Modular CHR with *ask* and *tell*

François Fages, Cleyton Mario de Oliveira Rodrigues, Thierry Martinez

Contraintes Project–Team, INRIA Paris–Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
http://contraintes.inria.fr

Abstract. In this paper, we introduce a modular version of the Constraint Handling Rules language CHR, called CHRat for modular CHR with *ask* and *tell*. Any constraint defined in a CHRat component can be reused both in rules and guards in another CHRat component to define new constraint solvers. Unlike previous work on modular CHR, our approach is completely general as it does not rely on an automatic derivation of conditions for checking entailment in guards, but on a programming discipline for defining both satisfiability (*tell*) and entailment (*ask*) checks by CHRat rules for each constraint. We define the operational and declarative semantics of CHRat, provide a transformation of CHRat components to flat CHR programs, and prove the preservation of the semantics. We then provide examples of the modularization of classical CHR constraint solvers and of the definition of complex constraint solvers in a modular fashion.

1 Introduction

The Constraint Handling Rules language CHR was introduced nearly two decades ago as a declarative language for defining constraint solvers by multiset rewriting rules with guards assuming some built-in constraints [1]. The CHR programming paradigm resolves implementing a constraint system into the declaration of guarded rewriting rules, that transform the store into a solved form allowing to decide the satisfiability. Each transformation is supposed to preserve the satisfiability of the system, and the solved form, reached when no more transformation can be applied, is unsatisfiable if it contains the constraint "false", and is operationnally satisfiable otherwise. One important, but not mandatory, property of these transformations is *confluence* which means that the solved form is always independent of the order of application of the rules, and is in fact a *normal form* for the initial constraint store [2].

Since then, CHR has evolved to a general purpose rule-based programming language [1] with some extensions such as for the handling of disjunctions [3] or for introducing types [4]. However, one main drawback of CHR as a language for defining constraint solvers, is the absence of *modularity*. Once a constraint system is defined in CHR with some built-in constraints, this constraint system cannot be reused in another CHR program taking the defined constraints as new built-in constraints. The reason for this difficulty is that a CHR program defines

a satisfiability check but not the *constraint entailment* check that is required in guards.

Previous approaches to this problem have studied conditions under which one can derive automatically an entailment check from a satisfiability check. In [5] such conditions are given based on the logical equivalence:

$$D \models C \rightarrow c \Leftrightarrow D \models (C \wedge c) \leftrightarrow C$$

In this paper, we propose a different paradigm for *modular* CHR, called CHR with *ask* and *tell*, and denoted CHRat. This paradigm is inspired by the framework of concurrent constraint programming [6, 7]. The programming discipline in CHRat for programming modular constraint solvers is to enforce, for each constraint c, the definition of simplification and propagation rules for the constraint tokens **ask**(c) and **entailed**(c). Solvers for *asks* and *tells* are already required for the built-in constraint system implementation [8]; the discipline we propose consists in the internalization of this requirement in the CHR solver itself. A constraint c is *operationally entailed* in a constraint store containing **ask**(c) when its solved form contains the token **entailed**(c). Beside the simpagation rule c **ask**(c) $\Longrightarrow$ **entailed**(c) which will be always assumed to provide a minimalist entailment-solver, arbitrarily complex entailment checks can be programmed with rules, as opposed to event-driven imperative programming[9]. With this programming discipline, CHRat constraints can be reused both in rules and guards in other components to define new constraint solvers.

In the next section, we illustrate this approach with a simple example. Then we define the syntax and declarative semantics of CHRat. Section 4 describes the transformation of CHRat programs into flat CHR programs and proves its correctness. Section 5 provides examples of the modularization of classical CHR constraint solvers and of the definition of complex constraint solvers in a modular fashion. Finally we conclude with a discussion on the simplicity and expressiveness of this approach and its current limitation to non-quantified constraints.

2 Introductory Example

2.1 CHRat Components for leq/2 and min/3

We begin with the pedagogical CHR constraint solver for ordering relations. This solver defines the CHR constraint leq/2. The first task is to define, as usual, the satisfiability solver associated to this constraint: this is done by the following four rules. The first three rules translate the axioms for ordering relations, and the rule redundant gives set semantics to the constraint leq/2.

File leq_solver.cat

```
component leq_solver.
export leq/2.
reflexive     @ leq(X,X) <=> true.
antisymmetric @ leq(X,Y), leq(Y,X) <=> X = Y.
transitive    @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
redundant     @ leq(X,Y) \ leq(X,Y) <=> true.
```

There is a second task for defining a constraint solver in CHRat: the definition of rules for checking the entailment of leq(X, Y) constraint. These rules have to rewrite the constraint token **ask**(leq(X, Y)) into the constraint token **entailed**(leq(X, Y)). The rule leq(X,Y) \ **ask**(leq(X,Y)) $\Longleftrightarrow$**entailed**(leq(X,Y)) is always assumed and provides a minimalist entailment-solver for free. In this simple example, since checking leq(X, Y) for $X \neq Y$ is directly observable in the store, there is only a single rule to add for the reflexivity.

```
reflexiveAsk @ ask(leq(X,X)) <=> entailed(leq(X,X)).
```

The satisfiability solver and the entailment solver together define a CHRat component for the CHR-constraint leq(X, Y). Our implementation of CHRat relies on a simple atom-based component separation mechanism: there is a component by file; exported CHR-constraints are prefixed with the name of the component; and the choice for the prefixes of internal CHR-constraints is done so as to avoid collisions.

