Expressiveness of multiple heads in CHR

Cinzia Di Giusto[1], Maurizio Gabbrielli[1], and Maria Chiara Meo[2]

[1] Dipartimento di Scienze dell'Informazione,
Università di Bologna, Italia
[2] Dipartimento di Scienze, Università di Chieti Pescara, Italia

Abstract. Constraint Handling Rules (CHR) is a general purpose, committed-choice declarative language which, differently from other similar languages, uses multi-headed (guarded) rules.
In this paper we prove that multiple heads augment the expressive power of the language. In fact, we first show that restricting to single head rules affects the Turing completeness of CHR, provided that the underlying signature (for the constraint theory) does not contain function symbols. Next we show that, also when considering generic constraint theories, under some rather reasonable assumptions it is not possible to encode CHR (with multi-headed rules) into a single-headed CHR language while preserving the semantics of programs. As a corollary we obtain that, under these assumptions, CHR can be encoded neither in (constraint) logic programming nor in pure Prolog.

1 Introduction

Constraint Handling Rules (CHR) [7,9] is a committed-choice declarative language which has been originally designed for writing constraint solvers and which is nowadays a general purpose language. A CHR program consists of a set of multi-headed guarded (simplification and propagation) rules which allow one to rewrite constraints into simpler ones until a solved form is reached. The language is parametric w.r.t. an underlying constraint theory CT which defines the meaning of basic built-in constraints.

The presence of multiple heads is a crucial feature which differentiates CHR from other existing committed choice (logic) languages. Many examples in the vast literature on CHR provide empirical evidence for the claim that such a feature is needed in order to obtain reasonably expressive constraint solvers in a reasonably simple way (see the discussion in [9]). However this claim was not supported by any formal result, so far.

In this paper we prove that multiple heads do indeed augment the expressive power of CHR. Since we know that CHR is Turing powerful [16] we first show that CHR with single heads, called CHR-s in what follows, is also Turing powerful, provided that the underlying constraint theory allows the equality predicate (interpreted as pattern matching) and that the signature contains at least one function symbol (of arity greater than zero). This result is certainly not surprising; however it is worth noting that, as we prove later, when considering an underlying (constraint theory defined over a) signature containing finitely many constant symbols and no function symbol CHR (with multiple heads) is still Turing complete, while this is not the case for CHR-s.

This provide a first separation result which is however rather weak, since usual constraint theories used in CHR do allow function symbols. Moreover computability theory is not always the right framework for comparing the expressive power of concurrent languages, since often one has to compare languages which are Turing powerful (see [4] for one of the first discussions on the subject). Hence in the second part of the paper we compare the expressive power of CHR and CHR-s by using the notion of language encoding, first formalized in [4,15,17][3]. Intuitively, a language $\mathcal{L}$ is more expressive than a language $\mathcal{L}'$ or, equivalently, $\mathcal{L}'$ can be encoded in $\mathcal{L}$, if each program written in $\mathcal{L}'$ can be translated into an $\mathcal{L}$ program in such a way that: 1) the intended observable behaviour of the original program is preserved (under some suitable decoding); 2) the translation process satisfies some additional restrictions which indicate how easy this process is and how reasonable the decoding of the observables is. For example, typically one requires that the translation is compositional w.r.t. (some of) the syntactic operators of the language [4].

We prove that CHR cannot be encoded into CHR-s under the following three assumptions. First we assume that the observable properties to be preserved are the constraints computed by a program for a goal, more precisely we consider data sufficient answers and qualified answers. Since these are the two typical CHR observables for most CHR reference semantics, assuming their preservation is rather natural. Secondly we require that both the source CHR language and the target CHR-s share the same constraint theory defining built-in constraints. This is also a natural assumption, as CHR programs are usually written to define a new (user-defined) predicate in terms of the existing built-in constraints. Finally we assume that the translation of a goal is compositional w.r.t. conjunction of goals, that is, we assume that $[\![A, B]\!]_g = [\![A]\!]_g, [\![B]\!]_g$ for any conjunctive goal A, B, where $[\![\ ]\!]_g$ denotes the translation of a goal. We believe this notion of compositionality to be reasonable as well, since essentially it means that the translated program is not specifically designed for a single goal. It is worth noticing that we do not impose any restriction on the translation of the program rules.

From this main separation result follows that CHR cannot be encoded in (constraint) logic programs nor in pure Prolog. This does not conflict with the fact that there exist many CHR to Prolog compilers: it simply means that these compilers do not satisfy our assumptions (typically, they do not translate goals in a compositional way).

