A Type System for CHR

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

[1] INRIA Rocquencourt, Projet Contraintes,
BP 105, F-78153 Le Chesnay, France
[2] Conservatoire National des Arts et Métiers,
292 rue Saint Martin, 75141 Paris Cedex 03, France

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 when 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.

1 Introduction

The language of *Constraint Handling Rules* (CHR) of T. Frühwirth [1] 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 [2], Java [3] or Haskell [4], 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 commited-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 [5, 6] or for handling overloading in functional languages [7] and constraint logic languages [8, 6]. There were however only few works the the typing of CHR itself. In [4], Chin, Sulzmann and Wang propose a monomorphic type system for the embedding of CHR into Haskell.

In this article, we propose a generic type system for CHR inspired by the TCLP type system for constraint logic programs [9]. 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 [9]. We show a subject reduction theorem for the CLP+CHR execution model [1] 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 [10]. Finally, we conclude in section 6.

2 Preliminaries on CHR

Here, we recall the syntax and semantics of CHR, as given in [1]. 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 $s, t, \ldots$ We note $\mathcal{X}$ the domain of native constraints, and $\mathcal{CT}$ its (possibly incomplete) first-order logic theory.

2.1 Syntax

Definition 1. *A CHR rule is either:*

– *a* simplification *rule of the form:*
  $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
– *a* propagation *rule of the form:*
  $H_1, \ldots, H_i$ `==>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
– *or a* simpagation *rule of the form:*
  $H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$

with $i > 0$, $j \geq 0$, $k \geq 0$, $l > 0$ *and* $H_1, \ldots, H_i$ *is a nonempty sequence of CHR constraints, the guard* $G_1, \ldots, G_j$ *being a sequence of native constraints and the body* $B_1, \ldots, B_k$ *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 | 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 $H_{l+1}, \ldots, H_i$ are replaced by the body, while the constraints $H_1, \ldots, H_l$ 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

$$H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i \texttt{ <=> } G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

is just an abbreviation for the simplification rule

$$H_1, \ldots, H_i \texttt{ <=> } G_1, \ldots, G_j \mid H_1, \ldots, H_l, B_1, \ldots, B_k.$$

