



A Type System for CHR

Emmanuel Coquery, Francois Fages

► **To cite this version:**

Emmanuel Coquery, Francois Fages. A Type System for CHR. [Research Report] RR-5565, INRIA. 2005, pp.17. inria-00070441

HAL Id: inria-00070441

<https://hal.inria.fr/inria-00070441>

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Type System for CHR

Emmanuel Coquery — François Fages

N° 5565

Mai 2005

Thème SYM



R
apport
de recherche

A Type System for CHR

Emmanuel Coquery , François Fages

Thème SYM — Systèmes symboliques
Projet Contraintes

Rapport de recherche n° 5565 — Mai 2005 — 17 pages

Abstract: We propose a generic type system for the *Constraint Handling Rules* (CHR), a rewriting rule language for implementing constraint solvers. CHR being a high-level extension of a host language, such as Prolog or Java, this type system is parameterized by the type system of the host language. We show the consistency of the type system for CHR w.r.t. its operational semantics. We also study the case where the host language is a constraint logic programming language, typed with the prescriptive type system we developed 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 report on experimental results about the type-checking of 12 CHR solvers and programs, including TCLP itself.

Key-words: type system, constraint programming, constraint logic programming, constraint handling rules

Un système de types pour CHR

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.

Mots-clés : système de types, programmation par contraintes, programmation logique avec contraintes, constraint handling rules

1 Introduction

The language of *Constraint Handling Rules* (CHR) of T. Frühwirth [8] is a successful rule-based language for implementing constraint solvers in a wide variety of domains. It is an extension of a host language, such as Prolog [11] or Java [1], allowing the introduction of new constraints in a declarative way. CHR is used to handle user-defined constraints while the host language deals with other computations using *native* constraints. CHR is a committed-choice language of guarded rules that rewrite constraints into simpler ones until they are in solved forms. One peculiarity of CHR is that it allows multiple heads in rules.

Typed languages have numerous advantages from the point of view of program development, such as the static detection of programming errors or program composition errors, and the documentation of the code by types. CHR has already been used for the typing of programming languages, either for solving subtyping constraints [6, 5] or for handling overloading in functional languages [17] and constraint logic languages [3, 5]. However, to our knowledge, there is no type system for CHR.

In this article, we propose a type system for CHR inspired by the TCLP type system for constraint logic programs [7]. CHR being an extension of a host language, this system is parameterized by the type system of the host language. We will make three assumptions on the type system of the host language:

- Typing judgments of the form $\Gamma \vdash t : \tau$ are considered, where τ is a type associated to the term t in a typing environment Γ . Moreover *well-typed* constraints in a typing environment Γ are defined by a derivation system for typing judgments.
- The constraint $t_1 = t_2$ is well-typed in the environment Γ if there exists a type τ such as $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$.
- If a conjunct c of native constraints is well-typed in an environment Γ and is equivalent to a conjunct d , then d is also well-typed in Γ .

Using these assumptions, we show the consistency of the type system for CHR w.r.t. its operational semantics. This is expressed by a subject reduction theorem which establishes that if a program is well-typed then all the derived goals from a well-typed goal are well-typed.

We also study the instantiation of this type system with the TCLP type system for constraint logic programs [7]. We show a subject reduction theorem for the CLP+CHR execution model [8] in which it is possible to extend the definition of constraints by clauses. This result is interesting because constraint logic programming is a natural framework for using constraint solvers. A type system for CLP+CHR allows us to type-check a solver together with the program that uses it, as well as complex CHR solvers written as a combination of clauses and rules, where CHR rules use CLP predicates and CLP clauses post CHR constraints.

The rest of the paper is organized as follows. Section 2 recalls the syntax and operational semantics of CHR, including the CLP+CHR execution model. Section 3 presents

the type system and section 4 presents its instantiation with the type system for CLP. Section 5 presents some experimental results on the typing of some CHR solvers, using the implementation of the system in TCLP [4]. Finally, we conclude in section 6.

2 Preliminaries on CHR

Here, we recall the syntax and semantics of CHR, as given in [8]. We distinguish the user-defined *CHR constraints* from the *native constraints* of the host language, which represent auxiliary computations that take place during the application of a CHR rule. We assume that native constraints are handled by a predefined solver of the host language. We also assume that native constraints include the equality constraint $=/2$ and the constraint *true*. The terms of the host language are noted s, t, \dots . We note \mathcal{X} the domain of native constraints, and CT its (possibly incomplete) first-order logic theory.

2.1 Syntax

Definition 1 A CHR rule is either:

- a simplification rule of the form:

$$H_1, \dots, H_i \Leftrightarrow G_1, \dots, G_j \mid B_1, \dots, B_k$$
- a propagation rule of the form:

$$H_1, \dots, H_i \Rightarrow G_1, \dots, G_j \mid B_1, \dots, B_k$$
- or a simpagation rule of the form:

$$H_1, \dots, H_l \setminus H_{l+1}, \dots, H_i \Leftrightarrow G_1, \dots, G_j \mid B_1, \dots, B_k$$

with $i > 0, j \geq 0, k \geq 0, l > 0$ and H_1, \dots, H_i is a nonempty sequence of CHR constraints, the guard G_1, \dots, G_j being a sequence of native constraints and the body B_1, \dots, B_k being a sequence of CHR and native constraints.