The remainder of the paper is organized as follows. Section 2 introduces the languages under consideration. We then provide the encoding of Minsky machines in CHR-s and discuss the Turing completeness of this language in Section 3. Section 4 contains the main separation results while Section 5 concludes by discussing some related works.

2 Preliminaries

In this section we give an overview of CHR syntax and operational semantics following [9]. We first need to distinguish the constraints handled by an existing solver, called built-in (or predefined) constraints, from those defined by the CHR program, called user-defined (or CHR) constraints. Therefore we assume that the signature contains two disjoint sets of predicate symbols for built-in and CHR constraints. A built-in constraint c is defined by: $c ::= a \mid c \wedge c \mid \exists_x c$ where a is an atomic built-in constraint (an

[3] The original terminology of these papers was "language embedding".

atomic constraint is a first-order atomic formula). For built-in constraints we assume given a (first order) theory CT which describes their meaning. A user-defined constraint is a conjunction of atomic user-defined constraints. We use c, d to denote built-in constraints, h, k to denote CHR constraints and a, b, f, g to denote both built-in and user-defined constraints (we will call these generally constraints). The capital versions of these notations will be used to denote multisets of constraints. We also denote by `false` any inconsistent conjunction of constraints and with `true` the empty multiset of built-in constraints. We will use "," rather than $\wedge$ to denote conjunction and we will often consider a conjunction of atomic constraints as a multiset of atomic constraints[4]. In particular, we will use this notation based on multisets in the syntax of CHR. The notation $\exists_V \phi$, where V is a set of variables, denotes the existential closure of a formula ϕ w.r.t. the variables in V, while the notation $\exists_{-V} \phi$ denotes the existential closure of a formula ϕ with the exception of the variables in V which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in ϕ. Moreover, if $\bar{t} = t_1, \ldots t_m$ and $\bar{t}' = t'_1, \ldots t'_m$ are sequences of terms then the notation $p(\bar{t}) = p'(\bar{t}')$ represents the set of equalities $t_1 = t'_1, \ldots, t_m = t'_m$ if $p = p'$, and it is undefined otherwise. This notation is extended in the obvious way to sequences of constraints.

A CHR program is defined as a sequence of two kinds of rules: simplification and propagation[5]. Intuitively, simplification rewrites constraints into simpler ones, while propagation adds new constraints which are logically redundant but may trigger further simplifications.

Definition 1. *A CHR* simplification *rule has the form:* $\quad r \ @\ H \Leftrightarrow C \mid B$
while a CHR propagation *rule has the form:* $\quad r \ @\ H \Rightarrow C \mid B,$
where r is a unique identifier of a rule, H (the head) is a (non-empty) multiset of user-defined constraints, C (the guard) is a possibly empty multiset of built-in constraints and B is a possibly empty multiset of (built-in and user-defined) constraints.

A CHR program *is a finite set of CHR simplification and propagation rules.*

In the following when the guard is `true` we omit `true` $\mid$. Also the names of rules are omitted when not needed. A CHR *goal* is a multiset of (both user-defined and built-in) constraints. An example of CHR Program is shown in Figure 3.

We describe now the operational semantics of CHR by slightly modifying the transition system defined in [9]. We use a transition system $T = (\mathit{Conf}, \longrightarrow)$ where configurations in *Conf* are triples of the form $\langle G, K, d \rangle$, where G are the constraints that remain to be solved, K are the user-defined constraints that have been accumulated and d are the built-in constraints that have been simplified.

An *initial configuration* has the form $\langle G, \emptyset, \emptyset \rangle$ while a *final configuration* has either the form $\langle G, K, \mathtt{false} \rangle$ when it is *failed*, or the form $\langle \emptyset, K, d \rangle$ when it is successfully terminated because there are no applicable rules. Given a program P, the transition relation $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ is the least relation satisfying the rules in Table 1 (for the sake of simplicity, we omit indexing the relation with the name of the program). The **Solve**

[4] We prefer to use multisets rather than sequences (as in the original CHR papers) because our results do not depend on the order of atoms in the rules.

[5] Some papers consider also simpagation rules. Since these are abbreviations for propagation and simplification rules we do not need to introduce them.

