_1)$ est linéaire. Donc $\sigma_\infty(G_1) = G^\infty$ est linéaire. □

[2]Nous dénoterons $Var(E)$ l'ensemble des variables présentes dans E.

8.2 Les Systèmes Pondérés d'Equations

Les systèmes d'équations et leurs notions d'unification, solvabilité, test d'occurence, etc., fournissent un bon cadre mathématique pour l'étude de la sémantique opérationnel des langages à clauses de Horn. Ils ont été étendus dans [13] aux systèmes pondérés d'équations définis à partir d'une relation pondérée (proche d'un GOP) et d'un système d'équations sur des classes d'équivalence (une variable X représentant la classe de toutes les variables indicées X_i). Ce formalisme rend compte exactement du comportement itératif d'une règle binaire. Il a été étendu ensuite aux liens exceptions et aux liens linéaires dans [17] afin de pouvoir tenir compte des but et fait, éventuellement non–linéaires.

Dans ce qui suivra, $\mathcal{I}$ désignera un intervalle de $\mathbb{Z}$.

8.2.1 Relations Pondérées de Base

Definition 22 Une *relation pondérée de base* sur $Var \times \mathcal{I}$ est formalisée par le graphe élémentaire suivant, où X et Y sont dans Var et ω *dans* $\mathbb{N}$:

$$X \xrightarrow{\omega} Y$$

Son interprétation sur $\mathcal{I}$ est :

$$\forall (i, i+\omega) \in \mathcal{I}^2, X_i = Y_{i+\omega}.$$

Definition 23 Une *relation pondérée simple*, $\mathcal{R}$, est un ensemble de relations de base, c'est–à–dire un graphe orienté dont les noeuds sont étiquetés par des symboles de variable et dont les flèches sont pondérées par des entiers relatifs.

L'interprétation sur $\mathcal{I}$ de $\mathcal{R}$ est la clôture transitive de l'union de l'interprétation sur $\mathcal{I}$ de toutes ses relations de base.

Théorème 5 [13] *L'équivalence de deux relations pondérées est décidable.*

Théorème 6 [13] *Etant donné un ensemble de variables, il ne peut y avoir de suite infinie croissante de relations pondérées.*

8.2.2 Systèmes Pondérés d'Equations

Definition 24 Un *système pondéré d'équations* est composé d'une relation pondérée sur un ensemble $\mathcal{S}$ de variables et d'un ensemble d'équations liant ces variables.

Son interprétation sur $\mathcal{I}$, est le système obtenu par union des interprétations de sa relation pondérée et de ses équations.

Exemple 12 Voici un système pondéré :

$$\alpha = succ(X)$$
$$X \xrightarrow{1} \alpha$$

Son interprétation sur $\mathcal{I}$=$[1, n]$ est :

$$\forall i \in [1, n], \alpha_i = succ(X_i)$$
$$\forall i \in [1, n-1], X_i = succ(\alpha_{i+1})$$

c'est–à–dire, sous la forme résolue *:*

$$\forall i \in [1, n], X_i = succ^{n-i}(X_n)$$
$$\forall i \in [2, n-1], \alpha_i = succ^{n-i+1}(X_n)$$

□

Il est facile d'étendre les notions et algorithmes classiques d'unification, solvabilité, test d'occurence, etc. des systèmes d'équations habituels. Il suffit en effet d'ajouter quelques règles aux algorithmes existants. Voici par exemple une des règles qu'il faut ajouter à l'*algorithme de simplification* :

Fusion haut–vers–bas	
$X = f(U^1, U^2, \cdots, U^n)$	pour tout i ajouter les relations de base
$Y = f(V^1, V^2, \cdots, V^n)$	héritées sur U^i et V^i de X et Y.

Cet algorithme permet alors de calculer finiment, l'*unique* forme minimale du système considéré.

Ces systèmes pondérés formalisent le comportement des règles binaires. Les variables du système correspondent aux variables formelles de la clause et les indices au numéro de l'inférence. En effet, comme nous l'avons vu à la section 6.2, le comportement itératif d'une clause de Horn binaire telle que

$$gauche \leftarrow droite$$

peut être, facilement et naturellement, exprimée par un système de la forme

$$\{gauche_{i+1} = droite_i \mid \forall i \in [1, n-1]\}$$

Ce système peut lui–même être traduit directement en un système pondéré d'équations :

$$\text{relation pondérée} : \quad \alpha \xleftarrow{1} \beta$$
$$\text{équation} : \quad \alpha = gauche, \beta = droite$$

En fait l'application itérative de toute clause binaire peut être totalement caractérisée par un système pondéré solvable, $\mathcal{S}$. Le système simplifié d'équations correspondant à n inférences de la règle binaire est obtenu par interprétation de $\mathcal{S}$ sur $[1, n]$. De ce fait, il a été établi dans [13] une nouvelle preuve, plus simple que celle de [16], pour la décidabilité des problèmes de l'arrêt et du vide dans le cas d'un but et d'un fait linéaires.

Bien que plus puissants que les GOPs, les systèmes d'équations pondérés fournissent des outils et des preuves plus faciles à utiliser. Ils restent cependant insuffisants si l'on veut rendre compte exactement de l'influence du fait ou du but lors de l'exécution d'un programme à une clause binaire. Ainsi dans

$$\begin{cases} p(X, Y) \leftarrow p(Y, Z) \ . \\ \leftarrow p(U, b) \ . \end{cases}$$

il est nécessaire de considérer les équations $X_1 = U$ et $Y_1 = b$ qui sont les contraintes imposées par le but. Pour cela nous allons introduire la notion de *lien exception*[17].

8.2.3 Relations Pondérées Exceptions

Definition 25 Un *lien exception* entre deux variables X et Y et d'indices i_1 et i_2 (avec $(i_1, i_2) \in \mathbb{N}_+^2$) est l'égalité entre les variables indicées X_{i_1} et Y_{i_2} formalisée par le graphe :

$$X \ {}_{i_1} - - - - - - - \ {}_{i_2} \ Y$$

son interprétation sur $\mathcal{I}$ est : si $(i_1, i_2) \in \mathcal{I}^2, X_{i_1} = Y_{i_2}$.

On peut alors étendre la définition des relations pondérées (et donc des systèmes pondérés d'équations) :

Definition 26 Une *relation pondérée étendue* est un ensemble de relations de base et de liens exceptions. Son interprétation sur $\mathcal{I}$ est la clôture transitive de l'union des interprétations sur $\mathcal{I}$ de toutes ses relations de base et liens exceptions.

On prouve également :

Théorème 7 *L'équivalence de deux relations pondérées étendues est décidable.*

Principe de preuve. La preuve de ce théorème est longue, technique et n'apporterait rien à notre propos. Je ne la donnerai donc pas. Elle peut être communiquée à toute personne intéressée.

Le principe en est le suivant : on prouve que deux relations sont équivalentes si il est possible de les transformer en une même relation grâce à des transformations données. Cette preuve constructive fournit en fait un algorithme décidant de l'équivalence de deux relations. □

Ces liens exceptions vont permettre de prendre en compte l'effet d'un but linéaire. Si l'on veut tenir compte du fait, il faut faire intervenir des égalités de la forme $X_n = V$, où n est le nombre d'inférences de la règle. Il serait donc nécessaire d'introduire une notion comparable à celles des variables degrés de [6].

Mais une fois encore, ce formalisme va se révéler limité. Si l'on veut prendre en considération un but non–linéaire comme dans

$$\begin{cases} p(s(X), s(s(Y))) \leftarrow p(X, Y) \ . \\ \leftarrow p(U, U) \ . \end{cases}$$

on constate qu'à chaque itération la taille de la seconde variable décroit de deux alors que celle de la première de un. Le but force l'égalité des deux arguments et on ressent le besoin d'exprimer une équation telle que $X_{2i} = Y_i$. Cela peut être réalisé à l'aide des *liens linéaires* [17].

8.2.4 Relations Pondérées Linéaires

Definition 27 Un *lien linéaire* est formalisé par le graphe suivant, où X et Y sont des variables et $(a, a', b, b') \in \mathbb{N}^2 \times \mathbb{N}_+^2$:

$$X \ {}_{(ai+b)} \text{———} {}_{(a'i+b')} \ Y$$

Son interprétation sur $\mathcal{I}$ (de la forme $[1, n]$) est :

$$\forall i \in \mathbb{N}, \ \textit{tel que } (ai + b, a'i + b') \in \mathbb{N}^2, X_{ai+b} = Y_{a'i+b'}.$$

Remarquons que les relations de base et les liens exceptions sont inclus dans les liens linéaires :

$$\begin{array}{rcl} X \xrightarrow{\omega} Y & \text{est équivalent à} & X \ {}_{(i+1)} \text{———} {}_{(i+\omega)} \ Y \\ X \ {}_{i_1} - - - - - - - {}_{i_2} \ Y & \text{est équivalent à} & X \ {}_{(0i+i_1)} \text{———} {}_{(0i+i_2)} \ Y \end{array}$$

On peut donc étendre de manière tout à fait naturelle la définition des relations pondérées ainsi :

Definition 28 Une *relation pondérée linéaire* est un ensemble de liens linéaires. Son interprétation sur $\mathcal{I}$ est la clôture transitive des interprétations sur $\mathcal{I}$ de tous ses liens linéaires.

Ces relations linéaires ont été à l'origine des travaux qui vont suivre, sur la codification des fonctions de Conway du chapitre suivant par exemple. Il serait possible d'exprimer à l'aide de ces relations la plupart des résultats à venir. La proposition 9 établit en effet qu'à tout couple (clause binaire,but) (but a priori non–linéaire), on peut associer une telle relation.

Ainsi si le théorème suivant peut être établi comme une conséquence de l'indécidabilité du problème de l'arrêt (Théorème 10), j'aurai pu, en présentant les choses différemment inverser les rôles (Cf. [17]).

Théorème 8 *L'équivalence de deux relations pondérées linéaires est indécidable.*

Preuve. [3]On construit deux relations pondérées linéaires. La première contient tous les liens linéaires

$$X \ {}_{(ai+b)} \overline{\qquad\qquad} {}_{(a'i+b')} \ X$$

traduisant une fonction de Conway nulle g. La seconde contient ces mêmes liens avec en plus le lien exception traduisant l'égalité $X_{2^n} = X_1$:

$$X \ {}_{(0i+2^n)} \overline{\qquad\qquad} {}_{(0i+1)} \ X$$

On en déduit que ces deux relations seront équivalentes si et seulement si n est dans le domaine de g, ce qui est indécidable. Cette démonstration revient exactement au même que celle du théorème 10 de la page 87, mais en utilisant cette fois les liens linéaires plutôt que les clauses de Horn. □

[3]Il est conseillé de ne revenir sur cette preuve qu'après avoir étudié celle du théorème 10 de la page 87.

Chapitre 9

Clauses Binaires et Fonctions de Conway

Nous allons maintenant établir le lien entre les clauses de Horn binaires qui sont le point central de cette étude et les fonctions de Conway présentées précédemment. Une fonction de Conway g s'exprime par des relations de la forme $g(n) = a_i n$ avec $n = \alpha d + i$ et telles que $a_i n$ soit toujours entier. C'est–à–dire en fait, qu'une fonction g associe à un nombre de la forme $\alpha d + i$ un nombre de la forme $(a_i\alpha)d + (a_i i)$ avec $\alpha, i, a_i\alpha, a_i i$ tous entiers.

Après avoir montré que l'on peut exprimer à l'aide d'une clause binaire et d'un but, toute relation qui associe à un nombre $ai + b$ un autre nombre $ci + d$ avec a, b, c, d entiers, nous établirons de manière immédiate que le codage des fonctions de Conway par une clause binaire est possible. Ce codage sera décrit explicitement. Nous en déduirons une caractérisation des ensembles récursivement énumérables par une clause binaire.

9.1 La Codification

Exemple 13 Considérons le programme suivant *:*

$$\begin{cases} p((s(x), s(s(Y))) \leftarrow p(X, Y) \ . \\ \leftarrow p(U, U) \ . \end{cases}$$

il crée les égalités entre les variables indicées comme indiquées dans le schéma qui suit et d'où l'on déduit la relation *:*

$$Y_i = X_{2i}.$$

En effet la taille de la seconde variable augmente de 2 pendant que celle de la première n'augmente que de 1.

$$
\begin{array}{lccccccccccccc}
U = & s & s & s & s & s & & U = & s & s & s & s \\
 & X_1 & s & s & s & s & & & s & s & s & s \\
 & & X_2 & s & s & \vdots & \cdots\cdots\cdots & Y_1 & s & s & \vdots \\
 & & & X_3 & s & \vdots & & & s & s & \vdots \\
 & & & & X_4 & \vdots & \cdots\cdots\cdots\cdots\cdots & Y_2 & s & s & \vdots \\
 & & & & & X_n & & & & s & \vdots \\
 & & & & & & & & & Y_3 & \vdots \\
 & & & & & & & & & & Y_n
\end{array}
$$

□

D'une manière générale, nous allons établir ici que n'importe quelle relation de la forme

$$X_{ai+b} = Y_{a'i+b'}$$

peut être obtenue à l'aide d'une clause binaire et d'un but non linéaire. Le codage sera tout à fait similaire à celui de l'exemple : il ne nécessite que l'emploi d'un symbole de fonction. Toutefois, par souci de lisibilité j'utiliserai de preférence le constructeur de listes plutôt que la fonction $s(\_)$ de l'exemple.

En fait, nous n'étudierons que des relations de la forme :

$$X_{ai+b} = X_{a'i+b'},$$

puisque ce sont les seules qui nous seront utiles par la suite. La production de relations entre deux variables distinctes X et Y sera immédiate à partir de ce qui va être présenté.