Such a component can then be used to define new constraint solvers using the leq(X, Y) constraint both in rules and guards. For instance, a component for the minimum constraint min(X,Y,Z), stating that Z is the minimum value among X and Y, can be defined in CHRat as follows:

File min_solver.cat

```
component min_solver.
import leq/2 from leq_solver.
export min/3.
minLeft     @ min(X,Y,Z) <=> leq(X,Y) | Z=X.
minRight    @ min(X,Y,Z) <=> leq(Y,X) | Z=Y.
minGen      @ min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).

minAskLeft  @ ask(min(X, Y, X)) <=> leq(X, Y) |
                  entailed(min(X, Y, X)).
minAskRight @ ask(min(X, Y, Y)) <=> leq(Y, X) |
                  entailed(min(X, Y, Y)).
```

The three first rules describe the satisfiability check for min(X,Y,Z). The relevant rules to be discussed are the minAskLeft and minAskRight: it is worth noticing that the entailment of min(X,Y,Z) can be stated if, and only if, Z is already known to be equal to X or Y.

2.2 Transformation to a Flat CHR Program

The guards in CHR rules are restricted to built-in constraints [1]. In order to translate CHRat programs into CHR programs, we proceed with a program transformation which removes all the user defined constraints from the guard. This transformation also renames the constraints **ask**(constraint) to ask_constraint and **entailed**(constraint) to entailed_constraint . The resulting CHR program for the min(X, Y, Z) CHRat solver is the following:

```
min-auto-ask      @ min(X,Y,Z) \ ask_min(X,Y,Z) ⟹ entailed_min(X,Y,Z).
minLeft-ask       @ min(X,Y,Z) ⟹ ask_leq(X,Y).
minLeft-fire      @ entailed_leq(X,Y), min(X,Y,Z) ⟺ Z=X.
minRight-ask      @ min(X,Y,Z) ⟹ ask_leq(Y,X).
minRight-fire     @ entailed_leq(Y,X), min(X,Y,Z) ⟺ Z=Y.
minGen            @ min(X,Y,Z) ⟹ leq(Z,X), leq(Z,Y).
minAskLeft-ask    @ ask_min(X,Y,X) ⟹ ask_leq(X,Y).
minAskLeft-fire   @ entailed_leq(X,Y), ask_min(X,Y,X) ⟺
                      entailed_min(X,Y,X).
minAskRight-ask   @ ask_min(X,Y,Y) ⟹ ask_leq(Y,X).
minAskRight-fire  @ entailed_leq(Y,X), ask_min(X,Y,Y) ⟺
                      entailed_min(X,Y,Y).
```

It is worth noting that the transformed program can be executed with any regular CHR implementation [10, 11].

3 Syntax and Semantics of CHRat Components

Let V be a countable set of variables. Let $\mathrm{fv}(e)$ denotes the set of free variables of a formula e.

3.1 Syntax

Definition 1. *A* built-in constraint system *is a pair* $(\mathcal{C}, \vdash_{\mathcal{C}})$*, where:*

- $\mathcal{C}$ *is a set of formulas over the variables* V*, closed by logical operators and quantifiers;*
- $\Vdash_{\mathcal{C}} \subseteq \mathcal{C}^2$ *defines the non-logical axioms of the constraint system;*
- $\vdash_{\mathcal{C}}$ *is the least subset of* $\mathcal{C}^2$ *containing* $\Vdash_{\mathcal{C}}$ *and closed by the logical rules.*

Let $(\mathcal{C}, \vdash_{\mathcal{C}})$ be a built-in constraint system over a domain $\mathcal{D}$ with variables V, assumed to contain the standard axiom schemas for equality.

Let $\mathcal{T}$ be a set of constraint *tokens* of the form $c(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n \in \mathcal{D}$ and disjoint from $\mathcal{C}$. We suppose that for any $t \in \mathcal{T}$, $\mathrm{ask}(t) \notin \mathcal{T}$ and $\mathrm{entailed}(t) \notin \mathcal{T}$, and we define

$$\mathcal{T}_a \doteq \{\mathrm{ask}(t) \mid t \in \mathcal{T}\}$$
$$\mathcal{T}_e \doteq \{\mathrm{entailed}(t) \mid t \in \mathcal{T}\}$$

$\mathcal{T}$, $\mathcal{T}_a$, $\mathcal{T}_e$ and $\mathcal{C}$ are thus pairwise disjoint. Let:

$$\mathcal{T}^\bullet \doteq \mathcal{T} \uplus \mathcal{T}_a \uplus \mathcal{T}_e$$

where $\uplus$ denotes disjoint set union.

$\mathrm{ask}^*(\cdot)$ and $\mathrm{entailed}^*(\cdot)$ are the homomorphic extensions of $\mathrm{ask}(\cdot)$ and $\mathrm{entailed}(\cdot)$ respectively to functions from multisets to multisets.

