

Scaling termination proofs by a characterization of cycles in CHR

Paolo Pilozzi
Danny De Schreye

Report CW 541, April 2009



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Scaling termination proofs by a characterization of cycles in CHR

Paolo Pilozzi
Danny De Schreye

Report CW541, April 2009

Department of Computer Science, K.U.Leuven

Abstract

In the current paper, we discuss cycles in Constraint Handling Rules for the purpose of scaling termination proofs. In order to obtain a useful characterization, our approach differs from the ones used in other declarative languages, such as Logic Programming and Term Rewrite Systems. Due to multi-headed rules, the notion of a cycle is not in direct correspondence with the recursive calls of a program. Our characterization has to be more refined as we have to consider also partner constraints. Furthermore, a second, more challenging problem, due to the multi-set semantics of CHR, makes it unclear how cycles structurally compose. To tackle this problem, we develop a new abstraction for computations in CHR based on hypergraphs. On the basis of this abstraction, we define CHR constructs as a representation for sub-computations. These constructs introduce a concept of minimality and structural composability, making it a useful abstraction. On the basis of this abstraction we define the meaning of a CHR cycle. These are special kinds of CHR constructs. We have developed a verification for detecting whether constructs are CHR cycles and for deriving the minimal CHR cycles of a program. We motivate why and how this will lead to scalable automated termination proof procedures for CHR.

Keywords : Constraint Handling Rules, Termination Analysis.

Scaling termination proofs by a characterization of cycles in CHR

Paolo Pilozzi* and Danny De Schreye

Dept. of Computer Science, K.U.Leuven, Belgium
Firstname.Lastname@cs.kuleuven.be

Abstract. In the current paper, we discuss cycles in Constraint Handling Rules for the purpose of scaling termination proofs. In order to obtain a useful characterization, our approach differs from the ones used in other declarative languages, such as Logic Programming and Term Rewrite Systems. Due to multi-headed rules, the notion of a cycle is not in direct correspondence with the recursive calls of a program. Our characterization has to be more refined as we have to consider also partner constraints. Furthermore, a second, more challenging problem, due to the multi-set semantics of CHR, makes it unclear how cycles structurally compose. To tackle this problem, we develop a new abstraction for computations in CHR based on hypergraphs. On the basis of this abstraction, we define CHR constructs as a representation for sub-computations. These constructs introduce a concept of minimality and structural composability, making it a useful abstraction. On the basis of this abstraction we define the meaning of a CHR cycle. These are special kinds of CHR constructs. We have developed a verification for detecting whether constructs are CHR cycles and for deriving the minimal CHR cycles of a program. We motivate why and how this will lead to scalable automated termination proof procedures for CHR.

1 Preliminaries

In this Section, we provide notation and semantics. We first discuss the syntax of CHR. Then we define multisets, after which we discuss the multiset semantics of CHR. Finally, we introduce hypergraphs and their formal and graphical foundations.

1.1 Syntax of CHR

CHR manipulates conjunctions of *constraints*.

Definition 1 (Constraint). *A constraint is syntactically defined as a special first-order predicate $c(t_1, \dots, t_n)$ of arity $n \geq 0$. We distinguish built-in constraints from CHR constraints. Built-in constraints are pre-defined and solved by an underlying constraint solver CT . CHR constraints are user-defined and solved by a CHR program P . \square*

* Supported by I.W.T. Flanders - Belgium

A *CHR program* relates conjunctions of constraints by three different kinds of rules. A *simplification rule* replaces constraints by simpler constraints, a *simpagation rule* replaces constraints conditional on the presence of other constraints and a *propagation rule* adds redundant constraints without replacing any existing constraints.

Definition 2 (CHR program). *Let H_k , H_r and C denote conjunctions of CHR constraints and let G and B denote conjunctions of built-in constraints. Then, a CHR program P is a finite set of CHR rules of the following form:*

$$\begin{array}{lll} \text{Simplification rule:} & \text{Propagation rule:} & \text{Simpagation rule:} \\ \text{true} \setminus H_r \Leftrightarrow G \mid B, C. & H_k \setminus \text{true} \Leftrightarrow G \mid B, C. & H_k \setminus H_r \Leftrightarrow G \mid B, C. \\ \text{or } H_r \Leftrightarrow G \mid B, C. & \text{or } H_k \Rightarrow G \mid B, C. & \end{array}$$

CHR rules are named by adding "rulename @" in front of the rule. \square

Note that an empty conjunction is denoted by the built-in constraint *true* and that *CHR rules* are generalized by simpagation rules.

