Un système de types pour CHR

Emmanuel Coquery, Francois Fages

Un système de types pour CHR

Emmanuel Coquery[1,2] **François Fages**[1]

[1] Projet Contraintes, INRIA Rocquencourt, BP 105, F-78153 Le Chesnay Cedex
[2] Conservatoire National des Arts et Métiers, 292 rue St Martin, 75141 Paris cedex 03
Emmanuel.Coquery@inria.fr Francois.Fages@inria.fr

Résumé

Nous proposons un système de types général pour le langage des *Constraint Handling Rules* (CHRs), un langage de règles de réécriture destiné à l'implantation de solveurs de contraintes. Les CHRs étant en fait une extension de haut niveau d'un langage hôte, tel que Prolog ou Java, le système de types est ainsi paramétré par le système de types du langage hôte. Nous montrons la cohérence du système de types pour les CHRs par rapport à leur sémantique opérationnelle. Nous étudions également le cas particulier où le langage hôte est un langage de programmation en logique avec contraintes, typé avec le système de types prescriptif que nous avons développé dans nos précédents travaux. En particulier nous montrons que le système résultant est cohérent avec le modèle d'exécution étendu CLP+CHR. Ce système est implanté à travers une extension du logiciel TCLP de typage des programmes logiques avec contraintes. Nous exposons nos résultats expérimentaux sur la vérification des types de solveurs et programmes utilisant les CHRs, dont le logiciel TCLP lui-même.

Abstract

We propose a generic type system for the *Constraint Handling Rules* language (CHRs), a rewriting rule language for implementing constraint solvers. The CHRs being a high-level extension of a host language, such as Prolog or Java, this type system is parametrized by the type system of the host language. We show the consistency of the type system for the CHRs w.r.t. their operational semantics. we also study the particular case where the host language is a constraint logic programming language, typed with the prescriptive type system we developped in our previous work. In particular, we show the consistency of the resulting type system w.r.t. the extended execution model CLP+CHR. This system is implemented through an extension of our type checker TCLP for constraint logic languages. We also present some experimental results on the type-checking of solvers and programs using the CHRs, including TCLP itself.

1 Introduction

Le langage des *Constraint Handling Rules* (en abrégé CHR) de Frühwirth [8] est un langage de règles de réécriture destiné à l'implantation de solveurs de contraintes. Il se présente sous la forme d'une extension d'un langage hôte, tel que Prolog [10] ou Java [2], permettant d'introduire de nouvelles contraintes de manière déclarative dans celui-ci. Ainsi les CHRs sont utilisés pour gérer les contraintes définies par l'utilisateur, le langage hôte se chargeant des autres calculs, par exemple à l'aide de contraintes *natives*. Par la suite nous supposons l'existence de la contrainte native $= /2$. Les CHRs sont un langage de règles gardées qui réécrivent des multi-ensembles de contraintes en les simplifiant, et ceci jusqu'à leur résolution. Ainsi, une des particularités des CHRs est qu'ils autorisent plusieurs contraintes dans la tête d'une règle.

Les langages typés possèdent de nombreux avantages en termes de développement, tels que la détection statique d'erreurs de programmation ou de composition de programmes, et la documentation du code en utilisant les types. Les CHRs ont déjà été utilisés dans le cadre du typage des langages de programmation, que ce soit pour la résolution des contraintes de sous-typage [6, 4] ou pour la gestion de la surcharge dans le cadre des langages fonctionnels [16] et des langages avec contraintes [5, 4]. Cependant, il n'existe, à notre connaissance, aucun système de types pour les CHRs.

Dans cet article, nous comblons ce vide en proposant un système de types pour les CHRs, inspiré du système de types TCLP pour les programmes logiques avec contraintes [7]. Les CHRs étant une extension d'un langage hôte, ce système est paramétré par le système de types du langage hôte. Nous ferons trois hypothèses sur le système de types du langage hôte.

– Il permet de dériver des jugements de typage de la

forme $\Gamma \vdash t : \tau$ qui associe le type τ au terme (ou à l'expression) t dans l'environnement (ou, plus généralement sous l'hypothèse) Γ. Il définit également une notion de contrainte *bien typée* dans un environnement Γ.
- La contrainte $t_1 = t_2$ est bien typée dans un environnement Γ s'il existe un type τ tel que $\Gamma \vdash t_1 : \tau$ et $\Gamma \vdash t_2 : \tau$.
- Si une conjonction c de contraintes natives est bien typée dans un environnement Γ et est équivalente à une conjonction d, alors d est également bien typée dans Γ.

En utilisant ces hypothèses, nous exprimons la cohérence du système de types pour les CHRs par rapport à leur sémantique opérationnelle à travers un théorème d'auto-réduction qui exprime que si un programme est bien typé alors toutes les dérivations issues d'un but bien typé sont bien typées.

Nous étudions également le cas de l'instanciation du système de types pour les CHRs avec le système de types TCLP pour les programmes logiques avec contraintes [7], et montrons un théorème d'auto-réduction pour le système de types résultant par rapport au modèle d'exécution CLP+CHR [8] dans lequel il est possible d'étendre la définition des contraintes CHRs par des clauses. Ce résultat est intéressant car la programmation logique avec contraintes constitue un cadre naturel pour l'utilisation des solveurs de contraintes. Un système de type pour CLP+CHR permet de typer un solveur avec le programme qui l'utilise. De plus il arrive fréquemment que les solveurs CHR complexes soient écrits comme une combinaison de clauses et de règles CHR, les règles faisant appel à des prédicats et les clauses à des contraintes CHR.

Le reste de l'article est organisé comme suit. La section 2 rappelle la syntaxe et la sémantique opérationnelle des CHRs, y compris le modèle CLP+CHR. La section 3 présente le système de types que nous proposons et la section 4 son instanciation avec le système de types pour CLP. La section 5 présente quelques résultats expérimentaux sur le typage de solveurs écrits en CHR à partir de l'implantation du système dans le logiciel TCLP [3]. Enfin, nous concluons avec la section 6.

2 Préliminaires sur les CHRs

Nous rappelons ici la syntaxe et la sémantique des CHRs, issues de [8]. Nous distinguons les *contraintes CHR*, définies par le programmeur, des *contraintes natives* du langage hôte, qui représentent les calculs auxiliaires ayant lieu lors de l'application d'une règle CHR. Ces dernières sont gérées par un solveur prédéfini du langage hôte. Nous supposons que la contrainte d'égalité $= /2$ et la contrainte *true* font partie de ces contraintes natives. On note $\mathcal{X}$ le domaine des contraintes natives et $\mathcal{CT}$ la théorie, éventuellement incomplète, qui correspond au solveur de contraintes natives. On notera $s, t, \ldots$ les expressions du langage hôte, $\mathcal{S}_{CHR}$ l'ensemble des symboles de contraintes CHR, données avec leur arité, $\mathcal{S}_N$ l'ensemble des symboles de contraintes natives, également données avec leur arité.