Solve	$\dfrac{CT \models c \wedge d \leftrightarrow d' \text{ and c is a built-in constraint}}{\langle (c,G), K, d\rangle \longrightarrow \langle G, K, d'\rangle}$
Introduce	$\dfrac{\text{h is a user-defined constraint}}{\langle (h,G), K, d\rangle \longrightarrow \langle G, (h,K), d\rangle}$
Simplify	$\dfrac{H \Leftrightarrow C \mid B \in P \;\; x = Fv(H) \;\; CT \models d \rightarrow \exists_x((H = H') \wedge C)}{\langle G, H' \wedge K, d\rangle \longrightarrow \langle B \wedge G, K, H = H' \wedge d\rangle}$
Propagate	$\dfrac{H \Rightarrow C \mid B \in P \;\; x = Fv(H) \;\; CT \models d \rightarrow \exists_x((H = H') \wedge C)}{\langle G, H' \wedge K, d\rangle \longrightarrow \langle B \wedge G, H' \wedge K, H = H' \wedge d\rangle}$

Table 1. The standard transition system for CHR

transition allows to update the constraint store by taking into account a built-in constraint contained in the goal. The **Introduce** transition is used to move a user-defined constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules. The transitions **Simplify** and **Propagate** allow to rewrite user-defined constraints (which are in the CHR constraint store) by using rules from the program. As usual, in order to avoid variable name clashes, both these transitions assume that all variables appearing in a program clause are fresh ones. Both the **Simplify** and **Propagate** transitions are applicable when the current store (d) is strong enough to entail the guard of the rule (C), once the parameter passing has been performed (this is expressed by the equation $H = H'$). Note that, due to the existential quantification over the variables x appearing in H, in such a parameter passing the information flow is from the actual parameters (in H') to the formal parameters (in H), that is, it is required that the constraints H' which have to be rewritten are an instance of the head H[6]. The difference between **Simplify** and **Propagate** lies in the fact that while the former transition removes the constraints H' which have been rewritten from the CHR constraint store, this is not the case for the latter.

Given a goal G, the operational semantics that we consider observes the final stores of computations terminating with an empty goal and an empty user-defined constraint. Following the terminology of [9], we call such observables *data sufficient answers*.

Definition 2. *[Data sufficient answers [9]] Let P be a program and let G be a goal. The set $\mathcal{SA}_P(G)$ of data sufficient answers for the query G in the program P is defined as:* $\mathcal{SA}_P(G) = \{\exists_{-Fv(G)} d \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow^* \langle \emptyset, \emptyset, d\rangle \not\longrightarrow\}$.

We also consider the following different notion of answer, obtained by computations terminating with a user-defined constraint which does not need to be empty.

Definition 3. *[Qualified answers [9]] Let P be a program and let G be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query G in the program P is defined as:* $\mathcal{QA}_P(G) = \{\exists_{-Fv(G)}(K \wedge d) \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow^* \langle \emptyset, K, d\rangle \not\longrightarrow\}$.

Both previous notions of observables characterize an input/output behavior, since the input constraint is implicitly considered in the goal. Clearly in general $\mathcal{SA}_P(G) \subseteq$

[6] This means that the equations $H = H'$ express pattern matching rather than unification.

$\mathcal{QA}_P(G)$ holds, since data sufficient answers can be obtained by setting $K = \emptyset$ in Definition 3.

Note that in presence of propagation rules, the abstract (naive) operational semantics that we consider here introduces redundant infinite computations (because propagation rules do not remove user defined constraints). It is possible to define different operational semantics (see [1] and [6]) which avoids these infinite computations by allowing to apply at most once a propagation rule to the same constraints. The results presented in this paper hold also in case these more refined semantics are considered, essentially because the number of applications of propagations rules does not matter. We refer here to the naive operational semantics because it is much simpler than those in [1] and [6].

3 Turing completeness of CHR-s

In this section we discuss the Turing completeness of CHR-s by taking into account also the underlying constraint theory.

In order to show the Turing completeness of a language we encode Minsky machines [12] into it, hence we recall some basic notions on this Turing equivalent formalism. A Minsky machine $M(v_0, v_1)$ is a two-counter machine which consists of two registers r_1 and r_2 holding arbitrary large natural numbers and initialized with the values v_0 and v_1, and a program, i.e. a finite sequence of numbered instructions which modify the two registers. There are three types of instructions:

- $Succ(R_i)$: adds 1 to the content of register R_i and goes to the next instruction;
- $DecJump(R_i, l)$: if the content of the register R_i is not zero, then decreases it by 1 and goes to the next instruction, otherwise jumps to instruction l;
- $Halt$: stops computation and returns the value in register R_1.

An internal state of the machine is given by a tuple (p_i, r_1, r_2) where the program counter p_i indicates the next instruction and r_1, r_2 are the current contents of the two registers. Given a program its computation proceeds by executing the instructions as indicated by the program counter. The execution stops when the program counter reaches the $Halt$ instruction.