1.2 Multisets

A multiset is a collection of elements in which elements may occur multiple times. Multisets generalize normal sets, where every element may only occur once.

Definition 3 (Multiset). *A multi-set is defined as a pair $\mathcal{M} = (A, m)$, where A is a normal set and where $m : A \mapsto \mathbb{N}_0$ is a function mapping the elements in A to the set of positive natural numbers. The set A is called the underlying set of elements and for every element $a \in A$, the multiplicity $m(a)$ is the number of occurrences of a .*

The function m is a set of ordered pairs $\{(a, m(a)) : a \in A\}$. For example, the multiset written as $[a, a, b]$ is defined as $\{(a, 2), (b, 1)\}$, and the multiset $[a, b]$ is defined as $\{(a, 1), (b, 1)\}$. The empty multiset $[\]$ is defined as $\{\}$.

We denote by $a \in \mathcal{M}$ that a is in the multiset $\mathcal{M} = (A, m)$. We denote by $a \in^n \mathcal{M}$ that a occurs n times in the multiset \mathcal{M} . We denote by $a \notin \mathcal{M}$ that a is not in the multiset \mathcal{M} .

Definition 4 (Multi-subset). *A multiset $\mathcal{M}_1 = (A, m_1)$ is a multi-subset of another multiset $\mathcal{M}_2 = (B, m_2)$ if $A \subseteq B$ and $\forall a \in A : m_1(a) \leq m_2(a)$. We denote multi-subsets by $\mathcal{M}_1 \subseteq \mathcal{M}_2$ and complement them in the obvious way.*

We define *equality* between multisets as $\mathcal{M}_1 \subseteq \supseteq \mathcal{M}_2$ and denote it as $\mathcal{M}_1 = \mathcal{M}_2$. We define *strict subset* as $\mathcal{M}_1 \subseteq \not\supseteq \mathcal{M}_2$ and denote it as $\mathcal{M}_1 \subset \mathcal{M}_2$. Next we define the *join*, the *union* and the *intersection* of multisets.

Definition 5 (Join). *The join or sum of two multisets $\mathcal{M}_1 = (A, m_1)$ and $\mathcal{M}_2 = (B, m_2)$ is defined as $\mathcal{M}_1 \uplus \mathcal{M}_2 = (A \cup B, m)$, where $\forall x \in A \setminus B : m(x) = m_1(x)$, $\forall x \in B \setminus A : m(x) = m_2(x)$ and $\forall x \in A \cap B : m(x) = m_1(x) + m_2(x)$.*

Definition 6 (Union). *The union of two multisets $\mathcal{M}_1 = (A, m_1)$ and $\mathcal{M}_2 = (B, m_2)$ is $\mathcal{M}_1 \cup \mathcal{M}_2 = (A \cup B, m)$, where $\forall x \in A \setminus B : m(x) = m_1(x), \forall x \in B \setminus A : m(x) = m_2(x)$ and $\forall x \in A \cap B : m(x) = \max\{m_1(x), m_2(x)\}$.*

Definition 7 (Intersection). *The intersection of two multisets $\mathcal{M}_1 = (A, m_1)$ and $\mathcal{M}_2 = (B, m_2)$ is $\mathcal{M}_1 \cap \mathcal{M}_2 = (A \cap B, m)$, where $\forall x \in A \cap B : m(x) = \min\{m_1(x), m_2(x)\}$.*

Example 1 (Multisets). Consider the following multisets: $\mathcal{M}_1 = [1, 2, 2, 2, 4]$ and $\mathcal{M}_2 = [2, 2, 8, 8, 9, 9, 9]$. We obtain the multiset $[1, 2, 2, 2, 2, 4, 8, 8, 9, 9, 9]$ by joining \mathcal{M}_1 and \mathcal{M}_2 . The union of \mathcal{M}_1 and \mathcal{M}_2 results in $[1, 2, 2, 2, 4, 8, 8, 9, 9, 9]$ and the intersection in $[2, 2]$. Neither one of the multisets is a subset of the other: $\mathcal{M}_1 \not\subseteq \mathcal{M}_2$ and $\mathcal{M}_2 \not\subseteq \mathcal{M}_1$. \square

1.3 The theoretical operational multiset semantics of CHR

A CHR program defines a state transition system, where the transition relation is given by the rules in the CHR program. Declaratively, rules define a logical equivalence between removed constraints H_r and added constraints $B \wedge C$, provided that the kept constraints H_k are present in the constraint store and that the guard G holds: $G \rightarrow (H_k \rightarrow H_r \leftrightarrow B \wedge C)$. Operationally, rules are applied exhaustively on the CHR constraints in the constraint store until an answer state is reached. Rule application is non-deterministic, meaning that we can choose to fire any of the rules applicable to some CHR state. However, this choice is a committed choice, it cannot be undone.