A CHR program is a finite sequence of CHR rules.

The constraint *true* is used to represent empty sequences. The empty guard can be omitted, together with the \mid symbol. The notation $name@R$ gives a name to a CHR rule R .

Informally, a simplification rule replaces the constraints of the head by the constraints of the body. A propagation rule adds the constraints of the body while keeping the constraints of the head in the store. A simpagation rule is a mix of the two preceding kind of rules: the constraints H_{l+1}, \dots, H_i are replaced by the body, while the constraints H_1, \dots, H_l are kept.

For the sake of simplicity, and because the distinction of propagation and simpagation rules are not needed for typing purposes, we will consider that a propagation or a simpagation rule of the form

$$H_1, \dots, H_l \setminus H_{l+1}, \dots, H_i \Leftrightarrow G_1, \dots, G_j \mid B_1, \dots, B_k$$

is just an abbreviation for the simplification rule

$$H_1, \dots, H_i \text{ <=> } G_1, \dots, G_j \mid H_1, \dots, H_i, B_1, \dots, B_k.$$

Example 1 *The following CHR program, taken from [8], defines a solver for a general ordering constraint =<.*

```

reflexivity @ X=<Y <=> X=Y | true.
antisymmetry @ X=<Y , Y=<X <=> X=Y.
transitivity @ X=<Y , Y=<Z ==> X=<Z.
identity @ X=<Y \ X=<Y <=> true.

```

The rule reflexivity eliminates the =< constraints when its two arguments are equal. Rule antisymmetry simplifies a double inequality into an equality. The rule transitivity adds constraints corresponding to the transitive closure of =<. Finally, identity eliminates redundant =< constraints.

2.2 Operational Semantics

The operational semantics of CHR is expressed by a transition system, noted \mapsto , over states which are triples $\langle F, E, D \rangle$, where F is a goal, that is a multiset of native and CHR constraints, E is a CHR constraint store and D is a native constraint store. A state is thus a conjunction of CHR and native constraints.

In the following definition, the equality is extended to constraints by morphism, that is $c(t_1, \dots, t_n) = c(t'_1, \dots, t'_n)$ if $t_1 = t'_1 \wedge \dots \wedge t_n = t'_n$. The conjunction notation \wedge is used to express the matching of a constraint in a multiset, The equality is also extended to conjunctions of constraints: $H_1 \wedge \dots \wedge H_n = H'_1 \wedge \dots \wedge H'_n$ if $H_1 = H'_1 \wedge \dots \wedge H_n = H'_n$.

Definition 2 *Let P be a CHR program. The transition relation \mapsto is given by the following rules, where the variables appearing in triples stand for conjunctions of constraints and \bar{x} represents the set of variables appearing in the head H .*

Solve

$$\langle C \wedge F, E, D \rangle \mapsto \langle F, E, D' \rangle$$

if C is a native constraint and $CT \models (C \wedge D) \Leftrightarrow D'$.

Introduce

$$\langle H \wedge F, E, D \rangle \mapsto \langle F, H \wedge E, D \rangle$$

if H is a CHR constraint.

Simplify

$$\langle F, H' \wedge E, D \rangle \mapsto \langle B \wedge F, E, H = H' \wedge D \rangle$$

if $(H \text{ <=> } G \mid B)$ is in P renamed with fresh variables, and $CT \models D \Rightarrow \exists \bar{x}(H = H' \wedge G)$.

Propagate

$\langle F, H' \wedge E, D \rangle \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D \rangle$
 if $(H \implies G \mid B)$ is in P renamed with fresh variables,
 and $CT \models D \Rightarrow \exists \bar{x}(H = H' \wedge G)$.

The **Solve** transition corresponds to a transition of the native constraint solver. The **Introduce** transition simply transfers a CHR constraint from the goal to the CHR constraint store. The **Simplify** transition correspond to the application of CHR simplification. The **Propagate** transition is indicated for the sake of clarity, although it is treated as an abbreviation for a simplification rule in the rest of the paper. The condition for applying these rules is that the head of the rule can be instantiated such that the guard and the matching condition of the head are implied by the current native constraint store. The body of the rule is then added to the current goal and, when applying a **Simplify** transition, the constraints matching the head are removed from the constraint store.

Definition 3 An initial state consists in a goal F and two empty constraint stores: $\langle F, true, true \rangle$. A final state is either of the form $\langle F, E, false \rangle$ (failure), or of the form $\langle true, E, D \rangle$ where D is satisfiable (success).

The following example illustrates the execution of a CHR program.