We first show that CHR-s is Turing powerful, provided that the constraint theory allows the built-in = (interpreted as pattern matching) and that the underlying signature contains at least a function symbol (of arity one) and a constant symbol. This result is obtained by providing an encoding $[\![\ ]\!] : Machines \rightarrow CHR$ of a Minsky machine $M(v_0, v_1)$ in CHR as shown in Figure 1: Every rule takes as input the program counter and the two registers and updates the state according to the instruction in the obvious way. The variable X is used for outputting the result at the end. Note that, due to the pattern matching mechanism, a generic goal $i(p_i, s, t, X)$ can fire at most one of the two rules encoding the $DecJump$ instruction (in fact, if s is a free variable no rule in the encoding of $DecJump(r_1, p_l)$ is fired). Without loss of generality we can assume that the counters are initialized with 0, hence the encoding of a machine M with n instructions has the form: $[\![M(0,0)]\!] := \{[\![Instruction_1]\!], \dots, [\![Instruction_n]\!]\}$ (note that the initial values of the register are not considered in the encoding of the machine: they will be used in the initial goal, as shown below). The following theorem states the correctness of the encoding. The proof is immediate.

$[\![p_i : Halt]\!] :=$	$i(p_i, R_1, R_2, X) \Leftrightarrow X = R_1$
$[\![p_i : Succ(r_1)]\!] :=$	$i(p_i, R_1, R_2, X) \Leftrightarrow i(p_{i+1}, succ(R_1), R_2, X)$
$[\![p_i : Succ(r_2)]\!] :=$	$i(p_i, R_1, R_2, X) \Leftrightarrow i(p_{i+1}, R_1, succ(R_2), X)$
$[\![p_i : DecJump(r_1, p_l)]\!] :=$	$i(p_i, 0, R_2, X) \Leftrightarrow i(p_l, 0, R_2, X)$
	$i(p_i, succ(R_1), R_2, X) \Leftrightarrow i(p_{i+1}, R_1, R_2, X)$
$[\![p_i : DecJump(r_2, p_l)]\!] :=$	$i(p_i, R_1, 0, X) \Leftrightarrow i(p_l, R_1, 0, X)$
	$i(p_i, R_1, succ(R_2), X) \Leftrightarrow i(p_{i+1}, R_1, R_2, X)$

Fig. 1. Minsky machine encoding in CHR-s

Theorem 1. *A Minsky machine $M(0,0)$ halts with output k if and only if the goal $i(1,0,0,X)$ in the program $[\![M(0,0)]\!]$ has a data sufficient answer $X = k$.*

It is worth noting that the presence of a function symbol ($succ()$ in our case) is crucial in order to encode natural numbers and therefore to obtain the above result. Indeed, as we prove below, when considering a signature containing only a finite number of constant symbols the language CHR-s, differently from the case of CHR, is not Turing powerful. To be more precise, assume that CT defines only the = symbol (to be interpreted as pattern matching, as in the previous case) and assume that such a theory is defined over a signature containing finitely many constant symbols and no function symbol (of arity > 0). Let us call $CT_\emptyset$ the resulting theory.

When considering $CT_\emptyset$, CHR-s is computationally equivalent w.r.t. termination to place/transition nets [14], a formalism where termination is known to be decidable. Hence we have the following result.

Theorem 2. *CHR-s on $CT_\emptyset$ is not Turing complete.*

On the other hand, CHR (with multiple heads) is still Turing powerful also when considering the theory $CT_\emptyset$. Indeed, as we show in Figure 2, we can encode Minsky machines into CHR (defined on $CT_\emptyset$)[7]. The basic idea here is that to encode the values of the registers we use chains (conjunctions) of atomic formulas of the form $s(R_1, SuccR_1)$, $s(SuccR_1, SuccR_1')\ldots$ (recall that R_1, $SuccR_1$, $SuccR_1'$ are variables and we have countably many variables; moreover recall that the CHR computation mechanism avoid variables capture by using fresh names for variables each time a rule is used). It is also worth noting that for the correctness of the encoding it is essential that pattern matching rather than unification is used when applying rules (this ensures that in the case of the decrement only one of the two instructions can match the goal and therefore can be used). The correctness of the encoding is stated by the following theorem whose proof is immediate.

