Extension au premier ordre de l'unification des termes par CHR

Khalil Djelloul, Thi-Bich-Hanh Dao, Thom Fruehwirth

Extension au premier ordre de l'unification des termes par CHR*

Khalil Djelloul [1] **Thi-Bich-Hanh Dao** [2] **Thom Fruehwirth** [1]

[1] Fakultat fur Informatik. Universitat Ulm, Germany.
[2] Laboratoire d'Informatique Fondamentale d'Orléans, France.
{khalil.djelloul,thom.fruehwirth}@uni-ulm.de dao@univ-orleans.fr

Résumé

Prolog, acronyme de PROgrammation LOGique, est l'un des principaux langages de programmation logique. Un de ces concepts de base est *l'unification des termes* qui consiste à instancier les variables de deux termes afin que ces derniers soient syntaxiquement égaux. Techniquement parlant, l'unification est équivalente à la résolution de conjonctions d'équations dans la théorie des arbres finis ou infinis. Nous présentons dans ce papier une extension au premier ordre du mécanisme de l'unification des termes par un algorithme de résolution de contraintes du premier ordre (toute quantification et tout symbole logique) dans une théorie étendue T d'arbres finis ou infinis. Pour cela, nous étendons tout d'abord la théorie des arbres finis ou infinis par la relation $fini(t)$ qui permet de contraindre le terme t à être un arbre fini et présentons une axiomatisation au premier ordre de cette théorie étendue. Nous montrons ensuite la complétude de T par un solveur de contraintes du premier ordre sous forme de 16 règles de réécriture. Nous terminons ce papier par une implémentation en CHR de notre solveur. CHR (Constraint Handling Rules) est un langage de programmation logique à base de règles de réécriture qui a fait ses preuves dans la résolution de plusieurs problèmes complexes en IA. Cependant, il n'existe à ce jour aucun solveur CHR de contraintes du premier ordre. La complexité et la puissance de nos 16 règles de réécriture posent un véritable challenge pour CHR et ses programmeurs. Nous montrons alors comment représenter en CHR la structure complexe des formules du premier ordre et comment traduire chaque règle de notre solveur en règles CHR afin d'obtenir le premier solveur CHR de contraintes du premier ordre !

Abstract

Prolog, which stands for PROgramming in LOGic, is the most widely used language in the logic programming paradigm. One of its main concepts is unification. It represents the mechanism of binding the contents of variables and can be seen as solving conjunctions of equations over finite or infinite trees. We present in this paper a first-order extension of Prolog's unification by giving a general algorithm for solving any first-order constraint in an extended theory T of finite or infinite trees. For that, we extend the theory of finite or infinite trees by a relation which allows to distinguish between finite and infinite trees and present a first-order axiomatization of this extended theory. We then show the completeness of T by a giving a first-order constraint solver in T in the form of 16 rewriting rules. We end this paper describing a CHR implementation of our algorithm. CHR (Constraint Handling Rules) has originally been developed for writing constraint solvers, but the constraints here go much beyond implicitly quantified conjunctions of atomic constraints and are considered as arbitrary first-order formulas built on the signature of T. We discuss how we implement nested local constraint stores and what programming patterns and language features we found useful in the CHR implementation of our algorithm. This is the first CHR first-order constraint solver !

1 Introduction

Prolog : acronyme de PROgrammation LOGique [24, 5], est l'un des principaux langages de programmation logique. Un de ces concepts de base est *l'unification des termes* qui consiste à instancier les variables de deux termes afin que ces derniers soient syntaxiquement égaux. Par exemple, l'unification de $f(g(a))$ avec $f(x)$ est possible en instanciant x par $g(a)$. Cette unification a subi une première évolution dans la deuxième version de Prolog, connue sous le nom de PrologII, dans laquelle A. Colmerauer a intro-

*Une version complète de cet article vient d'être acceptée pour paraître dans la revue *"Theory and Practice of Logic Programming" (TPLP)*.

duit le concept de l'unification des termes infinis [4, 6]. Ainsi, l'unification des deux termes x et $f(x)$ est possible en instanciant x par le *terme infini* $f(f(f(...)))$. Mais la plus importante évolution qu'a connu Prolog a été effectuée dans Prolog III et IV où l'algorithme de l'unification de Prolog a été remplacé par un solveur de conjonctions de contraintes atomiques dans la théorie de M. Maher [20] des arbres finis ou infinis [7, 3]. Ainsi l'unification des deux termes $f(f(x))$ et $f(x)$ est obtenue par la résolution de la contrainte $f(f(x)) = f(x)$ dans la théorie des arbres finis ou infinis. Le solveur calcule alors une formule résolue qui présente les instanciations nécessaires pour une telle unification. En l'occurrence $x = f(f(...))$.

Nous présentons dans ce papier deux nouvelles contributions dans l'idée des extensions possibles du mécanisme de l'unification de Prolog :

(1) *Une contribution théorique* : Tout d'abord, nous étendons la signature de la théorie de M. Maher des arbres finis ou infinis [20] par la relation *fini*(t) qui contraint le terme t à être un arbre fini. Cette extension est essentiellement motivée par le manque d'expressivité de la théorie originale de M. Maher qui ne permet pas de contraindre certaines variables d'un problème à être des arbres strictement finis (ou strictement infinis). En effet, si un problème donné fait intervenir deux variables x et y et si x doit être un arbre fini et y un arbre infini alors il n'existe aucun moyen pour modéliser ce problème par des formules du premier ordre dans la théorie des arbres finis ou infinis ! L'extension de la signature de cette théorie par la relation *fini*(t) apporte donc une solution à ce problème. Afin de maintenir la complétude de cette théorie nous proposons deux nouveaux axiomes et montrons la complétude de la théorie résultante, notée T, par un algorithme non pas seulement de décision mais de résolution effective de contraintes du premier ordre, capable de résoudre n'importe quel problème de satisfaction de contraintes du premier ordre dans T. L'algorithme est donné sous forme de 16 règles de réécriture qui transforment toute contrainte du premier ordre φ en une équivalente disjonction ϕ de formules résolues dans lesquelles les solutions des variables libres sont exprimées d'une manière claire et explicite. On garantie entre autres qu'une formule résolue est soit la formule *vrai*, soit la formule *faux*, soit une formule ayant au moins une variable libre et n'étant équivalente ni à *vrai* ni à *faux*. Ce tout dernier point est extrêmement important et ne peut être garanti par les procédures de décision qui existent à ce jour pour la théorie des arbres finis ou infinis [20, 9] et leurs récentes extensions aux théories décomposables [11]. En effet, les procédures de décision garantissent uniquement la complétude d'une théorie par un algorithme qui pour toute proposition du premier ordre (formules dans lesquelles toutes les variables sont quantifiées) produit soit la formule *vrai* soit la formule *faux*, mais en aucun cas ce dernier n'est capable de produire des solutions claires et explicites pour une formule du premier ordre avec variables libres.

(2) *Une contribution d'ordre pratique* qui consiste en une implémentation CHR de notre solveuur. CHR : *Constraint Handling Rules* [15, 17, 28] est un langage de programmation logique à base de règles de réécriture, créé par Thom Fruehwirth dans les années 90 et dont le fonctionnement est régit par une sémantique raffinée [13]. L'idée de base consiste à appliquer des règles de réécriture sur une contrainte initiale jusqu'à ce qu'un point fixe soit atteint. Depuis plusieurs années CHR est utilisé avec succès pour le développement de solveurs de différents types de problèmes : Allocation de ressources [16], modélisation spatio-temporelle [14], systèmes multi-agents [25, 2], scheduling [1], semantic web [19], software engineering [22]...etc. Cependant, il n'existe à ce jour aucun solveur CHR de contraintes du premier ordre (toute quantification et tout symbole logique). La complexité et la puissance de nos 16 règles de réécriture posent un véritable challenge pour CHR et ses programmeurs. Nous montrons alors comment représenter en CHR la structure complexe d'une formule du premier ordre et comment gérer les différentes phases de notre solveur en s'appuyant entre autres sur la sémantique raffinée de CHR [13]. Le solveur CHR obtenu est le tout premier solveur CHR capable de résoudre des contraintes du premier ordre !