Example 2 Let us consider the solver given in example 1 together with the initial state $\langle X=<Y \wedge Y=<Z \wedge Z=<X, true, true \rangle$. One possible execution is:

$\langle Z=<X, X=<Y \wedge Y=<Z, true \rangle$	<i>(Introduce ×2)</i>
$\langle X=<Z \wedge Z=<X, X=<Y \wedge Y=<Z, true \rangle$	<i>(Propagate transitivity)</i>
$\langle true, X=<Z \wedge Z=<X \wedge X=<Y \wedge Y=<Z, true \rangle$	<i>(Introduce ×2)</i>
$\langle X=Z, X=<Y \wedge Y=<Z, true \rangle$	<i>(Simplify antisymmetry)</i>
$\langle true, X=<Y \wedge Y=<Z, X=Z \rangle$	<i>(Solve)</i>
$\langle X=Y, true, X=Z \rangle$	<i>(Simplify antisymmetry)</i>
$\langle true, true, X=Y \wedge X=Z \rangle$	<i>(Solve)</i>

One can remark that in this operational semantics, once a propagation rule can be applied, it can be applied infinitely often, which leads to a trivial case of non termination. In the preceding example, one could have applied the **transitivity** rule instead of the **antisymmetry** rule, thus reintroducing the constraint $X=<Z$ that was eliminated at the fourth step. In [2], Abdennadher gives refined operational semantics that are more faithful to the actual implementation of CHR. In particular the previous behaviour is avoided by restricting the application of a rule only once on the same constraints. The subject reduction theorems given in the following sections express that given a well-typed program, a transition occurring from a well-typed state leads to a well-typed state. It is worth noting that they thus hold also in these more realistic semantics.

2.3 CLP+CHR

When the host language belongs is a constraint logic programming language $CLP(\mathcal{X})$ [12], it is possible to tightly integrate CHR to the host language. To this end, Frühwirth [8] proposed to extend CHR with the construct *label_with* used to define CHR constraints by CLP clauses. We recall here the syntax and operational semantics of this extension. We note \mathcal{S}_F (resp. \mathcal{S}_P) the set of function (resp. predicate) symbols, given with their arity, and \mathcal{V} the set of variables. An *atom* is either a native constraint, a CHR constraint or of the form $p(t_1, \dots, t_n)$, where p/n is a program predicate symbol. The declaration *label_with* $c(t_1, \dots, t_n)$ *if* G_1, \dots, G_j expresses that G_1, \dots, G_j is a guard for the clauses of the CHR constraint c/n .

Definition 4 A labeling declaration for a CHR constraint H is an expression of the form:

$$\text{label_with } H \text{ if } G_1, \dots, G_j$$

where G_1, \dots, G_j is a conjunction of native constraints.

Clauses are of the form:

$$H \text{ :- } B_1, \dots, B_n$$

where H an atom corresponding either to a predicate or to a CHR constraint but not to a native constraint, and B_1, \dots, B_k is a sequence of atoms.

Definition 5 The relation transition between CHR states is extended by the two following rules:

Unfold

$$\langle H' \wedge F, E, D \rangle \mapsto \langle B \wedge F, E, H = H' \wedge D \rangle$$

if $(H \text{ :- } B)$ is in P renamed with fresh variables,
and H is not a CHR constraint.

Label

$$\langle F, H' \wedge E, D \rangle \mapsto \langle B \wedge F, E, H = H' \wedge D \rangle$$

if $(H \text{ :- } B)$ and $(\text{label_with } H'' \text{ if } G)$ are in P renamed with fresh variables, and
 $\mathcal{X} \models \exists \bar{x}(H' = H'' \wedge G)$

The **Unfold** transition is close to the CSLD resolution rule [12]. The difference is that, under CSLD resolution, the constraints in the body of the resolving clause are added to the native constraint store and the resulting store, i.e. $H = H' \wedge D \wedge C$, must be satisfiable, which is not demanded here. The CLP clauses for CHR constraints can only be used in a **Label** transition, requiring that the guards declared using *label_with* are implied by the current native constraint store.

3 Type System

3.1 Assumptions about the type system of the host language

Since CHR is an extension of a host language, the type system we propose is parameterized by the type system, noted \vdash_N , of the host language. We will make the following assumptions on \vdash_N .

We suppose that \vdash_N is based on a type algebra, the set of types being noted \mathcal{T} . Types are noted using the letter τ . Typing environments, noted Γ , associate types to variables. Given an expression t and a typing environment Γ , \vdash_N is used to deduce typing judgments of the form $\Gamma \vdash_N t : \tau$. Similarly, \vdash_N is used to deduce *well-typed* constraints in a typing environment Γ , a conjunction $C_1 \wedge \dots \wedge C_n$ of native constraints being well-typed in Γ if for each $i \in \{1, \dots, n\}$, C_i is well-typed in Γ . We note $\Gamma \vdash_N C \text{ Atom}$, the fact that C is well-typed in the typing environment Γ . We also assume that the equality constraint $s = t$ between s and t is well-typed in Γ if there exists a type τ such that $\Gamma \vdash_N s : \tau$ and $\Gamma \vdash_N t : \tau$.