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

```
reflexivity  @ X=<Y <=> X=Y | true.
antisymetry  @ 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 $\longmapsto$, 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 $c(t_1, \ldots, t_n) = c(t'_1, \ldots, t'_n)$ if $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$. 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: $H_1 \wedge \ldots \wedge H_n = H'_1 \wedge \ldots \wedge H'_n$ if $H_1 = H'_1 \wedge \ldots \wedge H_n = H'_n$.

Definition 2. *Let P be a CHR program. The transition relation $\longmapsto$ 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
$\langle C \wedge F, E, D\rangle \longmapsto \langle F, E, D'\rangle$
if C is a native constraint and $\mathcal{CT} \models (C \wedge D) \Leftrightarrow D'$.

Introduce
$\langle H \wedge F, E, D\rangle \longmapsto \langle F, H \wedge E, D\rangle$
if H is a CHR constraint.

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

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

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

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 [11], Abdennadher gives refined operational semantics that are more faithful to the actual implementation of CHR. In particular the previous behavior 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 is a constraint logic programming language of the class $CLP(\mathcal{X})$ [12], it is possible to tightly integrate CHR to the host language. To this end, Frühwirth [1] 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 $p(t_1, \ldots, t_n)$, where p/n is a program predicate symbol. The declaration *label_with* $c(t_1, \ldots, t_n)$ *if* $G_1, \ldots, G_j$ expresses that $G_1, \ldots, G_j$ 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:*

label_with H *if* $G_1, \ldots, G_j$

where $G_1 \ldots, G_j$ is a conjunction of native constraints.
Clauses are of the form:

H *:-* $B_1, \ldots, B_n$

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

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

Unfold

$\langle H' \wedge F, E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
if $(H$ *:-* $B)$ *is in* P *renamed with fresh variables,*
and H *is not a CHR constraint.*

Label

$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
if $(H$ *:-* $B)$ *and (label_with* H'' *if* $G)$ *are in* P *renamed with fresh variables,*
and $\mathcal{CT} \models D \Rightarrow \exists \bar{x}(H' = H'' \wedge G)$

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 program 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 $C_1 \wedge \ldots \wedge C_n$ of native constraints being well-typed in Γ if for each $i \in \{1, \ldots, n\}$, C_i is well-typed in Γ. We note $\Gamma \vdash_N C$ *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 a set of types $types(c/n)$, each type being of the form $\tau_1 \times \ldots \times \tau_n$. This set of types is assumed

to be fixed, for example using some declarations provided by the programmer. This framework allows the use of parametric polymorphism [13]. A parametric type scheme $\forall\alpha_1 \ldots \alpha_k.\ \tau_1 \times \ldots \times \tau_n$ is represented by the set of all its possible instantiations. For example, declaring that $\mathit{types}(\texttt{append}/3) = \{\mathit{list}(\tau)\times\mathit{list}(\tau)\times \mathit{list}(\tau) \mid \tau \in \mathcal{T}\}$ allows one to give the type $\forall\alpha.\ \mathit{list}(\alpha) \times \mathit{list}(\alpha) \times \mathit{list}(\alpha)$ to the constraint `append`/3.

$$(\mathit{Native}) \quad \frac{\Gamma \vdash_N C\ \mathit{Atom}}{\Gamma \vdash C\ \mathit{Atom}} \qquad \text{if } C \text{ is a native constraint}$$