Ce papier est organisé en quatre sections suivies d'une conclusion. Cette introduction constitue la première section. Dans la section 2, nous présentons notre théorie étendue T des arbres finis ou infinis construite sur une signature contenant un ensemble infini de symboles ainsi qu'une relation unaire *fini*(t) qui contraint le terme t à être un arbre fini. Dans la section 3, nous présentons une structure élaborée de formules du premier ordre que l'on appelle *formules de travail* et montrons quelques unes de leurs propriétés. Nous terminons cette section par un algorithme de résolution de contraintes du premier ordre dans la théorie T. Ce dernier manipule des formules de travail et transforme toute contrainte du premier ordre en une disjonction de formules résolues chacune ayant des solutions claires et explicites. La correction de cet algorithme implique la complétude de notre théorie T. Enfin, nous présentons à la section 4 notre implémentation en CHR. Par manque de place nous ne pouvons présenter toutes les preuves des propriétés énoncées dans ce papier, cependant une version complète de cet article contenant toutes les preuves en détail est disponible en ligne [12].

2 Théorie étendue T des arbres finis ou infinis

2.1 Syntaxe

Soient V un ensemble infini dénombrable de *variables* et L l'ensemble des symboles *logiques*

$$=, \mathit{vrai}, \mathit{faux}, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists, (,).$$

Soit maintenant un ensemble supplémentaire S de symboles, appelé *signature* et partitionné en deux sous-ensembles : l'ensemble F des symboles *de fonction* et l'ensemble R des symboles *de relation*. A chaque symbole de fonction et de relation est attaché un entier n positif ou nul, son *arité*. Un symbole n-aire est un symbole d'arité n. Un symbole de fonction 0-aire est appelé *constante*. Fixons nous alors une signature $S = F \cup R$ pour toute cette sous section.

Un *terme* ou *S-terme*, est un mot construit sur $L \cup S \cup V$, de l'une des deux formes suivantes

$$x,\ ft_1 \dots t_n,$$

avec x pris dans V, f un symbole de fonction n-aire pris dans F et les t_i des termes de tailles plus petites que celui qui est en train d'être défini.

Une *formule* ou *S-formule* est un mot construit sur $L \cup S \cup V$ de l'une des onze formes suivantes

$$\begin{array}{c} s = t,\ rt_1 \dots t_n,\ \mathit{vrai},\ \mathit{faux}, \\ \neg\varphi,\ (\varphi \wedge \psi),\ (\varphi \vee \psi),\ (\varphi \rightarrow \psi),\ (\varphi \leftrightarrow \psi), \\ (\forall x\, \varphi),\ (\exists x\, \varphi), \end{array} \quad (1)$$

avec $x \in V$, s, t et t_i des termes, r un symbole de relation n-aire pris dans R et φ et ψ des formules de tailles plus petites que celle qui est en train d'être définie. Les formules de la première ligne de (1) sont dites *atomiques*, et *à plat* si elles sont de l'une des formes suivantes

$$\mathit{vrai},\ \mathit{faux},\ x_0 = fx_1 ... x_n,\ x_0 = x_1,\ rx_1 ... x_n,$$

où les x_i sont des variables éventuellement non distinctes prises dans $\mathbf{V}$, f un symbole de fonction n-aire pris dans F et r un symbole de relation n-aire pris dans R. Si φ est une formule, alors on note $\mathit{var}(\varphi)$ l'ensemble des variables libres de φ. Une *proposition* est une formule sans variables libres. Nous utiliserons également les quantificateurs $\exists?$ (au plus un) et $\exists!$ (un et un seul). Ces deux quantificateurs ne sont que des notations commodes que l'on peux exprimer facilement au premier ordre [11].

2.2 Axiomes

Soit F une ensemble de symboles de fonction contenant une infinité de symboles de fonction d'arités non nulles et au moins une constante. Soit *fini* un symbole de relation unaire. La théorie étendue T des arbres finis ou infinis construite sur la signature $S = F \cup \{\mathit{fini}\}$ consiste en la suite des propositions suivantes :

$\forall \bar{x} \forall \bar{y}$	$\neg(f(\bar{x}) = g(\bar{y}))$	[1]
$\forall \bar{x} \forall \bar{y}$	$f(\bar{x}) = f(\bar{y}) \rightarrow \bigwedge_i x_i = y_i$	[2]
$\forall \bar{x} \exists! \bar{z}$	$\bigwedge_i z_i = t_i[\bar{x}\bar{z}]$	[3]
$\forall \bar{x} \forall u$	$\neg(u = t[u, \bar{x}] \wedge \mathit{fini}(u))$	[4]
$\forall \bar{x} \forall u$	$(u = f(\bar{x}) \wedge \mathit{fini}(u)) \leftrightarrow (u = f(\bar{x}) \wedge \bigwedge_i \mathit{fini}(x_i))$	[5]

où f et g sont des symboles de fonction pris dans F, $\bar{x}$ est un vecteur de variables x_i éventuellement non distinctes, $\bar{y}$ est un vecteur de variables y_i éventuellement non distinctes, $\bar{z}$ est un vecteur de variables z_i toutes distinctes, $t_i[\bar{x}\bar{z}]$ est un terme qui commence par un élément pris dans F suivi par des variables prises dans $\bar{x}$ ou $\bar{z}$, et $t[u, \bar{x}]$ est un terme contenant au moins une occurrence d'un élément de F et de la variable u et éventuellement d'autres variables prises dans $\bar{x}$. Par exemple, on a $T \models \forall x_1 x_2 \forall u\, \neg(u = f_1(x_1, f_2(u, x_2)) \wedge \mathit{fini}(u))$ et $T \models \forall u \neg(u = f_1(f_2(u, f_0), f_0) \wedge \mathit{fini}(u))$ avec f_1 et f_2 deux symboles de fonction d'arité 2 et f_0 une constante.

Les formes [1],..., [5] sont appelées *schémas d'axiomes* de la théorie T. La proposition [1] dite *de conflit de symboles* montre que deux opérations distinctes produisent deux individus distincts. La proposition [2] dite *d'explosion* montre que deux individus commençant par deux symboles distincts sont toujours distincts. La proposition [3] dite de *solution unique* montre qu'une certaine forme de conjonctions d'équations a une solution unique. En particulier, la formule $\exists z\, z = f(z)$ a une solution unique qui est l'arbre infini $f(f(f(...)))$. Cet axiome est particulier à la théorie des arbres finis ou infinis et n'est pas valide par exemple pour la théorie Q des rationnels additifs ordonnés. En effet, on a $T \models \exists x!\, x = f(x, 1)$ mais en aucun cas on a $Q \models \exists x\, x = x + 1$. La proposition [4] exprime le fait qu'un arbre fini ne peut être un sous arbre infini de lui même. Nous insistons particulièrement sur le fait que le terme $t[u, \bar{x}]$ doit contenir au moins une occurrence d'un élément de F et de la variable u. Enfin, dans la proposition [5], si $\bar{x}$ est un vecteur vide de variables et f une constante alors nous obtenons $\forall u\, u = f \wedge \mathit{fini}(u) \leftrightarrow u = f$, qui exprime le fait que toute constante de F est un arbre fini.