We assume that the union of type environments over disjoint sets of variables can be formed with an operation noted \uplus such that if $\Gamma \vdash_N t : \tau$ then $\Gamma \uplus \Gamma' \vdash_N t : \tau$ for any typing environment Γ' disjoint from Γ . We also assume that if a conjunction of native constraints C is well-typed in a typing environment Γ and $\mathcal{CT} \models C \Leftrightarrow D$, then there exists a typing environment Γ' , such that the conjunction of constraints D is well-typed in $\Gamma \uplus \Gamma'$.

3.2 Type System for CHR

The type system we propose for CHR defines a notion of well-typedness for CHR rules. To each CHR constraint symbol c/n is associated one, and only one, type of the form $\tau_1 \times \dots \times \tau_n$. This type is assumed to be fixed, for example using some declarations provided by the programmer. One can remark that the limitation to one type for each constraint symbol prevents the use of type schemes, such as those used for parametric polymorphism [14]. For example, it is forbidden to give the type $\forall \alpha. \text{list}(\alpha) \times \text{list}(\alpha) \times \text{list}(\alpha)$ to a CHR constraint `append/3`, because it is equivalent to give it all the types of the form $\text{list}(\tau) \times \text{list}(\tau) \times \text{list}(\tau)$. This restriction is motivated by the example 3 given below, which shows that the allowance of type schemes in CHR would make the type system inconsistent.

The rules of the type system for CHR are given in table 1. A CHR constraint H is *well-typed* in Γ if the judgment $\Gamma \vdash H \text{ Atom}$ can be derived from the typing rule. Terms or expressions appearing as arguments of the constraints are typed using the type system \vdash_N for the host language. The rule (*Simpl CHR*) expresses that a CHR rule is *well-typed* if its head, its guard and its body are well-typed in the same typing environment Γ . A CHR program is well-typed if all its clauses are well-typed.

The following lemma expresses that the well-typedness of goals is preserved by extension of the environment:

Lemma 6 *Let G be a goal and Γ a typing environment such that $\Gamma \vdash G \text{ Goal}$. Let Γ' be a typing environment such that $\Gamma \uplus \Gamma'$ is defined. Then $\Gamma \uplus \Gamma' \vdash G \text{ Goal}$.*

(Native)	$\frac{\Gamma \vdash_N C \text{ Atom}}{\Gamma \vdash C \text{ Atom}}$	if C is a native constraint
(CHR Atom)	$\frac{\Gamma \vdash_N t_1 : \tau_1 \quad \dots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \dots, t_n) \text{ Atom}}$	if c/n a CHR constraint and if c/n has type $\tau_1 \times \dots \times \tau_n$
(Goal)	$\frac{\Gamma \vdash B_1 \text{ Atom} \quad \dots \quad \Gamma \vdash B_n \text{ Atom}}{\Gamma \vdash B_1, \dots, B_n \text{ Goal}}$	
(Simpl CHR)	$\frac{\Gamma \vdash H_1, \dots, H_i \text{ Goal} \quad \Gamma \vdash G_1, \dots, G_j \text{ Goal} \quad \Gamma \vdash B_1, \dots, B_k \text{ Goal}}{\vdash H_1, \dots, H_i \Leftrightarrow G_1, \dots, G_j \mid B_1, \dots, B_k \text{ Rule}}$	

Table 1: Type system for CHR

Proof: By induction on the derivation and by using the assumption that if $\Gamma \vdash_N t : \tau$ then $\Gamma \uplus \Gamma' \vdash_N t : \tau$ for any expression t . \square

The consistency of the type system w.r.t. the operational semantics of CHR is given by the following subject reduction theorem, which expresses that the well-typedness of goals is preserved by transitions:

Theorem 7 *Let P be a well-typed CHR program. Let $\langle F, E, D \rangle$ and $\langle F', E', D' \rangle$ be two states such that $\langle F, E, D \rangle \mapsto \langle F', E', D' \rangle$. If there exists a typing environment Γ such that $\Gamma \vdash F, E, D \text{ Goal}$, then there exists a typing environment Γ' such that $\Gamma' \vdash F', E', D' \text{ Goal}$. Moreover, if the transition rule contains a guard G then $\Gamma' \vdash G \text{ Goal}$.*

Proof: By case on the transition.

Solve By hypothesis, $\Gamma \vdash D \text{ Goal}$ and $\Gamma \vdash C \text{ Atom}$. Since $CT \models (C \wedge D) \Leftrightarrow D'$, and by assumption on \vdash_N , there exists a typing environment Γ'' such that $\Gamma \uplus \Gamma'' \vdash_N D' \text{ Goal}$. By posing $\Gamma' = \Gamma \uplus \Gamma''$ and by lemma 6, we obtain $\Gamma' \vdash F, E \text{ Goal}$, thus $\Gamma' \vdash F, E, D' \text{ Goal}$.

Introduce This transition only moves a constraint from the goal to the CHR constraint store, thus the resulting state is also well-typed in Γ .