Definition 8 (CHR state). *A CHR state is a tuple $\langle \mathcal{G}, \mathcal{S} \rangle$, where the goal \mathcal{G} and the constraint store \mathcal{S} are multisets of CHR constraints such that $\mathcal{G} \uplus \mathcal{S}$ represent a conjunction of constraints. An initial state or query is an arbitrary multiset of constraints $\langle \mathcal{G}, \emptyset \rangle$. In a final state or answer no more transitions are possible.*

We do not consider built-in constraints as part of a state. After all, every added built-in b/m with arguments \bar{t} can be replaced by a CHR constraints c/m with identical arguments \bar{t} and a CHR rule R_p having b/m as its guard:

$$(H_k \setminus H_r \leftrightarrow G \mid \mathbf{b}(\bar{t}), B, C) \text{ is equivalent to } (H_k \setminus H_r \leftrightarrow G \mid \mathbf{c}(\bar{t}), B, C) \text{ and } (R_p @ \mathbf{c}(\bar{t}) \leftrightarrow \mathbf{b}(\bar{t}) \mid \text{true})$$

Note that when the guard fails the CHR constraint remains in the store, becoming inapplicable. As such, the behavior of added body built-ins is represented by a guard and a CHR constraint instead. We obtain therefore the following generalized transition relation.

In the following definition let H_k , H_r and C denote multisets of CHR constraints, let G and B denote multisets of built-in constraints, let σ denote a matching substitution of the global variables appearing in the head of the rule R and let θ denote an answer substitution of the local variables appearing in the

guard but not in the head of R . Then the *transition relation*, \mapsto , between CHR states, given CT for the built-in constraints and P for the CHR constraints, is defined as follows.

Definition 9 (Transition relation).

Apply	<p><i>if</i> $(R @ H_k \setminus H_r \Leftrightarrow G \mid B, C)$ is a copy of a rule in P <i>and</i> $CT \models S \rightarrow \exists \sigma \theta ((H'_k \uplus H'_r) = (H_k \uplus H_r)\sigma) \wedge G\sigma\theta$ <i>and</i> $\mathcal{G} \uplus \mathcal{S} = H'_k \uplus H'_r \uplus \mathcal{G}' \uplus \mathcal{S}'$ <i>then</i> $\langle \mathcal{G}, \mathcal{S} \rangle \mapsto \langle \mathcal{G}', H'_k \uplus B\sigma \uplus C\sigma \uplus \mathcal{S}' \rangle$</p>
--------------	---

The definition states that for a rule to be applicable there must exist a multiset of matching constraints in the constraint store for which the guard can be satisfied. In CHR, a transition is often called a *computation step* and a sequence of computation steps is called a *computation*.

1.4 Introduction to hypergraphs

Hypergraphs generalize standard graphs by defining edges between multiple nodes instead of only two nodes.

Definition 10 (Hypergraph). A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is defined as an ordered pair, where $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ is a set of vertices or nodes and $\mathcal{E} = \{E_1, E_2, \dots, E_m\}$ is a set of hyperedges. Every hyperedge relates a set of vertices and therefore $\mathcal{E} \subseteq \mathcal{V}^2$, where \mathcal{V}^2 denotes the power-set of \mathcal{V} . \square

Many of the definitions of graphs carry verbatim to hypergraphs. The cardinality of a hyperedge E is the number of vertices the edge contains and is denoted by $c(E)$. \mathcal{H} is said to be k -uniform if every edge $E_i \in \mathcal{E}$ has cardinality $c(E_i) = k$. Therefore, a standard graph is a 2-uniform hypergraph. The degree of a vertex v is the number of edges in \mathcal{E} that contain this vertex, often denoted by $d(v)$. \mathcal{H} is called k -regular if every vertex has degree k .