Cette théorie est une extension de la théorie originale de M. Maher des arbres finis ou infinis [20] qui est construite sur signature contenant uniquement un ensemble infini de symboles de fonction. La théorie de M. Maher est composée des trois premiers axiomes de notre théorie T et sa complétude a été démontrée par une procédure de décision de propositions du premier ordre qui transforme toute proposition (formules sans variables libres) en une combinaison booléenne de conjonctions de formules atomiques quantifiées exis-

tentiellement. K. Djelloul a récemment proposé une procédure de décision sous forme de cinq règles de réécriture pour l'ensemble des théories dites décomposables [11] : arbres finis ou infinis, ordre denses sans extrêmes, rationnels et réels additifs ordonnés, égalité de Clark,...etc. Malheureusement cette procédure de décision ainsi que celle de M. Maher permettent uniquement de décider de la valeur de vérité des propositions du premier ordre (formules sans variables libres) : elles ne garantissent pas que les solutions des variables libres d'une formule résolue soient exprimées d'une manière claire et peuvent même générer une formule résolue ϕ qui contient au moins une variable libre mais qui est toujours équivalente à *vrai* ou à *faux* dans T. La formule résolue doit être dans ce cas directement la formule *vrai* ou *faux* et non pas ϕ. De bien plus puissants algorithmes doivent être alors utilisés, surtout lorsque l'on veut résoudre un problème de satisfaction de contrainte du premier ordre avec variables libres. En effet, dans ce genre de problèmes, notre but n'est pas de savoir s'il existe ou pas une solution mais d'obtenir ces solutions d'une manière claire et explicite, c'est-à-dire sous forme d'une formule résolue ϕ qui est soit la formule *vrai* (le problème est donc toujours vrai quelque soit les valeurs des variables libres), soit la formule *faux* (le problème est donc toujours faux), soit une formule très simple, n'étant équivalente ni à *vrai* ni à *faux* et dans laquelle les solutions des variables libres sont exprimées d'une manière claire et explicite. Ce sont ces algorithmes que l'on appelle : **solveur de contraintes du premier ordre**. Nous allons alors en présenter un dans la théorie T.

3 Résolution de contraintes du premier ordre dans T

Nous présentons dans cette section une structure de formules élaborées que l'on va utiliser ensuite dans notre solveur de contraintes du premier ordre. Plusieurs propriétés de ces formules sont également énoncées dans cette section.

3.1 Formules normalisées et formules de travail

Supposons que l'ensemble $\mathbf{V}$ des variables soit ordonné par une relation d'ordre strict, dense et sans extrêmes, notée $\succ$. A partir de cette section, nous imposons la discipline suivante à toute formule φ dans T : Les variables quantifiées de φ sont renommées telles que : (1) Les variables quantifiées de φ ont des noms distincts et différents de ceux des variables libres (2) Pour toutes variables x, y et toute sous-formule φ_i de φ, si y a une occurrence libre dans φ_i et x a une occurrence liée dans φ_i alors $x \succ y$. Il est évident que l'on peut toujours transformer toute formule φ en une formule équivalente ϕ qui respecte la discipline des formules dans T, simplement **en renommant les variables quantifiées** de φ. Il est également nécessaire de préciser que dans toutes les propriétés de cette section, chaque formule respecte la discipline des formules dans T. Cette discipline de formules est nécessaire dans les preuves de ces propriétés.

Définition 3.1.1 *Soit $v_1, ..., v_n, u_1, ..., u_m$ des variables. Une* formule de base *est une formule de la forme*

$$(\bigwedge_{i=1}^{n} v_i = t_i) \wedge (\bigwedge_{i=1}^{m} \mathit{fini}(u_i)) \qquad (2)$$

dans laquelle toutes les équations $v_i = t_i$ sont à plat. Notons que si $n = m = 0$ alors la formule (2) se réduit à la formule vrai. La formule de base (2) est dite résolue *si toutes les variables $v_1, ..., v_n, u_1, ..., u_m$ sont distinctes et pour toute équation de la forme $x = y$ on a $x \succ y$. Si α est une formule de base alors on note : (i) $MBG(\alpha)$ l'ensemble des variables qui apparaissent dans le membre gauche d'une équation de α. (ii) $FINI(\alpha)$ l'ensemble des variables qui apparaissent dans une sous-formule de α de la forme $\mathit{fini}(x)$.*

Définition 3.1.2 *Soit α une formule de base et $\bar{x}$ un vecteur de variables. Les variables et équations accessibles dans α à partir de la variable x_0 sont celles qui apparaissent dans une sous formule de α de la forme :*

$$x_0 = t_0(x_1) \wedge x_1 = t_1(x_2) \wedge ... \wedge x_{n-1} = t_{n-1}(x_n),$$

où x_{i+1} apparaît dans le terme $t_i(x_{i+1})$. Les variables et équations accessibles de $\exists \bar{x}\, \alpha$ sont celles qui sont accessible dans α à partir des variables libres de $\exists \bar{x}\, \alpha$. Une sous formule de α de la forme $\mathit{fini}(u)$ est dite accessible dans $\exists \bar{x}\, \alpha$ si $u \notin \bar{x}$ ou u est une variable accessible de $\exists \bar{x}\, \alpha$.

Exemple 3.1.3 *Dans la formule : $\exists uvw\, z = f(u, v) \wedge v = g(v, u) \wedge w = f(u, v) \wedge \mathit{fini}(u) \wedge \mathit{fini}(x)$, l'équations $z = f(u, v)$ et $v = g(v, u)$, les variables z, u et v ainsi que les formules $\mathit{fini}(u)$ et $\mathit{fini}(x)$ sont accessibles. D'autre part, l'équation $w = f(u, v)$ et la variable w ne sont pas accessibles.*

La notion d'accessibilité est vitale pour notre algorithme de résolution dans T car elle permet d'identifier les sous-formules qui n'admettent pas d'élimination de quantificateurs et donc qui ne peuvent plus être simplifiées davantage. En effet, si α est une formule de base et si toutes les variables de $\bar{x}$ sont accessibles dans $\exists \bar{x}\, \alpha$ alors la formule $\exists \bar{x}\, \alpha$ n'admet pas d'élimination de quantificateurs. Si l'on revient à l'exemple précédent, les variables u et v sont accessibles dans la sous-formule $\exists uv\, z = f(u, v) \wedge v = g(v, u)$ et donc cette sous-formule ne peut être simplifiée davantage.

Définition 3.1.4 *Une formule normalisée φ de profondeur $d \geq 1$ est une formule de la forme*

$$\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i \in I} \varphi_i),$$

où I est un ensemble fini (éventuellement vide), α une formule de base et les φ_i des formules normalisées de profondeur d_i avec $d = 1 + \max\{0, d_1, ..., d_n\}$.

Exemple 3.1.5 *Soient f et g deux symboles de fonction unaire pris dans F. La formule*

$$\neg \left[\begin{array}{l} \exists \varepsilon\, \mathit{fini}(u) \wedge \\ \neg \left[\begin{array}{l} \exists x\, y = f(x) \wedge x = g(y) \wedge \\ \neg \left[\ \exists \varepsilon\, y = g(x) \wedge \mathit{fini}(x) \ \right] \end{array} \right] \end{array} \right]$$

est une formule normalisée de profondeur égale à trois.

Propriété 3.1.6 *Toute formule φ est équivalente dans T à une formule normalisée.*

Une preuve détaillée de cette propriété est disponible dans [11] où l'on a utilisé une notion similaire de formules normalisées pour construire une procédure de décision pour les théories décomposables.

Définition 3.1.7 *Une formule* résolue générale *est une formule de la forme $\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \neg(\exists \bar{y}_i\, \beta_i))$, avec $n \geq 0$ et telle que :*

1. *la formule α ainsi que chaque β_i, avec $i \in \{1, ..., n\}$, sont des formules de base résolues.*
2. *Si α' est une conjonction d'équations de α alors toutes les conjonctions $\alpha' \wedge \beta_i$, avec $i \in \{1, ..., n\}$, sont des formules de base résolues.*
3. *Toutes les variables de $\bar{x}$ sont accessibles dans $\exists \bar{x}\, \alpha$.*
4. *Les variables de $\bar{y}_i$ sont accessibles dans $\exists \bar{y}_i\, \beta_i$ pour tout $i \in \{1, ..., n\}$.*
5. *Si $\mathit{fini}(u)$ est une sous-formule de α alors pour tout $i \in \{1, ..., n\}$, la formule β_i contient soit $\mathit{fini}(u)$, soit $\mathit{fini}(v)$ où v est une variable accessible à partir de u dans $\alpha \wedge \beta_i$ et n'apparaît dans aucun membre gauche d'une équation de $\alpha \wedge \beta_i$.*
6. *Pour tout $i \in \{1, ..., n\}$, la formule β_i contient au moins une formule atomique qui n'apparaît pas dans α.*

Enonçons maintenant une des propriétés les plus importantes de ce papier :

Propriété 3.1.8 *Soit φ une formule générale résolue. Si φ n'a pas de variables libres alors φ est la formule $\neg(\exists \varepsilon\, \mathit{vrai})$ sinon on a ni $T \models \neg\varphi$ ni $T \models \varphi$.*

Cette propriété montre qu'une formule résolue générale est ni vraie ni fausse dans T et ne peut être simplifiée davantage du fait des propriétés d'accessibilité sur ses variables quantifiées. Toute formule générale résolue a donc un ensemble non vide de solutions ainsi qu'un ensemble non vide de non solutions.