Proposition 9 *Pour tout quadruplet d'entiers naturels a, a', b, b', il existe une variable X, une clause binaire linéaire droite $p(t) \leftarrow p(tt)$ et un but $\leftarrow p(\gamma)$ tels que :*

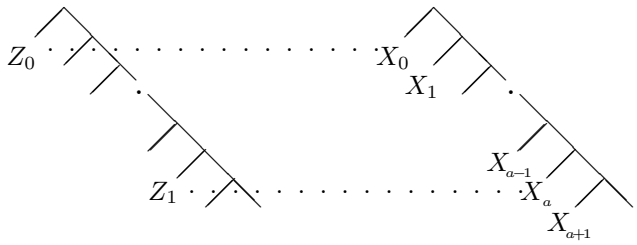
$$(\{\gamma = t_1\} \cup \{tt_i = t_{i+1} \mid \forall i > 0\}) \uparrow_{\{X\}} \equiv \{X_{ai+b} = X_{a'i+b'} \mid i > 0\}$$

où $S \uparrow_{\{X\}}$ est la projection sur les variables X_i des équations exprimées dans S.

Preuve. Nous allons en premier lieu nous intéresser au cas où a' vaut 1 et b et b' sont nuls. Etudions ensemble le comportement de la variable X dans le programme suivant :

$$\begin{cases} p(\overbrace{[Z, \_, \cdots, \_}^{a} | L, [X|LL]) \leftarrow p(L, LL) \ . \\ \leftarrow p(L, L) \ . \end{cases}$$

Comme dans l'exemple précédent, la taille de la première variable de la clause de Horn diminue de a pendant que celle de la seconde diminue de 1, ainsi on obtient :



L'égalité des deux arguments dans le but provoque l'égalité des deux termes ci–dessus. En suivant les liens dessinés en pointillés, on déduit la relation $Z_i = X_{ai}$.

Supposons maintenant que nous voulions établir une relation telle que $Z_i = X_{ai+b}$, il résulte de ce qui précède que nous n'avons qu'à décaler les égalités des deux termes. Cela se fait très facilement en prenant pour but :

$$\leftarrow p([\underbrace{\_, \cdots, \_}_{b} \mid L], L).$$

On comprend bien que dans ce cas, les relations n'affecteront les variables Z_i qu'à partir de $i \geq b$.

Maintenant si nous combinons deux relations $Z_i = X_{ai+b}$ et $Z_i = X_{a'i+b'}$. La clôture transitive de ces relations et leur projection sur la variable X produira la relation souhaitée

$$X_{ai+b} = X_{a'i+b'}$$

obtenue par le programme :

$$\begin{cases} p([\overbrace{Z, \_, \cdots, \_}^{a}|L1], [X|L2], [\overbrace{Z, \_, \cdots, \_}^{a'}|L3], [X|L4]) \\ \qquad\qquad \leftarrow p(L1, L2, L3, L4) \ . \\ \leftarrow p([\underbrace{\_, \cdots, \_}_{b}|L], L, [\underbrace{\_, \cdots, \_}_{b'}|LL], LL) \ . \end{cases}$$

□

Remarque 3 Une autre codification de la relation $X_{ai+b} = X_{a'i+b'}$, lorsque $b < a$ et $b' < a'$, est possible *:*

$$\begin{cases} p([\underbrace{\_, \cdots, Z}_{b}, \_, \cdots|L1], [X|L2], [\underbrace{\_, \cdots, Z}_{b'}, \_, \cdots|L3], [X|L4]) \\ \qquad\qquad \leftarrow p(L1, L2, L3, L4) \ . \\ \leftarrow p(L, L, LL, LL) \ . \end{cases}$$

Remarque 4 Un quadruplet d'entiers (a, a', b, b') comme celui dont il est question dans la proposition caractérise en fait le lien linéaire :

$$X \ {}_{(ai+b)} \text{———} {}_{(a'i+b')} \ X$$

La codification des fonctions de Conway par une clause binaire récursive est maintenant immédiate. On peut donc établir la proposition suivante.

Proposition 10 *Pour toute fonction de Conway g, il existe une variable X, une clause binaire linéaire droite $p(t) \leftarrow p(tt)$ et un but $\leftarrow p(\gamma)$ tels que :*

$$(\{\gamma = t_1\} \cup \{tt_i = t_{i+1} \mid \forall i > 0\}) \uparrow_{\{X\}} \equiv \left\{X_n = X_{g(n)} \mid i > 0\right\}$$

Preuve. Soit g une fonction de Conway. Elle est définie par la donnée de d, $a_1, \cdots, a_{d-1}$. Comme nous l'avons étudié précédemment, g peut être décomposée en un nombre fini de relations de la forme $(X_{ai+b} = X_{a'i+b'})_{i>0}$ où a, b, a' et b' sont des entiers naturels. D'après la proposition précédente, il est possible d'associer à chacune de ces relations une clause de Horn binaire et un but qui peuvent tous être fusionnés pour ne former qu'une seule règle linéaire droite et un but qui satisfont la proposition. □

On peut noter ici, que l'on code en fait les relations de Conway autant que les fonctions elle–mêmes.

Exemple 14 Reprenons la conjecture de Collatz et montrer qu'elle peut être exprimée par une règle récursive binaire et un but. Cela n'est d'ailleurs pas pour nous surprendre, elle peut en effet être exprimée en terme de relation d'équivalence sur $Var \times \mathbb{N}$ ainsi *:*

$$\forall k \in \mathbb{N}, \mathsf{Si}\ k \text{ est pair } \mathsf{Alors}\ X_k = X_{\frac{k}{2}} \mathsf{Sinon}\ X_k = X_{3k+1}$$

Soit f la fonction telle que $\forall i > 0$, $f(2i) = i$ et $f(2i-1) = 6i - 2$. Puisqu'il n'existe pas de $k \in \mathbb{N}$ tel que $f^{(k)}(1) = n$ $(\forall n > 4)$, il est possible d'exprimer cette relation par le système de relations suivant *:*

$$\left\{ \begin{array}{rcl} X_i & = & X_{2i} \\ X_{2i-1} & = & X_{3(2i-1)+1} \end{array} \right.$$

Or nous avons vu que ce type de système est exprimable par une clause binaire récursive. Voici cette clause, légèrement arrangée par rapport à ce qui est décrit précédemment, deux des arguments résultants de la construction décrite dans les preuves précédentes ont été rassemblés en un seul.

$$\left\{ \begin{array}{l} p(\overbrace{[X|U]}^{L_1}, \overbrace{[Y, X|V]}^{L_2}, \overbrace{[\_, \_, \_, Y, \_, \_|W]}^{L_3}) \leftarrow p(U, V, W) \ . \\ \leftarrow p(Z, Z, Z) \ . \end{array} \right.$$

A partir d'un but général $\leftarrow p(L_1, L_2, L_3)$, par itérations successives de la règle on obtiendrait *:*

$$\begin{array}{lll} 1. & L_1 = [X_1|U_1] & L_2 = [Y_1, X_1|V_1] \\ 2. & L_1 = [X_1, X_2|U_2] & L_2 = [Y_1, X_1, Y_2, X_2|V_2] \\ \vdots & \vdots & \vdots \\ n. & L_1 = [X_1, X_2, \cdots, X_n|U_n] & L_2 = [Y_1, X_1, Y_2, X_2 \cdots, Y_n, X_n|V_n] \end{array}$$

$$\begin{array}{ll} 1. & L_3 = [\_, \_, \_, Y_1, \_, \_ \mid W_1] \\ 2. & L_3 = [\_, \_, \_, Y_1, \_, \_, \_, \_, Y_2, \_, \_ \mid W_2] \\ \vdots & \vdots \\ n. & L_3 = [\_, \_, \_, Y_1, \_, \_, \cdots, \_, \_, \_, Y_n, \_, \_ \mid W_n] \end{array}$$

C'est–à–dire après n itérations *:*

$$\begin{array}{lllllllll} L_2 = & [Y_1, & X_1, & Y_2, & X_2, & Y_3, & X_3, & \cdots, Y_n, & X_n|V_n] \\ L_1 = & [X_1, & X_2, & X_3, & X_4, & X_5, & X_6, & \cdots, X_{n-1}, & X_n|U_n] \\ L_3 = & [\_, & \_, & \_, & Y_1, & \_, & \_, & \cdots, \_, Y_n, & \_, \_|W_n] \end{array}$$

Et donc avec le but $\leftarrow p(Z, Z, Z)$ qui force les égalités entre L_1 et L_2 et entre L_1 et L_3, on obtient *:*

1. $L_1 = L_2 \Rightarrow X_{2i-1} = Y_i$ et $X_{2i} = X_i$

2. $L_1 = L_3 \Rightarrow X_{6i-2} = Y_i$

c'est–à–dire

$$X_i = X_{2i} \text{ et } X_{2i-1} = X_{6i-2}$$

Maintenant avec un but de la forme

$$\leftarrow p([\underbrace{a, \_, \cdots, \_, \bar{a}}_{n} \mid L], [\underbrace{a, \_, \cdots, \_, \bar{a}}_{n} \mid L], [\underbrace{a, \_, \cdots, \_, \bar{a}}_{n} \mid L]).$$

nous forçons les égalités $X_1 = a$ et $X_n = \bar{a}$. En conséquence, avec un tel but, la résolution sera finie si et seulement si une unification échoue du fait de $X_n \neq X_1$, c'est–à–dire si le programme de Collatz termine avec l'entrée n. Autrement dit, la conjecture de Collatz est équivalente à prouver que pour n'importe quel n, avec un but de la forme ci–dessus, la résolution termine. □

9.2 Clauses et Récursivement Enumérable

Dans le chapitre concernant les fonctions de Conway, nous avons défini la notion de relation de Conway. Nous avons montré qu'elle permettait de caractériser les ensembles récursivement énumérables contenant $\{0\}$. Nous allons maintenant associer une clause binaire linéaire droite récursive et un but à de tels ensembles en accord avec ce qui a été présenté dans la section précédente.

Théorème 9 *Soit $\sharp$ un symbole particulier. Pour tout ensemble récursivement énumérable Σ contenant $\{0\}$, il existe une clause linéaire droite binaire récursive et un but tels que, un entier naturel n appartient à Σ si et seulement si, à partir d'un certain nombre d'étapes de résolution SLD, le premier argument du but initial est une liste dont le $(2^n)^{ème}$ élément est marqué par $\sharp$. Nous appellerons cette liste liste caractéristique de Σ.*

Preuve. En accord avec les deux propositions de la section précédentes, soit X la variable utilisée pour coder la relation de Conway associée à Σ (Cf. proposition 6). La liste L est alors progressivement construite comme $[X_1, X_2, \ldots, X_n, \ldots]$, avec les variables X_i liées entre elles par les relations $X_i = X_{g(i)}$. D'après la proposition 6, on en déduit donc

$$\Sigma = \{n \in \mathbb{N} \mid X_{2^n} \leftrightarrow_g X_1\}$$

Si de plus, au départ, la variable X_1 est instanciée à $\sharp$, alors cette marque sera propagée à tous les X_{2^n} tels que n appartienne à Σ. □

Ce théorème est fondamental dans la mesure où il est à la base de toutes les preuves qui suivront. En effet, la clause associée à un ensemble récursivement énumérable peut être considéré comme un processus d'énumération de cet ensemble, il suffit d'initialiser X_1 à $\sharp$ comme expliqué et d'appliquer itérativement cette clause. Nous construirons ainsi des ensembles et utiliserons les résultats liés à ceux–ci. En fait la clause peut être considéré comme un processus d'énumération de l'ensemble récursivement énumérable qui lui est associé.

Exemple 15 Dans le cas où Σ est égal à $\mathbb{N}$, le programme suivant satisfait le théorème précédent *:*

$$\begin{cases} p([X|L], [\_, X|LL]) \leftarrow p(L, LL) \ . \\ \leftarrow p([\sharp|L], [\sharp|L]) \ . \end{cases}$$

En effet ce programme marque par $\sharp$ toutes les $(2^n)^{\text{èmes}}$ positions de la liste $[\sharp \mid L]$ du but.

A titre de curiosité, il est possible de modifier légérement ce programme pour qu'il marque non seulement d'un $\sharp$ les $(2^n)^{\text{èmes}}$ positions, mais qu'il marque d'un autre symbôle ($\flat$) les autres *:*

$$\begin{cases} p([X|L],[Y,X|LL],[\flat,Z|LLL]) \leftarrow p(L,LL,LLL) \ . \\ \leftarrow p([\sharp,\sharp|L],[\sharp,\sharp|L],L) \ . \end{cases}$$

Ce petit programme nous sera utile par la suite. □

Chapitre 10

Le Problème de l'Arrêt

Nous allons répondre dans ce chapitre au premier des problèmes que nous avions soulevés : celui de l'arrêt d'une clause binaire récursive pour un but donné. Alors qu'il avait été établi comme étant décidable dans le cas d'un but clos [46] ou linéaire [16], nous allons établir ici l'indécidabilité de ce problème dans le cas d'un but général[1]. En fait il suffit que la règle soit linéaire droite. Pour compléter la réponse, nous établirons ensuite la décidabilité dans le cas où la règle est linéaire gauche.

10.1 Le Cas Général

La preuve est facile à partir du théorème 9, nous provoquerons l'arrêt grâce à l'échec d'une unification. Cet échec ne se produisant que si un élément appartient à un ensemble récursivement énumérable...

Théorème 10 *Le problème de l'arrêt d'une clause de Horn binaire linéaire droite pour un but donné est indécidable, que la résolution soit faite avec ou sans le test d'occurence.*

Preuve. C'est une conséquence directe du théorème 9 utilisant le même principe que l'exemple sur la codification du problème de Collatz . En initialisant la liste L du but à $[\sharp, X_2, \ldots, X_{2^n-1}, \flat \mid LL]$ où la marque $\flat$ est placée sur le $(2^n)^{\text{ème}}$ élément de L, alors la résolution finira si et seulement si l'équation $X_1 = X_{2^n}$, soit $\sharp = \flat$, se produit, c'est–à–dire, si et seulement si n est un élément de Σ. Nous avons vu en première partie que le problème de déterminer si un élément appartenait à un ensemble récursivement énumérable était indécidable (Cf. page 28), ce qui établit le théorème. On vérifie aisément que le test d'occurence ne joue aucun rôle ici. □

[1] Ce résultat a été publié initialement dans [18].

Mais il est possible de faire encore mieux ! Le théorème suivant établit le même résultat mais pour *une* clause de Horn *particulière*. La preuve est un peu plus délicate que la précédente...