Definition 2. *A* CHRat rule *is of one of the three forms that follow:*

Simplification *rule-name* @ $H \Leftrightarrow G \mid B$.
Propagation *rule-name* @ $H \Rightarrow G \mid B$.
Simpagation *rule-name* @ $H \setminus H' \Leftrightarrow G \mid B$.

where

- *rule-name is an optional name for the rule;*
- *the* heads H *and* H' *are non-empty multisets of elements of* $\mathcal{T} \uplus \mathcal{T}_a \uplus \mathcal{C}$*;*
- *the* guard G *is a multiset of elements of* $\mathcal{T} \uplus \mathcal{C}$*;*
- *the* body B *is a multiset of elements of* $\mathcal{T} \uplus \mathcal{T}_e \uplus \mathcal{C}$.

In guards, the built-in constraints will be distinguished from the user-defined constraints. For a guard C, we write $C_{\text{built-in}} = C \cap \mathcal{C}$ and $C_{\text{CHR}} = C \cap \mathcal{T}$.

Definition 3. *A* CHRat program *is a tuple* $(\{r_1, \ldots, r_n\}, \Sigma)$ *where* $r_1, \ldots, r_n$ *are* CHRat *rules, and* Σ *is the signature of* $\mathcal{T}$*, with the following side condition: for every rule, all variables which appear in the* CHR*-constraint part of the guard* C_{CHR}*, also appear in the head or in the built-in constraints of the guard.*

Remark 1. It is worth noticing that the restriction for guards in CHRat only concerns the CHR-constraint part. In particular, since a CHR program has no CHR-constraint in its guards, every CHR program is a valid CHRat program.

In principle, CHRat programs for *ask* should satisfy some further properties. Putting an ask(·) token should indeed never lead to a failure, and an ask solver should restrict its interaction with the store such that, as far as other components are concerned, only consumption of ask(·) tokens and addition of entailed(·) tokens can be observed, in particular rules for *ask* should not add *tell* constraints to the store. However, the formal semantics described in the following sections will not assume these further restrictions.

As usual, and without loss of generality, we will focus on slightly generalized simpagation rules where one of the heads can be empty. Simplification rules and propagation rules will then be mapped to simpagation rules, by assuming that left heads are empty in translations of simplification rules, and that right heads are empty in translations of propagation rules.

3.2 Operational Semantics

As usual, the operational semantics of CHRat is defined as a transition system between states, called configurations, defined as for CHR [1] by:

Definition 4. *A* configuration *is a tuple* $\langle F, E, D \rangle_{\mathcal{V}}$ *where:*

- *the query* F *is a multiset of elements from* $\mathcal{C} \uplus \mathcal{T}^{\bullet}$*;*
- *the* CHRat *constraint store* E *is a multiset of elements from* $\mathcal{T}^{\bullet}$*;*
- *the built-in store* D *is an element of* $\mathcal{C}$*;*
- $\mathcal{V} \subset V$ *is the set of variables of the initial query.*

Let $\mathfrak{C}$ denotes the set of all configurations.

Definition 5. *We distinguish some relevant configurations:*

- *an* initial *configuration is of the form* $\langle F, \emptyset, \text{true}\rangle_{\mathcal{V}}$ *where* $\mathcal{V} = \text{fv}\,(F)$*;*
- *a* failed *configuration is of the form* $\langle F, E, D\rangle_{\mathcal{V}}$ *where* $D \vdash_{\mathcal{C}} \text{false}$*;*
- *a* successful *configuration is of the form* $\langle \emptyset, E, D\rangle_{\mathcal{V}}$ *where* $D \not\vdash_{\mathcal{C}} \text{false}$*.*

The set of variables of the initial query is written in the configuration to keep these variables free when we consider the logical meaning of the configuration:

Definition 6. *The* logical meaning *of a configuration:*

$$\langle F, E, D\rangle_{\mathcal{V}}$$

is:

$$\exists \boldsymbol{y}(F \wedge E \wedge D)$$

where $\boldsymbol{y}$ *enumerates* $\text{fv}\,(F, E, D) \setminus \mathcal{V}$*.*

Definition 7. *Let* P *be a* CHRat *program. The* transition relation $\mapsto \subseteq \mathfrak{C}^2$ *is the least binary relation closed by the following induction rules:*

Solve

$$\frac{c \in \mathcal{C}}{\langle \{c\} \uplus F, E, D\rangle_{\mathcal{V}} \mapsto \langle F, E, c \wedge D\rangle_{\mathcal{V}}}$$

Introduce

$$\frac{t \in \mathcal{T}^{\bullet}}{\langle \{t\} \uplus F, E, D\rangle_{\mathcal{V}} \mapsto \langle F, \{t\} \uplus E, D\rangle_{\mathcal{V}}}$$

Trivial Entailment

$$\frac{t \in \mathcal{T}}{\langle F, \{\text{ask}(t), t\} \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle \{\text{entailed}(t)\} \uplus F, \{t\} \uplus E, D\rangle_{\mathcal{V}}}$$