2.1 Syntaxe

Définition 1 *Une règle CHR peut être :*
- *une règle de* simplification *de la forme :* $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
- *une règle de* propagation *de la forme :* $H_1, \ldots, H_i$ `==>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
- *ou bien une règle de* simpagation *de la forme :* $H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$

avec $i > 0$, $j \geq 0$, $k \geq 0$, $l > 0$ et $H_1, \ldots, H_i$ est une suite non vide de contraintes CHR, la garde $G_1, \ldots, G_j$ est une suite de contraintes native et le corps $B_1, \ldots, B_k$ est une suite de contraintes CHR et de contraintes natives.

Un programme CHR est une suite finie de règles CHR.

La contrainte *true* est utilisée pour représenter les séquences vides. La garde vide peut être omise, avec l'opérateur |. La notation $nom@R$ donne le nom *nom* à la règle CHR R.

Informellement, la règle de simplification remplace les contraintes de la tête par celles du corps. La règle de propagation ajoute les contraintes du corps en conservant celles de la tête. La règle de simpagation est un mélange des deux précédentes : les contraintes $H_{l+1}, \ldots, H_i$ sont remplacées par le corps, les contraintes $H_1, \ldots, H_l$ étant conservées.

Par souci de simplicité, et parce que le typage des programmes CHR ne nécessite pas de distinguer les règles de simpagation, nous considérerons qu'une règle de simpagation de la forme $H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$ est une abréviation pour la règle $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid H_1, \ldots, H_l, B_1, \ldots, B_k$. De même, les règles de propagation pourraient être réécrites sous forme de règles de simplification, mais nous les conservons ici par souci de clarté.

Exemple 1 *Le programme CHR suivant, tiré de [8], définit un solveur pour la contrainte `=<` "plus petit ou égal", pouvant gérer des arguments qui sont des variables :*

```
reflexivite @ X=<Y <=> X=Y | true.
```

```
antisymetrie @ X=<Y , Y=<X   <=> X=Y.
transitivite @ X=<Y, Y=<Z ==> X=<Z.
ident @ X=<Y \ X=<Y <=> true.
```

La règle `reflexivite` *élimine les contraintes* `=<` *si les deux arguments sont égaux. La règle* `antisymetrie` *simplifie une double inégalité en une contrainte d'égalité. La règle* `transitivite` *permet d'ajouter les contraintes correspondant à la clôture transitive de* `=<`*. Enfin, la règle* `ident` *élimine les contraintes* `=<` *en double.*

2.2 Sémantique opérationnelle

La sémantique opérationnelle des CHRs s'exprime sous forme d'un système de transitions, noté $\longmapsto$, sur des états qui sont des triplets $\langle F, E, D\rangle$, où F est un but, c'est-à-dire un multi-ensemble de contraintes CHR et de contraintes natives, E est une mémoire de contraintes CHR et D est une mémoire de contraintes natives, c'est-à-dire des multi-ensemble de contraintes (respectivement CHR et natives). Un état est donc une conjonction de contraintes CHR et de contraintes natives.

Dans la définition suivante, l'égalité est étendue aux contraintes par morphisme, c'est-à-dire $c(t_1, \ldots, t_n) = c(t'_1, \ldots, t'_n)$ si pour tout $i \in \{1, \ldots, n\}$, $t_i = t'_i$. La notation $\wedge$ est utilisée pour exprimer le filtrage de contraintes dans un multi-ensemble. L'égalité est alors étendue aux conjonctions de contraintes de la manière suivante : si H et H' sont deux conjonctions de contraintes, alors $H = H'$ s'il existe deux suites de contraintes $H_1, \ldots, H_n$ et $H'_1, \ldots, H'_n$ correspondant respectivement à H et H', et, telles que pour tout $i \in \{1, \ldots, n\}$, $H_i = H'_i$.

Définition 2 *Soit P un programme CHR. La relation de transition $\longmapsto$ pour les CHRs est donnée par les règles suivantes. Les majuscules apparaissant dans les triplets représentent des conjonctions de contraintes et $\bar{x}$ représente les variables apparaissant dans la tête H.*

Solve
$\langle C \wedge F, E, D\rangle \longmapsto \langle F, E, D'\rangle$
si C est une contrainte native
et si $\mathcal{CT} \models (C \wedge D) \Leftrightarrow D'$.

Introduce
$\langle H \wedge F, E, D\rangle \longmapsto \langle F, H \wedge E, D\rangle$
si H est une contrainte CHR.

Simplify
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
si (H `<=>` $G \mid B$) est dans P renommé avec des variables fraîches,
et si $\mathcal{CT} \models D \Rightarrow \exists \bar{x}(H = H' \wedge G)$.

Propagate
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D\rangle$
si (H `==>` $G \mid B$) est dans P renommé avec des variables fraîches,
et si $\mathcal{CT} \models D \Rightarrow \exists \bar{x}(H = H' \wedge G)$.

La transition **Solve** correspond à une transition effectuée par le solveur de contraintes natives. La transition **Introduce** exprime simplement le déplacement d'une contrainte CHR du but vers la mémoire de contraintes CHR où elle pourra être utilisée avec d'autres contraintes pour l'application d'une règle. Les règles **Simplify** et **Propagate** correspondent, respectivement, à l'application des règles CHR de simplification et de propagation. La condition d'application des règles exprime que la tête de la règle doit être instanciée de façon à ce que la garde et la condition de filtrage de la tête soient impliquées par la mémoire courante de contraintes natives. Le corps de la règle est alors ajouté au but courant et les contraintes H' sont retirées de la mémoire de contraintes lors de l'application de la règle **Simplify**.

Définition 3 *Un* état initial *consiste en un but F et des mémoires de contraintes vides : $\langle F, true, true\rangle$. Un* état final *est soit de la forme $\langle F, E, false\rangle$ (*échec*), soit de la forme $\langle true, E, D\rangle$ (*succès*).*

L'exemple suivant montre l'exécution d'un programme CHR.