$$(\mathit{CHR\ Atom}) \quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ \mathit{Atom}} \qquad \begin{array}{r}\text{if } c/n \text{ a CHR constraint}\\ \text{and if } \tau_1 \times \ldots \times \tau_n \in \mathit{types}(c/n)\end{array}$$

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

$$(\mathit{CHR\ Head}) \quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ \mathit{Head}_{\tau_1\times\ldots\times\tau_n}} \qquad \begin{array}{r}\text{if } c/n \text{ a CHR constraint}\\ \text{and if } \tau_1 \times \ldots \times \tau_n \in \mathit{types}(c/n)\end{array}$$

$$(\mathit{MultiHead}) \quad \frac{\Gamma \vdash H_1\ \mathit{Head}_{\sigma_1} \qquad \ldots \qquad \Gamma \vdash H_i\ \mathit{Head}_{\sigma_i}}{\Gamma \vdash H_1, \ldots, H_i\ \mathit{MHead}_{\sigma_1,\ldots,\sigma_i}}$$

$$(\mathit{Simpl\ CHR}) \quad \frac{\begin{array}{ll} & \Gamma_{\sigma_1,\ldots,\sigma_n} \vdash H_1, \ldots, H_n\ \mathit{MHead}_{\sigma_1,\ldots,\sigma_n}\\ \forall(\sigma_1, \ldots, \sigma_n) \in S_1 \times \ldots \times S_n & \Gamma_{\sigma_1,\ldots,\sigma_n} \vdash G_1, \ldots, G_r\ \mathit{Goal}\\ & \Gamma_{\sigma_1,\ldots,\sigma_n} \vdash B_1, \ldots, B_q\ \mathit{Goal}\end{array}}{\vdash H_1, \ldots, H_n\ \texttt{<=>}\ G_1, \ldots, G_r \mid B_1, \ldots, B_q\ \mathit{Rule}}$$

where for all $i \in \{1, \ldots, n\}$, $H_i = c_i(t^i_1, \ldots, t^i_{m_i})$ and $S_i = \mathit{types}(c_i/m_i)$

Table 1. Type system for CHR

The rules of the type system for CHR are given in table 1, where σ's represent types of CHR constraints and S's represent sets of such types.

The typing rules resemble to the rules of Chin, Sulzmann and Wang [4], with the addition of the possibility to have more than one type for a CHR constraint and the abstraction w.r.t. the host type system.

A CHR constraint H is *well-typed* in Γ if the judgment $\Gamma \vdash H\ \mathit{Atom}$ 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 rules (*CHR Head*) and (*MultiHead*), differ from (*CHR Atom*) and (*Goal*) in that they add an annotation for keeping track of the type used for typing each head. The rule (*Simpl CHR*) requires, for each combination $\sigma_1, \ldots, \sigma_n$ of the types of the different occurrences of the CHR constraints of the head of the CHR rule, that the head, the guard and the body of the CHR rule are well-typed in some typing environment $\Gamma_{\sigma_1,\ldots,\sigma_n}$. As shown in section 4.3, in the case of parametric polymorphism, this can be ensured by renaming the type scheme

of each occurrence of CHR constraints in the head with distinct variables, which can be seen as applying the principle of *definitional genericity* [14] to the typing of CHR constraints. In the context of logic programming, this principle 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 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 1. *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 $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$. If there exists a typing environment Γ such that $\Gamma \vdash F, E, D$ Goal, then there exists a typing environment Γ' such that $\Gamma' \vdash F', E', D'$ Goal. Moreover, if the transition rule contains a guard G then $\Gamma' \vdash G$ Goal.*

The following example show the necessity of considering all possible combinations of types when typing a CHR rule with multiple heads.

Example 3. Let us assume that the constraint `=<` has the type scheme $\forall\alpha.\alpha \times \alpha$. Let us consider the polymorphic type $list(\alpha)$ for lists and the types *int* and *string*. We assume that the empty list `[]` has type $\forall\alpha.\ list(\alpha)$, that the list constructor has type $\forall\alpha.\ \alpha \times list(\alpha) \rightarrow \alpha$, and that $list(int)$ and $list(string)$ are incompatible. Then the `transitivity` rule is not well-typed:

```
X=<Y, Y=<Z  ==> X=<Z
```

For example, one might consider the type $list(int) \times list(int)$ for the first occurrence of `=<`/2 and the type $list(string) \times list(string)$ for the second one, in which case the head is not well-typed because `Y` can not have both types $list(int)$ and $list(string)$.

The rule can produce an ill-typed state from a well-typed one. The state $\langle true, \texttt{["a"] =< []} \wedge \texttt{[] =< [1]}, true\rangle$ is well-typed. However the rule would add $\texttt{X = ["a"]} \wedge \texttt{Z = [1]} \wedge \texttt{X =< Z}$ to the current goal. This subgoal is not well-typed because, when typing `X =< Z`, the type chosen for `=<`/2 must be compatible both with $list(string)$ and $list(int)$.

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 [9]. 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 $boolean \leq int$ 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 $list(\alpha) \leq term$ allows one to see homogeneous lists as terms and to apply decomposition predicates to them, such as `functor`/3, `arg`/3 or `=..`/2.

In [9], 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 [9], or, in the case of the equality constraint, by using modes to fix the dataflow [15].

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 [16]. 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 [5], 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 $\alpha, \beta, \ldots$. We note $\mathcal{T}_{\mathcal{W}}$ the set of types built on $\mathcal{K} \cup \mathcal{W}$.

Definition 6. *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 $\rho : \mathcal{W} \to \mathcal{T}$* 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 [5], 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 $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ is associated to each function symbol f/n (resp. predicate symbol p/n), where $\bar{\alpha}$ is the set of parameters occurring in types $\tau_1, \ldots, \tau_n, \tau$. 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.

A typing environment is a partial mapping $\Gamma : \mathcal{V} \mapsto \mathcal{T}_{\mathcal{W}}$, also noted $\{X_1 : \tau_1, \ldots, X_n : \tau_n\}$. 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.

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

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

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

$$(Head)\quad \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash p(t_1, \ldots, t_n)\ Head_{\tau_1 \times \ldots \times \tau_n}} \quad \begin{array}{r}\rho \text{ is a type renaming} \\ \tau_1 \times \ldots \tau_n \in types(p/n)\end{array}$$

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

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

Table 2. Type system for CLP and *label_with*

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 [13] with the addition of subtyping and overloading. A predicate

call $p(t_1, \ldots, t_n)$ (resp. a native constraint) is *well-typed* in a typing environment Γ if $\Gamma \vdash p(t_1, \ldots, t_n)$ *Atom* can be derived from the rules. A clause H `:-` B is *well-typed* if $\vdash H$ `:-` B *Clause* can be derived from the rules. A labeling declaration *label_with H if G* is well-typed if $\vdash$ *label_with H if G 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 distinction between rules (*Atom*) and (*Head*) expresses the principle of *definitional genericity* [14], 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, in a typing environment Γ_σ that depends on the considered type σ. This 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.

Theorem 2. *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* $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$*, then there exists a typing environment* Γ' *such that* $\Gamma' \vdash F', E', D'$ *Goal. Moreover, if the transition rules contains a guard* G *then* $\Gamma' \vdash G$ *Goal.*

4.3 Typing of polymorphic CHR constraints

In this section, we show how to type check CHR constraints in the presence of parametric polymorphism. The set $types(c/n)$ of the types of the constraint c/n is restricted to a finite set of type schemes of the form $\forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n$. This does not mean that $types(c/n)$ itself is finite, as a type scheme represent an infinite set of types. More precisely, we assume a finite set $types_p(c/n)$ of type schemes and define $types(c/n) = \{\tau_1\rho \times \ldots \times \tau_n\rho \mid \forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n \in types_p(c/n)$ and $\rho : \bar{\alpha} \to \mathcal{T}\}$.

A type system that deals directly with parametric polymorphism can be obtained by replacing rules (*CHR Atom*), (*CHR Head*) and (*MultiHead*) by their counterparts given in table 3 and by replacing $types(c_i/m_i)$ is replaced by $types_p(c_i/m_i)$ in rule (*Simpl CHR*). The resulting type system is noted $\vdash_p$. The following proposition expresses the equivalence of the two type systems:

Proposition 1. *A CHR rule (resp. goal) is well-typed in* $\vdash$ *if and only if it is well-typed in* $\vdash_p$*.*

5 Experimental Results

The type system for CLP+CHR has been implemented as an extension of the TCLP software [10], which is a type checker for constraint logic programming.

$$(\mathit{CHR\ Atom}')\ \frac{\Gamma \vdash t_1 : \tau_1\rho \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash c(\tau_1, \ldots \tau_n)\ \mathit{Atom}} \quad \begin{array}{r}\rho \text{ is a type substitution} \\ \forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n \in \mathit{types}_p(c/n)\end{array}$$