Ask

$$\frac{(H \setminus H' \Leftrightarrow C_{\text{built-in}}, C_{\text{CHR}} \mid B.)\,\sigma \in P \qquad D \vdash_{\mathcal{C}} C_{\text{built-in}}}{\langle F, H \uplus H' \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle \text{ask}^*(C_{\text{CHR}}) \uplus F, H \uplus H' \uplus E, D\rangle_{\mathcal{V}}}$$

Fire

$$\frac{(H \setminus H' \Leftrightarrow C_{\text{built-in}}, C_{\text{CHR}} \mid B.)\,\sigma \in P \qquad D \vdash_{\mathcal{C}} C_{\text{built-in}}}{\langle F, H \uplus H' \uplus \text{entailed}^*(C_{\text{CHR}}) \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle B \uplus F, H \uplus E, D\rangle_{\mathcal{V}}}$$

where σ *denotes some variable substitution: the support of* σ *consists of the free variables appearing in the rule which* σ *is applied to; in the* ***Ask*** *rule, variables which do not appear in* H*,* H'*,* $C_{\text{built-in}}$ *have to be mapped to fresh variables; in the* ***Fire*** *rule, variables which do not appear in* H*,* H'*,* $C_{\text{built-in}}$*,* C_{CHR} *have to be mapped to fresh variables.*

Whereas a CHR rule is reduced in only one step, CHRat reduces it in two steps: first, if the heads and builtin guards match, ask solvers are awoken with ask(·) tokens (**Ask** rule); then, when all ask solvers have answered positively to the guard with entailed(·) tokens, the body of the rule is fired (**Fire** rule). Unlike deep guards [12], asks are thus checked in CHRat in the same constraint store as tells.

Definition 8. *A* computation of a goal G *is a sequence* $S_0, S_1, \ldots$ *of configurations with* $S_i \mapsto S_{i+1}$*, beginning with* $S_0 = \langle G, \emptyset, \text{true} \rangle_{\mathcal{V}}$ *and ending in a final configuration or diverging. A finite computation is* successful *if the final configuration is successful. It is* failed *otherwise. The logical meaning of the final configuration of a finite computation is called the* answer *of the computation.*

3.3 Declarative Semantics

CHR programs enjoy a logical semantics that is better suited than the operational semantics to reason about programs and establish program equivalence for instance. In this section, we show that this logical reading of the rules applies as well to CHRat programs.

Let $\mathcal{C}^\bullet$ be the closure of $\mathcal{C} \uplus \mathcal{T}^\bullet$ by logical operators and quantifiers, and let $\vdash_{\mathcal{C}^\bullet}$ be the logical extension of $\vdash_{\mathcal{C}}$ to $\mathcal{T}^\bullet$ with equality and no other non-logical axiom. More precisely, $\vdash_{\mathcal{C}^\bullet}$ is the closure of $\Vdash_{\mathcal{C}^\bullet}$ by logical rules with:

$$\Vdash_{\mathcal{C}^\bullet} \doteq \Vdash_{\mathcal{C}} \uplus \left\{ \begin{array}{r} (c(x_1, \ldots, x_n), c(x'_1, \ldots, x'_n)) \in (\mathcal{T}^\bullet)^2 \\ \mid \vdash_{\mathcal{C}} x_1 = x'_1 \wedge \cdots \wedge x_n = x'_n \end{array} \right\}$$

Definition 9. *Let:*

$$(\cdot)^\ddagger : \mathsf{CHRat} \to \mathcal{C}^\bullet$$

be defined for CHRat *rules as follows:*

$$\begin{aligned} (\mathit{rule} \ @ \ H \ \setminus \ H' \ \Leftrightarrow \ C_{\text{built-in}}, C_{\text{CHR}} \mid B.)^\ddagger \doteq \\ \forall \boldsymbol{y}(C_{\text{built-in}} \to \overline{H} \wedge \overline{H'} \to \overline{\text{ask}^*(C_{\text{CHR}})}) \\ \wedge \forall \boldsymbol{y}(C_{\text{built-in}} \to (\overline{H} \wedge \overline{H'} \wedge \overline{\text{entailed}^*(C_{\text{CHR}})} \leftrightarrow \exists \boldsymbol{y}'(\overline{H} \wedge \overline{B}))) \end{aligned}$$

where:

- $\boldsymbol{y}$ *enumerates the variables occurring in the head and the guard, and* $\boldsymbol{y}'$ *enumerates the other variables occurring in the body (without occurring neither in the head nor in the guard);*
- *for each multiset of constraints* $S = \{c_1, \ldots, c_n\}$*,* $\overline{S}$ *denotes the constraint* $c_1 \wedge \cdots \wedge c_n$*;*

The declarative semantics *of a program* $P \doteq (\{r_1, \ldots, r_n\}, \Sigma)$ *is:*

$$(P)^\ddagger \doteq \left(\bigwedge_{1 \leq i \leq n} (r_i)^\ddagger \right) \wedge \left(\bigwedge_{(f/k) \in \Sigma} \begin{array}{r} \forall \boldsymbol{x}(f(x_1, \ldots, x_k) \to \\ (\text{ask}(f(x_1, \ldots, x_k)) \leftrightarrow \text{entailed}(f(x_1, \ldots, x_k)))) \end{array} \right)$$

Remark 2. Let $(\cdot)^\dagger$ denote the usual CHR declarative semantics. For all CHR program P, we have $\vdash_{\mathcal{C}^\bullet} (P)^\ddagger \leftrightarrow (P)^\dagger$: CHR is thus a proper sublanguage of CHRat and CHRat, both syntactically and semantically.