Théorème 11 *Il existe une clause de Horn binaire linéaire droite, explicitement constructible, pour laquelle le problème de l'arrêt pour un but donné est indécidable. La résolution peut être faite avec ou sans le test d'occurence.*

Preuve. Considérons une fonction récursive partielle ϕ associée au programme :

- lire f, n
- calculer $f(n)$
- écrire 0

où f est une fonction récursive partielle. Ce programme répond 0 si et seulement si n est dans le domaine de la fonction f (c'est–à–dire si et seulement si f s'arrête avec l'entrée n). Il est possible de caractériser f par son nombre de Gödel[2]. En conséquence, on peut considérer l'entrée de ϕ comme un entier : le nombre $2^f 3^n$. Ainsi ϕ n'a qu'un seul argument. Il est évident que l'on peut imposer $\phi(0) = 0$ sans influencer quoique ce soit.

En résumé, nous avons une fonction ϕ telle que $\phi(0) = 0$ et telle que pour toute entrée de la forme $2^f 3^n$, ϕ donne 0 si et seulement si n est dans le domaine de f. Donc, $2^f 3^n$ est dans le domaine de ϕ si et seulement si n est dans le domaine de f. Soit Σ le domaine de ϕ (remarquons que $\{0\} \in \Sigma$), $2^f 3^n$ appartient à Σ si et seulement si n est dans le domaine de f, ce qui est indécidable. Maintenant, si l'on associe une clause de Horn à Σ en accord avec le théorème 9 et comme dans la preuve du théorème 10, en initialisant la liste du but à $[\sharp, \ldots, \flat \mid L]$ où le symbole $\flat$ est en position $2^{2^f 3^n}$, il est alors impossible de décider si ce programme particulier s'arrêtera. On en conclut donc que cette clause et ce but satisfont le résultat. On vérifie que le test d'occurence ne joue aucun rôle. □

On pourrait éventuellement discuter le "explicitement constructible" de l'énoncé du théorème, mais il se justifie. Il "suffit" en effet de :

- construire la machine de Minsky asssociée au petit programme présenté,
- trouver le nombre de Gödel de la fonction f passée comme argument,
- construire la fonction de Conway associée à la sus–citée machine de Minsky,

et voilà... un jeu d'enfant...

[2]Dans la suite de la démonstration, le symbôle f représentera indifféremment la fonction ou le nombre de Gödel associé. Le contexte devrait lever toute ambiguité éventuelle.

10.2 Conséquences

Quelques corollaires peuvent être établis immédiatement. Dans chaque cas, on peut évidemment établir une version générale et une version "explicitement constructible". Nous ne donnerons que la seconde qui implique évidemment la première.

10.2.1 Nombre de Solutions Fini et Programmes Bornés.

Corollaire 12 *Il existe un programme explicitement constructible de la forme*

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \end{cases}$$

où $fait$, $gauche$ et $droit$ sont des termes, tel qu'il soit indécidable de savoir, pour un but "$\leftarrow p(but)$" donné, si il existe un nombre fini de substitutions réponses.

Preuve. Considérons le programme construit à partir de :

- la cause binaire et le but décrits dans la preuve précédente,
- un fait "$p(X) \leftarrow$" où X est une variable.

A chaque fois que le fait est examiné, on obtient une solution. Ce programme en aura donc un nombre fini si et seulement si la clause binaire ne s'arrête pas pour le but donné, ce qui a été montré indécidable par le théorème 11. □

Ce résultat a pour conséquence immédiate que :

Corollaire 13 *Il existe un programme particulier explicitement constructible construit à partir d'une clause de Horn binaire linéaire droite, d'un but et d'un fait pour lequel il est indécidable de déterminer si il est borné*[3] *ou non.*

Preuve. Considérons le programme utilisé dans la preuve précédente, étant donné qu'il est indécidable de savoir si il possède un nombre fini de solutions, le fait que ce programme soit borné est a fortiori indécidable. □

Cette propriété était déjà connue comme indécidable, même pour des programmes DATALOG[4] linéaires [21]. Cependant, elle est décidable pour les programmes linéaires à un seul prédicat binaire, la complexité étant alors NP–complète [48]. Ici, nous avons donc résolu le cas des programmes à une règle binaire linéaire droite : il est indécidable.

[3] Le terme anglais correspondant est "*bounded*". Un programme possède cette propriété si il existe un programme constitué d'un nombre fini de clauses unité (c'est–à–dire de faits) qui lui soit équivalent, c'est–à–dire si il est possible d'en éliminer la récursivité.

[4] C'est–à–dire sans symbole de fonction.

10.2.2 Test d'Occurrence

Les résultats précédents ont été établis indépendamment de la présence du test d'occurrence. A partir de ceux–ci nous pouvons de plus établir qu'il est indécidable de déterminer pour un programme donné si ce test d'occurrence doit ou non être appliqué lors de la résolution.

Corollaire 14 *Il existe un programme explicitement constructible linéaire droit pour lequel il est indécidable, pour un but donné, de déterminer si le test d'occurrence est nécessaire ou pas.*

Preuve. Il suffit dans la preuve du théorème 11 de remplacer les égalités :

$$X_1 = \sharp \text{ et } X_{2^n} = \flat$$

du but par :

$$X_1 = h(Y, s(Y)) \text{ et } X_{2^n} = h(Z, Z)$$

où h et s sont des symboles de fonction d'arité 2 et 1 respectivement. Il en découle qu'il est indécidable de savoir si le programme s'arrêtera ou non du fait des équations $Z = Y$ et $Z = s(Y)$, c'est–à–dire à cause du test d'occurrence. □

Pouvoir décider d'une telle propriété aurait pourtant été bien intéressant. Rappelons, cela a déjà été dit précédemment, que ce test est très couteux et est donc généralement omis lors des résolutions. Etre assuré qu'en s'en passant on ne risque pas de le regretter un jour serait très intéressant. On parle alors de *programmes NSTO* pour "Non Soumis au Test d'Occurence" (ou "Non Subject To Occur–check" pour les anglophones). Malheureusement, nous venons d'établir ici que même pour les programmes les plus simples, cela n'est pas possible.

10.2.3 Décoration Totale

Voici le dernier résultat découlant de l'indécidabilité de l'arrêt de notre classe de programmes. Il concerne la propriété de *décoration totale* dans la résolution d'un programme logique. Cette propriété est utilisée pour optimiser le passage de la programmation logique aux grammaires attribuées [5]. Une résolution SLD d'un programme est dite à décoration totale si et seulement si à chaque étape de la résolution toutes les clauses de Horn de ce programme peuvent être utilisées. Dans le cas qui nous concerne ici, il n'y a qu'une seule clause. La propriété de décoration totale est donc ici équivalente au problème de l'arrêt, ou plutôt du non–arrêt, de cette clause. On peut donc affirmer :

Corollaire 15 *Il existe une clause binaire particulière linéaire droite, pour laquelle il est indécidable, pour un but donné, de déterminer si sa résolution sera à décoration totale.*

Preuve. Le paragraphe précédent tient lieu de preuve. □

Ce problème était donné ouvert dans [5].

10.3 Cas de la Clause Linéaire Gauche

Arrivés à ce point de notre discours, nous savons que le problème de l'arrêt est décidable si le but est linéaire et indécidable si la règle est linéaire droite. Pour compléter la réponse et satisfaire une curiosité naturelle et bien venue, il est maintenant légitime d'examiner le cas où elle est linéaire gauche. La preuve du résultat suivant utilise une méthode totalement différente de celle qui vient d'être utilisée. En effet, elle est basée sur une utilisation des graphes pondérés. Nous utiliserons donc certains des résultats présentés au chapitre 8.

Théorème 16 *Le problème de l'arrêt d'une clause de Horn linéaire gauche, pour un but quelconque donné est décidable.*

Preuve. Considérons la clause de Horn

$$G \leftarrow D$$

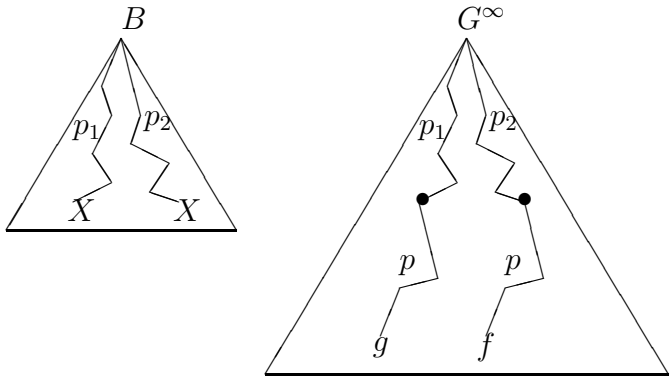
où G est linéaire. Je considérerai dans la suite que cette règle peut s'appliquer une infinité de fois à elle même, c'est–dire que G et D sont unifiables, sinon le problème devient trivial. Etant donné un but B, nous pourrons avoir une résolution infinie si et seulement si l'unification de B et de G^{∞}[5] est possible. Si B est clos ou linéaire, le problème est résolu dans [46] et [16] respectivement. Supposons donc que B est non–linéaire et qu'il existe donc deux chemins p_1 et p_2 dans B étiquetés par la même variable X. Notons N le nombre d'inférences nécessaire pour que la taille de toute branche croissante de G^N soit supérieure à celle des branches de B correspondantes. Deux cas sont possibles :

- si p_1, p_2 ou les deux à la fois n'appartiennent pas à $Dom(G^{\infty})$ (rappel : $Dom(G^{\infty})$ est l'ensemble des chemins menant à un noeud de G^{∞}). Alors après N applications de la règle binaire, une infinité d'autres est possible.

- si $(p_1, p_2) \in Dom(G^{\infty})^2$. Puisque G est linéaire, G^{∞} est linéaire, cela d'après la proposition 8. Nous pouvons donc considérer l'unification de chaque paires de sous–termes de G^{∞} et de B indépendamment les unes des autres. La résolution sera donc finie si et seulement si il existe deux chemins, $p_1.p$ et $p_2.p$, qui mènent à deux symboles de fonctions différents (Cf. la figure ci–dessous). C'est–à–dire si et seulement si

$$\exists p, \exists (f, g) \in \mathcal{F}^2 \mid G^{\infty}(p_1.p) = f, G^{\infty}(p_2.p) = g$$

[5] Je reprends ici les notations du chapitre 8



Ceci est équivalent à dire que :

$$p_1^{-1}.Dom_f(G^\infty) \cap p_2^{-1}.Dom_g(G^\infty) \neq \emptyset$$

Cela correspond à l'intersection de deux langages rationnels (Cf. proposition 7) qui donne un langage rationnel. Le problème du vide pour les langages rationnels étant connu comme décidable.

Nous en déduisons la décidabilité du problème de l'arrêt, pour un but donné, d'une clause de Horn binaire récursive linéaire droite. □

10.4 Conclusion

Il résulte des résultats des paragraphes précédents que pour un programme de la forme

$$\begin{cases} p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

dès que l'un des termes *gauche* ou *but* est linéaire, le problème de l'arrêt pour un but donné est décidable. Ces résultats sont récapitulés dans le tableau suivant :

but	*gauche*	*droit*	Arrêt ?
clos	quelconque	quelconque	décidable
linéaire	quelconque	quelconque	décidable
quelconque	linéaire	quelconque	décidable
quelconque	quelconque	linéaire	indécidable

Chapitre 11

Le Problème du Vide

Le second problème que nous avions dégagé était celui de l'existence d'au moins une solution, que nous appellerons également problème du vide par analogie au problème du mot vide. Bien qu'il ait été montré décidable dans le cas d'un fait et d'un but clos [46] ou linéaires [16], nous allons montrer dans ce chapitre qu'il est lui aussi indécidable dans le cas général[1]. Une autre preuve, que nous examinerons également, a été établie simultanément et indépendamment par J. Würtz et P. Hanschke de l'Université de Saarbrücken. Elle est basée sur le problème de Post. Notre preuve est sans doute plus difficile d'abord que la leur, mais elle a pour avantage, en utilisant les fonctions de Conway, de rentrer dans un cadre homogène pour les preuves de nos différents résultats. Elle permettra de plus d'obtenir facilement une preuve pour l'un des cas particuliers, décidables ou indécidables, que nous présenterons ensuite.

11.1 Remarque Préliminaire

Effectuons dès maintenant la moitié du travail ! En effet, en ce qui concerne l'*existence* de solutions, les programmes

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases} \qquad \text{et} \qquad \begin{cases} p(but) \leftarrow \ . \\ p(droit) \leftarrow p(gauche) \ . \\ \leftarrow p(fait) \ . \end{cases}$$

sont équivalents.

Dans le premier cas, il faut résoudre, pour déterminer l'existence de solutions après n inférences, le système d'équations :

$$\{but = gauche_1, gauche_{i+1} = droit_i (1 \leq i \leq n-1), droit_n = fait\}$$

[1] Ce résultat a été initialement publié dans [19].

dans le second :

$$\{but = gauche_n, gauche_i = droit_{i+1}(1 \leq i \leq n-1), droit_1 = fait\}$$

A un renommage des variables près (pour tout $1 \leq j \leq n$, X_j renommée en X_{n-j+1}), les deux systèmes sont identiques. L'existence de solutions pour l'un entraine donc l'existence de solutions pour l'autre. Donc, lorsqu'un résultat aura été établi pour certaines propriétés d'un quadruplet ($fait$,$gauche$,$droit$,but), on pourra en déduire un similaire pour les mêmes propriétés du quadruplet ($but, droit, gauche, fait$).

11.2 La Preuve via Conway

C'est sans doute la preuve la plus complexe que nous rencontrerons ensemble. Elle demande une bonne compréhension du fonctionnement du codage des fonctions de Conway par les clauses binaires. Le principe est le suivant : imaginons que nous ayons une fonction (ou la relation de Conway associée) de Conway linéaire (se rapporter dans le chapitre 2 au passage sur les machines de Minsky linéaires), alors dans le programme associé, la propagation de la marque ♯ dans la liste sera linéaire aussi. Il est alors possible d'écrire un programme pour lequel une solution à la $(2^n)^{\text{ème}}$ itération équivaut à dire que n n'appartient pas à l'ensemble récursif linéaire codé par la relation de Conway. On peut alors, à partir du résultat du corollaire 2, déduire l'indécidabilité de l'existence de solutions.