Theorem 3. *A Minsky machine $M(0,0)$ halts with output $k > 0$ (or $k = 0$) if and only if the goal $zero(R_1) \wedge zero(R_2) \wedge i(1, R_1, R_2, X)$ in the program $[\![M(0,0)]\!]_2$ produces a qualified answer $\exists_{-X,R_1}(X = R_1 \wedge s(R_1, SuccR_1^1) \bigwedge_{i=1\ldots k-1}(SuccR_1^i, SuccR_1^{i+1}))$ (or $\exists_{-X,R_1}(x = R_1 \wedge zero(R_1))$).*

[7] We thank an anonymous reviewer for having suggested this encoding.

$$
\begin{aligned}
&[\![p_i : Halt]\!]_2 := i(p_i, R_1, R_2, X) \Leftrightarrow X = R_1\\
&[\![p_i : Succ(r_1)]\!]_2 := i(p_i, R_1, R_2, X) \Leftrightarrow s(R_1, SuccR_1), i(p_{i+1}, SuccR_1, R_2, X)\\
&[\![p_i : Succ(r_2)]\!]_2 := i(p_i, R_1, R_2, X) \Leftrightarrow s(R_2, SuccR_2), i(p_{i+1}, R_1, SuccR_2, X)\\
&[\![p_i : DecJump(r_1, p_l)]\!]_2 := i(p_i, R_1, R_2, X), s(PreR_1, R_1) \Leftrightarrow i(p_{i+1}, PreR_1, R_2, X)\\
&\qquad\qquad\qquad\qquad zero(R_1), i(p_i, R_1, R_2, X) \Leftrightarrow i(p_l, R_1, R_2, X), zero(R_1)\\
&[\![p_i : DecJump(r_2, p_l)]\!]_2 := i(p_i, R_1, R_2, X), s(PreR_2, R_2) \Leftrightarrow i(p_{i+1}, R_1, PreR_2, X)\\
&\qquad\qquad\qquad\qquad zero(R_2), i(p_i, R_1, R_2, X) \Leftrightarrow i(p_l, R_1, R_2, X), zero(R_2)
\end{aligned}
$$

Fig. 2. Minsky machine encoding in CHR on $CT_\emptyset$

Previous theorems state a separation result between CHR and CHR-s, however this is rather weak since the real implementations of CHR usually consider a non-trivial constraint theory which includes function symbols. Therefore we are interested in proving finer separation results which hold for Turing powerful languages. This is the content of the following section.

4 Separating CHR and CHR-s

In this section we consider a generic non-trivial constraint theory CT. We have seen that in this case both CHR and CHR-s are Turing powerful, which means that in principle one can always encode CHR into CHR-s. The question is how difficult and how acceptable such an encoding is and this question can have important practical consequences: for example, a distributed algorithm can be implemented in one language in a reasonably simple way and cannot be implemented in another (Turing powerful) language, unless one introduces rather complicated data structures or loses some compositionality properties (see [18]).

We prove now that, when considering acceptable encodings and generic goals whose components can share variables, CHR cannot be embedded into CHR-s while preserving data sufficient answers. As a corollary we obtain that also qualified answers cannot be preserved.

First we have to formally define what an acceptable encoding is. We define a *program encoding* as any function $[\![\ ]\!] : \mathcal{P}_{CHR} \rightarrow \mathcal{P}_{CHR-s}$ which translates a CHR program into a (finite) CHR-s program ($\mathcal{P}_{CHR}$ and $\mathcal{P}_{CHR-s}$ denote the set of CHR and CHR-s programs, respectively). To simplify the treatment we assume that both the source language CHR and the target language CHR-s use the same built-in constraints semantically described by a theory CT (actually this assumption could be relaxed). Note that we do not impose any other restriction on the program translation (which, in particular, could also be non compositional). Next we have to define how the initial goal of the source program has to be translated into the target language. Here we require that the translation is compositional w.r.t. the conjunction of atoms, as mentioned in the introduction. Moreover since both CHR and CHR-s share the same CT we assume that the built-ins present in the goal are left unchanged. These assumptions essentially mean that our encoding respects the structure of the original goal and does not introduce new relations among the variables which appear in the goal. Finally, as mentioned before, we are interested in preserving data sufficient and qualified answers. Hence we have the following definition where we denote by $\mathcal{G}_{CHR}$ and $\mathcal{G}_{CHR-s}$ the class of CHR and CHR-s goals, respectively (we differentiate these two classes because, for example, a CHR-s goal could use some user defined predicates which are not allowed in the goals of the original program[8]). Note that the following definition is parametric w.r.t. a class $\mathcal{G}$ of goals: clearly considering different classes of goals could affect our encodability results. Such a parameter will be instantiated when the notion of acceptable encoding will be used.