The fundamental link between the operational and the declarative semantics is stated by:

Lemma 1. *Let P be a* CHRat *program and $S \mapsto S'$ be a transition. Let C and C' denote the logical reading of S and S' respectively. We have:*

$$(P)^{\ddagger} \vdash_{\mathcal{C}^\bullet} \forall \boldsymbol{x}(C \leftrightarrow C')$$

where $\boldsymbol{x}$ enumerates $fv(C) \cup fv(C')$.

Proof. Case analysis over the kind of the transition $S \mapsto S'$:

- immediate for **Solve** and **Introduce**; **Trivial Entailment** derives from $(P)^{\ddagger} \vdash_{\mathcal{C}^\bullet} c \rightarrow (\mathrm{ask}(c) \leftrightarrow \mathrm{entailed}(c))$ for all $c \in \mathcal{T}$;
- **Ask** and **Fire** derive from the logical translation of the CHRat rule which they are applied to.

This result is the direct translation for CHRat of Fruhwirth's soundness and completeness results for CHR. As such, this lemma entails the soundness and completeness of the operational semantics with respect to the declarative semantics:

Theorem 1 (Soundness). *Let P be a* CHRat *program and G be a goal. If G has a computation with answer C then:*

$$(P)^{\ddagger} \vdash_{\mathcal{C}^\bullet} \forall \boldsymbol{x}(C \leftrightarrow G)$$

where $\boldsymbol{x}$ enumerates free variables of C and G.

Theorem 2 (Completeness). *Let P be a* CHRat *program and G be a goal with at least one finite computation. For all conjunctions of constraints C such that $(P)^{\ddagger} \vdash_{\mathcal{C}^\bullet} \forall \boldsymbol{x}(C \leftrightarrow G)$, there exists a computation of answer C' from G such that:*

$$(P)^{\ddagger} \vdash_{\mathcal{C}^\bullet} \forall \boldsymbol{x}(C \leftrightarrow C')$$

where $\boldsymbol{x}$ enumerates free variables of C and C'.

4 Program Transformation of CHRat to CHR

Definition 10. *Let $[\![\cdot]\!] : \mathsf{CHRat} \rightarrow \mathsf{CHR}$ be defined for every* CHRat *rule by $[\![\cdot]\!]$ as follows:*

$$[\![\mathit{rule} \ @ \ H \ \setminus \ H' \ \Leftrightarrow \ C_{\text{built-in}}, C_{\text{CHR}} \mid B.]\!] \doteq \begin{cases} \mathit{rule\text{-}ask} \ @ \ H, H' \ \Rightarrow \ C_{\text{built-in}} \mid \mathrm{ask}^*(C_{\text{CHR}}). \\ \mathit{rule\text{-}fire} \ @ \ H \ \setminus \ H', \mathrm{entailed}^*(C_{\text{CHR}}) \ \Leftrightarrow \ C_{\text{built-in}} \mid B. \end{cases} \quad (1)$$

Transformations for simplification and propagation rules follow from 1 by immediate specialization. The image of a whole CHRat *program* (R, Σ) *by* $[\![\cdot]\!]$ *is the concatenation of images of the individual rules, with the propagation rules:*

$$f(x_1, \ldots, x_k) \;\Rightarrow\; \text{entailed}(f(x_1, \ldots, x_k)).$$

implicitly added for each constraint declaration $(f/k) \in \Sigma$*, if such a rule was not already written by the user in* R*.*

To take benefits of first functor symbol indexing, instead of using compound terms **ask** (...) or **entailed** (...) in the implementation, we rather prefix the constraint symbol in the generated CHR code: constraint (x, y, z) becomes ask_constraint (x, y, z) and entailed_constraint (x, y, z). We could otherwise relying upon automatic program transformations to make this optimization [13].

It is worth noticing that the first and the second CHR rules produced for each CHRat rule respectively follow the right and the left operands of $\wedge$ in the declarative semantics of this rule. That leads to the following result which states the soundness of the transformation:

Theorem 3. *For all* CHRat *program* P*, we have:*

$$\vdash_{\mathcal{C}^\bullet} (P)^\ddagger \leftrightarrow ([\![P]\!])^\dagger$$

where $(P)^\ddagger$ *is the* CHRat *declarative semantics of* P *(see definition 9) and* $(\cdot)^\dagger$ *denotes the usual* CHR *declarative semantics (see remark 2)*