Enfin, examinons plutôt la preuve exacte.

Théorème 17 *Pour un programme de la forme*

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où $fait$ et $droit$ sont des termes linéaires, l'existence d'au moins une solution (le problème du vide) est indécidable.

Lemme 18 *Pour tout ensemble récursivement énumérable linéaire Σ_r (contenant $\{0\}$), il existe une clause de Horn binaire linéaire droite et un but tels que : un entier naturel n appartient à Σ_r si et seulement si après au plus 2^n étapes de la résolution SLD, le premier argument du but initial est une liste dont le $(2^n)^{ème}$ élément est marqué par un symbole ♯.*

Preuve. Soit Σ_r un ensemble récursivement énumérable linéaire, par définition il existe une fonction de Conway linéaire nulle dont le domaine est Σ_r. Il est facile de vérifier que la relation de Conway associée est calculée au pire α–linéairement par la clause ($\mathcal{C}$) et le but ($\mathcal{B}$) obtenus par la codification présentée à la proposition

10. En d'autres termes, la marque ♯ est propagée de manière linéaire dans la liste placée comme premier argument du but. Il est alors facile de définir une clause 1–linéaire, $\mathcal{C}'$, et un but, $\mathcal{B}'$, à partir de $\mathcal{C}$ et de $\mathcal{B}$, tels que chaque étape dans la résolution de $\mathcal{C}'$ corresponde à α étapes de la résolution de $\mathcal{C}$. J'appellerai *paire caractéristique* de Σ_r, le couple ($\mathcal{C}'$,$\mathcal{B}'$) et *liste caractéristique* de Σ_r la liste qui contient les ♯ caractérisant Σ_r. □

Preuve du Théorème. Nous allons reprendre le programme présenté à l'exemple 15 page 84 et qui marque par ♯ les puissances de 2 et par ♭ les autres éléments de la liste :

$$\Pi_1 \qquad \begin{cases} p([X|L],[Y,X|LL],[\flat,Z|LLL]) \leftarrow p(L,LL,LLL) \ . \\ \leftarrow p([\sharp,\sharp|L],[\sharp,\sharp|L],L) \ . \end{cases}$$

Ce programme est donc tel que après n étapes de résolution SLD, le premier argument $\mathcal{L}$ est

$$\mathcal{L} = [X_1, X_2, \cdots, X_n, \cdots]$$

où $\forall k < n$, $X_k = \sharp$ si k est une puissance de 2 et $X_k = \flat$ sinon.

Je vais maintenant construire une classe de programmes pour lesquels l'existence de solutions est indécidable. Soit Σ_r un ensemble récursivement énumérable linéaire et sa paire caractéristique définie grâce au lemme précédent que je noterai :

$$\Pi_2 \qquad \begin{cases} p(t_1,t_2,\cdots,t_k) \leftarrow p(tt_1,tt_2,\cdots,tt_k) \ . \\ \leftarrow p(g_1,g_2,\cdots,g_k) \ . \end{cases}$$

Maintenant voici notre classe particulière de programmes construite à partir de Π_1 et Π_2 :

$$\prod : \begin{cases} p(Y_1,Y_2,\cdots,Y_k,\flat,Z,[\sharp|L],LL,LLL) \leftarrow \ . \\ p(t_1,t_2,\cdots,t_k,W,[U|V],[X|L],[Y,X|LL],[\flat,Z|LLL]) \\ \qquad\qquad \leftarrow p(tt_1,tt_2,\cdots,tt_k,U,V,L,LL,LLL) \ . \\ \leftarrow p(g_1,g_2,\cdots,g_k,X,g_1,[\sharp,\sharp|L],[\sharp,\sharp|L],L) \ . \end{cases}$$

Les k premiers arguments codent Σ_r. Le $(k+1)^{\text{ème}}$ permet d'extraire à la $n^{\text{ème}}$ étape de résolution, le $n^{\text{ème}}$ argument de la liste caractéristique de Σ_r. Le $(k+2)^{\text{ème}}$ est la liste caractéristique de Σ_r (du fait de l'unification avec g_1 dans le but) privée de ses n premiers arguments à la $n^{\text{ème}}$ itération (on perd un élément à chaque application de la règle). Et les trois derniers arguments codent la liste caractéristique des puissances de 2. En conséquence, de par la forme du fait, il y aura une solution à la $n^{\text{ème}}$ étape de résolution si et seulement si :

- n est une puissance de 2 (les trois derniers arguments et en particulier le terme [♯ | L] du fait)

- le $n^{\text{ème}}$ élément de la liste caractéristique de Σ_r n'est pas marqué par $\sharp$ puisqu'il doit être unifiable avec $\flat$ (le $k{+}1^{\text{ème}}$ élément du fait et la variable U de la règle).

Donc, puisque l'on sait que le marquage par $\sharp$ est 1–linéaire, il y aura une solution à la $(2^n)^{\text{ème}}$ étape si et seulement si le $(2^n)^{\text{ème}}$ élément de la liste caractéristique n'est pas marqué par $\sharp$, c'est–à–dire si n n'appartient pas à Σ_r. En conséquence, Π n'aura pas de solution si et seulement si Σ_r est égal à $\mathbb{N}$. Nous avons montré au Corollaire 2 que cette propriété était indécidable, on en déduit le résultat. □

Le principe de cette preuve est repris plus informellement dans le cadre qui suit.

$\mathcal{L}_1 = [\sharp, \sharp, \flat, \sharp, \flat, \cdots, \sharp, \cdots, \flat, \cdots]$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \uparrow \qquad \uparrow$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad n = 2^p \;\; n \neq 2^p$

le $(n = 2^p)^{\text{ème}}$ élément est instancié à $\sharp$ en moins de 2^p itérations.

$\mathcal{L}_2 = [\sharp, \_, \cdots, \sharp, \cdots]$
$\qquad\qquad\qquad\quad \uparrow$
$\qquad\qquad\qquad\quad n = 2^p \text{ et } p \in \Sigma_r$

le $(n = 2^p)^{\text{ème}}$ élément est instancié à $\sharp$ en moins de 2^p itérations ssi $p \in \Sigma_r$.

Dans Π, le fait impose à la $n^{\text{ème}}$ itération d'unifier le $n^{\text{ème}}$ élément de $\mathcal{L}_1$ avec $\sharp$ et le $n^{\text{ème}}$ de $\mathcal{L}_2$ avec $\flat$. A cette étape, trois situations d'unification peuvent se produire :

$$
\begin{array}{lrcl}
 & & \boxed{n} & \\
 & & \sharp & \\
n \neq 2^p \Rightarrow & \mathcal{L}_1 = [\sharp, \sharp, \flat, \sharp, \cdots, & \flat \quad , \cdots] & \longrightarrow \text{ échec} \\
 & \mathcal{L}_2 = [\sharp, X_2, \cdots, & ? \quad , \cdots] & \\
 & & \flat & \\
 & & \sharp & \\
n = 2^p \text{ et } p \in \Sigma_r \Rightarrow & \mathcal{L}_1 = [\sharp, \sharp, \flat, \sharp, \cdots, & \sharp \quad , \cdots] & \longrightarrow \text{ échec} \\
 & \mathcal{L}_2 = [\sharp, X_2, \cdots, & \sharp \quad , \cdots] & \\
 & & \flat & \\
 & & \sharp & \\
n = 2^p \text{ et } p \notin \Sigma_r \Rightarrow & \mathcal{L}_1 = [\sharp, \sharp, \flat, \sharp, \cdots, & \sharp \quad , \cdots] & \longrightarrow \text{ succès} \\
 & \mathcal{L}_2 = [\sharp, X_2, \cdots, & X_{2^p} \quad , \cdots] & \\
 & & \flat &
\end{array}
$$

On aura donc une solution à cette étape si et seulement si :

$$n = 2^p \text{ et } p \notin \Sigma_r$$

Par symétrie du problème, conformément à ce que nous avons expliqué à la section 11.1, on en déduit immédiatement un théorème équivalent avec cette fois les termes *but* et *gauche* linéaires et les termes *droit* et $fait$ quelconques.

11.3 Un Corollaire Important

Une conséquence importante de ce théorème est qu'il permet de résoudre un problème ouvert en *logique du premier ordre* concernant la satisfiabilité de formules. En effet, pour les formules du premier ordre à quatre sous–formules telles que

$$\forall X_i, (P(t_1) \wedge Q(t_2) \vee R(t_3) \wedge S(t_4))$$

où P, Q, R et S sont des prédicats positifs ou négatifs et où les X_i sont les variables apparaissant dans t_1, t_2 t_3 et t_4. Une instance particulière de ce problème est :

$$\forall X_i, (P(t_1) \wedge (P(t_2) \vee \neg P(t_3)) \wedge \neg P(t_4))$$

pour laquelle, le problème de la non–satisfiabilité est équivalent à l'existence de solutions pour le programme

$$\begin{cases} p(t_1) \leftarrow \ . \\ p(t_2) \leftarrow p(t_3) \ . \\ \leftarrow p(t_4) \ . \end{cases}$$

où t_1, t_2, t_3 et t_4 sont des termes quelconques. Si cela vous rappelle quelque chose, c'est normal. Nous pouvons donc affirmer que :

Théorème 19 *La satisfiabilité de la classe des formules du premier ordre à quatre sous–formules est indécidable.*

Preuve. En accord avec ce qui précède, ceci est immédiat d'après le théorème 17. □

Ce résultat est à rapprocher d'un problème également ouvert : celui de la satisfiabilité des formules *quantifiées pures* (c'est–à–dire sans symbole de fonction et avec une infinité de constantes possible) :

$$\forall t \exists u \forall v \cdots \forall w (\mathcal{A}_1 \wedge \mathcal{A}_2 \vee \cdots \wedge \mathcal{A}_n)$$

où les $\mathcal{A}_i$ sont des formules atomiques positives ou négatives. La satisfiabilité du cas à cinq sous–formules a été montré indécidable dans [22]. W. Goldfarb et H.R. Lewis ont établi que ce problème était équivalent à celui de l'arrêt des machines à deux compteurs (qui est indécidable). Le cas des formules quantifiées à trois ou quatre sous–formules est donné ouvert par W. Goldfarb et H.R. Lewis dès 1973.

Par skolemisation de ces formules, il est possible de se ramener à des formules du premier ordre sans quantificateur existentiel. Mais la forme des termes fonctionnels que l'on crée est alors spéciale du fait des contraintes liées à cette opération. Il n'est donc pas possible d'étendre immédiatement le théorème précédent aux formules quantifiées pures à quatre sous–formules. On peut cependant espérer avoir trouvé ici la piste d'une preuve possible.

11.4 La Preuve via Post

Je vais maintenant présenter la preuve dûe à P. Hanschke et J. Würtz [25]. Elle n'est pas basée sur les fonctions de Conway mais sur le problème de Post [41], plus connu. Je rappellerai brièvement en quoi consiste celui–ci avant de présenter la preuve proprement dite. Pour peu que l'on ait compris le fonctionnement des listes–différence, celle–ci est d'une simplicité déconcertante. J'avais examiné la possibilité d'un codage de ce problème hélas sans succès et sans penser à utiliser ces listes...

11.4.1 Le Problème de Post

Considérons donné un alphabet Σ. Un *système de correspondance de Post* sur Σ est un ensemble non vide $\mathcal{S} = \{(g_i, d_i) \mid i \in [1, \cdots, m]\}$ où les g_i, d_i sont des mots de Σ. Une séquence non vide d'indices $1 \leq i_1, \cdots, i_n \leq m$ est appelée une solution de $\mathcal{S}$ si et seulement si

$$g_{i_1} \cdots g_{i_n} = d_{i_1} \cdots d_{i_n}$$

Il est bien établi que le problème de correspondance de Post, c'est–à–dire "Existe–t–il une solution pour un système donné ?", est en général indécidable si l'alphabet contient au moins deux symboles.

11.4.2 Codage du Problème de Post

Les éléments de l'alphabet Σ seront représentés par des symboles de fonctions unaires et un mot, $w = a_1 a_2 \cdots a_n$ de Σ sera donc noté $a_n(\cdots(a_2(a_1(\epsilon)))\cdots)$ où ϵ est la constante mot vide. Les listes–différence seront utilisées pour l'ajout d'élément à une liste.

On considère donné un problème de correspondance de Post défini avec les notations de la section précédente. Voici le programme avec un fait, une règle binaire et un but qui permet de coder ce problème :

$$\begin{cases} p([E \mid H_1] - H_2, [E \mid H_3] - H_4) \leftarrow \ . \\ p([C \mid G] - [g_1(C), \cdots, g_m(C) \mid X], \\ \qquad [CC \mid D] - [d_1(CC), \cdots, d_m(CC) \mid Y]) \\ \qquad\qquad \leftarrow p(G - X, D - Y) \ . \\ \leftarrow p([g_1(\epsilon), \cdots, g_m(\epsilon) \mid X] - X, [d_1(\epsilon), \cdots, d_m(\epsilon) \mid Y] - Y) \ . \end{cases}$$

Le but définit l'ensemble $\mathcal{S}$ de paires (g_i, d_i) et le fait vérifie, en unifiant les têtes des deux listes–différence, si une solution du système a été trouvé à la $n^{\text{ème}}$ itération en unifiant les suites correspondantes : $(g_{i_k})_{1 \leq k \leq n}$ et $(d_{i_k})_{1 \leq k \leq n}$.

Le principe de ce programme est le suivant. A chaque itération, une séquence $i_1 \cdots i_j$ est sélectionnée (C et CC) et remplacée par *toutes* les séquences de longueur (j+1) et ayant $i_1 \cdots i_j$ pour suffixe (respectivement $[g_1(C), \cdots, g_m(C)]$ et $[d_1(CC), \cdots, d_m(CC)]$). Le principe de la résolution SLD assure que toutes les séquences possibles seront tour à tour sélectionnées.

Ce programme n'aura donc de solution que si le système de Post correspondant en a une. Le problème de correspondance de Post étant indécidable, l'existence de solution pour ce programme l'est aussi. □

Hum ! Vraiment très simple ...