It is worth noting that there is a one-to-one correspondence between hypergraphs and Boolean matrices. Indeed, any $n \times m$ matrix $A = [a_{ij}]$ such that $a_{ij} \in \{0, 1\}$ may be considered as the *incidence matrix* of a hypergraph \mathcal{H} , where each row i is associated with a vertex v_i and each column j with a hyperedge E_j . This is illustrated in the following example.

Example 2 (A hypergraph and its incidence matrix). The hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ consists of vertices $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and hyperedges $\mathcal{E} = \{E_1, E_2, E_3\} = \{\{v_2, v_3\}, \{v_4, v_5\}, \{v_1, v_2, v_3, v_4\}\}$.

Note that we have replaced the zeros in the incidence matrix in Figure 1 by dots. We obtain the cardinality of the hyperedges by summing the values of the corresponding columns: $c(E_1) = 2$, $c(E_2) = 2$ and $c(E_3) = 4$. By summing the rows of the incidence matrix we obtain the degree of the vertices in the hypergraph: $d(v_1) = 1, d(v_2) = 2, d(v_3) = 2, d(v_4) = 2, d(v_5) = 1, d(v_6) = 0$. \square

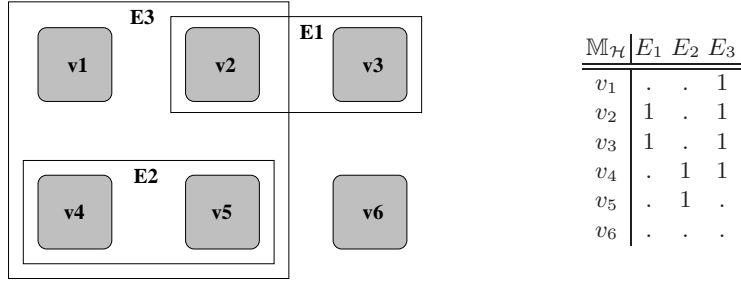


Fig. 1. A hypergraph and its incidence matrix

The generalization of a directed graph is a directed hypergraph. A directed hypergraph differs from a regular hypergraph by representing edges as ordered pairs of sets of vertices.

Definition 11 (Directed hypergraph). A directed hypergraph is a pair $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ represents the set of vertices and where $\mathcal{E} = \{E_1, E_2, \dots, E_m\}$, with E_i for $i = 1, 2, \dots, m$ directed hyperedges or hyperarcs which are ordered pairs, $E_i = (X_i, Y_i)$, of (possibly empty) subsets of vertices. Usually X_i is called the tail $T(E_i)$ of E_i and Y_i is called the head $H(E_i)$. \square

In the following, directed hypergraphs will simply be called hypergraphs. As for directed graphs, the incidence matrix of a hypergraph \mathcal{H} is an $n \times m$ matrix $[a_{ij}]$ which is defined as follows:

$$a_{ij} = \begin{cases} +2 & \text{if } v_i \in T(E_j) \wedge v_i \in H(E_j) \\ -1 & \text{if } v_i \in T(E_j) \\ +1 & \text{if } v_i \in H(E_j) \\ 0 & \text{otherwise} \end{cases}$$

Example 3 (A directed hypergraph and its incidence matrix). The hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ consists of vertices $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5\}$ and hyperedges $\mathcal{E} = \{E_1, E_2, E_3, E_4, E_5\}$ where $E_1 = (\{v_1\}, \{v_2, v_3\})$, $E_2 = (\{v_2\}, \{v_1, v_4\})$, $E_3 = (\{v_3\}, \{v_1, v_3, v_4\})$, $E_4 = (\{v_4\}, \{\})$ and $E_5 = (\{v_5\}, \{v_1, v_4, v_5\})$. Note that

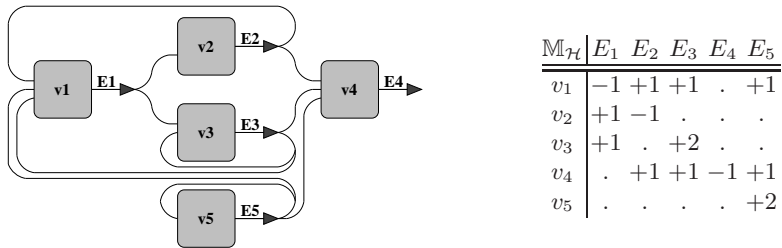


Fig. 2. A directed hypergraph and its incidence matrix

we have replaced the zeros in Figure 2 by dots. We obtain the cardinality of the hyperedges by summing the absolute values of the corresponding columns: $c(E_1) = 3, c(E_2) = 3, c(E_3) = 4, c(E_4) = 1, c(E_5) = 4$. By summing the absolute values of the rows of the incidence matrix we obtain the degree of the vertices in the hypergraph: $d(v_1) = 4, d(v_2) = 2, d(v_3) = 3, d(v_4) = 4, d(v_5) = 2$. \square

Finally, we are able to define hyperpaths and hypercycles. A hyperpath is an alternating sequence of nodes and hyperarcs that characterizes the way nodes can be reached from other nodes by using the hyperarcs of the graph.