Simplify By hypothesis, the rule $(H \Leftrightarrow G \mid B)$ is well-typed. Thus, there exists a typing environment Γ'' such that $\Gamma'' \vdash H, G, B \text{ Goal}$. By lemma 6, and by posing $\Gamma' = \Gamma \uplus \Gamma''$, we obtain $\Gamma' \vdash F, E, D \text{ Goal}$ and $\Gamma' \vdash G, B \text{ Goal}$. It remains to prove that $H = H'$ is well-typed in Γ' . H is of the form $c_1(t_1^1, \dots, t_{m_1}^1), \dots, c_i(t_1^i, \dots, t_{m_i}^i)$ and H' of the form $c_1(s_1^1, \dots, s_{m_1}^1), \dots, c_i(s_1^i, \dots, s_{m_i}^i)$. Let $l \in \{1, \dots, i\}$ and $p \in \{1, \dots, m_l\}$. Let $\tau_1 \times \dots \times \tau_{m_l}$ be the type of c_l . Since $\Gamma' \vdash c_l(t_l^1, \dots, t_{m_l}^l) \text{ Atom}$, we have $\Gamma' \vdash_N t_p^l : \tau_p$. Similarly, $\Gamma' \vdash_N s_p^l : \tau_p$. Thus $\Gamma' \vdash t_p^l = s_p^l \text{ Atom}$. Thus, we obtain $\Gamma' \vdash B, F, E, H = H', D \text{ Goal}$ and $\Gamma' \vdash H = H', G \text{ Goal}$. \square

The following example illustrates how the use of a type scheme for a CHR constraint symbol may lead to check an ill-typed conjunction of constraints.

Example 3 *Let us assume that the constraint $=<$ has the type scheme $\forall\alpha.\alpha \times \alpha$. Let us suppose that "a" and "b" have type string, that 1 and 2 have type int, and that these two types are incompatible. Let us consider the rule:*

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

*and the state $\langle \text{true}, "a"=<"b" \wedge 1=<2, \text{true} \rangle$. This state is well-typed. However if one tries to apply the rule transitivity with **Propagate**, one needs to check that the constraint store implies $X="a" \wedge Y="b" \wedge Y=1 \wedge Z=2$, which is not a well-typed constraint, because Y must have both types string and int.*

4 Integration with CLP

In this section we are interested in the particular case where the host language is a constraint logic language, typed using the prescriptive type system TCLP [7]. This system combines parametric polymorphism, subtyping and overloading to obtain the flexibility that is needed for typing CLP programs that are originally untyped. In particular, subtyping is used for typing the simultaneous use of different constraint domains: for instance, the relation $\text{boolean} \leq \text{int}$ allows one to see booleans as integers, and thus to type check constraints combining boolean variables with integer variables (such as in a sum of boolean variables). Subtyping is also used for the typing of programs using meta-programming techniques: the relation $\text{list}(\alpha) \leq \text{term}$ allows one to see homogeneous lists as terms and to apply decomposition predicates to them, such as `functor/3`, `arg/3` or `./2`.

In [7], the type system of TCLP is proved consistent w.r.t. the CSLD execution model [12], which is an abstract model of execution proceeding by constraint accumulation. In particular, the transformations that can be made by the constraint solver are not considered. In the following, we assume that the solver for native constraints only performs simplifications that preserve well-typedness, according to the assumptions of section 3.1. This can be obtained either by using a typed execution model, as proposed in [7], or, in the case of the equality constraint, by using modes to fix the dataflow [16].

First, we present the type algebra used in the system, then we recall the typing rules for CLP, together with a typing rule for the labeling declaration `label _with`. The resulting system is proven consistent w.r.t. the CLP+CHR execution model.

4.1 Type Structure

We consider a partial order $(\mathcal{K}, \leq_{\mathcal{K}})$ of *type constructors*, given with their arity. The set \mathcal{T} of types is the set of finite or infinite types built on \mathcal{K} .

Subtyping Relation The use of subtyping for meta-programming purposes requires to consider relations like $list(\alpha) \leq term$. This form of non-structural non-homogeneous subtyping links different constructors of different arities. Such subtyping relations require to express the correspondence between the different arguments of type constructors. For example, by writing $k_1(\alpha, \beta) \leq k_2(\beta)$, we specify that types built with k_1 are subtypes of those built with k_2 , provided that the second argument of k_1 is a subtype of the argument of k_2 , the first argument of k_1 being forgotten in the subtyping relation. One way to express the correspondence is to use a formalism of labels, as proposed by Pottier [15]. In this formalism, a label is associated to each argument of type constructors, the correspondence being expressed by the fact that two arguments of type constructors have the same label. The subtyping order \leq is built from the order $\leq_{\mathcal{K}}$ on type constructors and from the labels. A formal description of the type structure is given in [6], where the structures of types and type constructors are quasi-lattices, i.e. partial orders in which two elements have a least upper (resp. greatest lower) bound if and only if they have an upper (resp. lower) bound.

Subtyping constraints Let \mathcal{W} be a set of *type variables*, or *parameters*, noted α, β, \dots . We note $\mathcal{T}_{\mathcal{W}}$ the set of types built on $\mathcal{K} \cup \mathcal{W}$.