Définition 3.1.9 *Soit φ une formule de la forme*

$$\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \neg(\exists \bar{y}_i\, \beta_i), \tag{3}$$

avec $\bar{x}$ et $\bar{y}$ deux vecteurs de variables, $n \geq 0$ et α et les β_i, avec $i \in \{1, ..., n\}$, des formules de base. On dit que φ est sous forme résolue explicite *si et uniquement si la formule $\neg\varphi$, c'est-à-dire i.e.*

$$\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \neg(\exists \bar{y}_i\, \beta_i)), \tag{4}$$

est une formule résolue générale.

Cette définition montre comment facilement extraire à partir d'une formule générale résolue φ, une formule ϕ qui ne contient qu'un seul niveau de négation et dans laquelle les solutions des variables libres sont exprimées d'une manière claire et explicite.

Exemple 3.1.10 *Soit w, v, u_1, u_2, u_3 des variables telles que $w \succ v \succ u_1 \succ u_2 \succ u_3$. Soit φ la formule résolue générale suivante :*

$$\neg \left[\begin{array}{l} \exists v\, u_1 = f(v) \wedge v = u_2 \wedge \mathit{fini}(u_2) \wedge \\ \neg(\exists w\, u_2 = f(w) \wedge \mathit{fini}(w) \wedge \mathit{fini}(u_3)) \end{array} \right].$$

D'après la définition 3.1.9, la formule ϕ suivante est sous forme résolue explicite :

$$\left[\begin{array}{l} \exists v\, u_1 = f(v) \wedge v = u_2 \wedge \mathit{fini}(u_2) \wedge \\ \neg(\exists w\, u_2 = f(w) \wedge \mathit{fini}(w) \wedge \mathit{fini}(u_3)) \end{array} \right]. \tag{5}$$

En effet, prenons le modèle $\mathcal{T}r$ des arbres finis ou infinis et donnons toutes les instanciations possibles u_1^*, u_2^*, u_3^* des variables libres u_1, u_2, u_3 telles que la formule instanciée de ϕ soit vraie dans $\mathcal{T}r$. De la formule (5) nous identifions clairement les ensembles de solutions suivants :

- *Solution 1 :*
  - *u_3^* est un arbre infini quelconque.*
  - *u_2^* est un arbre fini quelconque.*
  - *u_1^* est l'arbre $f(u_2^*)$.*
- *Solution 2 :*
  - *u_3^* est un arbre fini quelconque.*
  - *u_2^* est un arbre fini quelconque qui commence par un symbole de fonction distinct de f.*
  - *u_1^* est l'arbre $f(u_2^*)$.*

Définition 3.1.11 *Une formule de travail est une formule normalisée dans laquelle toutes les occurrences de* $\neg$ *sont remplacées par* $\neg^k$ *avec* $k \in \{0, ..., 5\}$ *et telle que chaque occurrence d'une sous formule de la forme*

$$p = \neg^k(\exists \bar{x}\, \alpha \wedge q), \quad avec \;\; k > 0, \tag{6}$$

satisfait les k ***premières conditions*** *de la liste des conditions ci-dessous. Dans (6),* α *est une formule de base,* q *est une conjonction de formules de travail de la forme* $\bigwedge_{i=1}^{n} \neg^{k_i}(\exists \bar{y}_i\, \beta_i \wedge q_i)$*, avec* $n \geq 0$*,* β_i *est une formule de base,* q_i *est une conjonction de formules de travail et dans la liste des conditions ci-dessous* α' *est la formule de base de la sur-formule de travail immédiate*[1] p' *de* p *(si elle existe)*

1. *Si* p' *existe alors* $T \models \alpha \rightarrow \alpha'$ *et* $T \models \alpha_{eq} \rightarrow \alpha'_{eq}$ *où* α_{eq} *et* α'_{eq} *sont les conjonctions des équations de* α *et* α'*. De plus, l'ensemble des variables de* $MBG(\alpha') \cup FINI(\alpha')$ *est inclus dans celui de* $MBG(\alpha) \cup FINI(\alpha)$*.*
2. *Les membres gauches des équations de* α *sont tous distincts et pour toute équation de la forme* $u = v$ *on a* $u \succ v$*.*
3. α *est une formule de base résolue.*
4. *Si* p' *existe alors l'ensemble des équations de* α' *est inclus dans celui de* α*.*
5. *Les variables de* $\bar{x}$*, les équations de* α *ainsi que les contraintes de la forme* $fini(x)$ *de* α *sont tous accessibles dans* $\exists \bar{x}\, \alpha$*. De plus, si* $n > 0$ *alors pour tout* $i \in \{1, ..., n\}$ *la conjonction* β_i *contient au moins une formule atomique qui n'apparaît pas dans* α*.*

Une formule de travail est dite initiale *si elle commence par* $\neg^4$ *et* $k = 0$ *pour toutes les autres occurrences de* $\neg^k$*. Une formule de travail est dite* finale *si elle est de profondeur inférieure ou égale à deux avec* $k = 5$ *pour toute occurrence de* $\neg^k$*.*

Propriété 3.1.12 *Soit* p *la formule de travail finale suivante* $\neg^5(\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \neg^5(\exists \bar{y}_i\, \beta_i)$*. La formule* $\neg(\exists \bar{x}\, \alpha \wedge \bigwedge_{i=1}^{n} \neg(\exists \bar{y}_i\, \beta_i^*))$ *est une formule générale résolue équivalente à* p *dans* T*, où* β_i^* *est la formule de base* β_i *dans laquelle nous avons supprimé toutes les équations qui apparaissent également dans* α*.*

3.2 Les règles de réécriture

Nous présentons dans la figure 1, les règles de réécriture qui transforment toute formule de travail initiale en une conjonction équivalente de formules de travail finales. Dans ces règles les lettres u, v et w représentent des variables, les lettres $\bar{x}$, $\bar{y}$ et $\bar{z}$ représentent des vecteurs de variables, les lettres a, b et c représentent des formules de base, la lettre q représente une conjonction de formules de travail, la lettre r représente une conjonction d'équations à plat, de formules de la forme $fini(x)$ et de formules de travail. Toutes ces lettres peuvent être indicées et avoir des primes. De plus, $u \succ v$, et f et g sont deux symboles de fonction distincts. Dans la règle (3), t est un terme à plat (soit une variable soit un terme de la forme $f(x_1, ..., x_n)$ avec f un symbole de fonction n-aire et $x_1, ..., x_n$ des variables). Dans la règle (6), les équations de a ont des membres gauches distincts et pour toute équation de la forme $u = v$ on a $u \succ v$. Dans la règle (9), la variable u est accessible à partir de u dans a. Dans la règle (10), la variable u est non accessible à partir de u dans a. De plus, si f est une constante alors $n = 0$. Dans la règle (11), a est une formule de base résolue. Dans la règle (13), a et a'' sont des conjonctions d'équations ayant les mêmes membres gauches et a' est une conjonction de formules de la forme $fini(u)$. Dans la règle (15), $n \geq 0$ et pour tout $i \in \{1, ..., n\}$ la formule b_i est différente de la formule a. Les couples $(\bar{x}', a')$ et $(\bar{y}'_i, b'_i)$ sont obtenus par une décomposition de $\bar{x}$ et a en $\bar{x}'\bar{x}''\bar{x}'''$ et $a' \wedge a'' \wedge a'''$ de la manière suivante :