Exemple 2 *Considérons le solveur donné dans l'exemple 1 et l'état initial* $\langle$`X=<Y` $\wedge$ `Y=<Z` $\wedge$ `Z=<X`, *true*, *true*$\rangle$*. Une exécution possible est :*

$\langle$`Z=<X`, `X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ ***(Introduce*** $\times 2$***)***
$\langle$`X=<Z` $\wedge$ `Z=<X`, `X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ ***(Propagate*** `trans.`***)***
$\langle$*true*, `X=<Z`$\wedge$`Z=<X`$\wedge$`X=<Y`$\wedge$`Y=<Z`, *true*$\rangle$ ***(Introduce*** $\times 2$***)***
$\langle$`X=Z`, `X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ ***(Simplify*** `anti.`***)***
$\langle$*true*, `X=<Y` $\wedge$ `Y=<Z`, `X=Z`$\rangle$ ***(Solve)***
$\langle$`X=Y`, *true*, `X=Z`$\rangle$ ***(Simplify*** `anti.`***)***
$\langle$*true*, *true*, `X=Y` $\wedge$ `X=Z`$\rangle$ ***(Solve)***

On peut remarquer que la sémantique donnée ci-dessus n'empêche pas l'application répétée d'une règle de propagation sur les mêmes contraintes, provoquant de manière triviale la non terminaison du programme CHR. Dans l'exemple précédent, on aurait ainsi pu appliquer la règle `transitivite` au lieu de la règle `antisymetrie`, réintroduisant ainsi la contrainte `X=<Z` éliminée à la quatrième étape. Dans [1], Abdennadher donne une sémantique opérationnelle plus fidèle à l'implantation des CHRs permettant d'éviter ce comportement, en restreignant l'ensemble des dérivations possibles. Les théorèmes d'auto-réduction donnés dans les sections suivantes, qui expriment que dans un programme bien typé une transition à partir d'un état bien typé conduit à un état bien typé, restent ainsi valables pour cette sémantique plus réaliste.

2.3 CLP+CHR

Lorsque le langage hôte est un langage de la classe CLP($\mathcal{X}$) [11] des langages langages logiques avec contraintes sur un domaine de contraintes $\mathcal{X}$, il est possible d'intégrer plus fortement les CHRs au langage. Frühwirth [8] propose ainsi d'étendre le langage des CHRs avec une construction *label_with* et la possibilité d'introduire des clauses pour les contraintes CHR. Nous rappelons ici la syntaxe et la sémantique opérationnelle de cette extension. Un *atome* est soit une contrainte native, soit une contrainte CHR, soit de la forme $p(t_1, \ldots, t_n)$ où p/n est un symbole de prédicat. On note $\mathcal{S}_F$ (resp. $\mathcal{S}_P$) l'ensemble des symboles de fonction (resp. prédicats), donnés avec leur arité, et $\mathcal{V}$ l'ensemble des variables.

Définition 4 *Une* déclaration d'énumération *pour une contrainte CHR H est de la forme :*

$label\_with\ H\ if\ G_1, \ldots, G_j$

où $G_1 \ldots, G_j$ est une conjonction de contraintes natives.

De manière usuelle, les clauses sont de la forme :

H :- $B_1, \ldots, B_n$

où H est soit un atome correspondant à un prédicat, soit une contrainte CHR, mais pas une contrainte native, et $B_1, \ldots, B_k$ est une suite d'atomes.

La déclaration *label_with* $c(t_1, \ldots, t_n)$ *if* $G_1, \ldots, G_j$ exprime que $G_1, \ldots, G_j$ est une garde pour les clauses de la contrainte CHR c/n.

Définition 5 *La relation de transition entre les états CHRs est étendue par les deux règles suivantes :*

Unfold

$\langle H' \wedge F, E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
si (H :- B) est dans P renommé avec des variables fraîches,
et si H n'est pas une contrainte CHR.

Label

$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
si (H :- B) et (label_with H'' if G) sont dans P renommé avec des variables fraîches,
et si $\mathcal{X} \models \exists \bar{x}(H' = H'' \wedge G)$

La règle **Unfold** est proche de la règle de résolution CSLD [11]. La différence est que, dans le cadre de la résolution CSLD, les contraintes C présentes dans le corps de la clause sont également ajoutées à la mémoire de contraintes natives et que la mémoire résultante, c'est-à-dire $H = H' \wedge D \wedge C$, doit être satisfiable dans domaine des contraintes $\mathcal{X}$. Les clauses pour les contraintes CHR ne peuvent être utilisées que par application de la règle **Label**, qui nécessite que l'une des gardes déclarées avec *label_with* pour la contrainte considérée soit impliquée par la mémoire de contraintes natives courante.

3 Système de types

3.1 Hypothèses sur le système de types du langage hôte

Comme les CHRs sont une extension d'un langage hôte, le système de types que nous proposons pour les CHRs est paramétré par celui du langage hôte, noté $\vdash_N$. Nous supposerons que $\vdash_N$ vérifie les hypothèses qui suivent.

On considère que $\vdash_N$ est basé sur une algèbre de types, l'ensemble des types étant noté $\mathcal{T}$. On notera les types avec la lettre τ. Un environnement de typage, noté Γ, associe un type aux variables.

Étant donné une expression t, on suppose que $\vdash_N$ permet de déduire des jugements de la forme $\Gamma \vdash_N t : \tau$. De manière similaire, nous supposons également que $\vdash_N$ permet de définir une notion de contrainte *bien typée* dans un environnement Γ, une conjonction de contraintes natives $C_1 \wedge \ldots \wedge C_n$ étant bien typée dans un environnement Γ si pour tout $i \in \{1, \ldots, n\}$, C_i est bien typé dans Γ. On notera $\Gamma \vdash_N C\ Atom$, le fait que la contrainte C est bien typée dans l'environnement Γ. Nous supposons également que la contrainte d'égalité $s = t$ entre s et t est bien typée dans Γ s'il existe un type τ tel que $\Gamma \vdash_N s : \tau$ et $\Gamma \vdash_N t : \tau$.

Nous supposons que l'union d'environnements de typage sur des ensembles de variables disjoints est définie par une opération, notée $\uplus$, telle que si $\Gamma \vdash_N t : \tau$, alors $\Gamma \uplus \Gamma' \vdash_N t : \tau$. Dans de nombreux systèmes de types, Γ est une fonction associant des types aux variables apparaissant dans t et vérifie donc cette hypothèse. Cependant, nous n'avons pas besoin de considérer la forme exacte des environnements de typage, qui peuvent donc prendre des formes plus complexes sans que cela n'ait de conséquences sur le système de types pour CHR.

Enfin, nous supposons que si une conjonction de contraintes natives C est bien typée dans un environnement Γ et si $\mathcal{CT} \models C \Leftrightarrow D$, alors il existe un environnement de typage Γ', tel que la conjonction de contraintes D est bien typée dans $\Gamma \uplus \Gamma'$.