Definition 8 A subtyping constraint is of the form $\tau_1 \leq \tau_2$, where $\tau_1, \tau_2 \in \mathcal{T}_{\mathcal{W}}$ are finite types. A substitution $\rho : \mathcal{W} \rightarrow \mathcal{T}$ satisfies the constraint $\tau_1 \leq \tau_2$, noted $\rho \models \tau_1 \leq \tau_2$, if $\rho(\tau_1) \leq \rho(\tau_2)$. The subtyping constraint $\tau_1 \leq \tau_2$ is satisfiable if there exists a substitution ρ such that $\rho \models \tau_1 \leq \tau_2$.

In [6], sufficient conditions on $(\mathcal{K}, \leq_{\mathcal{K}})$ are given for the decidability of the satisfiability of subtyping constraints in quasi-lattices, this problem is shown to NP-complete, and a practical algorithm (used in section 5) is given for computing explicit solutions.

4.2 Type system for CLP+CHR

In order to support the overloading of CLP function and predicate symbols, we assume that a set $types(f/n)$ of type schemes of the form $\forall \bar{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau$ is associated to each function symbol f/n (resp. predicate symbol p/n), where $\bar{\alpha}$ is the set of parameters occurring in types $\tau_1, \dots, \tau_n, \tau$. These sets of types are supposed to be fixed, for example using declarations provided by the programmer. We also assume that the type of the constraint $=/2$ is the type scheme $\forall \alpha. \alpha \times \alpha$. For the sake of simplicity, the quantification $\forall \bar{\alpha}$ will be omitted in type schemes, each occurrence of a type scheme being renamed with fresh parameters. For a CHR constraint c/n of type $\tau_1 \times \dots \times \tau_n$, $types(c/n)$ is defined as the singleton $\{\tau_1 \times \dots \times \tau_n\}$.

A typing environment is a partial mapping $\Gamma : \mathcal{V} \mapsto \mathcal{T}_{\mathcal{W}}$, also noted $\{X_1 : \tau_1, \dots, X_n : \tau_n\}$. The operation \uplus on typing environments is defined as disjoint union, that is $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_1(X)$ if $X \in dom(\Gamma_1)$, $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_2(X)$ if $X \in dom(\Gamma_2)$, and $(\Gamma_1 \uplus \Gamma_2)(X)$ is undefined otherwise.

Table 2 gives the typing rules for CLP, together with the typing rule for the declaration *label_with*. The typing rules for CLP resemble the rules of Mycroft and O’Keefe [14] with

	$(Var) \frac{X : \tau \in \Gamma}{\Gamma \vdash X : \tau}$	$(Sub) \frac{\Gamma \vdash t : \tau \quad \tau \leq \tau'}{\Gamma \vdash t : \tau'}$
(Func)	$\frac{\Gamma \vdash t_1 : \tau_1 \rho \quad \dots \quad \Gamma \vdash t_n : \tau_n \rho}{\Gamma \vdash f(t_1, \dots, t_n) : \tau \rho}$	ρ is a type substitution $\tau_1 \times \dots \times \tau_n \rightarrow \tau \in types(f/n)$
(Atom)	$\frac{\Gamma \vdash t_1 : \tau_1 \rho \quad \dots \quad \Gamma \vdash t_n : \tau_n \rho}{\Gamma \vdash p(t_1, \dots, t_n) \text{ Atom}}$	ρ is a type substitution $\tau_1 \times \dots \times \tau_n \in types(p/n)$
(Head)	$\frac{\Gamma \vdash t_1 : \tau_1 \rho \quad \dots \quad \Gamma \vdash t_n : \tau_n \rho}{\Gamma \vdash p(t_1, \dots, t_n) \text{ Head}_{\tau_1 \times \dots \times \tau_n}}$	ρ is a type renaming $\tau_1 \times \dots \times \tau_n \in types(p/n)$
(Clause)	$\frac{\forall \sigma \in types(p/n) \quad \Gamma_\sigma \vdash p(t_1, \dots, t_n) \text{ Head}_\sigma \quad \Gamma_\sigma \vdash B_1 \text{ Atom} \quad \dots \quad \Gamma_\sigma \vdash B_k \text{ Atom}}{\vdash p(t_1, \dots, t_n) :- B_1, \dots, B_k \text{ Clause}}$	
(Label with)	$\frac{\Gamma \vdash H \text{ Atom} \quad \Gamma \vdash G \text{ Goal}}{\vdash \text{label_with } H \text{ if } G \text{ Label_with}}$	

Table 2: Type system for CLP and *label_with*

the addition of subtyping and overloading. A predicate call $p(t_1, \dots, t_n)$ (resp. a native constraint) is *well-typed* in a typing environment Γ if $\Gamma \vdash p(t_1, \dots, t_n) \text{ Atom}$ can be derived from the rules. A clause $H :- B$ is *well-typed* if $\vdash H :- B \text{ Clause}$ can be derived from the rules. A labeling declaration *label_with H if G* is well-typed if $\vdash \text{label_with } H \text{ if } G \text{ Label_with}$ can be derived. The *(Sub)* rule gives the semantics of subtyping by expressing that if a term t has type τ , then it has all types that are greater than τ .