11.5 Des Cas Particuliers

Nous établirons tout d'abord la décidabilité du problème du vide pour les cas où trois des éléments caractéristiques de nos programmes sont linéaires et dans celui où l'un seulement est clos. Puis nous montrerons que le résultat reste cependant indécidable lorsque seule la clause est linéaire (gauche et droite).

11.5.1 Décidabilité

Comme je viens de le dire, nous allons voir deux cas particuliers décidables. La preuve du premier est immédiate à partir du théorème 16. Celle du second l'est beaucoup moins. Elle fait appel à la fois aux GOPs et aux systèmes pondérés d'équations. Examinons–les.

Théorème 20 *Pour la classe de programme*

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où trois des quatre termes gauche, droit, fait et but sont linéaires, le problème du vide est décidable.

Preuve. Dès que le but et le fait sont linéaires, le problème est connu pour décidable depuis [16]. Supposons maintenant que la clause est linéaire ainsi que le fait. L'autre cas est symétrique.

Comme dans la preuve du théorème 16, puisque la règle est linéaire, le terme G^∞ est linéaire et donc le langage décrit par le graphe pondéré de la clause est rationnel. Puisque le fait est linéaire, en unifiant les graphes pondérés de la règle et du fait, nous conservons la rationnalité. Nous pouvons alors tenir le même raisonnement que dans cette même preuve pour l'unification avec le but non–linéaire. Le problème du vide est donc décidable dans ce cas. □

Théorème 21 *Le problème de l'existence de solutions pour la classe de programmes*

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

est décidable dès que l'un des termes gauche, droit, fait ou but est clos.

Preuve. Le problème est trivial lorsque *gauche* ou *droit* est clos. Supposons que $fait$ est clos, l'autre cas étant symétrique.

Si *but* est ou clos ou linéaire, le problème est déjà résolu, dans [46] et [16] respectivement, et connu pour décidable. Supposons donc que *but* est non–linéaire.

J'utiliserai ici à la fois les graphes pondérés et les systèmes pondérés d'équations.

Notons que dès que l'un des sous–termes de $droit^n$ croit sans limite, on peut assurer qu'après un nombre fini d'inférences il n'y aura plus de solutions possibles. En effet, le fait étant clos, sa taille est limitée. Par conséquence, nous pouvons ne pas considérer le cas où il existe une boucle négative dans le graphe pondéré puisqu'il est évidemment décidable.

Considérons donc qu'il n'y a que des boucles positives dans le graphe pondéré. Nous devons vérifier si la non–linéarité du but provoque la croissance d'un terme de $droit^n$. Pour cela nous allons utiliser l'algorithme de simplification des systèmes pondérés d'équations. Considérons le système pondéré simplifié de la règle et ajoutons lui le système d'égalités (les liens exceptions) dû à l'équation $but = gauche^1$. Nous appliquons la règle de fusion "haut–vers–bas" (Cf. page 74).

L'effet de cette règle est : si l'on a deux équations

$$X = f(U^1, \cdots, U^n) \text{ et } Y = f(V^1, \cdots, V^n)$$

et une relation $X_i = Y_{i+k}$, on peut ajouter les relations héritées sur les variables U^i et V^i à partir de X et Y.

Premier cas, la propagation "haut–vers–bas" est finie, c'est–à–dire que la non–linéarité n'interfère pas avec les boucles positives, le problème devient alors similaire au cas où le but est linéaire et est donc décidable.

Second cas, la non–linéarité interfère avec certaines boucles positives. Deux cas sont à nouveau possibles :

1. lors de la propagation, on génère des égalités de la forme :

$$X_k = X_{k'} \Rightarrow X_{k+a} = X_{k'+a} \Rightarrow X_{k+2a} = X_{k'+2a} \cdots$$

   Il est alors possible de déduire de cette suite infinie de relations (qui sont en fait des liens exception) une relation pondérée de base telle $X \xrightarrow{k-k'} X$ (a pouvant être considéré égal à 1). Puisqu'il ne peut y avoir de suite infinie croissante de relations pondérées (Cf. Théorème 6), le calcul est fini et convergent. Nous pouvons alors décider si il existe ou non au moins une solution en unifiant avec le fait clos.

2. la propagation a pour conséquence la production d'égalités de la forme :

$$X_k = Y_{k'} \Rightarrow X_{k+a} = Y_{k'+a'} \Rightarrow X_{k+2a} = Y_{k'+2a'} \cdots$$

   ce qui correspond à une relation pondérée linéaire de la forme

$$X_{ai+k} \xrightarrow{\quad} {}_{a'i+k'}X$$

   On peut alors choisir X et Y tels qu'il existe une relation "récursive" comme

$$X_i = f(\cdots g(\cdots X_{i+k} \cdots) \cdots) \text{ (et idem pour } Y)$$

   En effet, comme nous l'avons vu, de telles équations correspondent aux boucles positives du graphe pondéré.

   Il est maintenant facile de vérifier que, si $k < k'$ (resp. $k' < k$), le terme X_n (resp. Y_n) de $droit^n$ croitra infiniment en fonction de n. En conséquence après un nombre calculable d'inférences de la règle, aucune nouvelle unification avec le fait clos ne sera possible.

Dans tous les cas, on vérifie donc bien que l'on peut décider de l'existence ou non d'une solution. □

11.5.2 Indécidabilité

Nous allons nous intéresser ici au cas où les termes droit *et* gauche de la règle sont linéaires. En parvenant à transformer les clauses non–linéaires codant les fonctions de Conway en des clauses équivalentes linéaires, nous montrerons que la démonstration du théorème 17 peut s'appliquer dans ce cas et prouverai ainsi que :

Théorème 22 *Pour la classe de programmes*

$$\begin{cases} p(fait) \leftarrow \ . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

où gauche et droit sont des termes linéaires et but et fait sont quelconques, le problème de l'existence de solutions est indécidable.

Preuve. Considérons le programme suivant :

$$\begin{cases} p([X|LX], [\overbrace{U, \_, \cdots, \_}^{a}|LU], [\overbrace{V, \_, \cdots, \_}^{c}|LV]) \leftarrow p(LX, LU, LV) \ . \\ \leftarrow p(L, [\underbrace{\_, \cdots, \_}_{b}|L], [\underbrace{\_, \cdots, \_}_{d}|L]) \ . \end{cases}$$

Depuis le chapitre 9, nous savons déchiffrer ces programmes. Nous en déduisons qu'il produit les égalités :

$$\begin{cases} X_{ai+b} = U_i \\ X_{ci+d} = V_i \end{cases}$$

Si nous le modifions légèrement et lui ajoutons un fait comme dans :

$$\begin{cases} p(\_, \_, \_, L, L) \leftarrow \ . \\ p([X|LX], [\overbrace{U, \_, \cdots, \_}^{a}|LU], [\overbrace{V, \_, \cdots, \_}^{c}|LV], LLU, LLV) \\ \qquad\qquad \leftarrow p(LX, LU, LV, [U|LLU], [V|LLV]) \ . \\ \leftarrow p(L, [\underbrace{\_, \cdots, \_}_{b}|L], [\underbrace{\_, \cdots, \_}_{d}|L], [\,], [\,]) \ . \end{cases}$$

de par la non–linéarité du fait, nous avons ajouté l'égalité $U_i = V_i$, donc nous déduisons que pour la variable X, on a :

$$X_{ai+b} = X_{ci+d}$$

et aucune autre relation sur X n'est définie.

Il est facile de créer n autres égalités sur X. Par exemple si n vaut 2 :

$$\left\{\begin{array}{l} p\ (\_, X, \_, \_, L1, L1, \_, \_, L2, L2) \leftarrow \quad . \\ p\ ([X|LX], Y, [\overbrace{U1, \_, \cdots, \_}^{a_1}|LU1], [\overbrace{V1, \_, \cdots, \_}^{c_1}|LV1], LLU1, LLV1, \\ \qquad\qquad [\underbrace{U2, \_, \cdots, \_}_{a_2}|LU2], [\underbrace{V2, \_, \cdots, \_}_{c_2}|LV2], LLU2, LLV2) \\ \qquad\qquad\qquad \leftarrow p(LX, X, LU1, LV1, [U1|LLU1], [V1|LLV1], \\ \qquad\qquad\qquad\qquad\qquad LU2, LV2, [U2|LLU2], [V2|LLV2])\ . \\ \leftarrow p([\sharp|L1], \_, [\overbrace{\_, \cdots, \_}^{b_1}, \sharp|L1], [\overbrace{\_, \cdots, \_}^{d_1}, \sharp|L1], [\ ], [\ ], \\ \qquad\qquad [\underbrace{\_, \cdots, \_}_{b_2}, \sharp|L2], [\underbrace{\_, \cdots, \_}_{d_2}, \sharp|L2], [\ ], [\ ])\ . \end{array}\right.$$

De la même manière que précédemment, ce programme produit les égalités :

$$X_{a_1 i+b_1} = X_{c_1 i+d_1} \text{ et } X_{a_2 i+b_2} = X_{c_2 i+d_2}$$

Remarquons que dans le fait, après p itérations, X est égal à X_p. L'extension pour n'importe quel $n \leq 2$ est maintenant triviale. Il est donc possible de coder n'importe quelle fonction de Conway, c'est–à–dire n'importe quel ensemble récursif Σ_r (contenant $\{0\}$). En effet, si $p = 2^k$ alors, dans le fait, $X = X_p = X_{2^k} = \sharp$ si et seulement si $k \in \Sigma_r$. Appelons Π_{Σ_r} un tel programme associé à un ensemble Σ_r.

Considérons maintenant cet autre programme :

$$\Pi' : \left\{\begin{array}{l} p(\_, \_, L, L) \leftarrow \quad . \\ p([X|LX], [\_, Y, \flat|LY], LLX, LLY) \\ \qquad\qquad\qquad \leftarrow p(LX, [\flat|LY], [X|LLX], [Y|LLY]). \\ \leftarrow p([\sharp, \sharp|L], [\sharp, \sharp|L], [], [])\ . \end{array}\right.$$

Il est le pendant à clause linéaire du programme que nous avons déjà rencontré page 84 et qui construit, dans la liste

$$L = [X \mid LX] = [X_1, X_2, \cdots, X_p, \cdots],$$

les relations $X_p = \sharp$ si p est une puissance de 2 et $X_p = \flat$ sinon.

Maintenant exactement comme dans la preuve du cas général (Cf. Théorème 17), en mélangeant les clauses et les buts du programme ci–dessus et d'un programme Π_{Σ_r}, et en choisissant un fait tel que :

$$p(\underbrace{\_, \flat, \_, \_, L1, L1, \cdots, \_, \_, LN, LN}_{\text{partie } \Pi_{\Sigma_r}}, \underbrace{\_, \_, [\sharp \mid L], [\sharp \mid L]}_{\text{partie } \Pi'}) \leftarrow .$$

on obtient un programme à clause binaire linéaire qui n'aura de solutions que si et seulement si Σ_r n'est pas égal à $\mathbb{N}$. Cette propriété est indécidable. □

11.6 Conclusion

Nous avons donc établi l'indécidabilité de l'existence de solutions dans le cas général. La preuve fournie par J. Würtz et P. Hanschke est plus facilement abordable mais la notre a ensuite permis d'établir l'indécidabilité dans le cas de la règle linéaire. Nous avons de plus montré la décidabilité du problème du vide dès que trois des termes caractéristiques du programme sont linéaires ou dès que l'un seulement est clos. Tous ces résultats sont récapitulés dans le tableau suivant :

but	*gauche*	*droit*	*fait*	Solution ?
clos	quelconque	quelconque	clos	décidable
linéaire	quelconque	quelconque	linéaire	décidable
clos	quelconque	quelconque	quelconque	décidable
quelconque	quelconque	quelconque	clos	
linéaire	linéaire	linéaire	quelconque	décidable
quelconque	linéaire	linéaire	linéaire	
quelconque	quelconque	linéaire	linéaire	indécidable
linéaire	linéaire	quelconque	quelconque	
quelconque	linéaire	linéaire	quelconque	indécidable

Chapitre 12

Puissance de Calcul

Dans ce chapitre nous allons présenter ce qui est sans doute le résultat le plus important de tous ceux qui ont été établis. D'une part parce qu'il les implique tous et d'autre part parce qu'il montre à quel point la puissance d'expression des clauses de Horn est élevée. Nous allons montrer que la classe des programmes, que nous avons attentivement examinée depuis le début, a en fait la même puissance de calcul que les machines de Turing (et tous les équivalents)[1]. Cela signifie donc que tout ce qui est programmable l'est par un programme appartenant à cette classe. On pourrait donc, bien que cela ne soit pas, mais alors pas du tout, raisonnable, imaginer un langage de programmation n'utilisant que ce schéma structurellement simple, bien que sémantiquement difficilement pénétrable.

Pour aboutir à ce résultat j'utiliserai les principes des deux preuves de l'existence de solutions que nous avons présentées précédemment. Dans les grandes lignes, la preuve suivante va consister en l'établissement d'un méta–interpréteur Prolog sous la forme d'un programme à une seule clause binaire. Cela se fera progressivement : nous construirons d'abord un générateur de mots, puis un méta–interpréteur qui ne s'arrête jamais et enfin nous ajouterons un contrôle sur la terminaison. Le seul point réellement délicat est l'"astuce" utilisant les fonctions de Conway et permettant d'assurer l'arrêt de notre méta–interpréteur.

12.1 Méta–Programmes/Méta–Interpréteurs

Je commencerai par une petite mise au point sur la différence que j'établis entre les dénominations "méta–programme" et "méta–interpréteur". Cette distinction n'est sans doute pas universelle, mais elle permettra de préciser la terminologie.

Nous appellerons donc *méta–programme,* une structure de programmes telle que l'on puisse construire à partir de tout programme Π, un programme équivalent

[1]Ce résultat a été initialement publié dans [20], il a été réalisé en collaboration avec Jörg Würtz.

en terme de solutions et d'arrêt à Π (quelque soit le but) et qui satisfasse cette structure.