- a' est la conjonction des équations et des formules de la forme $fini(x)$ qui sont accessibles dans $\exists \bar{x}\, a$.
- $\bar{x}'$ est le vecteur des variables de $\bar{x}$ qui sont accessibles dans $\exists \bar{x}\, a$.
- a'' est la conjonction des formules de la forme $fini(x)$ qui sont non accessibles dans $\exists \bar{x}\, a$.
- $\bar{x}''$ est le vecteur des variables de $\bar{x}$ qui sont non accessibles dans $\exists \bar{x}\, a$ et n'apparaissent dans aucun membre gauche des équations de a.
- $\bar{x}'''$ est le vecteur des variables de $\bar{x}$ qui sont non accessibles dans $\exists \bar{x}\, a$ et apparaissent dans les membres gauches des équations de a.
- a''' est la conjonction des équations qui sont non accessibles dans $\exists \bar{x}\, a$.
- b_i^* est la formule obtenue en supprimant de b_i les formules de la forme $fini(u)$ qui apparaissent également dans a''.
- $\bar{y}'_i$ est le vecteur des variables de $\bar{y}_i\bar{x}'''$ qui sont accessibles dans $\exists \bar{y}_i\bar{x}'''\, b_i^*$.
- b'_i est la conjonction des équations et de formules de la forme $fini(x)$ qui sont accessibles dans $\exists \bar{y}_i\bar{x}'''\, b_i^*$.
- $K \subseteq \{1, ..., n\}$ est l'ensemble des indices i tels que $i \in K$ si et seulement si aucune variable de $\bar{x}''$ n'apparaît dans b'_i.
- La formule $\bigwedge_{i \in K} \neg^5(\exists \bar{y}'_i\, b'_i)^*$ est la formule $\bigwedge_{i \in K} \neg^5(\exists \bar{y}'_i\, b'_i)$ dans laquelle nous avons renommé les variables quantifiées de tel façon à ce

[1] En d'autres termes p' est de la forme $\neg^{k'}(\exists \bar{x}'\, \alpha' \wedge p^* \wedge p)$ où p^* est une conjonction de formules de travail et p est la formule (6).

$$
\begin{array}{lcc}
(1) & \neg^1(\exists \bar{x}\, u = u \wedge r) & \Longrightarrow \quad \neg^1(\exists \bar{x}\, r) \\
(2) & \neg^1(\exists \bar{x}\, v = u \wedge r) & \Longrightarrow \quad \neg^1(\exists \bar{x}\, u = v \wedge r) \\
(3) & \neg^1(\exists \bar{x}\, u = v \wedge u = t \wedge r) & \Longrightarrow \quad \neg^1(\exists \bar{x}\, u = v \wedge v = t \wedge r) \\
(4) & \neg^1(\exists \bar{x}\, u = fv_1 ... v_n \wedge u = gw_1 ... w_m \wedge r) & \Longrightarrow \quad vrai \\
(5) & \neg^1(\exists \bar{x}\, u = fv_1 ... v_n \wedge u = fw_1 ... w_n \wedge r) & \Longrightarrow \quad \neg^1(\exists \bar{x}\, u = fv_1 ... v_n \wedge \bigwedge_{i=1}^{n} v_i = w_i \wedge r) \\
(6) & \neg^1(\exists \bar{x}\, a \wedge q) & \Longrightarrow \quad \neg^2(\exists \bar{x}\, a \wedge q) \\
(7) & \neg^2(\exists \bar{x}\, \mathit{fini}(u) \wedge \mathit{fini}(u) \wedge r) & \Longrightarrow \quad \neg^2(\exists \bar{x}\, \mathit{fini}(u) \wedge r) \\
(8) & \neg^2(\exists \bar{x}\, u = v \wedge \mathit{fini}(u) \wedge r) & \Longrightarrow \quad \neg^2(\exists \bar{x}\, u = v \wedge \mathit{fini}(v) \wedge r) \\
(9) & \neg^2(\exists \bar{x}\, \mathit{fini}(u) \wedge a \wedge q) & \Longrightarrow \quad vrai \\
(10) & \neg^2(\exists \bar{x}\, u = f(v_1, ..., v_n) \wedge \mathit{fini}(u) \wedge r) & \Longrightarrow \quad \neg^2(\exists \bar{x}\, u = f(v_1, ..., v_n) \wedge \bigwedge_{i=1}^{n} \mathit{fini}(v_i) \wedge r) \\
(11) & \neg^2(\exists \bar{x}\, a \wedge q) & \Longrightarrow \quad \neg^3(\exists \bar{x}\, a \wedge q) \\
(12) & \neg^4(\exists \bar{x}\, a \wedge q \wedge \neg^0(\exists \bar{y}\, r)) & \Longrightarrow \quad \neg^4(\exists \bar{x}\, a \wedge q \wedge \neg^1(\exists \bar{y}\, a \wedge r)) \\
(13) & \neg^4(\exists \bar{x}\, a \wedge a' \wedge q \wedge \neg^3(\exists \bar{y}\, a'' \wedge r)) & \Longrightarrow \quad \neg^4(\exists \bar{x}\, a \wedge a' \wedge q \wedge \neg^4(\exists \bar{y}\, a \wedge r)) \\
(14) & \neg^4(\exists \bar{x}\, a \wedge q \wedge \neg^5(\exists \bar{y}\, a)) & \Longrightarrow \quad vrai \\
(15) & \neg^4(\exists \bar{x}\, a \wedge \bigwedge_{i=1}^{n} \neg^5(\exists \bar{y}_i\, b_i)) & \Longrightarrow \quad \neg^5(\exists \bar{x}'\, a' \wedge \bigwedge_{i \in K} \neg^5(\exists \bar{y}'_i\, b'_i)^*) \\
(16) & \neg^4 \left[\begin{array}{l} \exists \bar{x}\, a \wedge q \wedge \\ \neg^5 \left[\begin{array}{l} \exists \bar{y}\, b \wedge \\ \bigwedge_{i=1}^{n} \neg^5(\exists \bar{z}_i\, c_i) \end{array} \right] \end{array} \right] & \Longrightarrow \quad \left[\begin{array}{l} \neg^4(\exists \bar{x}\, a \wedge q \wedge \neg^5(\exists \bar{y}\, b)) \wedge \\ \bigwedge_{i=1}^{n} \neg^4(\exists \bar{x} \bar{y} \bar{z}_i\, c_i \wedge q_0)^* \end{array} \right]
\end{array}
$$

Fig 1. Transformation d'une formule de travail initiale en une formule de travail finale.

qu'elles respectent la discipline des formules dans T.

Dans la règle (16), $n > 0$ et q_0 est la formule q dans laquelle toutes les occurrences de $\neg^k$ ont été remplacées par $\neg^0$. La formule $\bigwedge_{i=1}^{n} \neg^4(\exists \bar{x} \bar{y} \bar{z}_i\, c_i \wedge q_0)^*$ est la formule $\bigwedge_{i=1}^{n} \neg^4(\exists \bar{x} \bar{y} \bar{z}_i\, c_i \wedge q_0)$ dans laquelle nous avons renommé les variables quantifiées de tel façon à ce qu'elles respectent la discipline des formules dans T.

Le fait d'utiliser des indices sur les négations nous permet de guider l'application de nos règles et de forcer notre algorithme à suivre une stratégie jusqu'à ce que l'on atteigne une conjonction de formules finales. En effet, l'application des règles suit deux phases :

- (i) Une résolution descendante qui propage les formules de base suivant la structure arborescente des formules de travail en utilisant les règles (1),...,(13). Ainsi, les formules de base sont résolues puis propagées à toutes les sous-formules de travail. Le test de contradiction entre arbres finis ou infinis est réalisé par la règle (9).
- (ii) Une analyse ascendante qui élimine une partie des quantificateurs et diminue la profondeur des formules de travail par les règles (14),...,(16). Les formules inconsistantes sont également supprimées dans cette phase.