3.2 Système de types pour les CHRs

Le système de types que nous proposons pour CHR permet de définir une notion de contraintes et de règles CHR bien typées. On associe à chaque symbole de contrainte CHR c/n un, et un seul, type de la forme $\tau_1 \times \ldots \times \tau_n$. Ce type est supposé fixé, par

$$(\textit{Native}) \quad \frac{\Gamma \vdash_N C\ \textit{Atom}}{\Gamma \vdash C\ \textit{Atom}} \qquad \text{si } C \text{ est une contrainte native}$$

$$(\textit{CHR Atom}) \quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ \textit{Atom}} \qquad \begin{array}{c}\text{si } c/n \text{ est une contrainte CHR} \\ \text{et si } c/n \text{ a le type } \tau_1 \times \ldots \times \tau_n\end{array}$$

$$(\textit{Goal}) \quad \frac{\Gamma \vdash B_1\ \textit{Atom} \quad \ldots \quad \Gamma \vdash B_n\ \textit{Atom}}{\Gamma \vdash B_1, \ldots, B_n\ \textit{Goal}}$$

$$(\textit{Prop CHR}) \quad \frac{\Gamma \vdash H_1, \ldots, H_i\ \textit{Goal} \quad \Gamma \vdash G_1, \ldots, G_j\ \textit{Goal} \quad \Gamma \vdash B_1, \ldots, B_k\ \textit{Goal}}{\vdash H_1, \ldots, H_i\ \texttt{==>}\ G_1, \ldots, G_j \mid B_1, \ldots, B_k\ \textit{Rule}}$$

$$(\textit{Simpl CHR}) \quad \frac{\Gamma \vdash H_1, \ldots, H_i\ \textit{Goal} \quad \Gamma \vdash G_1, \ldots, G_j\ \textit{Goal} \quad \Gamma \vdash B_1, \ldots, B_k\ \textit{Goal}}{\vdash H_1, \ldots, H_i\ \texttt{<=>}\ G_1, \ldots, G_j \mid B_1, \ldots, B_k\ \textit{Rule}}$$

TAB. 1 – Système de types pour CHR

exemple à l'aide de déclarations fournies par le programmeur. On peut remarquer que la limitation à un seul type pour chaque symbole de contrainte interdit l'utilisation de schéma de types, comme ceux que l'on peut trouver dans les systèmes utilisant le polymorphisme paramétrique. Par exemple, il est interdit de donner à une contrainte CHR `append`/3 le type $\forall\alpha.list(\alpha)\times list(\alpha)\times list(\alpha))$, car cela revient à lui donner tous les types de la forme $list(\tau)\times list(\tau)\times list(\tau)$. Cette restriction est due à la possibilité d'avoir plusieurs contraintes dans la tête d'une règle CHR. Si les contraintes CHR peuvent avoir plusieurs types, il faut pouvoir corréler les types choisis pour chaque contrainte dans la tête d'une règle. De plus rien ne garantit à priori que les types des contraintes considérées lors l'application de la règle CHR soient corrélés de manière similaire, comme le montre l'exemple 3.

Les règles du système de types pour CHR sont données dans la table 1. Une contrainte CHR H est *bien typée* dans Γ si on peut dériver le jugement $\Gamma \vdash H\ \textit{Atom}$. Les termes ou expressions qui apparaissent en tant qu'arguments de la contrainte sont typés avec le système de types $\vdash_N$ du langage hôte. Les règles (*Prop CHR*) et (*Simpl CHR*) expriment simplement qu'une règle CHR est *bien typée* si sa tête, sa garde et son corps sont bien typés dans un même environnement Γ. Un programme CHR est bien typé si toutes ses règles sont bien typées.

Le lemme suivant exprime simplement qu'un but bien typé dans un environnement Γ l'est également dans un environnement obtenu à partir de Γ par application de $\uplus$.

Lemme 1 *Soit un but B et un environnement de typage Γ tels que $\Gamma \vdash B\ \textit{Goal}$. Soit Γ' un environnement de typage tel que $\Gamma \uplus \Gamma'$ est défini. Alors $\Gamma \uplus \Gamma' \vdash B\ \textit{Goal}$.*

Preuve. Par induction et en utilisant l'hypothèse que si $\Gamma \vdash_N t : \tau$ alors $\Gamma \uplus \Gamma' \vdash_N t : \tau$. □

La cohérence du système de types par rapport à la sémantique opérationnelle des CHRs est donnée par le théorème d'auto-réduction suivant, qui exprime que si on considère un programme et un état $\langle F, E, D\rangle$ bien typés, et si $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$, alors $\langle F', E', D'\rangle$ est bien typé.

Théorème 1 *Soit un programme CHR bien typé P. Soient deux états $\langle F, E, D\rangle$ et $\langle F', E', D'\rangle$ tels que $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$. S'il existe un environnement Γ tel que $\Gamma \vdash F, E, D\ \textit{Goal}$ alors il existe un environnement Γ' tel que $\Gamma' \vdash F', E', D'\ \textit{Goal}$. De plus, dans le cas des transitions **Simplify** et **Propagate**, si la règle CHR contient une garde G, alors $\Gamma' \vdash G\ \textit{Goal}$.*

Preuve. Par cas sur la règle de transition utilisée :

Solve Par hypothèse, $\Gamma \vdash D\ \textit{Goal}$ et $\Gamma \vdash C\ \textit{Atom}$. Comme $\mathcal{CT} \models (C \wedge D) \Leftrightarrow D'$, et par hypothèse sur $\vdash_N$, il existe un environnement Γ'' tel que $\Gamma \uplus \Gamma'' \vdash_N D'\ \textit{Goal}$. En posant $\Gamma' = \Gamma \uplus \Gamma''$ et par le lemme 1, on obtient $\Gamma' \vdash F, E\ \textit{Goal}$, d'où $\Gamma' \vdash F, E, D'\ \textit{Goal}$.

Introduce Cette règle se contente de faire migrer une contrainte du but vers la mémoire de contraintes CHR, donc l'état résultant est bien typé dans Γ.