The set of rules of tables 1 and 2 define the type system for CLP+CHR. A CLP+CHR program is well-typed if all its CHR rules, all its clauses and all its labeling declarations are well-typed. The rule *(CHR Atom)* can be seen as a particular case of rules *(Atom)* and *(Head)*, where the symbol p/n has only one type scheme without any parameter, which corresponds to the restriction that a CHR constraint has only one type.

The distinction between rules *(Atom)* and *(Head)* expresses the principle of *definitional genericity* [13], which establishes that the type of the head of a clause must be equivalent to, up to renaming but not an instance of, the declared type of the predicate. The rule *(Clause)* imposes that a clause must be well-typed for all possible types of the defined predicate, which can be seen as a condition similar to definitional genericity for overloading. These two conditions are useful for the following subject reduction theorem which expresses the consistency of the type system with reference to the operational semantics of CLP+CHR. It is preceded by a lemma which expresses that in a derivation apart from *(Head)* or *(Clause)*, the types can be arbitrarily instantiated.

Lemma 9 *For any typing environment Γ , for any judgment R different from Head or Clause and any type substitution ρ , if $\Gamma \vdash R$ then $\Gamma\rho \vdash R\rho$.*

Proof: By induction on the derivation. □

Theorem 10 *Let us consider a well-typed CHR+CLP program. Let $\langle F, E, D \rangle$ and $\langle F', E', D' \rangle$ be two states and Γ be a typing environment such that $\Gamma \vdash F, E, D$ Goal. If $\langle F, E, D \rangle \mapsto \langle F', E', D' \rangle$, then there exists a typing environment Γ' such that $\Gamma' \vdash F', E', D'$ Goal. Moreover, if the transition rules requires a guard G then $\Gamma' \vdash G$ Goal.*

Proof: One can check that the assumptions of section 3.1 are correct for the the system of table 2. Moreover, an atom corresponding to a predicate call and an atom corresponding to a native constraint are typed in the same way. Therefore, by theorem 7, if the transition is one of **Solve**, **Introduce** or **Simplify**, then there exists a typing environment te' such that $\Gamma' \vdash F', E', D'$ Goal and $\Gamma' \vdash G$ Goal in case of need.

Let us consider the **Unfold** transition. We can assume, without loss of generality, that $H' = p(s)$ and $H = p(t)$. Since $\Gamma \vdash p(s)$ Atom, there exists a type scheme $\tau \in \text{types}(p)$ and a substitution ρ such that $\Gamma \vdash s : \tau\rho$. Since the program is well-typed, $\vdash H :- B$ Clause, thus there exists a typing environment Γ_τ such that $\Gamma_\tau \vdash B$ Goal and $\Gamma_\tau \vdash p(t)$ Head $_\tau$, that is $\Gamma_\tau \vdash t : \tau\rho_\tau$ where ρ_τ is a renaming of τ . By posing $\rho' = \rho_\tau^{-1}\rho$, and by lemma 9, we obtain $\Gamma_\tau\rho' \vdash t : \tau\rho$ and $\Gamma_\tau\rho' \vdash B$ Goal. By posing $\Gamma' = \Gamma_\tau\rho' \uplus \Gamma$, we obtain $\Gamma' \vdash t = s$ Atom. Thus $\Gamma' \vdash B, F, E, s = t, D$ Goal.

Let us finally consider the case of a **Label** transition. Similarly to the case of the **Unfold** transition, there exists a typing environment Γ' , such that $\Gamma' \vdash B, F, E, t = s, D$. Since H' is a CHR constraint, τ does not contain any parameter, that is ρ'' is the identity substitution. We have $H'' = p(u)$ for some term u . Since $\vdash \text{label_with } H'' \text{ if } G \text{ Label_with}$, there exists a typing environment Γ_{lw} such that $\Gamma_{lw} \vdash G$ Goal and $\Gamma_{lw} \vdash H''$ Atom, that is $\Gamma_{lw} \vdash u : \tau$. By posing $\Gamma'' = \Gamma' \uplus \Gamma_{lw}$, we obtain $\Gamma'' \vdash s = u$ Atom, $\Gamma'' \vdash B, F, E, s = t, D$ and $\Gamma'' \vdash G$ Goal, and thus $\Gamma'' \vdash s = u, G$ Goal. □

5 Experimental Results

The type system for CLP+CHR has been implemented as an extension of the TCLP software [4], which is a type checker for constraint logic programming. Furthermore a type inference algorithms makes it possible to infer types for variables and for program predicates automatically. In a lattice of types with top element *term* however, the type $\text{term} \times \dots \times \text{term}$ is always a possible type for predicates. For this reason, a heuristic type inference algorithm is used, providing a more informative type and often the expected type [7, 5]. This algorithm can also be used to infer the type of CHR constraints that are not declared by the user.

TCLP uses several solvers written in CHR. The main solver is the one for subtyping constraints. We also use a CHR solver to handle overloading of function and predicate

symbols during type checking. Some other small CHR solvers are also used for handling typing environments and preliminary computations on the structure of type constructors. Hence, the possibility to type check CHR programs makes it possible that TCLP type checks its own source code.