Definition 4 (Acceptable encoding). *Let $\mathcal{G}$ be a class of CHR goals. An* acceptable encoding *(of CHR into CHR-s, for the class of goals $\mathcal{G}$) is a pair of mappings $[\![\ ]\!] : \mathcal{P}_{CHR} \rightarrow \mathcal{P}_{CHR-s}$ and $[\![\ ]\!]_g : \mathcal{G}_{CHR} \rightarrow \mathcal{G}_{CHR-s}$ which satisfy the following conditions:*

- *$\mathcal{P}_{CHR}$ and $\mathcal{P}_{CHR-s}$ share the same CT;*
- *for any goal $(A, B) \in \mathcal{G}_{CHR}$, $[\![A, B]\!]_g = [\![A]\!]_g, [\![B]\!]_g$ holds. We also assume that the built-ins present in the goal are left unchanged;*
- *Data sufficient (qualified) answers are preserved for the class of goals $\mathcal{G}$, that is, for all $G \in \mathcal{G} \subseteq \mathcal{G}_{CHR}$, $\mathcal{SA}_P(G) = \mathcal{SA}_{[\![P]\!]}([\![G]\!]_g)$ ($\mathcal{QA}_P(G) = \mathcal{QA}_{[\![P]\!]}([\![G]\!]_g)$) holds.*

Note that, since we consider goals as multisets, with the second condition here we are not requiring that the order of atoms in the goals is preserved by the translation: We are only requiring that the translation of A, B is the conjunction of the translation of A and of B. Weakening this condition by requiring that the translation of A, B is some form of composition of the translation of A and of B does not seem reasonable, as conjunction is the only form for goal composition available in these languages.

In order to prove our separation result we need the following lemma which states a key property of CHR-s computations. Essentially it says that if the conjunctive G, H with input constraint c produces a data sufficient answer d, then when considering one component, say G, with the input constraint d we obtain the same data sufficient answer. Moreover the same answer can be obtained, either for G or for H, also starting with an input constraint c' weaker than d.

Lemma 1. *Let P be a CHR-s program and let (c, G, H) be a goal, where c is a built-in constraint, G and H are multisets of CHR constraints and $V = Fv(c, G, H)$. Assume that (c, G, H) in P has the data sufficient answer d. Then the following holds:*

- *Both the goals (d, G) and (d, H) have the same data sufficient answer d.*
- *If $CT \models c \not\rightarrow d$ then there exists a built-in constraint c' such that $Fv(c') \subseteq V$, $CT \models c' \not\rightarrow d$ and either (c', G) or (c', H) has the data sufficient answer d.*

Proof. The proof of the first statement is straightforward (since we consider single headed programs). In fact, since the goal (c, G, H) has the data sufficient answer d in P, the goal (d, G) can either answer d or can produce a configuration where the user

[8] This means that in principle the signatures of (language of) the original and the translated program are different.

defined constraints are waiting for some guards to be satisfied in order to apply a rule r, but since the goal contains all the built-in constraints in the answer all the guards are satisfied letting the program to answer d.

We prove the second statement. Let $\delta = \langle (c, G, H), \emptyset, \emptyset \rangle \longrightarrow^* \langle \emptyset, \emptyset, d' \rangle \not\longrightarrow$ be the derivation producing the data sufficient answer $d = \exists_{-V} d'$ for the goal (c, G, H).

By definition of derivation and since by hypothesis $CT \models c \not\rightarrow d$, δ must be of the form $\langle (c, G, H), \emptyset, \emptyset \rangle \longrightarrow^* \langle (c_1, G_1), S_1, d_1 \rangle \longrightarrow \langle (c_2, G_2), S_2, d_2 \rangle \longrightarrow^* \langle \emptyset, \emptyset, d' \rangle \not\rightarrow$, where for $i \in [1, 2]$, c_i and d_i are built-in constraints such that $CT \models c_1 \wedge d_1 \not\rightarrow d$ and $CT \models c_2 \wedge d_2 \rightarrow d$. We choose $c' = \exists_{-V}(c_1 \wedge d_1)$. By definition of derivation and since P is a CHR-s program, the transition $\langle (c_1, G_1), S_1, d_1 \rangle \longrightarrow \langle (c_2, G_2), S_2, d_2 \rangle$ must be a rule application of a single headed rule r, which must match with a constraint k that was derived (in the obvious sense) by either G or H. Without loss of generality, we can assume that k was derived from G. By construction c' suffices to satisfy the guards needed to reproduce k, which can then fire the rule r, after which all the rules needed to let the constraints of G disappear can fire. Therefore we have that $\langle (c', G), \emptyset, \emptyset \rangle \longrightarrow^* \langle \emptyset, \emptyset, d'' \rangle \not\rightarrow$, where $CT \models \exists_{-V} d'' \leftrightarrow \exists_{-V} d' (\leftrightarrow d)$ and then the thesis. □

Note that Lemma 1 is not true anymore if we consider (multiple headed) CHR programs. Indeed if we consider the program P consisting of the single rule `rule`@H, $H \Leftrightarrow$ `true` $\mid c$ then goal (H, H) has the data sufficient answer c in P, but for each constraint c' the goal (H, c') has no data sufficient answer in P. With the help of the previous lemma we can now prove our main separation result. The idea of the proof is that any possible encoding of the rule `r` @ $H, G \Leftrightarrow$ `true` $\mid c$ into CHR-s would either produce more answers for the goal H (or G), or would not be able to provide the answer c for the goal H, G.