Proof. Let $P = (\{r_1, \ldots, r_n\}, \Sigma)$. According to the definitions:

$$\vdash_{\mathcal{C}^\bullet} (P)^\ddagger \leftrightarrow \left(\bigwedge_{1 \le i \le n} (r_i)^\ddagger \right) \wedge e \text{ and } \vdash_{\mathcal{C}^\bullet} ([\![P]\!])^\dagger \leftrightarrow \left(\bigwedge_{1 \le i \le n} ([\![r_i]\!])^\dagger \right) \wedge e$$

where:

$$e \doteq \left(\bigwedge_{(f/k) \in \Sigma} \begin{array}{l} \forall \boldsymbol{x}(f(x_1, \ldots, x_k) \rightarrow \\ \quad (\text{ask}(f(x_1, \ldots, x_k)) \leftrightarrow \text{entailed}(f(x_1, \ldots, x_k)))) \end{array} \right)$$

Then it suffices to show that, for all $1 \le i \le n$, $\vdash_{\mathcal{C}^\bullet} (r_i)^\ddagger \leftrightarrow ([\![r_i]\!])^\dagger$. Let r_i be the simpagation rule:

$$H \;\setminus\; H' \;\Leftrightarrow\; C_{\text{built-in}}, C_{\text{CHR}} \mid B.$$

then:

$$\begin{array}{rl} [\![r_i]\!] = \{ & (H, \text{entailed}^*(C_{\text{CHR}}) \;\setminus\; H' \;\Leftrightarrow\; C_{\text{built-in}} \mid B.)\,, \\ & (H, H' \;\Rightarrow\; C_{\text{built-in}} \mid \text{ask}^*(C_{\text{CHR}}).)\} \end{array}$$

then we have:

$$\begin{array}{rl} ([\![r_i]\!])^\dagger = & \forall \boldsymbol{x} \left(C_{\text{built-in}} \rightarrow \overline{H} \wedge \overline{H'} \rightarrow \overline{\text{ask}^*(C_{\text{CHR}})} \right) \\ & \wedge \forall \boldsymbol{x} \left(C_{\text{built-in}} \rightarrow \left(\overline{H} \wedge \overline{H'} \wedge \overline{\text{entailed}^*(C_{\text{CHR}})} \leftrightarrow \overline{H} \wedge \exists \boldsymbol{z}(\overline{B}) \right) \right) \end{array}$$

where $\boldsymbol{x}$ enumerates variables of $C_{\text{built-in}}$ and H, and $\boldsymbol{z}$ enumerates variables of B. Since P is a CHRat program, variables in $\text{ask}^*(C_{\text{CHR}})$ and $\text{entailed}^*(C_{\text{CHR}})$ are in $\boldsymbol{x}$. Thus $\vdash_{\mathcal{C}\bullet} (r_i)^\ddagger \leftrightarrow ([\![r_i]\!])^\dagger$.

5 Examples

5.1 Union-Find Constraint Component

The union-find (or disjoint set union) algorithm [14] has been implemented in CHR with its best-known algorithmic complexity [15]. This positive result is remarkable because logic programming paradigm has been known to be ill-suited to such implementations [16], and because the algorithm given in [15] indeed benefits from the non-monotonic evolution of the store of the operational semantics.

The union-find algorithm maintains a partition of a universe, such that each equivalence class has a *representative* element. Three operations define this data structure:

- make(X) adds the element X to the universe, initially in an equivalence class reduced to the singleton {X}.
- find(X) returns the representative of the equivalence class of X.
- union(X,Y) joins the equivalence classes of X and Y (possibly changing the representative).

Naive Implementation The naive implementation, which [15] begins with, relies on the classical representation of equivalence classes by rooted trees. Roots are representative elements, they are marked as such with the CHR-constraint root(X). Tree branches are marked with A $\rightsquigarrow$ B, where A is the child and B the parent node.

File naive_union_find_solver.cat

```
component naive_union_find_solver.
export make/1, ≃/2.
make     @ make(A) ⟺ root(A).

union    @ union(A, B) ⟺ find(A, X), find(B, Y), link(X, Y).

findNode @ A ⇝ B \ find(A, X) ⟺ find(B, X).
findRoot @ root(A) \ find(A, X) ⟺ X = A.

linkEq   @ link(A, A) ⟺ true.
link     @ link(A, B), root(A), root(B) ⟺ B ⇝ A, root(A).
```

This implementation supposes that its entry-points make and union are used with constant arguments only, and that the first argument of find is always a constant.

In CHRat, one needs to add to this implementation the ability to check if two elements A and B are in the same equivalence class: we denote such a constraint

A ≃ B, where A and B are supposed to be constants. Telling this constraint just yields to the union of the two equivalence classes:

```
tellSame @ A ≃ B ⟹ union(A, B).
```

A way to provide a naive implementation for ask is to follow tree branches until possibly finding a common ancestor for A and B.

```
askEq    @ ask(A ≃ A) ⟺ entailed(A ≃ A).
askLeft  @ A ⇝ C \ ask(A ≃ B) ⟺ C ≃ B | entailed(A ≃ B).
askRight @ B ⇝ C \ ask(A ≃ B) ⟺ A ≃ C | entailed(A ≃ B).
```

The computation required to check the constraint entailment is done with the use of recursion in the definition of the guard $A = B$.

Optimized Implementation The second implementation proposed in [15] implements both *path-compression* and *union-by-rank* optimizations.