Definition 12 (Hyperpath). *A hyperpath (or simply a path) \mathcal{P}_{st} of length q in a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a sequence of nodes and hyperarcs $\mathcal{P}_{st} = (s = v_1, E_{i_1}, v_2, E_{i_2}, \dots, E_{i_q}, v_{q+1} = t)$, where $s \in T(E_{i_1})$, $t \in H(E_{i_q})$ and $v_j \in H(E_{i_{j-1}}) \cap T(E_{i_j})$ for $j = 2, \dots, q$. \square*

Nodes s and t are the *origin* and the *destination* of \mathcal{P}_{st} , respectively, and we say that t is *connected* to s . If $t \in T(E_{i_1})$, then \mathcal{P}_{st} is said to be a cycle. This is in particular true when $t = s$. In a *simple path* all hyperarcs are distinct and a simple path is *elementary* if all nodes v_1, v_2, \dots, v_{q+1} are distinct. Similarly, we may define *simple* and *elementary cycles*. A path is said to be *cycle-free* if it does not contain any subpath which is a cycle.

2 CHR Net

In this section, we represent CHR programs by hypergraphs. We first introduce the notion of an *abstract constraint* and its relation to actual CHR constraints. Then, we explain how these abstract constraints can represent constraints in computations of a CHR program. Afterwards, we relate abstract constraints by *rule* and *match transitions*. As such, we obtain a hypergraph, called a *CHR net*, which describes a CHR program both graphically and formally. In the next section, given a CHR net, we introduce the notion of a *CHR construct*. These constructs represent computations of a CHR program, using concepts of CHR nets.

2.1 Representing CHR programs as hypergraphs

In CHR, the position of identical constraints matters. CHR constraints in rules are therefore labeled as follows:

$$R_i @ H_{(i,1)}, \dots, H_{(i,k)} \setminus H_{(i,k+1)}, \dots, H_{(i,n)} \Leftrightarrow G_{(i,1)}, \dots, G_{(i,u)} \mid B_{(i,1)}, \dots, B_{(i,v)}, C_{(i,1)}, \dots, C_{(i,m)}.$$

The rules of a CHR program relate CHR constraints. The head constraints of a rule represent the constraints *required* for rule application. The kept heads and added CHR constraints represent the *available* constraints after rule application. To describe the constraints they represent, we introduce abstract CHR constraints.

Definition 13 (Abstract CHR constraint). Let CT be a constraint theory for built-ins. Then an abstract CHR constraint $\mathcal{C} = (C, B)$ represents the set of instances of a CHR constraint C using a variable substitution σ , such that the conjunction of built-in constraints $B\sigma$ is satisfiable in CT . We define a function $\wp : \mathcal{C} \mapsto \{C\sigma \mid CT \models \exists\theta : B\sigma\theta\}$, that maps an abstract CHR constraint to the set of its instances, conditional on the satisfiability of B . \square

There are two types of abstract CHR constraints in a CHR program. An *abstract input constraint* represents the set of CHR constraints that match with a CHR head constraint conditional on the guards of the rule in which the head appears.

Definition 14 (Abstract input constraint of a CHR rule). An abstract input constraint of a CHR rule R_i in a CHR program P is an abstract constraint, $in_{(i,j)} = (H_{(i,j)}, G_i)$, which represents the set of constraints $\wp(in_{(i,j)})$ that can be used to match with the head $H_{(i,j)}$ conditional on the guards $G_i = G_{(i,1)} \wedge \dots \wedge G_{(i,u)}$ in R_i . \square

An *abstract output constraint* represents the CHR constraints which become available after rule application. These are the kept head constraints and the added CHR constraints, both of which are conditional on the guards of the rule.