Simplify Par hypothèse, la règle (H `<=>` $G \mid B$) est bien typée. Il existe donc un environnement Γ'' tel que $\Gamma'' \vdash H, G, B\ \textit{Goal}$. Par le lemme 1, et en posant $\Gamma' = \Gamma \uplus \Gamma''$, on obtient $\Gamma' \vdash F, E, D\ \textit{Goal}$ et $\Gamma' \vdash G, B\ \textit{Goal}$. Il reste à montrer que $H = H'$ est bien typé dans Γ'. H est de la forme $c_1(t_1^1, \ldots, t_{m_1}^1), \ldots, c_i(t_1^i, \ldots, t_{m_i}^i)$ et H'

de la forme $c_1(s^1_1, \ldots, s^1_{m_1}), \ldots, c_i(s^i_1, \ldots, s^i_{m_i})$. Soit $l \in \{1, \ldots, i\}$ et $p \in \{1, \ldots, m_l\}$. Soit $\tau_1 \times \ldots \times \tau_{m_l})$ le type de c_l. Comme $\Gamma' \vdash c_l(t^1_l, \ldots, t^l_{m_l})$ *Atom*, on a $\Gamma' \vdash_N t^l_p : \tau_p$. De même, $\Gamma' \vdash_N s^l_p : \tau_p$. D'où $\Gamma' \vdash t^l_p = s^l_p$ *Atom*. On obtient donc $\Gamma' \vdash B, F, E, H = H', D$ *Goal* et $\Gamma' \vdash H = H', G$ *Goal*.

Propagate La preuve est similaire au cas **Simplify**. □

L'exemple suivant montre que l'utilisation d'un schéma de types pour une contrainte CHR peut amener à tester une conjonction de contraintes mal typée.

Exemple 3 *Supposons que la contrainte* `=<` *aie le schéma de types* $\forall\alpha.\alpha \times \alpha$. *Supposons que* `"a"` *et* `"b"` *aient le type string et* `1` *et* `2` *aient le type int et que ces deux types soient incompatibles. Considérons la règle :*

```
transitivite @ X=<Y, Y=<Z ==> X=<Z.
```

et l'état $\langle$`true, "a"=<"b" ∧ 1=<2`, *true*$\rangle$. *Cet état est bien typé. Si on essaie d'appliquer la règle* `transitivite` *avec* ***Propagate****, il faut tester la contrainte* `X="a"∧Y="b"∧Y=1∧Z=2`, *qui est mal typée car* `Y` *doit avoir à la fois le type string et le type int.*

4 Intégration avec CLP

Dans cette section, nous nous intéressons au cas particulier où le langage hôte est un langage logique avec contraintes, typé avec le système de types prescriptifs TCLP [7]. Ce système combine polymorphisme paramétrique, sous-typage et surcharge afin d'obtenir la souplesse nécessaire au typage de langages CLP non typés au départ. En particulier, le sous-typage est utilisé pour typer l'utilisation conjointe de plusieurs domaines de contraintes : la relation *boolean* $\leq$ *int* permet de voir les booléens comme des entiers et par la même de typer des programmes combinant des contraintes booléennes et des contraintes de domaines finis. Le sous-typage est également utilisé pour typer les programmes faisant appel à la méta-programmation : la relation $list(\alpha) \leq term$ permet de voir les listes homogènes comme des termes et de leur appliquer des prédicats de décomposition, tels que `functor`/3, `arg`/3 ou `=..`/2.

Le système de types TCLP est prouvé cohérent par rapport au modèle d'exécution CSLD [11] qui est un modèle abstrait procédant par accumulation de contraintes. En particulier ce modèle ne tient pas compte des étapes de résolution qui peuvent être opérées par le solveur de contraintes. Nous considérerons donc par la suite que le solveur de contraintes natives opère uniquement des transformations de la mémoire des contraintes conservant la propriété d'être bien typé, conformément à l'hypothèse faite dans la section 3.1. Ceci peut être obtenu en ayant recours à un modèle d'exécution typé comme nous l'avons proposé dans [7], ou bien, dans le cas de la contrainte d'égalité, par l'utilisation de modes fixant le flot de données [15].

Nous présentons tout d'abord l'algèbre des types utilisée par le système, avant de rappeler les règles de typage pour CLP, ainsi que la règle de typage pour la déclaration d'énumération *label_with*. Le système résultant est prouvé cohérent par rapport au modèle d'exécution CLP+CHR.

4.1 Structure des types

On considère un ordre partiel $(\mathcal{K}, \leq_\mathcal{K})$ de *constructeurs de types*, donnés avec leur arité. L'ensemble $\mathcal{T}$ des types est l'ensemble des termes, finis ou infinis, construits à partir de $\mathcal{K}$.

Relation de sous-typage L'utilisation du sous-typage pour la méta-programmation nécessite de considérer des relations telle que $list(\alpha) \leq term$, ce qui correspond à un sous-typage non structurel non homogène, c'est-à-dire reliant des constructeurs d'arité différente. En général, de telles relations de sous-typage nécessitent d'exprimer la correspondance entre les différents arguments des constructeurs de types. Par exemple, en écrivant $k_1(\alpha, \beta) \leq k_2(\beta)$, nous spécifions que les types construits avec k_1 sont des sous-types de ceux construits avec k_2 lorsque le second argument de k_1 est un sous-type de l'argument de k_2, le premier argument de k_1 étant oublié dans la relation de sous-typage. Une manière d'exprimer cette correspondance est d'utiliser un formalisme d'étiquettes, comme proposé par Pottier [14]. Dans ce formalisme, on associe à chaque argument d'un constructeur une étiquette, la correspondance entre deux arguments de deux constructeurs étant exprimée par le fait qu'ils ont la même étiquette. L'ordre de sous-typage $\leq$ est construit à partir de l'ordre sur les constructeurs types $\leq_\mathcal{K}$ et les étiquettes. Une description plus formelle de la structure des types est donnée dans [6], où la structure des types et des constructeurs de types forme un quasi-treillis, c'est-à-dire un ordre partiel dans lequel deux éléments ont une borne inférieure (resp. supérieure) si et seulement si ils sont minorés (resp. majorés).

Contraintes de sous-typage Soit $\mathcal{W}$ un ensemble de *variables de types*, ou *paramètres*, notées $\alpha, \beta, \ldots$. On note $\mathcal{T}_\mathcal{W}$ l'ensemble des types construits sur $\mathcal{K} \cup \mathcal{W}$.

Définition 6 *Une* contrainte de sous-typage *est de la forme* $\tau_1 \leq \tau_2$, *où* $\tau_1, \tau_2 \in \mathcal{T}_\mathcal{W}$ *sont des types finis.*

Une substitution $\rho : \mathcal{W} \to \mathcal{T}$ satisfait *la contrainte* $\tau_1 \leq \tau_2$, *noté* $\rho \models \tau_1 \leq \tau_2$, *si* $\rho(\tau_1) \leq \rho(\tau_2)$.

La contrainte de sous-typage $\tau_1 \leq \tau_2$ *est* satisfiable *s'il existe une substitution* ρ *telle que* $\rho \models \tau_1 \leq \tau_2$.