Plus précisément, partant d'une formule initiale φ de la forme $\neg^4(\exists \bar{x}\, a \wedge \bigwedge_{i \in I} q_i)$, où tous les q_i sont des formules de travail dont les négations sont toutes de la forme $\neg^0$, la règle (12) propage les formules atomiques de a dans les sous-formules q_i, avec $i \in I$, et change la première négation des q_i en $\neg^1$. Les règles (1),...,(5) peuvent maintenant être appliquées jusqu'à ce que les équations de a aient des membres gauches distincts et que pour toute équation de la forme $u = v$ on ait $u \succ v$. La règles (6) est ensuite appliquée et change la première négation de q_i en $\neg^2$. L'algorithme commence maintenant une nouvelle étape qui consiste à résoudre les conjonctions de formules de base en utilisant les règles (7),...,(10). En particulier, un test sur les arbres finis ou infinis est effectué par la règle (9). Lorsqu'une formule de base résolue est obtenue, la règle (11) est appliquée et change la négation en $\neg^3$. Notons entre autres que si une formule de travail commence par $\neg^3$ alors sa sur-formule de travail commence par $\neg^4$. La règle (13) est ensuite appliquée. Elle restaure les équations et change la première négation en $\neg^4$. La règle (12) peut maintenant être appliquée une seconde fois car toutes les négations imbriquées sont de la forme $\neg^0$ et ainsi de suite. Ceci est la première phase de notre algorithme. Une fois que toutes les formules de travail de profondeur 1 sont de la forme $\neg^4(\exists \bar{y}_i\, b_i)$, la deuxième phase de notre algorithme peut commencer en utilisant la règle (15) avec $n = 0$ sur ces formules de travail de profondeur 1 et transforme leurs négations en $\neg^5$. Les formules de travail inconsistantes de la forme $\neg^4(\exists \bar{x}\, \alpha \wedge \neg^5(\exists \bar{y}\, \alpha) \wedge q)$ sont ensuite supprimées par la règle (14). Lorsque toutes les formules inconsistantes ont été supprimées, la règle (15) avec $n \neq 0$ peut être appliquée sur

les sous-formules de travail de profondeur 2 de la forme $\neg^4(\exists\bar{x}\, a \wedge \bigwedge_{i\in I} \neg^5(\exists\bar{y}_i\, b_i))$ et produit une formule de travail de la forme $\neg^5(\exists\bar{x}\, a \wedge \bigwedge_{i\in I} \neg^5(\exists\bar{y}_i\, b_i))$. La règle (16) peut alors être appliquée sur les formules de travail de profondeur $d > 2$ de la forme $\neg^4(\exists\bar{x}\, a \wedge q \wedge \neg^5(\exists\bar{y}\, b \wedge \bigwedge_{i=1}^{n} \neg^5(\exists\bar{z}_i\, c_i)))$. Après chaque application de cette règle, de nouvelles formules de travail sont générées avec des négations de la forme $\neg^0$ ce qui implique l'exécution à nouveau des règles de la première phase de notre algorithme en commençant par la règle (12) et ainsi de suite. Après plusieurs applications des deux phases nous obtenons une conjonction de formules de travail de profondeur inférieure ou égale à 2. Les règles sont alors appliquées une dernière fois jusqu'à ce que toutes les négations soient de la forme $\neg^5$. **C'est une conjonction de formules finales**.

Propriété 3.2.1 *Toute application répétée des règles sur une formule initiale p termine et produit une conjonction de formules finales équivalente à p dans T.*

3.3 Le solveur de contraintes du premier ordre dans T

Soit p une formule quelconque du premier ordre. La résolution de p dans T s'effectue de la manière suivante :

(1) Transformer la formule $\neg p$ **la négation de la formule p** en une formule équivalente normalisée p_1.

(2) Transformer p_1 en la formule de travail initiale p_2 suivante

$$p_2 = \neg^4(\exists\varepsilon\ vrai \wedge \neg^0(\exists\varepsilon\ vrai \wedge p_1)),$$

dans laquelle toutes les occurrences de $\neg$ dans p_1 sont remplacées par $\neg^0$.

(3) Appliquer les 16 règles de réécriture sur p_2 jusqu'à ce qu'aucune règle ne soit applicable. D'après la propriété 3.2.1, nous obtenons à la fin une formule de travail finale p_3 de la forme

$$\bigwedge_{i=1}^{n} \neg^5(\exists\bar{x}_i\, \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg^5(\exists\bar{y}_{ij}\, \beta_{ij})).$$

D'après la propriété 3.1.12, la formule p_3 est équivalente dans T à la conjonction suivante p_4 de formules générales résolues

$$\bigwedge_{i=1}^{n} \neg(\exists\bar{x}_i\, \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\, \beta^*_{ij})),$$

où β^*_{ij} est la formule β_{ij} dans laquelle nous avons supprimé toutes les équations qui apparaissent également dans α_i. Du fait que p_4 soit équivalente à $\neg p$ (la négation de p) alors p est équivalente à

$$\neg \bigwedge_{i=1}^{n} \neg(\exists\bar{x}_i\, \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\, \beta^*_{ij})),$$

qui est équivalente à la disjonction p_5 suivante

$$\bigvee_{i=1}^{n} (\exists\bar{x}_i\, \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\, \beta^*_{ij})).$$

Ceci est la réponse de notre solveur à la contrainte du premier ordre p. Notons que les négations qui étaient au début de chaque formule générale résolue p_4 ont été supprimées et que la conjonction a été remplacée par une disjonction. Par conséquent, l'ensemble des solutions des variables libres de p_5 n'est rien d'autre que l'union des solutions de chaque formule $\exists\bar{x}_i\, \alpha_i \wedge \bigwedge_{j=1}^{n_i} \neg(\exists\bar{y}_{ij}\, \beta^*_{ij})$. Chacune de ces formules est sous forme *résolue explicite*.

En utilisant la propriété 3.1.8 nous obtenons notre résultat principal :

Théorème 3.3.1 *Toute formule est équivalente dans T soit à vrai, soit à faux, soit à une disjonction de formules simples n'étant équivalente ni à vrai ni à faux et dans laquelle les solutions des variables libres sont exprimées d'une manière claire et explicite.*

4 Implémentations

Deux implémentations de nos 16 règles ont été effectuées : la première en C++ en utilisant pas moins de 1700 lignes de code et la deuxième en CHR en utilisant uniquement 73 règles CHR. Les 14 premières règles de notre solveur (figure 1) ont été traduites directement en règles CHR avec un taux d'une règle CHR pour chaque règle de notre solveur. Les règles (15) et (16), étant très complexes, ont nécessité l'utilisation de plusieurs règles CHR. Notre implémentation est disponible à l'adresse suivante `http://khalil.djelloul.free.fr/solver.txt` et l'on peut également la tester en ligne en utilisant le compilateur `webchr` disponible à l'adresse suivante `http://chr.informatik.uni-ulm.de/~webchr`

4.1 Constraint Handling Rules (CHR)

CHR [15, 17, 28] a depuis plusieurs années fait ses preuves dans le développement de solveurs de différents types de problèmes : allocation de ressources [16], modélisation spatio-temporelle [14], systèmes multi-agents [25, 2], scheduling [1], semantic web [19], software engineering [22]...etc. Sa sémantique raffinée a permis d'aboutir à des implémentations à la fois concises et efficaces. Cependant, il n'existe à ce jour

aucun solveur CHR de contraintes du premier ordre. Nous présentons dans cette section le premier solveur CHR de contraintes du premier ordre. Nous montrons alors comment représenter la structure complexe des formules de travail en CHR et comment gérer les changements de phases de notre algorithme.

4.1.1 Exécution de CHR

CHR manipule des conjonctions de contraintes qui sont stockées dans une zone appelées *local store*. Soient H, C et B des conjonctions de contraintes. Trois types de règles existe dans la syntaxe de CHR : les règles de *simplification* $H \Leftrightarrow C \mid B$ remplacent la contrainte H par B à condition que la contrainte C soit satisfaite. Les règles de *propagation* $H \Rightarrow C \mid B$ ajoutent la contrainte B à H. Enfin, les règles de *simpagation* $H_1 \backslash H_2 \Leftrightarrow C \mid B$ remplacent dans toute conjonction $H_1 \wedge H_2$ la contrainte H_2 par B si la condition C est satisfaite.