Theorem 4. *Let $\mathcal{G}$ be a class of goals such that if H is an head of a rule then $H \in \mathcal{G}$. When considering data sufficient or qualified answers, there exists no acceptable encoding of CHR in CHR-s for the class $\mathcal{G}$.*

Proof. We first consider data sufficient answers. The proof is by contradiction. Consider the program P consisting of the single rule

$$\mathtt{r} \ @ \ H, G \Leftrightarrow \mathtt{true} \mid c$$

and assume that $[\![P]\!]$ is the translation of P in CHR-s. Assume also that c (restricted to the variables in H, G) is not the weakest constraint, i.e. assume that there exist d such that $CT \models d \not\rightarrow \exists_{-V} c$ where $V = Fv(H, G)$. Note that this assumption does not imply any loss of generality, as we consider non trivial constraint systems containing at least two different constraints.

Since the goal (H, G) has the data sufficient answer $\exists_{-V} c$ in the program P and since the encoding preserves data sufficient answers the goal $[\![(H, G)]\!]_g$ has the data sufficient answer $\exists_{-V} c$ also in the program $[\![P]\!]$. From the compositionality of the translation of goals and the previous Lemma 1 it follows that there exists a constraint c' such that $Fv(c') \subseteq V$, $CT \models c' \not\rightarrow \exists_{-V} c$ and either the goal $[\![(c', H)]\!]_g$, or the goal $[\![(c', G)]\!]_g$ has the data sufficient answer c in the encoded program $[\![P]\!]$. However

```
reflexivity @ $Lessequal(X, Y) \Leftrightarrow X = Y \mid$ true
antisymmetry @ $Lessequal(X, Y), Lessequal(Y, X) \Leftrightarrow X = Y$
transitivity @ $Lessequal(X, Y), Lessequal(Y, Z) \Rightarrow Lessequal(X, Z)$
```

Fig. 3. A program for defining $\leq$ in CHR

neither (c', H) nor (c', G) has any data sufficient answer in the original program P. This contradicts the fact that $[\![P]\!]$ is an acceptable encoding for P, thus concluding the proof for data sufficient answers. The thesis for qualified answers follows immediately from the previous part, as qualified answers contain the set of data sufficient answers. $\square$

The hypothesis made on the class of goals $\mathcal{G}$ is rather weak, as typically heads of rules have to be used as goals. As an example of the application of the previous theorem consider the program (from [9]) contained in Figure 3 which allows one to define the user-defined constraint *Lessequal* (to be interpreted as $\leq$) in terms of the only built-in constraint = (to be interpreted as syntactic equality). For example, given the goal $\{Lessequal(A, B), Lessequal(B, C), Lessequal(C, A)\}$ after a few computational steps the program will answer $A = B, B = C, C = A$. Now for obtaining this behaviour it is essential to use multiple heads, as already claimed in [9] and formally proved by previous theorem. In fact, following the lines of the proof of Theorem 4, one can show that if a single headed program P' is any translation of the program in Figure 3 which produces the correct answer for the goals above, then there exists a subgoal which has an answer in P' but not in the original program.

4.1 A note on logic programs and Prolog

(Constraint) Logic programming and Prolog are programming languages quite different from CHR, mainly because they are sequential ones, without any guard mechanism and commit operator. Nevertheless, since many CHR implementations are built on top of a Prolog system, by using a compiler which translates CHR programs to Prolog, it is meaningful to compare these sequential languages with CHR.

Note that here, following the general terminology (see for example [2]), a (constraint) logic program is a set of (definite) clauses, to be interpreted operationally in terms of SLD-resolution, thus using a non deterministic computational model. Real logic programming systems eliminate such a non determinism by choosing a specific selection rule (for selecting the atom in the goals to be evaluated) and a specific rule for searching the SLD-tree. Following [2] we call pure Prolog a logic programming language which uses the leftmost selection rule and the depth-first search (this corresponds to consider clauses top-down, according to the textual ordering in the program). Implemented Prolog systems are extensions of pure Prolog obtained by considering specific built-ins for arithmetic, control etc. Some of these built-ins have a non logical nature, which complicates their semantics.