Dans [6], nous donnons des conditions suffisantes sur $(\mathcal{K}, \leq_{\mathcal{K}})$ pour la décidabilité de la satisfiabilité des contraintes de sous-typage dans les quasi-treillis, ainsi qu'un algorithme de calcul explicite de solutions.

4.2 Système de types pour CLP+CHR

Afin de tenir compte de la surcharge des symboles de fonction et de prédicats CLP, on suppose qu'à chaque symbole de fonction f/n est associé un ensemble $types(f/n)$ de *schémas de types* de la forme $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \tau$, où $\bar{\alpha}$ est l'ensemble des paramètres apparaissant dans les types $\tau_1, \ldots, \tau_n, \tau$. De même, on suppose qu'à chaque symbole de prédicat p/n (resp. de contrainte native c/n) est associé un ensemble $types(p/n)$ de schémas de types de la forme $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n$, également noté σ. De même que pour le type des contraintes CHR, ces ensemble de types sont supposés fixé, par exemple à l'aide de déclarations fournies par le programmeur. On suppose également que la contrainte $= /2$ a le schéma de types $\forall\alpha.\alpha \times \alpha$. Par souci de simplicité, la quantification $\forall\bar{\alpha}$ sera omise dans les schémas de types, chaque occurrence d'un schéma de types étant renommée avec des paramètres frais. Étant donné une contrainte CHR c/n de type $\tau_1 \times \ldots \times \tau_n$, on définit $types(c/n)$ comme le singleton $\{\tau_1 \times \ldots \times \tau_n\}$.

Un environnement de typage est une fonction partielle $\Gamma : \mathcal{V} \mapsto \mathcal{T}_{\mathcal{W}}$, également noté $\{X_1 : \tau_1, \ldots, X_n : \tau_n\}$. L'opération $\uplus$ sur les environnements de typage est définie comme l'union disjointe, c'est-à-dire que $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_1(X)$ si $X \in dom(\Gamma_1) \setminus dom(\Gamma_2)$, $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_2(X)$ si $X \in dom(\Gamma_2) \setminus dom(\Gamma_1)$, et $(\Gamma_1 \uplus \Gamma_2)(X)$ est indéfini sinon.

La table 2 donne les règles de typage pour CLP, ainsi que la règle de typage pour la déclaration *label_with*. L'ensemble des règles des tables 1 et 2 définissent le système de types pour les langages CLP+CHR. Un appel de prédicat $p(t_1, \ldots, t_n)$ (resp. une contrainte native) est bien typé dans un environnement Γ si on peut dériver $\Gamma \vdash p(t_1, \ldots, t_n)$ *Atom*. Une clause H :- B est *bien typée* si on peut dériver $\vdash H$:- B *Clause*. Une déclaration d'énumération *label_with H if G* est bien typée si on peut dériver $\vdash$ *label_with H if G Label_with*. Un programme CLP+CHR est bien typé si toutes ses règles CHR, toutes ses clauses et toutes ses déclarations d'énumération sont bien typées.

Les règles de la table 2, à l'exception de la règle (*Label with*), correspondent au système de Mycroft et O'Keefe [13] auquel on a rajouté la règle de sous-typage et incorporé la notion de surcharge. La règle (*Sub*) donne la sémantique du sous-typage en exprimant que si un terme t a le type τ alors il a également tous les types qui sont plus grands que τ.

La règle (*CHR Atom*) peut être vue comme un cas particulier des règles (*Atom*) et (*Head*) où le symbole p/n n'a qu'un seul schéma de types et où ce schéma ne contient pas de paramètre, ce qui correspond bien à la restriction d'une contrainte CHR n'a qu'un seul type.

La distinction entre les règles (*Atom*) et (*Head*) exprime le principe de *généricité des définitions* [12] qui établit que le type d'une occurrence de définition d'un prédicat (c'est-à-dire la tête d'une clause) doit être équivalent à un renommage près au type assigné au prédicat. La règle (*Clause*) impose que la clause soit bien typée pour tous les types possibles pour le prédicat, ce qui peut être vu comme une condition similaire à celle de généricité des définitions pour la surcharge. Ces deux conditions sont utiles pour le théorème d'auto-réduction qui suit et qui exprime la cohérence du système de types par rapport à la sémantique opérationnelle CLP+CHR. Il est précédé par un lemme qui exprime que l'on peut instancier les types une dérivation de typage n'utilisant pas les règles (*Clause*) ou (*Head*).

Lemme 2 *Pour tout environnement de typage* Γ, *tout jugement* R *différent de Head ou Clause et toute substitution de type* ρ, *si* $\Gamma \vdash R$ *alors* $\Gamma\rho \vdash R\rho$.

Preuve. Par induction sur la dérivation. □

Théorème 2 *Soit un programme CHR+CLP bien typé. Soient deux états* $\langle F, E, D\rangle$ *et* $\langle F', E', D'\rangle$ *et un environnement de typage* Γ *tel que* $\Gamma \vdash F, E, D$ *Goal. Si* $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$, *alors il existe un environnement de typage* Γ' *tel que* $\Gamma' \vdash F', E', D'$ *Goal. De plus, si la transition nécessite de vérifier une implication de la forme* $\mathcal{CT} \models D \Rightarrow G$, *alors* $\Gamma' \vdash G$ *Goal.*

Preuve. On peut constater que le système de la table 2 vérifie bien les hypothèses de la section 3.1. De plus un atome correspondant à un appel de prédicat est typé de la même manière qu'une contrainte native. Par conséquent, de par le théorème 1, si la réduction s'effectue avec la règle **Solve**, **Introduce**, **Simplify** ou **Propagate**, alors on a bien qu'il existe un environnement te' tel que $\Gamma' \vdash F', E', D'$ *Goal* et $\Gamma' \vdash G$ *Goal* le cas échéant.