The following example shows the typical kind of errors detected by TCLP:

Example 4 *The following solver handles counters. The constraint `cpt/2` associates the name of the counter to its value, and has type $\text{atom} \times \text{int}$.¹ The constraint `val/2` also has type $\text{atom} \times \text{int}$ and constraints `incr/1` and `init/1` have 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).
```

The type checker produces the following message:

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

It is in fact an argument inversion: in the head of the last rule, the arguments of the constraint `cpt` were inverted.

The following example shows the result of type inference on a small solver:

Example 5 *The following solver, taken from [9], computes the greatest common divisor of two numbers.*

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

The type checker infers the following type:

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

that is `gcd` has type int .

Performances The speed of the type checker has been evaluated on ten CHR solvers taken from [9], on the solver for subtyping constraints, on the solver for overloading in TCLP, as well as on the complete TCLP source code. These tests were run on a 2 Ghz Pentium IV with 512 Mo of RAM, using the Sicstus Prolog implementation of TCLP for which the working memory space is limited to 256 Mo. The results are presented in table 3.

The first column indicates the CLP+CHR program. The second column indicates the number of lines of codes in the program and the third one indicates the number of CHR rules in the program. Next, in column “Type checking”, the type checking times are given with

¹The type *atom* corresponds to Prolog atoms, that is symbols of arity 0, and not to the logical atoms.

Program	# lines	# rules	Type check		Type inference	
			CHR	Total	CHR	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	NA	NA
subtyping	595	57	4.52 s	6.22 s	319.66 s	322.64 s
overloading	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

Table 3: Performances

type inference for variables, but without type inference for predicates or CHR constraints. Finally, the column “Type inference” indicates the times for inferring types to predicates and constraints. The typing times for CHR rules are given in columns “CHR”, while the typing times for the whole CLP+CHR programs are given in the column “Total”.

The type checking times without type inference for predicates and constraints show that the type checker is usable in practice. For example, it takes less than 18 s to check the 490 rules of the `allenComp` solver, or less than 54 s to check about 4600 lines of code constituting the source of TCLP.

In presence of subtyping, type inference needs 71 times more CPU time than type checking. In the case of `allenComp`, type inference even fails by lack of memory due to the restriction to 256 Mo. This is due to the fact that, when inferring the type of a constraint, the type checker must consider at the same time all the rules and clauses in a same connected component of the call graph, while type checking can be done rule by rule. CHR solvers often use large connected components however. One reason for this difficulty is that a few constraints used as data structures, appear in the head of numerous rules, thus creating large connected components. For example, the solver for subtyping constraints has a connected component of 54 predicates and CHR constraints. Such connected components thus require to deal with a very large number of subtyping constraints and overloaded symbols at once. Moreover, algorithms for solving subtyping constraints and overloading are potentially exponential [3, 6]. From this point of view, the performances of type inference are quite satisfactory. Type inference can be used the first time a solver is written, the inferred types being used afterwards as declarations during the rest of the development of the solver.

6 Conclusion

We have presented a type system for the *Constraint Handling Rules* CHR language [8], parameterized by the type system of the host language. In the particular case of constraint logic programming, its combination with the prescriptive type system TCLP [7] for CLP languages has been presented. Under the assumption that the well-typedness of native constraints is preserved by logical equivalence, the type system has been proved consistent w.r.t. the operational semantics of CHR and CLP+CHR respectively.

The type system for CLP+CHR is implemented as an extension of the TCLP software [4]. The reported experimental results on ten CHR solvers plus TCLP itself show that the system is already usable and useful.

As for future work, we plan to enhance the performances of type inference with large connected components, and to get some practical experience from the users of the system, in particular for the development of complex modular [10] and/or collaborative CHR solvers. It would also be interesting to study the instantiation of the type system with the one of Java in the framework of the JACK toolkit implementation of CHR [1].

References

- [1] 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.
- [2] Slim 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.
- [3] E. Coquery and F. Fages. Tcpl: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical report, INRIA Rocquencourt, 2002.
- [4] Emmanuel Coquery. TCLP. <http://contraintes.inria.fr/~coquery/tcpl/>.
- [5] Emmanuel Coquery. *Typage et programmation en logique avec contraintes*. PhD thesis, Université Paris 6 - Pierre et Marie Curie, December 2004.
- [6] Emmanuel Coquery and François Fages. Subtyping constraints in quasi-lattices. In P. Pandya and J. Radhakrishnan, editors, *Proceedings of the 23rd conference on foundations of software technology and theoretical computer science, FSTTCS'2003*, Lecture Notes in Computer Science, Mumbai, India, December 2003. Springer-Verlag.
- [7] François Fages and Emmanuel Coquery. Typing constraint logic programs. *Journal of 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] Rémy Haemmerlé and François Fages. Closures are needed for closed module systems. Technical report, INRIA, 2005.
 - [11] 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.
 - [12] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.
 - [13] 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.
 - [14] A. Mycroft and R.A. O’Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
 - [15] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.
 - [16] 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.
 - [17] 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.



Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399