Un programme CHR est donc une suite de règles CHR qui sont appliquées sur une contrainte initiale jusqu'à ce qu'un point fixe soit atteint ou que la contrainte *faux* apparaisse dans le local store. Le traitement de la contrainte se fait de gauche à droite et les règles sont appliquées dans l'ordre dans lequel elles apparaissent dans le programme, c'est ce qu'on appelle *refined semantics of CHR* [13].

4.1.2 Un solveur CHR dans T

Pour représenter la structure complexe des formules de travail nous avons développé deux contraintes : `nf/4` (negated formula) et `of/2` pour les équations et relations. Plus précisément, `nf(ParentId,Id,Phase,ExVars)` représente une structure de formule, son identifiant est `Id`, l'identifiant de son père est `parentId`, sa phase (un entier allant de 0 a 5) est `phase`, et la liste des variables quantifiées est `ExVars`. Une contrainte `Var=FlatTerm of Id` représente une équation entre une variable et un terme plat. Cette équation appartient à une contrainte nf/4 dont l'identifiant est `Id`. De même pour `finite(U) of Id` qui représente une relation *fini*(U) qui appartient à une contrainte nf/4 dont l'identifiant est Id.

Ainsi, représenter une formule de travail φ en CHR s'obtient par une conjonction de contraintes nf/4 et of/2. En effet, il suffit de créer une contrainte nf/4 pour chaque formule de base quantifiée de φ puis utiliser une conjonction de contraintes of/2 pour énumérer les formules atomiques liées à chaque formule de base quantifiée comme le montre l'exemple de la figure 2.

Les règles (1) à (14) de notre solveur (figure 1) se traduisent directement en CHR de la manière suivante :

```
%  Résolution locale des équations
(1) @ nf(Q,P,1,Xs) \ U=U of P <=> true.
(2) @ nf(Q,P,1,Xs) \ V=U of P <=> gt(U,V) | U=V of P.
(3) @ nf(Q,P,1,Xs), U=V of P \ U=G of P <=>
          gt(U,V) | V=G of P.
(4) @ nf(Q,P,1,Xs), U=F of P,  U=G of P <=>
          notsamefunctor(F,G) | P=true.
(5) @ nf(Q,P,1,Xs), U=F of P \ U=G of P <=>
          samefunctor(F,G) |  same_args(F,G,P).

% Passage à la phase 2
(6) @ nf(Q,P,1,Xs) <=> nf(Q,P,2,Xs).

% Résolution des relations finite/2
(7) @ nf(P0,P,2,Xs), finite(U) of P \ finite(U) of P
      <=>  true.
(8) @ nf(P0,P,2,Xs), U=V of P \ finite(U) of P <=>
          var(V) | finite(V) of P.
(9+10)@nf(P0,P,2,Xs),U=T of P \ finite(U) of P <=>
   nonvar(T) | reach_args(U,T,P), finite_args(U,T,P).

% Passage à la phase 3
(11) @ nf(Q,P,2,Xs) <=> nf(Q,P,3,Xs).

% Propagation des formules atomiques
(12a) @nf(Q,P,4,Xs), A of P,nf(P,P1,0,Ys) ==> A of P1.
(12b) @ nf(Q,P,4,Xs)        \ nf(P,P1,0,Ys) <=>
          nf(P,P1,1,Ys).

% Restauration des contraintes
(13a) @nf(Q,P,4,Xs),U=V of P,nf(P,P1,3,Ys)\ U=G of P1
          <=> V\==G | U=V of P1.

% Passage à la phase 4
(13b)@   nf(Q,P,4,Xs) \ nf(P,P1,3,Ys)   <=>
                nf(P,P1,4,Ys).

% Chaque contrainte A de P1 apparait également dans P
(14) @ nf(Q,P,4,Xs), nf(P,P1,5,Ys) <=>
              \+ (find_constraint(P1,(A of P1),_),
              \+ find_constraint(P,(A of P),_) )
              | P=true.
```

Dans les règles (2) et (3), le prédicat `gt(U,V)` est utilisé pour vérifier si `U` $\succ$ `V`. Dans les règles (4) et (5), les prédicats `samefunctor(F,G)` et `notsamefunctor(F,G)` sont utilisés pour vérifier si les termes F et G commencent par le même symbole de fonction. Notons également, que dans la règle (4) nous posons `P=true` qui a pour conséquence de supprimer du local store toutes les contraintes of/2 associées à P en utilisant une règle CHR de la forme `E of true <=> true`. Dans la règle (9+10), les deux règles (9) et (10) de notre algorithme (figure 1) ont été fusionnées. L'appel de `reach_args(U,T,P)` vérifie l'accessibilité de `U` à partir de lui même dans `P`. S'il est accessible alors `P` sera remplacé par `true` sinon l'appel de `finite_args(U,T,P)` propagera la relation `finite` de la variable `U` aux arguments

Figure 2. Représentation d'une formule de travail en CHR par une conjonction de contrainte of/2 et nf/4.

$$\neg^4 \left[\begin{array}{l} \exists u\, u = 1 \wedge \\ \left[\begin{array}{l} \neg^0(\exists \varepsilon\, u = s(v)) \wedge \\ \neg^0(\exists w_1\, u = s(w_1) \wedge w_1 = s(v)) \wedge \\ \neg^5(\exists \varepsilon\, v = s(u) \wedge u = 1 \wedge \left[\begin{array}{l} \neg^5(\exists \varepsilon\, v = s(u) \wedge u = 1 \wedge \mathit{fini}(w_1)) \wedge \\ \neg^5(\exists w_3\, v = s(u) \wedge u = 1 \wedge w_2 = s(w_3) \wedge \mathit{fini}(w_3)) \end{array} \right]) \end{array} \right] \end{array} \right].$$

La formule de travail précédente peut être représentée en CHR de la manière suivante :

$$\begin{array}{l} \texttt{nf}(\texttt{Q}, \texttt{P1}, 4, [\texttt{U}]), \texttt{U} = 1\ \texttt{of}\ \texttt{P1}, \\ \texttt{nf}(\texttt{P1}, \texttt{P2}, 0, [\,]), \texttt{U} = \texttt{S}(\texttt{V})\ \texttt{of}\ \texttt{P2}, \\ \texttt{nf}(\texttt{P1}, \texttt{P3}, 0, [\texttt{W1}]), \texttt{U} = \texttt{S}(\texttt{W1})\ \texttt{of}\ \texttt{P3}, \texttt{W1} = \texttt{S}(\texttt{V})\ \texttt{of}\ \texttt{P3}, \\ \texttt{nf}(\texttt{P1}, \texttt{P4}, 5, [\,]), \texttt{V} = \texttt{S}(\texttt{U})\ \texttt{of}\ \texttt{P4}, \texttt{U} = 1\ \texttt{of}\ \texttt{P4} \\ \texttt{nf}(\texttt{P4}, \texttt{P5}, 5, [\,]), \texttt{V} = \texttt{S}(\texttt{U})\ \texttt{of}\ \texttt{P5}, \texttt{U} = 1\ \texttt{of}\ \texttt{P5}, \texttt{finite}(\texttt{W1})\ \texttt{of}\ \texttt{P5} \\ \texttt{nf}(\texttt{P4}, \texttt{P6}, 5, [\texttt{W3}]), \texttt{V} = \texttt{S}(\texttt{U})\ \texttt{of}\ \texttt{P6}, \texttt{U} = 1\ \texttt{of}\ \texttt{P6}, \texttt{W2} = \texttt{S}(\texttt{W3})\ \texttt{of}\ \texttt{P6}, \texttt{finite}(\texttt{W3})\ \texttt{of}\ \texttt{P6} \end{array}$$

de `T`. Dans la règle (14), on vérifie si toutes les contraintes de P1 apparaissent également dans P. Pour cela, nous avons utilisé la contrainte CHR prédéfinie `find_constraint(Var,Pattern,Match)` qui renvoi toutes les contraintes `Match` qui correspondent avec `Pattern` et qui sont indexées par la variable `Var` ainsi que la fameuse negation-as-failure \+.