Nous appellerons *méta–interpréteur*, un programme qui ayant pour but un autre programme Π (ou une certaine codification du programme Π) et un but quelconque $\mathcal{B}$, produit les mêmes réponses que Π avec le but $\mathcal{B}$ et s'arrête si et seulement si Π s'arrête avec le but $\mathcal{B}$. Un méta–interpréteur est donc un programme universel au sens utilisé dans le Chapitre 1.

Dans le premier cas, le programme testé est différent pour chaque Π, dans le second, il reste toujours le même, seul le but change.

12.2 Un Générateur de Mots

Dans cette section nous montrons comment construire un programme à une clause binaire permettant de générer tous les mots de l'alphabet $\{a, b\}$. Ces mots seront représentés sous forme de liste (par exemple $[a, b, b, b, a]$). Le principe de la codification est similaire à celui utilisé pour le codage du problème de Post :

$$\begin{cases} gen([Mot|R] - RR, Mot) \leftarrow \ . \\ gen([Mot|R] - [[a|Mot], [b|Mot]|RR], M) \leftarrow gen(R - RR, M) \ . \\ \leftarrow gen([[]|R] - R, Mot) \ . \end{cases}$$

Voici comment se comporte la liste–différence qui constitue le premier argument :

$$[Mot, \overbrace{\underbrace{\_, \cdots, \_, [a|Mot], [b|Mot]}_{R-RR}, \underbrace{\_, \_, \cdots}_{RR}}^{R}]$$

Observons maintenant ensemble les premiers pas de ce programme. En unifiant le but et le fait, nous obtenons la solution $Mot = []$. En appliquant la clause binaire une fois, on obtient le nouveau but :

$$gen([[a], [b] \mid RR_1] - RR_1, M_1)$$

ce qui produit, par unification avec le fait, la solution $Mot = [a]$. Si l'on applique la clause plutôt que le fait, il en résulte le nouveau but courant suivant

$$gen([[b], [a, a], [b, a] \mid RR_2] - RR_2, M_2)$$

qui permet de déduire le nouveau but $Mot = [b]$ etc. Notez que a et b servent de préfixes à de nouveaux mots dont le suffixe et le premier élément de la liste

générée jusque là. Ces deux mots sont alors ajoutés au reste de cette liste. La liste (–différence) peut être vue comme une file LIFO (Last In First Out).

On retrouve exactement le principe du codage du problème de Post. Il est clair que ce générateur peut être étendu de manière évidente à n'importe quel alphabet fini.

12.3 Un Premier Méta–Interpréteur

Commençons par l'étude d'un méta–programme. Son schéma a été fourni par A. Parrain, P. Devienne et P. Lebègue dans [40, 39]. Il est constitué d'un fait, de deux règles récursives binaires et du but. Une forme équivalente, constituée de deux faits, d'une règle ternaire et du but, peut être établie.

Soit Π l'ensemble de clauses de Horn

$$\Pi = \{clause_1, clause_2, \cdots, clause_n\},$$

et "$\leftarrow b_1, \cdots, b_n$" un but, le méta–programme suivant génère les mêmes substitutions réponses dans le même ordre qu'un interpréteur SLD standard en largeur d'abord :

$$\begin{cases} solve([],[]) \leftarrow \ . \\ solve([But|R_1],[[But|R_2]-R_1|L]) \leftarrow solve(R_2,\mathcal{P}) \ . \\ solve(Buts,[Clause|Reste]) \leftarrow solve(Buts,Reste) \ . \\ \leftarrow solve(\mathcal{B},\mathcal{P}) \ . \end{cases}$$

$\mathcal{B}$ représente la liste $[b_1, b_2, \cdots, b_n]$ et $\mathcal{P}$ est la liste qui code des clauses du programme Π, $\mathcal{P}$=$[clause_1, \cdots, clause_n]$. Une clause (de Horn) $clause_i$, $a \leftarrow b_1, \cdots, b_m$, de Π est codée par la liste–différence $[a, b_1, \cdots, b_m \mid R] - R$. Dans ce méta–programme, la première clause binaire permet de sélectionner la première clause dans la liste courante et de vérifier si sa tête s'unifie avec le but courant. La seconde l'élimine de la liste courante. Un rapide examen permet de vérifier que de par ces choix, la résolution SLD est respectée.

Ce méta–programme est étudié en détail dans [39]. Sa complexité y est montré linéairement dépendante de celle du programme original (le programme Π) :

> "*... Appelons complexité d'un nœud-solution sol_n le nombre de nœuds de l'arbre SLD qu'on parcourt (par la stratégie en profondeur d'abord et de gauche à droite) avant d'atteindre sol_n. On notera φ_i la fonction de complexité du programme i, et N_i le nombre de règles qui le définissent. i prendra les valeurs o pour le programme objet, et m pour le méta–programme.*
>
> *Au mieux, le but s'unifie avec la premire rgle du programme original, et cette rgle est un fait. Au pire, il s'unifie avec la dernière règle du programme.*

Dans le cas où le fait du méta–programme est `solve([],[])`*, on obtient donc :*

$$\varphi_o(sol_n) + N_o \leq \varphi_m(sol_n) \leq (\varphi_o(sol_n) \times N_o) + N_o$$

".

Exemple 16 Voici le méta–programme associé au programme *"append" :*

1. `solve([],[]).`

2.
```
solve([But| L1],[[But| L3]-L1|Liste_de_règles]) :-
            solve(L3,[[append([],Lapp1,Lapp1)| L]-L,
                [append([X| Lapp2],Lapp3,[X| Lapp4]),
                append(Lapp2,Lapp3,Lapp4)|LL]-LL]).
```

3.
```
solve(Liste_de_buts,[Règle|Liste_de_Règles]) :-
                solve(Liste_de_buts,Liste_de_Règles).
```

Pour le but initial `append([1],[2,3],L)`, le but correspondant dans le méta–programme sera :

```
:- solve([append([1],[2,3],L1)],
       [[append([],Lapp1,Lapp1)|L]-L,
       [append([X| Lapp2],Lapp3,[X|Lapp4]),
       append(Lapp2,Lapp3,Lapp4)|LL]-LL]).
```
□

Pour coder un programme arbitraire Π, la première clause binaire et le but doivent être adaptés de manière adéquate (de ce fait, il s'agit bien d'un méta–programme et non d'un méta–interpréteur).

Maintenant, supposons que comme programme Π nous choisissions un méta–interpréteur Prolog. Alors cette codification me permet de définir un méta–interpréteur *MI* explicitement constructible ayant ce schéma de programme. Il s'agit bien maintenant d'un méta–interpréteur, puisque pour méta–interpréter par Π un programme quelconque, il ne nous reste qu'à coder correctement le but à transmettre à Π. On vérifie qu'il en est de même pour la version *MI*.

En résumé, dans cette section nous avons construit un méta–interpréteur, pour les langages à clauses de Horn, constitué d'un fait, de deux règles binaires et d'un but.

12.4 Un Générateur d'Arbres SLD

Considérons un programme Π constitué uniquement de deux clauses binaires

$$gauche_1 \leftarrow droite_1 \text{ et } gauche_2 \leftarrow droite_2$$

d'un fait $fait$ et d'un but but. En adaptant un peu le générateur de mots et en posant

$$\mathcal{A} = droite_1, gauche_1 \text{ et } \mathcal{B} = droite_2, gauche_2$$

on obtient le programme

$$\begin{cases} gen([M|R] - RR, M) \leftarrow \ . \\ gen([M|R] - [[\mathcal{A}|M], [\mathcal{B}|M]|RR], [H \mid RRR]) \\ \qquad\qquad\qquad\qquad \leftarrow gen(R - RR, [H, X, X \mid RRR]) \ . \\ \leftarrow gen([[but \mid L]|R] - R, [fait \mid LL]) \ . \end{cases}$$

Comment fonctionne ce programme ?

Après avoir appliqué n fois la clause binaire, nous obtenons comme second argument du nouveau but, la liste :

$$[fait, X_1, X_1, X_2, X_2, \cdots, X_n, X_n \mid Q]$$

De plus, conformément à ce que nous avions vu pour le générateur de mots, la tête de la liste placée en premier argument est

$$[droite_{i_m}, gauche_{i_m}, \cdots, droite_{i_1}, gauche_{i_1}, but \mid Q']$$

Les i_j valant indifféremment 1 ou 2 et les variables étant renommées à chaque étape de la résolution. Donc en effectuant l'unification du but courant avec le fait de ce programme, nous obtenons le problème d'unification suivant :

$$\begin{array}{l} [droite_{i_m}, gauche_{i_m}, droite_{i_{m-1}}, \cdots, droite_{i_1}, gauche_{i_1}, \ but \ , \_ \,] \\ \quad \updownarrow \qquad \updownarrow \qquad \updownarrow \qquad \cdots \qquad \updownarrow \qquad \updownarrow \qquad \updownarrow \ \updownarrow \\ [\ fait \ , \ X_1 \ , \ X_1 \ , \cdots , \ X_{m-1} \ , \ X_m \ , \ X_m \ , \cdots , X_n, Y | \_ \,] \end{array}$$

Ce qui revient à résoudre le système :

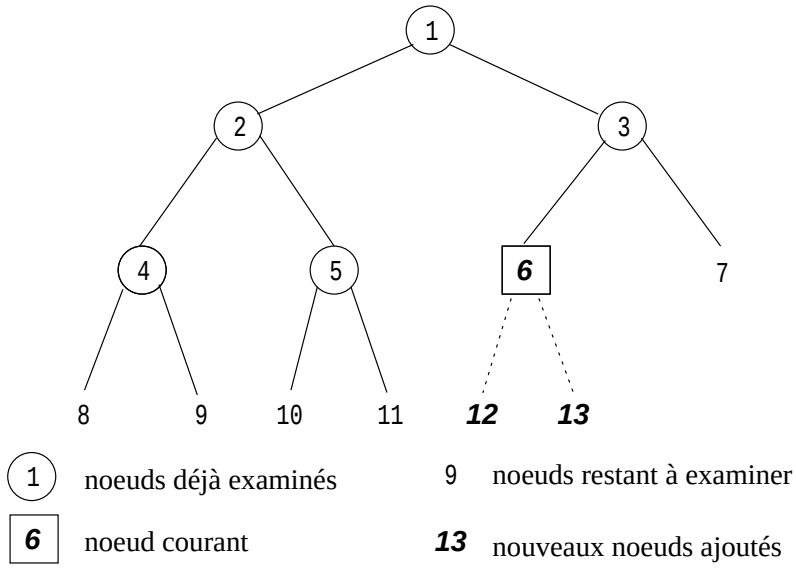
$$\begin{cases} fait &= droite_{i_m} \\ gauche_{i_k} &= droite_{i_k - 1}, 2 \leq k \leq m \\ but &= gauche_{i_1} \end{cases}$$

Il est important de remarquer que, par construction, nous sommes assurés que la liste contenant le fait est de longueur suffisante pour permettre ces équations. Toutes les séquences possibles

$$[droite_{i_m}, gauche_{i_m}, \cdots, droite_{i_1}, gauche_{i_1}, but \mid Q']$$

seront successivement sélectionnées. L'arbre de résolution SLD sera donc entièrement parcouru. En conséquence, ce système sera solvable si et seulement si il existe une réfutation du programme initial Π utilisant l'ordre de résolution imposé par ces équations (c'est–à–dire par les i_k).

En effet ce programme fonctionne comme un générateur d'arbres SLD. Il réalise la résolution avec une stratégie en largeur d'abord du programme constitué des deux règles, du fait et du but. La génération de l'arbre est reprise dans le schéma suivant :



A chaque fois qu'un nœud est sélectionné, les deux nouveaux nœuds correspondant aux inférences avec respectivement les règles $\mathcal{A}$ et $\mathcal{B}$ sont ajoutés à la fin de la liste des "nœuds à examiner" (c'est–à–dire à la fin de la liste $R - RR$), puis le nœud suivant est examiné, etc.

12.5 Un Méta–Interpréteur Binaire

Remarquons maintenant que notre méta–interpréteur *MI* est un programme de la famille Π. Si nous le codons comme ci–dessus, il est alors possible d'associer à tout programme logique un programme équivalent, en terme de solutions, contenant une clause binaire et deux clauses unaires. Malheureusement la terminaison n'est pas conservée par cette construction. En un certain sens nous pouvons dire que nous avons maintenant un (presque) méta–interpréteur qui ne s'arrête pas, nous l'appellerons $\mathcal{M}_{nS}$.

Dans ce qui suivra je vais m'intéresser à ajouter l'arrêt à $\mathcal{M}_{nS}$. Cela nécessite tout d'abord l'établissement d'un résultat technique et un peu délicat.

12.6 Le Préliminaire Technique

Notre objectif est de provoquer l'arrêt du programme précédent. Comme pour la preuve du problème de l'arrêt, nous provoquerons la terminaison de la résolution par l'échec d'une unification. La différence est ici que l'on veut que cet arrêt se fasse après au moins un certain nombre d'étapes puisque nous voulons que le programme ait produit les solutions auparavant. Cela n'est pas très difficile comme il sera montré dans la première partie de la preuve.

Remarque 5 Dans ce qui suit le mot "programme" sera à comprendre dans le sens habituel intuitif du terme. Il est possible, si l'on préfère, de considérer qu'il signifie "une machine (au sens des machines de Turing ou de Minsky) qui calcule une fonction récursive partielle", ou encore "un programme à une entrée et une sortie entières".