File union_find_solver.cat

```
component union_find.
export make/1, ≃/2.
make      @ make(A) ⟺ root(A, 0).

union     @ union(A, B) ⟺ find(A, X), find(B, Y), link(X, Y).

findNode  @ A ⇝ B, find(A, X) ⟺ find(B, X), A ⇝ X.
findRoot  @ root(A, _) \ find(A, X) ⟺ X = A.

linkEq    @ link(A, A) ⟺ true.
linkLeft  @ link(A, B), root(A, N), root(B, M) ⟺ N ≥ M |
              B ⇝ A, N1 is max(M+1, N), root(A, N1).
linkRight @ link(B, A), root(A, N), root(B, M) ⟺ N ≥ M |
              B ⇝ A, N1 is max(M+1, N), root(A, N1).
```

An optimized check for common equivalence class can rely on find to efficiently get the representatives and then compare them. check(A, B, X, Y) represents the knowledge that the equivalence class representatives of A and B are the roots X and Y respectively. When X and Y are known to be equal, **entailed**(A ≃ B) is put to the store (checkEq).

```
askEq     @ ask(A ≃ B) ⟺
                find(A, X), find(B, Y), check(A, B, X, Y).
checkEq   @ root(X) \ check(A, B, X, X) ⟺ entailed(A ≃ B).
```

These two rules are not enough to define a complete entailment-solver due to the non-monotonous nature of the changes applied to the tree structure. Indeed, roots found for A and B can be invalidated by subsequent calls to union, which may transform these roots into child nodes. When a former root becomes a child node, the following two rules put find once again to get the new root.

```
checkLeft  @ X ⇝ C \ check(A, B, X, Y) ⟺
                find(A, Z), check(A, B, Z, Y).
checkRight @ Y ⇝ C \ check(A, B, X, Y) ⟺
                find(B, Z), check(A, B, X, Y).
```

These rules define complete solvers for satisfaction and entailment checking for the $\simeq$ constraint.

5.2 Rational Tree Equality Constraint Component

Let us now consider rational terms, i.e. rooted, ordered, unranked, labelled, possibly infinite trees, with a finite number of structurally distinct sub-trees [17]. Nodes are supposed to belong to the universe considered by the union-find solver; two nodes belonging to the same equivalence class are supposed to be structurally equal. Each node X has a signature F/N, where F is the label of X and N its arity: the associated constraint is denoted fun(X, F, N). The constraint arg(X, I, Y), for each I between 1 and N, states that the Ith subtree of X is (structurally equal to) Y. These constraints have just to be compatible between elements of the same equivalence class:

File rational_tree_solver.cat

```
component rational_tree_solver.
import ≃/2 from union_find_solver.
export fun/3, arg/3, ∼/2.
eqFun @ fun(X0, F0, N0) \ fun(X1, F1, N1) ⟺ X0 ≃ X1 |
           F0 = F1, N0 = N1.
eqArg @ arg(X0, N, Y0) \ arg(X1, N, Y1) ⟺ X0 ≃ X1 |
           Y0 ≃ Y1.
```

Telling that two trees are structurally equal, denoted X ∼ Y, can be reduced to the union of the two equivalence classes.

```
eqProp @ X ∼ Y ⟺ X ≃ Y.
```

The computation associated to asking A ∼ B requires a coinductive derivation of structural comparisons to break infinite loops. That is done here by memoization. Each time a A ∼ B is asked, a new fresh variable M is introduced:

```
askEq @ ask(A ∼ B) ⟺ checkTree(M, A, B).
```

This variable marks the checking(M, A, B) tokens, signaling that A can be assumed to be equal to B since this check is already in progress.

```
checkTree(M, A, B) ⟺ eqTree(M, A, B) | entailed(A ∼ B).

ask(eqTree(M, A, B)) ⟺
   checking(M, A, B), fun(A, FA, NA), fun(B, FB, NB),
   checkTreeAux(M, A, B, FA, NA, FB, NB).
```

checkTreeAux firstly checks that signatures of A and B are equal, then compares arguments.

```
checkTreeAux(M, A, B, F, N, F, N) ⟺
    askArgs(M, A, B, 1, N), collectArgs(M, A, B, 1, N).
```

askArgs adds every askArg token corresponding to each pair of point-wise subtrees of A and B. askArg answers entailedArg if they match. collectArg ensures every entailedArg token have been put before concluding about the entailment of eqTree(M, A, B). It is very close to the definition of an ask solver, but the considered guard deals with a variable number of tokens equals to the arity of A and B.

```
askArgs(M, A, B, I, N) ⟺ I ≤ N |
    arg(A, I, AI), arg(B, I, BI),
    askArg(M, A, B, I, AI, BI),
    J is I + 1, askArgs(M, A, B, J, N).
askArgs(M, A, B, I, N) ⟺ true.
collectArgs(M, A, B, I, N), entailedArg(M, A, B, I) ⟺
    J is I + 1, collectArgs(M, A, B, J, N).
collectArgs(M, A, B, I, N) ⟺ I > N |
    entailed(eqTree(M, A, B)).
```

askArg firstly checks if the equality is memoized. Otherwise, the eqTree guard is recursively asked.

```
checking(M, AI, BI) \ askArg(M, A, B, I, AI, BI) ⟺
    entailedArg(M, A, B, I).
askArg(M, A, B, I, AI, BI) ⟺ eqTree(M, AI, BI) |
    entailedArg(M, A, B, I).
```

It is worth noticing that some garbage collection tasks are missing in this example: memoization tokens checking(M,A,B) are never removed from the store, and disentailment cases are not cleaned up.