Considérons le cas de la règle **Unfold**. On peut supposer, sans perte de généralité, que $H' = p(s)$ et $H = p(t)$. Comme $\Gamma \vdash p(s)$ *Atom*, il existe un schéma

$$(Var)\quad \frac{X:\tau \in \Gamma}{\Gamma \vdash X:\tau} \qquad (Sub)\quad \frac{\Gamma \vdash t:\tau \quad \tau \leq \tau'}{\Gamma \vdash t:\tau'}$$

$$(Func)\quad \frac{\Gamma \vdash t_1:\tau_1\rho \quad \ldots \quad \Gamma \vdash t_n:\tau_n\rho}{\Gamma \vdash f(t_1,\ldots,t_n):\tau\rho} \qquad \begin{array}{r}\rho \text{ est une substitution de types}\\ \tau_1 \times \ldots \tau_n \rightarrow \tau \in types(f/n)\end{array}$$

$$(Atom)\quad \frac{\Gamma \vdash t_1:\tau_1\rho \quad \ldots \quad \Gamma \vdash t_n:\tau_n\rho}{\Gamma \vdash p(t_1,\ldots,t_n)\ Atom} \qquad \begin{array}{r}\rho \text{ est une substitution de types}\\ \tau_1 \times \ldots \tau_n \in types(p/n)\end{array}$$

$$(Head)\quad \frac{\Gamma \vdash t_1:\tau_1\rho \quad \ldots \quad \Gamma \vdash t_n:\tau_n\rho}{\Gamma \vdash p(t_1,\ldots,t_n)\ Head_{\tau_1\times\ldots\times\tau_n}} \qquad \begin{array}{r}\rho \text{ est un renommage}\\ \tau_1 \times \ldots \tau_n \in types(p/n)\end{array}$$

$$(Clause)\quad \frac{\forall \sigma \in types(p/n) \quad \Gamma_\sigma \vdash p(t_1,\ldots,t_n)\ Head_\sigma \quad \Gamma_\sigma \vdash B_1\ Atom \quad \ldots \quad \Gamma_\sigma \vdash B_k\ Atom}{\vdash p(t_1,\ldots,t_n)\ \texttt{:-}\ B_1,\ldots,B_k\ Clause}$$

$$(Label\ with)\quad \frac{\Gamma \vdash H\ Atom \quad \Gamma \vdash G\ Goal}{\vdash label\_with\ H\ if\ G\ Label\_with}$$

TAB. 2 – Système de types pour CLP et *label_with*

de types $\tau \in types(p)$ et une substitution ρ tels que $\Gamma \vdash s : \tau\rho$. Comme le programme est bien typé, $\vdash H$:- B *Clause*, donc il existe un environnement de typage Γ_τ tel que $\Gamma_\tau \vdash B$ *Goal* et $\Gamma_\tau \vdash p(t)$ $Head_\tau$, c'est-à-dire $\Gamma_\tau \vdash t : \tau\rho_r$ où ρ_r est un renommage de τ. En posant $\rho' = \rho_r^{-1}\rho$, et de par le lemme 2, on obtient $\Gamma_\tau\rho' \vdash t : \tau\rho$ et $\Gamma_\tau\rho' \vdash B$ *Goal*. En posant $\Gamma' = \Gamma_\tau\rho' \uplus \Gamma$, on obtient $\Gamma' \vdash t = s$ *Atom*. D'où $\Gamma' \vdash B, F, E, s = t, D$ *Goal*.

Considérons enfin le cas de la règle **Label**. De même que pour la règle **Unfold**, il existe un environnement de typage Γ' tel que $\Gamma' \vdash B, F, E, t = s, D$. Comme H' est une contrainte CHR, τ ne contient pas de paramètre, c'est-à-dire que ρ'' est l'identité. On a $H'' = p(u)$ pour un certain terme u. Comme $\vdash$ *label_with* H'' *if* G *Label_with*, il existe un environnement de typage Γ_{lw} tel que $\Gamma_{lw} \vdash G$ *Goal* et $\Gamma_{lw} \vdash H''$ *Atom*, c'est-à-dire $\Gamma_{lw} \vdash u : \tau$. En posant $\Gamma'' = \Gamma' \uplus \Gamma_{lw}$, on obtient $\Gamma'' \vdash s = u$ *Atom*, $\Gamma'' \vdash B, F, E, s = t, D$ et $\Gamma'' \vdash G$ *Goal*, d'où $\Gamma'' \vdash s = u, G$ *Goal*. □

5 Résultats expérimentaux

Le système de types pour CLP+CHR à été implanté comme une extension du logiciel TCLP [3], qui est un typeur pour les langages logiques avec contraintes de la classe CLP. Le type des variables du programme est inféré et il est également possible d'inférer un type pour le type des prédicats. Le type $term \times \ldots \times term$ étant toujours un type possible pour un prédicat, nous utilisons une inférence heuristique donnant un type plus informatif [7, 4]. Cet algorithme est également utilisé pour inférer le type des contraintes CHRs qui n'ont pas été déclarées par l'utilisateur.

Un point intéressant est que TCLP utilise un certain nombre de solveurs écrits en CHR. Le principal solveur est celui pour les contraintes de sous-typage. Nous utilisons également un solveur CHR pour gérer la surcharge des symboles de fonction et de prédicats lors du typage. Enfin nous utilisons quelques autres petits solveurs qui permettent de gérer les environnements de typage et les calculs préliminaires sur la structure des constructeurs de types. Le fait de pouvoir typer les programmes CHR permet ainsi à TCLP de vérifier son propre code source.

L'exemple ci-dessous montre le genre d'erreur typiquement détectée par TCLP :

Exemple 4 *Le solveur suivant permet de gérer des compteurs. La contrainte* `cpt/2` *associe le nom du compteur à sa valeur et a le type atom* × *int.*[1] *La contrainte* `val/2` *a également le type atom* × *int et les contraintes* `incr/1` *et* `init/1` *ont le type atom.*

```
init(C) <=> cpt(C,0).
cpt(C,V) \ val(C,X) <=> X=V.
incr(C), cpt(V,C) <=> V1 is V+1, cpt(C,V1).
```

Le typeur produit le message suivant :

```
! Error in "count.pl", line 3 :
  Incompatible types for C : atom and int
```

Il s'agit en fait d'une inversion d'arguments : dans la tête de la dernière règle, les arguments de la contrainte `cpt` *ont étés inversés.*

[1] Le type *atom* correspond aux atomes Prolog, c'est-à-dire des symboles d'arité 0, et non pas aux atomes logiques.