All our technical lemmata about CHR-s can be stated also for (constraint) logic programming and pure Prolog (the proofs are similar, modulo some minor adjustments).

Hence our separation results hold also when considering these languages rather than CHR-s. These can be summarized as follows.

Corollary 1. *Let $\mathcal{G}$ be a class of goals such that if H is an head of a rule then $H \in \mathcal{G}$. When considering data sufficient answers or qualified answers there exists no acceptable encoding of CHR in constraint logic programming nor in pure Prolog for the class $\mathcal{G}$.*

As mentioned in the introduction, previous result does not conflict with the fact that there exist many CHR to Prolog compilers: it simply means that, when considering pure Prolog, these compilers do not satisfy our assumptions (typically, they do not translate goals in a compositional way). Moreover real Prolog systems use several non logical built-in's, which are out of the scope of previous results.

5 Conclusions and Related works

In this paper we have studied the expressiveness of CHR. We have proved that multiple heads augment the expressive power of the language, indeed we have shown that CHR cannot be encoded in CHR with single heads under quite reasonable assumptions. These results are then shown to hold also for (constraint) logic programming and pure Prolog.

There exists a very large literature on the expressiveness of concurrent languages, however there are only few papers which consider the expressive power of CHR: A recent study is [16], where the authors show that it is possible to implement any algorithm in CHR in an efficient way, i.e. with the best known time and space complexity. This result is obtained by introducing a new model of computation, called the CHR machine, and comparing it with the well-known Turing machine and RAM machine models. Earlier works by Frühwirth [8,10] studied the time complexity of simplification rules for naive implementations of CHR. In this approach an upper bound on the derivation length, combined with a worst-case estimate of (the number and cost of) rule application attempts, allows to obtain an upper bound of the time complexity. The aim of all these works is clearly completely different from ours, even though it would be interesting to compare CHR and CHR-s in terms of complexity.

When moving to other languages, somehow related to our paper is the work by Zavattaro [19] where the coordination languages Gamma [3] and Linda [11] are compared in terms of expressive power. Since Gamma allows multiset rewriting it reminds CHR multiple head rules, however the results of [19] are rather different from ours, since a process algebraic view of Gamma and Linda is considered where the actions of processes are atomic and do not contain variables. On the other hand, our results depend directly on the presence of logic variables in the CHR model of computation. Relevant for our approach is also [4] which introduces the original approach to language comparison based on encoding, even though in this paper rather different languages with different properties are considered.

We are extending this work along several lines. First in [5] we have considered weaker notion of acceptable encoding obtaining results similar to those of theorem 4. Moreover, in the same paper we have considered the different expressive power of

CHR_n, the language with at most n atoms in the heads, and CHR_{n+1}. Preliminary results show that CHR_{n+1} is strictly more expressive than CHR_n. We intend to extend these results by using the techniques described in [13,18] for showing that (under some reasonable hypothesis) some typical distributed problems can be solved in CHR_{n+1} and not in CHR_n. We also plan to investigate what happens when considering translation of CHR into real Prolog systems (with non logical built-ins). Some of the properties that we used in our technical lemma in this case do not hold anymore, however we believe that also in this case we can establish separation results similar to those shown in section 4.1.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Principles and Practice of Constraint Programming*, pages 252–266, 1997.
2. K. R. Apt. *From logic programming to Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
3. J.-P. Banâtre and D. L. Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993.
4. F. S. de Boer and C. Palamidessi. Embedding as a tool for language comparison. *Information and Computation*, 108(1):128–157, 1994.
5. C. Di Giusto, M. Gabbrielli, and M. C. Meo. On the expressive power of CHR. Technical report, 2008.
6. G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *ICLP*, LNCS, pages 90–104, 2004.
7. T. Frühwirth. Introducing simplification rules. Technical report, 1991.
8. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *KR 02*, 2002.
9. T. W. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
10. T. W. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. *Electr. Notes Theor. Comput. Sci.*, 59(3), 2001.
11. D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.
12. M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.
13. C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical. Structures in Comp. Sci.*, 13(5):685–719, 2003.
14. W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
15. E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.
16. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *CHR'05*, number CW 421, pages 3–17, 2005.
17. F. W. Vaandrager. Expressive results for process algebras. In *Proceedings of the REX Workshop on Sematics: Foundations and Applications*, pages 609–638, London, UK, 1993. Springer-Verlag.
18. M. G. Vigliotti, I. Phillips, and C. Palamidessi. Tutorial on separation results in process calculi via leader election problems. *Theor. Comput. Sci.*, 388(1-3):267–289, 2007.
19. G. Zavattaro. On the incomparability of gamma and linda. Technical report, Amsterdam, The Netherlands, 1998.