6 Conclusion and perspectives

We have shown that by letting the programmer define in CHRat not only satisfiability checks but also entailment checks for constraints, CHRat becomes fully modular, i.e. constraints defined in one component can be reused in rules and guards in other components without restriction. Furthermore, this programming discipline is not too demanding for the programmer, as for any constraint c, the CHRat rule c \ **ask**(c) ⟺**entailed**(c) constitutes a default rule for checking entailment by simple store inspection. In the general case, however, CHRat rules for **ask**(c) perform arbitrary complex simpagations, which lead either to the constraint token **entailed**(c), or to another store waiting for more information.

The operational and declarative semantics of CHRat have been defined and the program transformation from CHRat to CHR which is at the basis of our compiler has been proved to implement the formal semantics of CHRat. It is worth noticing that the described transformation uniform and compatible with other orthogonal approaches related to modularity, like methodologies to make several constraint solvers collaborate [18].

We have also shown that the classical examples of constraint solvers defined in CHR could easily be modularized in CHRat and reused for building complex constraint solvers.

As for future work, the CHRat scheme can be improved in several ways. Variables in a CHRat guard have to appear either in the head or in the built-in constraint part of the guard. One way to allow existentially quantified variables guards without this restriction is to explicitly stratify guards as proposed in [9].

On the programming discipline side, ask-solvers should never lead to a failure and should not interfere with the logical interpretation of exported constraints present in the CHR store. The union-find example is a typical case where the ask-solver does change the logical meaning of the store by path-compressions and meanwhile keeps the interpretation of the exported constraint $\simeq$ unchanged. Formalizing sanity conditions for ask-solvers will be a step towards establishing the link between ask-solvers and logical entailment in the semantics.

The transformation of CHRat programs into regular CHR programs make the implementation directly benefit from optimized CHR implementations. While the efficient management of **ask** and **entailed** is left to the underlying CHR implementation, garbage collection, caching and memoization for checking entailment are left to the CHRat programmer. Good strategies for garbage collection still need to be investigated as shown in the rational tree solver.

Finally, the issue of separate compilation has not been discussed here but is a natural subject for future work in this framework.

Acknowledgment

We are grateful to Rishi Kumar for his preliminary work along these lines at INRIA, and to Jacques Robin for his recent impulse for this work.

References

1. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. **37** (1998) 95–138
2. 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
3. Abdennadher, S., Schütz, H.: CHRv: A flexible query language. In: FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems, London, UK, Springer-Verlag (1998) 1–14
4. Coquery, E., Fages, F.: A type system for CHR. In: Recent Advances in Constaints, revised selected papers from CSCLP'05. Number 3978 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2006) 100–117
5. Schrijvers, T., Demoen, B., Duck, G., Stuckey, P., Fruhwirth, T.: Automatic implication checking for CHR constraint solvers. Electronic Notes in Theoretical Computer Science **147** (2006) 93–111
6. Saraswat, V.A.: Concurrent constraint programming. ACM Doctoral Dissertation Awards. MIT Press (1993)

7. Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). Journal of Logic Programming **37** (1998) 139–164
8. Duck, G.J., Stuckey, P.J., de la Banda, M.G., Holzbaur, C.: Extending arbitrary solvers with constraint handling rules. In: PPDP '03, Uppsala, Sweden, ACM Press (2003) 79–90
9. Duck, G.J., de la Banda, M.J.G., Stuckey, P.J.: Compiling ask constraints. In: ICLP. (2004) 105–119
10. Sneyers, J., Weert, P.V., Koninck, L.D., Demoen, B., Schrijvers, T.: The K.U.Leuven CHR system (2008)
11. Kaeser, M.: WebCHR (2007) `http://chr.informatik.uni-ulm.de/~webchr/`.
12. Schulte, C.: Programming deep concurrent constraint combinators. In Pontelli, E., Costa, V.S., eds.: Practical Aspects of Declarative Languages. PADL 2000. Volume 1753 of Lecture Notes in Computer Science., Boston, MA, USA, Springer-Verlag (2000) 215–229
13. Sarna-Starosta, B., Schrijvers, T.: Indexing techniques for CHR based on program transformation. Technical report, CW 500, K.U.Leuven, Department of Computer Science (2007)
14. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. In: J. ACM. (1984)
15. Schrijvers, T., Frühwirth, T.W.: Analysing the CHR implementation of unionfind. In: 19th Workshop on (Constraint) Logic Programming. (2005)
16. Ganzinger, H.: A new metacomplexity theorem for bottom-up logic programs. In: Proceedings of International Joint Conference on Automated Reasoning. (2001)
17. Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science **25** (1983) 95–169
18. Abdennadher, S., Fruhwirth, T.: Integration and optimization of rule-based constraint solvers (2003)