Les règles (15) et (16) de notre algorithme (figure 1) nécessitent une traduction en CHR plus longue car elles effectuent plusieurs traitements à la suite : élimination de quantificateurs, calcul d'accessibilité, renommage de variables, diminution de profondeur de formules de travail,...etc. Pour cela 40 règles CHR ont été utilisées et chaque traitement a été traduit en un ensemble de règles CHR. Par manque de place nous ne pouvant présenter ces règles.

5 Conclusion

Nous avons présenté dans ce papier deux contributions :

(1) Une extension au premier ordre du mécanisme de l'unification des termes qui est le le coeur même du langage Prolog. Pour cela, nous avons tout d'abord étendu le pouvoir d'expression de la théorie des arbres finis ou infinis en introduisant le symbole de relation *fini* qui permet de contraindre un terme à être un arbre fini. Nous avons ensuite proposé une axiomatisation au premier ordre de notre théorie étendue et avons montré sa complétude par un solveur de contraintes du premier ordre sous forme de 16 règles de réécriture. Face à la complexité en tour de puissances de 2 qui caractérise tout algorithme de résolution de contraintes du premier ordre dans des arbres finis ou infinis [29], nous avons utilisé deux stratégies : (i) propagation de contraintes et résolution locale en descendant le long de la structure des formules de travail, (ii) élimination de quantificateurs et distribution en remontant la structure des formules de travail. Ces deux stratégies nous évitent entre autres de résoudre des sous-formules qui contredisent leur sur ou sous formules. L'implémentation C++ nous a permis de résoudre des formules de positions gagnantes de jeux complexes à deux joueurs en résolvant entre autres des formules du premier ordre contenant plus 80 quantificateurs et négations imbriqués [12].

(2) Le tout premier solveur CHR de contraintes du premier ordre. Il s'agit d'un solveur de contraintes dans la théorie T implémenté en utilisant 73 règles CHR. Nous avons alors montré comment représenter la structure complexe des formules de travail en CHR et comment gérer les changements de phases de notre algorithme en utilisant notamment les propriétés élégantes de la sémantique raffinée de CHR [13]. Les premiers résultats obtenus en terme de performances [12] sont certes moins bons que ceux obtenus en utilisant C++ mais nous encouragent à optimiser notre implémentation en utilisant notamment des techniques évolués pour la gestion de l'espace mémoire en CHR, chose qui est vitale lors de la résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis. On prévoit également d'utiliser l'algorithme CHR quadratique de résolution de contraintes quantifiées dans la théorie des arbres finis ou infinis [21] tout en parallélisant certains traitements de notre algorithme en s'inspirant notamment de la récente prallélisation du fameux algorithme de *Union-Find* [18] en CHR.

Références

[1] S. Abdennadher a,d M. Marte. 2000. University timetabling using CHR. Journal of applied artificial

intelligence. 14(4) : 311-325.

[2] M. Alberti, M. Gavanelli, E. Lamma, P. Mello and P. Torroni. 2003. Specification and verification of agent interaction using social integrity contraints. ENTCS, 85(2) : 150-177.

[3] Benhamou, F., Colmerauer, A., Garetta, H., Pasero, R. and Van-caneghem, M. 1996. Le manuel de Prolog IV. PrologIA, Marseille, France.

[4] Colmerauer, A. 1982. Prolog and infinite trees. In K.L. Clark and S-A. Tarnlund, editors, Logic Programming. Academic Press. pp. 231–251.

[5] Colmerauer, A., Kanoui, H. and Van-caneghem,M. 1983. Prolog, Theoretical Basis and Current Developments. TSI 2(4) :271–311.

[6] Colmerauer, A. 1984. Equations and inequations on finite and infinite trees. Proceeding of the International conference on the fifth generation of computer systems, pp. 85–99.

[7] Colmerauer, A. 1990. An introduction to Prolog III. *Communication of the ACM*, 33(7) :68–90.

[8] Colmerauer, A. and Dao, T. 2003. Expressiveness of full first-order formulas in the algebra of finite or infinite trees, Constraints, 8(3) : 283–302.

[9] Comon, H. 1991a. Disunification : a survey. In J.L. Lassez and G. Plotkin, editors, Computational Logic : Essays in Honor of Alan Robinson. MIT Press.

[10] Dao, T. 2000. Resolution de contraintes du premier ordre dans la theorie des arbres finis ou infinis. These d'informatique, Universite de la mediterranee, France.

[11] Djelloul, K. 2006. Decomposable theories. Journal of Theory and practice of logic programming TPLP. (à paraître)

[12] Djelloul, K., Dao, T.B.H. and Fruehwirth, T. 2006. Theory of finite or infintie trees revisited. Journal of Theory and practice of logic programming TPLP. (à paraître). Disponible en ligne à l'adresse suivante `http://khalil.djelloul.free.fr/article/tplp2.pdf`

[13] Duck, G., Stuckey, P., Banda, M. and Holzbaur, C. 2004. The Refined Operational Semantics of Constraint Handling Rules. In Proc of the 20th International Conference on Logic Programming. LNCS 3132, pp. 105-119.

[14] M. Escrig and F. Toledo. 2000. Autonomous robot navigation using human spatial concepts. International journal of intelligent systems. 15(3) : 165-196.

[15] Fruehwirth, T. 1998. Theory and Practice of Constraint Handling Rules. Special Issue on Constraint Logic Programming. Journal of Logic Programming. 37(1–3) : 95-138.

[16] T. Fruehwirth and S. Abdennadher. 2001. The munich rent advisor : a success for logic programming on the internet. Journal of theory and practice of logic programming (TPLP), 1(3) : 303-319.

[17] Fruehwirth, T. and Abdennadher, S. 2003. Essentials of Constraint Programming. Springer.

[18] Fruehwirth, T. 2005. Parallelizing Union-Find in Constraint Handling Rules Using Confluence. In proc of the 21st International Conference of Logic Programming. LNCS, Vol 3668. pp : 113-127.

[19] C.H. Goh, S. Bressan, S.E. Madnick and M. Siegel. 1999. Context interchange : new features and formalisms for intelligent integration of information. ACM Trans. Inf. Syst, 17(3) : 270-293.

[20] Maher M. Complete axiomatization of the algebra of finite, rational and infinite trees. *Technical report, IBM*, 1988.

[21] Meister, M., Djelloul, K. and Fruehwirth, T. Complexity of CHR solveur for existentially quantified conjunctions of equations over trees. Recent Advances in Constraints. LNAI (à paraître)

[22] A. Pretschner. 2001. Classical search strategies for test case generation with CLP. In proceedings of formal approaches to testing of software. p : 47-60.

[23] Robinson, J.A. 1965. A machine-oriented logic based on the resolution principle. JACM, 12(1) :23–41.

[24] Roussel, F. 1975. Prolog : Manuel de Reference et d'Utilisation, Groupe d'Intelligence Artificielle, Marseille-Luminy.

[25] M. Thielscher. 2005. FLUX : A logic programming method for reasoning agents. Journal of theory and practice of logic programming (TPLP), 5(4-5) 533-565.

[26] Van Weert, P., Sneyers, J., Schrijvers, T. and Demoen, B. 2006. Constraint Handling Rules with Negations as Absence. In Proc of the third Workshop on Constraint Handling Rules.

[27] Schrijvers, T., Demoen, B., Duck, G., Stuckey, P. and Fruehwirth, T. 2006. Automatic implication checking for CHR constraints. In Proc of the 6th International Workshop on Rule-Based Programming. ENTC, vol 147, pp. 93-111.

[28] Schrijvers, T. and Fruehwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

[29] Vorobyov S. An Improved Lower Bound for Elementary Theories of Trees, Proceeding of the 13th Conference on Automated Deduction. LNAI, vol 1104, pp. 275-287.