Definition 15 (Abstract output constraint of a CHR rule). An abstract output constraint of a CHR rule R_i in a CHR program P is an abstract constraint, $out_{(i,j)}$, which represents the set of constraints $\wp(out_{(i,j)})$ that can become available after rule application conditional on the guards $G_i = G_{(i,1)} \wedge \dots \wedge G_{(i,u)}$ of R_i . Abstract output constraints relate to both kept constraints and added CHR constraints as follows:

$$\begin{cases} \text{For } j = 1, \dots, k & out_{(i,j)} = (H_{(i,j)}, G_i) \\ \text{For } j = k + 1, \dots, k + m & out_{(i,j)} = (C_{(i,j-k)}, G_i) \end{cases}$$

\square

Every CHR state can be represented by a multiset of abstract constraints taken from the set of output constraints of a CHR program. Then, every element in a CHR state is represented by an abstract output constraint: the constraint appears in the set of actual constraints that the abstract constraint represents. The notion of an abstract state as a multiset of the available constraints in a CHR program corresponds to multiple CHR states. A single CHR state corresponds usually to multiple abstract states as well as these abstract states encode the origin of the constraints. That is, which rule application gave rise to the constraint. The required constraints on the other hand describe required constraints and thus describe whether and in what way certain available constraints can be used to fire certain rules. This information will be used to describe dependencies between available constraints and next available constraints.

Example 4 (Greatest Common Divisor). The following implementation P for the Euclidian algorithm for calculating the GCD is a typical multi-headed terminating CHR program. The first rule removes only constraints, while the second replaces two constraints by simpler constraints:

$$\begin{aligned} R_1 @ gcd(0) &\Leftrightarrow true. \\ R_2 @ gcd(M) \setminus gcd(N) &\Leftrightarrow N \geq M, M > 0, L \text{ is } N - M \mid gcd(L). \end{aligned}$$

We denote by \mathcal{In}_{R_i} and \mathcal{Out}_{R_i} the sets of abstract input constraints and output constraints of a rule R_i . By \mathcal{In}_P and \mathcal{Out}_P , we denote the inputs and outputs of a set of rules P . For the GCD program we therefore obtain the following abstract inputs and outputs:

$$\begin{aligned} \mathcal{In}_P &= \mathcal{In}_{R_1} \cup \mathcal{In}_{R_2} = \{in_{(1,1)}, in_{(2,1)}, in_{(2,2)}\}, \text{ where} \\ &- in_{(1,1)} = (gcd(0), true) \\ &- in_{(2,1)} = (gcd(M), N \geq M, M > 0, L \text{ is } N - M) \\ &- in_{(2,2)} = (gcd(N), N \geq M, M > 0, L \text{ is } N - M) \end{aligned}$$

$$\begin{aligned} \mathcal{Out}_P &= \mathcal{Out}_{R_1} \cup \mathcal{Out}_{R_2} = \{out_{(2,1)}, out_{(2,2)}\}, \text{ where} \\ &- out_{(2,1)} = (gcd(M), N \geq M, M > 0, L \text{ is } N - M) \\ &- out_{(2,2)} = (gcd(L), N \geq M, M > 0, L \text{ is } N - M) \end{aligned}$$

The rules of a CHR program relate abstract inputs to abstract outputs. We call this relation a *rule transition relation*.

Definition 16 (Rule transition relation). A rule transition of a CHR program P is an ordered pair $T_i = (\mathcal{In}_{R_i}, \mathcal{Out}_{R_i})$, relating the set of abstract input constraints $\{in_{(i,1)}, \dots, in_{(i,n)}\} = \mathcal{In}_{R_i}$ of a rule $R_i \in P$ to the set of abstract output constraints $\{out_{(i,1)}, \dots, out_{(i,k+m)}\} = \mathcal{Out}_{R_i}$ of R_i . The rule transition relation $\mathcal{T} = \{T_i \mid R_i \in P\}$ of a CHR program P is the set of rule transitions T_i of a program P . \square

Example 5 (GCD cont.). The GCD program has two rules and thus two different rule transitions:

$$\begin{aligned} \mathcal{T} &= \{T_1, T_2\}, \text{ where} \\ &- T_1 = (\{in_{(1,1)}\}, \{\}) \\ &- T_2 = (\{in_{(2,1)}, in_{(2,2)}\}, \{out_{(2,1)}, out_{(2,2)}\}) \end{aligned}$$

The first rule has one abstract input constraint and no abstract output constraints. The second rule relates one input constraint and one output constraint, while the third rule relates an input constraint with two abstract output constraints. We can represent abstract input and abstract output constraints by vertices in a hypergraph. The rules then correspond to hyperarcs that connect the abstract input constraints of a rule to the abstract output constraints as is depicted in Figure 3). \square

Abstract outputs relate to abstract inputs by a *match transition relation*. This second kind of relation is the result of a dependency analysis between the inputs and outputs of a CHR program. It relates the constraints which become available after rule application to the constraints which are required for rule application.