$$(\mathit{CHR\ Head}_p)\ \frac{\Gamma \vdash t_1 : \tau_1\rho \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash c(\tau_1, \ldots \tau_n)\ \mathit{Head}_{\tau_1 \times \ldots \times \tau_n, \rho}} \quad \begin{array}{r}\rho \text{ is a type renaming} \\ \forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n \in \mathit{types}_p(c/n)\end{array}$$

$$(\mathit{MultiHead}_p)\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, n\}\ \Gamma \vdash H_i\ \mathit{Head}_{\sigma_i, \rho_i} \\ \forall\ 1 \leq i < j \leq n\ \mathit{codom}(\rho_i) \cap \mathit{codom}(\rho_j) = \emptyset\end{array}}{\Gamma \vdash H_1, \ldots, H_n\ \mathit{MHead}_{\sigma_1, \ldots, \sigma_n}}$$

Table 3. Typing rules for polymorphic CHR constraints

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 $\mathit{term} \times \ldots \times \mathit{term}$ 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 [9, 6]. 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 $\mathit{atom} \times \mathit{int}$.[3] The constraint `val`/2 also has type $\mathit{atom} \times \mathit{int}$ 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:

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

Example 5. The following solver, taken from [17], 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*.

Performance The speed of the type checker has been evaluated on ten CHR solvers taken from [17], 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 4.

			Type check		Type inference	
Program	# lines	# rules	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	8.93 s (99.58 s)	9.03 s (99.69 s)
domain	266	84	4.30 s	6.42 s	5.35 s (183.92 s)	7.75 s (186.94 s)
fourier-gauss	328	30	1.98 s	5.88 s	6.01 s (19.04 s)	16.16 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	9.96 s (319.66 s)	15.28 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	26.61 s (416.08 s)	96.09 s (518.39 s)

Table 4. Performance

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 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 times given between parenthesis are obtained without breaking connected components as explained in the following.

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 [8, 5]. From this point of view, the performance of type inference are quite satisfactory.

It is possible to reduce type inference type by breaking such connected components. This can be done by providing the type of the constraints that are used as data structures. This technique appear to be very efficient, reducing the time for type inference in `domain` from 184 s to 5.3 s, just by giving the type of one constraint. When no time is given between parenthesis, it means that the solver was already well stratified and thus didn't need the type for some CHR constraint to be given. Moreover, 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 [1], 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 [9] 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 [10]. 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 get some practical experience from the users of the system, in particular for the development of complex modular [18] 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 [3] and as well as with the Haskell implementation [4].

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
2. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules compiler and runtime system. Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules **14** (2000)
3. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java Constraint Kit. In: Electronic Notes in Theoretical Computer Science. Volume 64. Elsevier (2000)
4. Chin, W.N., Sulzmann, M., Wang, M.: A type-safe embedding of constraint handling rules into Haskell. Technical report, National University of Singapore (2003) `http://www.comp.nus.edu.sg/~sulzmann/chr/hchr/hchr-tr.ps`.
5. Coquery, E., Fages, F.: Subtyping constraints in quasi-lattices. In Pandya, P., Radhakrishnan, J., eds.: Proceedings of the 23rd conference on foundations of software technology and theoretical computer science, FSTTCS'2003. Lecture Notes in Computer Science, Mumbai, India, Springer-Verlag (2003)
6. Coquery, E.: Typage et programmation en logique avec contraintes. PhD thesis, Université Paris 6 - Pierre et Marie Curie (2004)
7. Stuckey, P.J., Sulzmann, M.: A theory of overloading. In Peyton-Jones, S., ed.: Proceedings of the International Conference on Functional Programming, ACM Press (2002) 167–178
8. Coquery, E., Fages, F.: Tclp: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical Report RR-4926, INRIA Rocquencourt (2002)
9. Fages, F., Coquery, E.: Typing constraint logic programs. Journal of Theory and Practice of Logic Programming **1** (2001) 751–777
10. Coquery, E.: TCLP (2003) `http//contraintes.inria.fr/~coquery/tclp/`.
11. Abdennadher, S.: 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., Linz, Springer-Verlag (1997) 252–266
12. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany, ACM (1987) 111–119
13. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial Intelligence **23** (1984) 295–307
14. Lakshman, T., Reddy, U.: Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Saraswat, V., Ueda, K., eds.: Proceedings of the 1991 International Symposium on Logic Programming, MIT Press (1991) 202–217
15. Smaus, J.G., Fages, F., Deransart, P.: 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. Pottier, F.: A versatile constraint-based type inference system. Nordic Journal of Computing **7** (2000) 312–347
17. Frühwirth, T., Schrijvers, T.: (CHR web page) `http//www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
18. Haemmerlé, R., Fages, F.: Closures are needed for closed module systems. Technical Report RR-5575, INRIA (2005)