Proposition 11 *Pour tout programme* Π *avec l'entrée* I*, il existe une clause binaire* $\mathcal{R}_\Pi$ *et un but, dépendant de l'entrée* I*, tel que* $\mathcal{R}_\Pi$ *s'arrêtera après au moins* n *itérations si* Π *s'arrête après* n *étapes élémentaires avec l'entrée* I*, et ne s'arrêtera pas sinon.*

Preuve. Soit Π un programme à une entrée et une sortie entières, et g la fonction de Conway nulle qui lui est associée (ou plus exactement qui est associée à la machine de Minsky codant Π, ce qui revient au même). g est caractérisée par sa "période" d (en fait la période de $\frac{g(n)}{n}$) et les nombres rationnels $a_0, a_1, \cdots, a_{d-1}$. Soit $\mathcal{R}$ la clause de Horn binaire associée à g comme il est indiqué dans la proposition 10. Par construction, à chaque itération, $\mathcal{R}$ construit d nouvelles égalités de la forme $X_i = X_{g(i)}$. En fait à la i$^{\text{ème}}$ itération, le programme produit les égalités

$$\forall 0 \leq l \leq d-1, X_{d(i-1)+p+1} = X_{a_k \times (d(i-1)+p+1)}$$

Etant donné que g est une fonction de Conway nulle, il existe un seul chemin de 2^n à 2^0 conformément à ce qui a été expliqué dans la Section 3.3. La clause de Horn produit aussi bien des itérés de g positifs que négatifs. Donc, à chaque application, $\mathcal{R}$ crée p_i égalités "positives" et n_i égalités "négatives" de la suite

$$(X_{g^{(i)}(2^n)} = X_{g^{(i+1)}(2^n)})_{i \in \mathbf{N}}$$

avec $p_i + n_i \leq d$. Ceci est repris dans le schéma :

$$2^n \text{ —} p_1 \overset{g^*}{\text{———}} g(2^n) \text{———} \cdots \text{———} g^{(-1)}(2^0) \overset{g^{(-1)*}}{\text{———}} n_1 \text{ —} 2^0$$

En conséquence, si il faut k_n itérations à partir de $g(2^n)$ pour obtenir 1, l'égalité $X_{2^n} = X_1$ sera générée en au minimum $\frac{k_n}{d}$ applications itératives de $\mathcal{R}$. Il est possible de ralentir d fois $\mathcal{R}$, en ajoutant des variables supplémentaires, de

telle sorte que la "vitesse" (que l'on peut considérer comme la complexité) de $\mathcal{R}$ soit au mieux égale à celle de g, nous obtenons ainsi $\mathcal{R}_\Pi$.

Maintenant, de manière identique à la preuve de l'indécidabilité du problème de l'arrêt, pour tout entrée I, nous pouvons choisir un but pour $\mathcal{R}_\Pi$ tel que celle–ci s'arrête si et seulement si la relation $X_{2^I} = X_{2^0}$ vient à se produire. Et nous pouvons assurer que le nombre d'itérations nécessaires avant l'arrêt est plus grand que le nombre d'étapes élémentaires avant arrêt du programme Π avec l'entrée I. □

Maintenant, construisons un programme particulier à partir du méta–interpréteur $\mathcal{M}_{nS}$. Voici le programme Π qui prend pour entrée un programme à clauses de Horn P :

1. lire P
2. exécuter P par une stratégie en largeur d'abord et garder les solutions en mémoire dans $\mathcal{S}_1$,
3. calculer $\mathcal{M}_{nS}(P)$ et garder les solutions dans $\mathcal{S}_2$, arrêter et écrire 0 dès que $\mathcal{S}_2 = \mathcal{S}_1$.

Il est clair que Π s'arrête si et seulement si P s'arrête et son temps d'arrêt (c'est–à–dire le nombre de pas avant l'arrêt) avec l'entrée P est plus grand que celui nécessaire à $\mathcal{M}_{nS}(P)$ pour produire toutes les solutions de P.

Maintenant en accord avec la proposition précédente, il existe une clause $\mathcal{R}_\Pi$, à laquelle on peut ajouter un fait (le plus général possible) et un but que l'on calculera en fonction de l'entrée, pour construire le programme $\mathcal{M}_S$:

$$\mathcal{M}_S \qquad \begin{cases} stop(fait_S) \leftarrow \ . \\ stop(gauche_S) \leftarrow stop(droit_S) \ . \quad (\mathcal{R}_\Pi) \\ \leftarrow stop(but_S) \ . \end{cases}$$

$\mathcal{M}_S$ est tel que avec comme entrée (dans but_S) un programme P (en fait une codification de P par son nombre de Gödel par exemple), il s'arrête si et seulement si P s'arrête et son temps d'arrêt est plus grand que celui nécessaire à $\mathcal{M}_{nS}(P)$ pour produire toutes les solutions de P. Remarquez que P n'est utilisé que dans le but.

12.7 Le Méta–Interpréteur

Utilisant $\mathcal{M}_{nS}$ et $\mathcal{M}_S$ il est possible de prouver que :

Théorème 23 *Il existe un méta–interpréteur pour les langages à clauses de Horn sous la forme d'un programme avec une seule clause de Horn binaire, un fait et un but, lequel, prenant comme entrée un programme à clauses de Horn P, a exactement les mêmes solutions que P et s'arrête si et seulement si P s'arrête.*

Preuve. Notons le programme $\mathcal{M}_{nS}$ ainsi :

$$\mathcal{M}_{nS} \quad \begin{cases} meta(fait_{nS}) \leftarrow \ . \\ meta(gauche_{nS}) \leftarrow meta(droit_{nS}) \ . \\ \leftarrow meta(but_{nS}) \ . \end{cases}$$

En fusionnant $\mathcal{M}_{nS}$ et $\mathcal{M}_S$ on obtient un nouveau méta–interpréteur :

$$\mathcal{MI} \quad \begin{cases} le\_meta(fait_{nS}, fait_S) \leftarrow \ . \\ le\_meta(gauche_{nS}, gauche_S) \leftarrow le\_meta(droit_{nS}, droit_S) \ . \\ \leftarrow le\_meta(but_{nS}, but_S) \ . \end{cases}$$

tel que, avec pour entrée un programme à clause de Horn P, il produit toutes les solutions de P, du fait de la partie $\mathcal{M}_{nS}$, et ensuite s'arrêtera si et seulement si P s'arrête du fait de la partie $\mathcal{M}_S$.

Ainsi nous avons obtenu un méta–interpréteur avec une clause binaire, un fait et un but qui préserve et les solutions (produites dans le même ordre qu'une résolution par une stratégie en largeur d'abord) et la terminaison de tout programme à clauses de Horn qui lui est fourni en entrée (ici la notion d'entrée est en fait celle de but). □

Ainsi, nous pouvons établir :

Théorème 24 *La classe des programmes à une clause de Horn binaire et deux clauses unaires a la même puissance d'expression que les machines de Turing.*

Preuve. Puisque nous avons obtenu un méta–interpréteur pour les langages à clauses de Horn construit à partir d'une seule clause récursive binaire, nous pouvons affirmer que cette classe de programmes a la même puissance de calcul que l'ensemble des programmes à clauses de Horn et donc que les machines de Turing (et tout formalisme équivalent). □

Ce méta–interpréteur est effectivement le plus simple que l'on puisse construire. Dans l'absolu il est constructible. Dans le même ordre d'idée que pour la clause explicitement constructible dont l'arrêt est indécidable, il suffit de créer un méta–interpréteur Prolog, de le coder dans le premier méta–programme, de lui associer une fonction de Conway et une clause de Horn, d'écrire le progamme $\mathcal{M}_S$ (bon courage) et le tour est joué. Je ne dévoilerai pas la solution et laisserai tout le loisir qu'il voudra au lecteur intéressé pour la trouver.

A. Parrain a galement tudi sa complexit dans [39] :

> "*... Sa complexité notée φ_u est majorée par (on reprend les notations précédentes) :*
>
> $$\varphi_u(sol_n) \leq N_m^{\varphi_m(sol_n)} \leq 2^{(\varphi_o(sol_n) \times N_o) + N_o}$$

> *φ_u reprsente bien la complexit du programme universel pour l'obtention des solutions. Pour l'arrt, la borne du nombre de nœuds parcourir est plus grande, puisqu'on ajoute la complexit du mcanisme d'arrt li aux fonctions de Conway...*"

Ce résultat peut être vu comme l'équivalent pour la programmation logique du théorème de Böhm–Jacopini. Donc par analogie, notre structure peut être vue comme le "tant-que" de la programmation logique.

Conclusion

Malheureusement notre étude des propriétés de terminaison et de satisfiabilité des programmes à clauses de Horn de plus petite structure non trivial possible :

$$\begin{cases} p(fait) \leftarrow \quad . \\ p(gauche) \leftarrow p(droit) \ . \\ \leftarrow p(but) \ . \end{cases}$$

a dégagé des résultats "négatifs" dans la mesure où elles sont toutes les deux indécidables dans le cas général. De ce fait, il ne sera pas possible comme nous l'espérions d'en déduire des propriétés ou des méthodes d'analyse pour les programmes de taille plus conséquente. De plus, la récursivité étant à la base de notre structure, il n'y a aucun espoir de pouvoir la contrôler.

Rappelons ces résultats :

- dès que l'un des termes *gauche* ou *but* est linéaire, le problème de l'arrêt est indécidable :

but	*gauche*	*droit*	Arrêt ?
clos	quelconque	quelconque	décidable
linéaire	quelconque	quelconque	décidable
quelconque	linéaire	quelconque	décidable
quelconque	quelconque	linéaire	indécidable

- il suffit que l'un des termes caractéristiques soit clos ou que trois d'entre eux soient linéaires pour que le problème du vide soit décidable :

but	*gauche*	*droit*	*fait*	Solution ?
clos	quelconque	quelconque	clos	décidable
linéaire	quelconque	quelconque	linéaire	décidable
clos	quelconque	quelconque	quelconque	décidable
quelconque	quelconque	quelconque	clos	
linéaire	linéaire	linéaire	quelconque	décidable
quelconque	linéaire	linéaire	linéaire	
quelconque	quelconque	linéaire	linéaire	indécidable
linéaire	linéaire	quelconque	quelconque	
quelconque	linéaire	linéaire	quelconque	indécidable

Cependant, nous avons pu déduire de ces résultats des preuves immédiates pour des problèmes de décision de certaines propriétés : programmes NSTO, à décoration totale, etc. De plus, l'indécidabilité du vide a permis de montrer l'insatisfiabilité des formules du permier ordre à quatre sous–formules.

En dernier lieu, ces résultats nous ayant amenés à considérer la puissance d'expression de cette classe de "petits" programmes, nous avons établi qu'elle avait la même puissance de calcul que les machines de Turing : tout ce qui est programmable peut l'être par un programme de cette forme ! Ce résultat peut être considéré comme l'analogue pour la programmation logique du théorème de Böhm–Jacopini pour les langages impératifs.

Tous ces résultats s'allient donc pour mettre en valeur la puissance des langages à clauses de Horn pourtant basés sur les principes simples de la résolution.

Lors de ces travaux, l'utilisation combinée des fonctions de Conway et de la technique d'indexation des variables lors de la résolution s'est révélée fournir un outil de preuve original et puissant. Même si dans le cas du problème du vide, une autre démonstration plus simple a été fournie par P. Hanschke et J. Würtz, l'utilisation de notre mécanisme de preuve a permis de résoudre ensuite simplement le cas de la clause linéaire.

En conséquence, il est légitime d'espérer appliquer cette technique pour la démonstration d'autres résultats. Ainsi, j'ai déjà montré précédemment, les liens entre l'implication de clause et notre schéma de programmes. J. Marcinkowski et L. Pacholski ont établi l'indécidabilité de l'implication $A \Rightarrow B$ si A est une clause de Horn à trois littéraux. Ils ne mettent cependant aucune condition sur la taille de B. Si B était binaire, ce problème serait équivalente à la satisfiabilité du programme :

$$\begin{cases} \delta \leftarrow \ . \\ \alpha \leftarrow \beta_1, \beta_2 \ . \\ \leftarrow \gamma \ . \end{cases}$$

où $A = \alpha \leftarrow \beta_1, \beta_2.$ et $B = \gamma \leftarrow \delta.$, avec γ et δ clos. Ce schéma de programmes est très proche de celui que nous avons étudié. Nous pouvons donc espérer pouvoir fournir une autre preuve du résultat de J. Marcinkowski et L. Pacholski, avec en plus une "contrainte" sur la taille de B.

A. Parrain, P. Devienne et P. Lebègue [39] se sont penché sur ce problème et ont montré que le problème de l'arrêt de cette classe de programmes pouvait être ramené à celui de l'arrêt, dans le cas général, des programmes que nous avons étudiés. Ils utilisent des techniques de transformations de programmes. Il est donc indécidable et la preuve est immédiate. Ils ont établi similairement que la satisfiabilité était indécidable si l'un des termes γ et δ était linéaire et l'autre clos. Si ces deux résultats ne permettent d'établir aucune conclusion immédiate quant à l'implication de clauses, ils semblent confirmer assez sérieusement l'espoir de l'établissement d'une preuve dans le cas où δ et γ sont clos.

D'autre part, comme nous l'avons dit précédemment, notre résultat sur les formules du premier ordre à quatre sous–formules nous amène à considérer le problème des formules quantifiées pures donné ouvert par W. Goldfarb et H.R. Lewis. Celui–ci est en effet un sous–cas de celui que nous avons résolu. Le problème est lié aux contraintes de la skolemisation et peut–être est–il possible par un bon choix de la forme des formules considérées d'appliquer notre méthode pour le résoudre. Ce problème, ouvert depuis longtemps, justifie en tout cas que cela soit essayé.

L'étude de ces deux problèmes : implication de clauses et satisfiabilité des formules quantifiées pures à quatre sous–formules semble constituer une suite logique aux travaux présentés ici.

Bibliographie

J'ai très brièvement précisé pour chaque référence son ou ses liens avec ce travail (ce qui n'exclut pas que d'autres sujets y soient abordés). J'espère que cela pourra guider le lecteur.

[1] Blair H.A. "The Recursion–Theoretic Complexity of the Semantics of Predicate Logic as a Programming Language." *Information and Control n°54. pp. 25–47.* 1982.

H.A. Blair montre qu'il est possible de tout calculer à l'aide de programmes logiques ne comportant que des clauses de Horn binaires récursives. Ce n'est pas le résultat principal de ce papier qui s'attache à donner une mesure de l'incomplétude de la négation par l'échec.

[2] Böhm C., Jacopini G. "Flow diagrams, Turing machines and languages with only two formation rules." *Communications of the Association for Computing Machinery, Vol.9, pp. 366–371.* 1966.

Les auteurs présentent la base de ce qui est maintenant connu comme le "théorème de Böhm–Jacopini". En fait dans cet article, ils ne se restreignaient pas à une seule boucle while.

[3] Bibel W., Hölldobler S., Würtz J. "Cycle Unification." *CADE pp. 94–108.* Juin 1992.