Programme	# lignes	# règles	Vérification CHR	Vérification Total	Inférence CHR	Inférence Total
gcd	10	2	0.03 s	0.03 s	0.04 s	0.04 s
varleq	30	4	0.04 s	0.26 s	0.07 s	0.43 s
bool	173	78	1.32 s	2.13 s	4.63 s	5.96 s
listdom	73	13	0.78 s	1.45 s	1.77 s	2.75 s
interval	145	24	3.41 s	3.5 s	99.58 s	99.69 s
domain	266	84	4.30 s	6.42 s	183.92 s	186.94 s
fourier-gauss	328	30	1.98 s	5.88 s	19.04 s	30.42 s
arc	47	2	0.14 s	0.81 s	0.23 s	1.09 s
allenComp	495	490	17.48 s	17.51 s	ND	ND
sous-typage	595	57	4.52 s	6.22 s	319.66 s	322.64 s
surcharge	465	10	0.43 s	3.99 s	1.10 s	8.01 s
TCLP	4594	82	5.22 s	53.97 s	416.08 s	518.39 s

TAB. 3 – Performance de TCLP

Cet exemple montre le résultat de l'inférence de types sur un petit solveur :

Exemple 5 *Le solveur suivant, issu de [9], calcule du plus grand diviseur commun de deux nombres.*

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=>
   N=<M | L is M mod N, gcd(L).
```

Le typeur infère le type suivant :

```
:- typeof gcd(int) is chr_constraint.
```

c'est-à-dire que `gcd` *a le type int.*

Performances La rapidité du typeur a été évaluée sur dix solveurs CHR issus de [9], sur le solveur de contraintes de sous-typage et le solveur pour la surcharge dans TCLP, ainsi que sur TCLP dans son intégralité. Ces tests on été réalisés sur une machine dotée d'un Pentium IV à 2 Ghz et de 512 Mo de RAM, en utilisant l'implantation de TCLP en Sicstus Prolog, pour laquelle l'utilisation de la mémoire est limitée à 256 Mo. Les résultats sont présentés dans le tableau 3.

La première colonne indique le programme CLP+CHR. La deuxième colonne indique le nombre de lignes de code dans le programme et la troisième colonne indique le nombre de règles CHR du programme. Nous indiquons ensuite, dans la colonne "Vérification", les temps de typage pour la vérification des types avec inférence du type des variables, mais sans inférence du type des prédicats et des contraintes CHR. Enfin, la colonne "Inférence" indique les temps de typage avec inférence de type pour les prédicats et les contraintes. Les temps de typage des règles CHR sont indiqués dans les colonnes "CHR", les temps de typage du programme CLP+CHR dans son intégralité étant donnés dans la colonne "Total".

Les temps de typage pour la vérification, sans inférence du type des prédicats et des contraintes, permettent de constater que le typeur est utilisable en pratique. Par exemple, il suffit de moins de 18 s pour vérifier les 490 règles du solveur `allenComp`, ou de 54 s pour vérifier les quelques 4600 lignes de code de TCLP. En revanche, les résultats sont beaucoup plus disparates lorsque l'on utilise l'inférence de types pour les prédicats et les contraintes. Par exemple, il faut environ 71 fois plus de temps pour inférer les types que pour effectuer la vérification dans le cas du solveur pour le sous-typage. Dans le cas de `allenComp`, l'inférence a même échoué pour cause d'un manque de mémoire dû à la restriction à 256 Mo. Ceci s'explique par le fait que, pour inférer le type d'une contrainte, le typeur doit considérer en une seule fois toutes les règles et les clauses faisant partie d'une même composante connexe du graphe d'appel. À l'inverse, la vérification peut se faire règle (resp. clause) par règle. Or les solveurs CHRs sont souvent peu stratifiés, c'est-à-dire qu'ils mettent souvent en jeu des composantes connexes très larges. Une raison à cette difficulté est qu'une ou deux contraintes sont utilisées comme des structures de données, par exemple pour représenter le domaine des variables. Ces contraintes apparaissent alors dans la tête de nombreuses règles CHR du solveur. Ainsi, le solveur pour les contraintes de sous-typage a une composante connexe comprenant 54 prédicats et contraintes CHR. De telles composantes connexes nécessitent ainsi de gérer de très nombreuses contraintes de sous-typage et occurrences de symboles surchargé en même temps. Les algorithmes de résolution des contraintes de sous-typage et de résolution de la surcharge sont cependant potentiellement exponentiels [6, 5]. De ce point de vue, les performances de l'inférence de types sont relativement satisfaisantes. En particulier, l'inférence de types peut être utilisée la

première fois qu'un solveur est typé, les types alors inférés étant ensuite utilisés comme déclarations durant la suite du développement du solveur.

6 Conclusion

Nous avons présenté un système de types pour le langage des *Constraint Handling Rules* [8], paramétré par le système de types du langage hôte. Nous avons également présenté son instanciation par le système de types prescriptifs TCLP [7] pour les langages logiques avec contraintes. Sous l'hypothèse que les étapes de résolution effectuées par solveur de contraintes natives transforment une mémoire bien typée en une mémoire bien typée, nous avons montré la cohérence des deux systèmes, respectivement par rapport à la sémantique opérationnelle de CHR et par rapport à la sémantique étendue CLP+CHR.

Le système de types pour CLP+CHR a été implanté comme une extension du logiciel TCLP [3]. Nous avons exposé des résultats expérimentaux montrant que le système est d'ores et déjà utilisable et utile en pratique.

Comme perspectives, nous prévoyons d'améliorer les performances de l'inférence des types pour les grandes composantes connexes, et d'acquérir plus d'expérience pratique de la part des utilisateurs du système, en particulier sur son impact lors du développement de solveurs CHRs complexes. Il serait également intéressant d'étudier l'instanciation du système de types avec celui du langage Java dans le cadre de l'implantation des CHRs dans la boîte à outils JACK [2].

Références

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266, Linz, 1997. Springer-Verlag.

[2] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK : A Java Constraint Kit. In *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier, 2000.

[3] E. Coquery. TCLP. `http://contraintes.inria.fr/~coquery/tclp/`.

[4] E. Coquery. *Typage et programmation en logique avec contraintes*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, December 2004.

[5] E. Coquery and F. Fages. Surcharge et sous-typage dans TCLP. In *Actes des Journées Francophones de la Programmation en Logique avec Contraintes JFPLC'2002*, 2002.

[6] E. Coquery and F. Fages. Subtyping constraints in quasi-lattices. In P. K. Pandya and J. Radhakrishnan, editors, *Foundations of Software Technology and Theoretical Computer Science, FSTTCS'03*, volume 2914 of *LNCS*, pages 136–148. Springer-Verlag, 2003.

[7] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6) :751–777, November 2001.

[8] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3) :95–138, October 1998.

[9] T. Frühwirth and T. Schrijvers. CHR web page. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.

[10] C. Holzbaur and T. Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.

[11] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[12] T.K. Lakshman and U.S. Reddy. Typed Prolog : A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[13] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23 :295–307, 1984.

[14] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4) :312–347, November 2000.

[15] Jan-Georg Smaus, François Fages, and Pierre Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In *Proceedings of FSTTCS '2000*, number 1974 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[16] P. J. Stuckey and M. Sulzmann. A theory of overloading. In S. Peyton-Jones, editor, *Proceedings of the International Conference on Functional Programming*, pages 167–178. ACM Press, 2002.