*Cet article introduit le terme "*Unification Cyclique*". Il présente un état de l'art complet (en juin 92) sur le sujet. Les auteurs résolvent certains cas particuliers décidables assez restrictifs.*

[4] Bratko I. "*Programmation Logique en PROLOG pour l'Intelligence Artificielle". InterEditions.* 1988.

Cet ouvrage présente la programmation en Prolog et l'applique à certaines techniques de l'IA.

[5] Bouquard J.L. "Logic programming and Attribute grammars." *Thèse de Doctorat. Orléans.* 1992.

*Dans cette thèse, J.L. Bouquart introduit le terme de programme à "*décoration totale*". Cette propriété est utilisée pour optimiser le passage de la programmation logique aux grammaires attribuées. Le problème de la décision de cette propriété pour un programme donné y est cité comme ouvert.*

[6] Chen H., Hsiang J. "Recurrence Domains : their Unification and Application to Logic Programming.", *Technical Report, Stoony Brook (available by anonymous ftp on* "sbcs.sunysb.edu"*).* July 1991.

*Les "recurrence domains" basés sur les ρ–*termes *sont introduits. ρ–Prolog, une extension du langage Prolog, est présenté. La preuve de l'algorithme d'unification donnée en annexe est assez coton !*

[7] Clocksin W. F., Mellish C. S., *"Programming in Prolog". Springer–Verlag.* 1981.

Une référence pour l'apprentissage de la Programmation Logique.

[8] Conway J.H. "Unpredictable Iterations." *Proc. 1972 Number Theory Conference. University of Colorado, pp. 49–52.* 1972.

Un article clé pour ce travail ! J.H. Conway y présente ce que nous avons appelé les fonctions de Conway *et en 4 pages seulement. Elles correspondent à une traduction arithmétique des machines de Minsky ([38]). Cet article n'est pas très facile à trouver...*

[9] Courcelle B. "*Fundamental Properties of Infinite Trees." Journal of Theoretical Computer Science, n°*17*, pp. 95–169.* 1983.

Comme le dit si bien le titre...

[10] Clark K.L., Tärnlund S.Å. "A First Order Theory of Data and Programs." *in Proc. IFIP 77. pp. 939–944.* 1977.

A l'origine des listes–différence.

[11] Dauchet M. "Simulation of Turing Machines by a regular rewrite rule." *Journal of Theoretical Computer Science. n°103. pp. 409–420.* 1992.

Le titre est explicite. Ce résultat a été présenté initialement à LICS.

[12] Dauchet M., Devienne P., Lebègue P. "Weighted Graphs : a Tool for Logic Programming." *11th Colloquium on Trees in Algebra and Programming (CAAP86).* 1986.

Première apparition officielle des graphes orientés pondérés. *Les décidabilités de la terminaison et de l'existence de solutions, pour nos programmes, quand le fait et le but sont linéaires y sont citées.*

[13] Dauchet M., Devienne P., Lebègue P. "Weighted Systems of Equations." *Informatika 91, Grenoble, Special issue of TCS.* 1991.

Les systèmes d'équations pondérés *sont introduits. Une preuve simple du résultat du papier précédent est donnée.*

[14] Delahaye J.P. "*Outils Logiques pour l'Intelligence Artificielle".* Editions Eyrolles. 1986.

Présentation très abordable des fondements de la programmation logique.

[15] Devienne P. "Les Graphes orientés pondérés : un outil pour l'étude de la terminaison et de la complexité dans les systèmes de réécritures et en programmation logique." *Thèse de Doctorat. Lille.* 1987.

Les principaux résultats sont repris dans l'article suivant.

[16] Devienne P. "Weighted graphs – tool for studying the halting problem and time complexity in term rewriting systems and logic programming." *Journal of Theoretical Computer Science, n°75, pp. 157–215.* 1990.

Présentation détaillée des GOPs. La preuve des problème de l'arrêt et du vide pour un but et un fait linéaires est donnée.

[17] Devienne P., Lebègue P., Routier J.C. "Weighted Systems of Equations revisited" *Analyse Statique, Actes WSA'92, Bordeaux, BIGRE 81–82. pp. 163–173.* September 1992.

Le formalisme de [13] est étendu aux liens exceptions *et* linéaires. *L'indécidabilité du problème de l'arrêt dans le cas général est trouvée grâce aux liens linéaires.*

[18] Devienne P., Lebègue P., Routier J.C. "Halting Problem of One Binary Horn Clause is Undecidable." *Proceedings of STACS'93, Springer–Verlag, Würzburg.* 1993.

Introduction de notre codage des fonctions de Conway à l'aide de clauses de Horn binaires. Grâce à lui, l'indécidabilité du problème de l'arrêt dans le cas général est prouvée. Des conséquences de ce résultat sont présentées.

[19] Devienne P., Lebègue P., Routier J.C. "The Emptiness Problem of One Binary Recursive Horn Clause is Undecidable." *International Logic Programming Symposium '93. The MIT Press. Vancouver.* Octobre 1993.

Grâce aux codages des fonctions de Conway dans une clause de Horn binaire, mais à l'aide de techniques différentes de [18], le problème du vide dans le cas général est montré est indécidable. Des sous–cas de l'arrêt et du vide sont également résolus.

[20] Devienne P., Lebègue P., Routier J.C., Würtz J. "One Binary Horn Clause is Enough" *LNCS. Springer–Verlag. STACS'94. Caen. à paraitre.* Mars 1993.

En combinant les preuves de [19] et [25], nous prouvons que les programmes à une seule clause binaire, un fait et un but ont la même puissance de calcul que les machines de Turing.

[21] Gaifman H., Mairson H., "Undecidable Optimisation Problems for Database Logic Programs." *Symposium on Logic in Computer Science, New–York, pp. 106–115.* 1987.

Où il est montré entre autres choses que la propriété de bornage est indécidable pour les programmes DATALOG linéaires.

[22] Goldfarb W., Lewis H.R. "The Decision Problem for Formulas with a Small Number of Atomic Subformulas" *J. Symbolic Logic 38(3), pp.471–480*, 1973.

La satisfiabilité des formules quantifiées pures à cinq sous–formules est montré indécidable puisqu'équivalente au problème de l'arrêt des machines à deux compteurs. Le cas à quatre sous–formules est donné ouvert.

[23] Guy R.K. "Conway's Prime Producing Machine" *Mathematics Magazine. n°56. pp. 26–33.* 1983.
R.K. Guy présente un exemple, réalisé en collaboration avec J.H. Conway, de passage d'une machine de Minsky à une fonction de Conway (ou plus exactement quelque chose de très proche, il manque quelques étapes pour avoir une vraie fonction de Conway).

[24] Hansson Å., Tärnlund S.Å "Program Transformation by Data Structure Mapping" *in "Logic Programming" K.L. Clark et S.Å. Tärnlund éditeurs. APIC Studies in Data Processing. Academic Press. pp. 117–122.* 1982.
Où l'on reparle des listes–différence et de la manière de passer d'un programme sans listes–différence à un autre équivalent, et plus efficace, qui en comporte.

[25] Hanschke P., Würtz J. "Satisfiability of the Smallest Binary Program." *Information Processing Letters, vol. 45, n°5. pp. 237–241.* Avril 1993.
Une très belle preuve du problème du vide dans le cas général. Elle a été obtenue simultanément et indépendamment de [19]. Elle est basée sur une codification très astucieuse du problème de Post dans une clause binaire récursive, un but et un fait.

[26] Harel D. "On folk theorems" *CACM, vol. 23, n°7. pp. 379–389.* 1980.
Après une définition de ce qu'est un "folk theorem", D. Harel décrit l'historique du théorème de Böhm–Jacopini. Trois preuves en existent et... aucune n'est dûe à G. Böhm et C. Jacopini puisque ce théorème leur est un peu abusivement attribué...

[27] Hofstadter D. "*Gödel Escher Bach. les Brins d'une Guirlande Eternelle."* InterEditions. 1979.
Faut–il vraiment présenter ce livre ? De plus est–il résumable en quelques mots ? De nombreux domaines sont abordés... On y trouve une présentation intuitive et progressive de la calculabilité, des théorèmes de Gödel, etc.

[28] Hopcroft J., Ullman J. "*Introduction to Automata Theory, Languages and Computation." Addison–Wesley Editeurs.* 1979.
Une référence en informatique théorique : recursivité, machines de Turing, calculabilité, etc.

[29] Kowalski R.A., "Predicate Logic as a Programming Language." *Information Processing 74, Stockholm, North Holland, pp. 569–574.* 1974.
L'auteur décrit la procédure de résolution qui est à la base de la programmation logique.

[30] Lagarias J.C. "The $3x + 1$ problem and its generalizations." *Amer. Math Monthly 92, pp. 3–23.* 1985.

Une présentation détaillée de nombreux travaux réalisé autour de la conjecture de Collatz. Il faut cependant noter que si ce papier a été à l'origine de nos travaux sur les fonctions de Conway, le théorème de Conway qui y est donné est faux ! Malheureusement, personne ne s'en était rendu compte et nous, cela nous a pris quelques temps...

[31] Lagarias J.C. "Bibliographie Commentée sur le Problème de Collatz". Communication privée. 1992.

Merci à J.C. Lagarias.

[32] Lebègue P. "Contribution à l'Etude de la Programmation Logique par les Graphes Orientés Pondérés" *Thèse de Doctorat. Lille.* 1988.

Des GOPs et de la programmation logique.

[33] Lewis H.R. "The decision Problem for Formulae with a Bounded Number of Atomic Subformulae." *Notices of the American Mathematical Society. vol. 20.* 1973.

Un autre travail de H.R. Lewis (Cf. [22]) sur la satisfiabilité des formules, le cas à quatre sous–formules y est donné ouvert.

[34] Lloyd J.W. *"Foundations of Logic Programming." Second, Extended Edition* Springer–Verlag. 1987.

Une référence en théorie de la programmation logique. Tout ce qui est fondamental y est présenté.

[35] Marcinkowski J. "A Horn Clause that Implies an Undecidable Set of Horn Clauses." *private communication.* 1993.

Le problème de l'implication de clause $A \Rightarrow B$ y est montré indécidable pour une clause de Horn particulière A à trois littéraux.

[36] Marcinkowski J., Pacholski L. "Undecidability of the Horn–Clause Implication Problem." *FOCS* 1992.

Le problème de l'implication de clause $A \Rightarrow B$ est montré indécidable dans le cas où A est une clause de Horn à trois littéraux. Ce cas était le dernier ouvert concernant ce problème.

[37] Marriott K., Søndergaard H. "Difference–List Transformation for Prolog." *New Generation Computing, n°11, Springer–Verlag. pp. 125–157.* 1993.

Les listes–différence et une étude fine des transformations de programmes les concernant.

[38] Minsky M. *"Computation : Finite and Infinite Machines."* Prentice–Hall. 1967.

Un autre ouvrage complet sur la théorie de la calculabilité. M. Minsky y propose des machines à états et registres que l'on appelle Machines de Minsky *(ou machines à compteurs). Celles–ci sont équivalentes aux machines de Turing, deux compteurs étant suffisants.*

[39] Parrain A. "Transformations de Programmes Logiques et Sémantique Opérationnelle" *Thèse de Doctorat. Lille. A paraître.* Fevrier 1994.

On y parle aussi de listes–différence, mais surtout du méta–interpréteur à deux clauses binaires que nous utilisons. Celui–ci est présenté en détail. Merci à Anne pour sa collaboration sur ce sujet.

[40] Parrain A., Devienne P., Lebègue P. "Prolog programs transformations and Meta–Interpreters." *Logic program synthesis and transformation, Springer–Verlag, LOPSTR'91, Manchester.* 1991.

C'est dans cet article que le méta–interpréteur à deux clauses binaires que nous utilisons a été présenté pour la première fois.

[41] Post E.M. "A Variant of a Recursively Unsolvable Problem" *Bulletin of American Mathematics Society. n°46. pp. 264–268.* 1946.

Où E.M. Post présente le problème qui porte maintenant son nom.

[42] Robinson J.A., "A Machine–oriented Logic Based on the Resolution Principle." *Journal of the ACM n°12 (1), pp. 23–41.* Janvier 1965.

Sept années avant les travaux de A. Colmerauer et R.K. Kowalski, J.A. Robinson présente la procédure originale de résolution. L'article qui met en place les bases d'une programmation logique.

[43] Rogers H. *"Theory of Recursive Functions and Effective Computability."* The MIT Press. 1987.

Une approche très mathématique de la théorie de la récursivité.

[44] Salomaa A. "*Computation and Automata.*" *Encyclopedia of Mathematics and Computation, Volume 25, Cambridge University Press.* 1985.

Encore un ouvrage sur la calculabilité et les machines de Turing.

[45] Salzer G. "Solvable Classes of Cycle Unification Problems." *IMYCS, Smolenice (CSFR).* 1992.

Un autre cas particulier d'unification cyclique ([3]) est montré décidable. L'auteur y utilise les ρ–termes (Cf. [6]).

[46] Schmidt–Schauß M. "Implication of clauses is Undecidable." *Journal of Theoretical Computer Science, n°59, pp. 287–296.* 1988.

Le problème $A \Rightarrow B$ est montré indécidable si A à quatre littéraux. Par contre M. Schmidt–Schauß le montre décidable si A n'a que deux littéraux. Ce résultat a pour conséquence la décidabilité de l'arrêt et du vide d'une clause de Horn binaire avec un but et un fait clos. L'indécidabilité de la satisfiabilité des ensembles à deux clauses binaires est également prouvée.

[47] Tärnlund S.Å. "Horn Clause Computability" *BIT 172, pp. 215–226.* 1977.

L'auteur montre que l'on peut coder toute machine de Turing par des programmes à clauses binaires (non nécessairement récursives) ou unaires.

[48] Vardi, "Decidability and Undecidability Results for Boundedness of Linear Recursive Queries." *Symposium on Principles of Database Systems, Austin, pp. 341–351.* 1988.

Le titre dit tout ce qu'il faut.

[49] Würtz J. "Unifying Cycles." *Proceedings of the European Conference on Artificial Intelligence. pp. 60–64.* Août 1992.

Version abrégée de [3].

Index

V