Definition 17 (Match transition relation). *A match transition of a CHR program P is an ordered pair $M_{(i,j,k,l)} = (\{out_{(i,j)}\}, \{in_{(k,l)}\})$, relating an output $out_{(i,j)} = (C_{(i,j)}, G_i)$ of $\mathcal{O}ut_P$ to an input $in_{(k,l)} = (C_{(k,l)}, G_k)$ of $\mathcal{I}n_P$ such that $CT \models \exists \sigma : ((C_{(i,j)} = C_{(k,l)}) \wedge G_i \wedge G_k)\sigma$. The match transition relation \mathcal{M} is the set of all match transitions $M_{(i,j,k,l)}$ in P . \square*

Note that a match transition exists if the input and output it connects represent sets of constraints with a common intersection: $\wp(out_{(i,j)}) \cap \wp(in_{(k,l)}) \neq \emptyset$.

Example 6 (GCD cont.). We obtain for the GCD program the following match transition relation:

$$\mathcal{M} = \{M_{(2,1,2,1)}, M_{(2,1,2,2)}, M_{(2,2,1,1)}, M_{(2,2,2,1)}, M_{(2,2,2,2)}\}.$$

We observe that the first output of the second rule does not match with the input of the first rule, as their intersection $\wp(out_{(2,1)}) \cap \wp(in_{(1,1)}) = \emptyset$. After all, the argument of the output constraint will always be greater than 0, while the input must have an argument which is valued 0. The match transition relation can be represented by arcs as well as is shown in Figure 3. \square

2.2 CHR net

A hypergraph representation of a CHR program takes the form of a *bipartite hypergraph*. A bipartite hypergraph, as for its corresponding definition in standard graph theory, can be decomposed into two distinct sets of vertices and two distinct sets of hyperedges. The two sets of hyperedges connect the vertices from the first set to those of the second set and the vertices from the second to those of the first, respectively.

Definition 18 (Directed bipartite hypergraph). *A directed bipartite hypergraph $\mathcal{N} = \langle \mathcal{V}_1, \mathcal{V}_2, \mathcal{E}_1, \mathcal{E}_2 \rangle$ is defined as a quadruple, where \mathcal{V}_1 and \mathcal{V}_2 are distinct sets of vertices and \mathcal{E}_1 and \mathcal{E}_2 are distinct sets of hyperedges. Every directed hyperedge in \mathcal{E}_1 relates elements from \mathcal{V}_1 to \mathcal{V}_2 . Every directed hyperedge in \mathcal{E}_2 relates elements from \mathcal{V}_2 to \mathcal{V}_1 . \square*

We define a CHR program as a directed bipartite hypergraph as illustrated in the example below. We obtain such a representation by combining previous results and call the obtained graph a CHR net.

Example 7 (CHR net of the GCD program). We derive the CHR net for P :

$$\begin{aligned} R_1 & @ \text{gcd}(0) \Leftrightarrow \text{true}. \\ R_2 & @ \text{gcd}(M) \setminus \text{gcd}(N) \Leftrightarrow M > 0, N \geq M, L \text{ is } N - M \mid \text{gcd}(L). \end{aligned}$$

CHR net of P

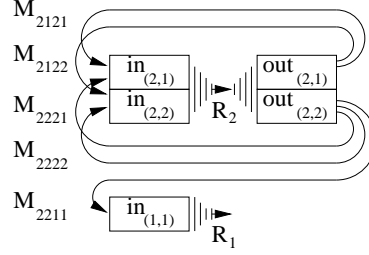


Fig. 3. CHR net of the GCD program

The CHR net of the GCD program depicted in Figure 3 is the directed bipartite hypergraph $\mathcal{N}_P = \langle \mathcal{In}, \mathcal{Out}, \mathcal{T}, \mathcal{M} \rangle$, where

$$\begin{aligned}
 \mathcal{In} &= \{in_{(1,1)}, in_{(2,1)}, in_{(2,2)}\} \\
 \mathcal{Out} &= \{out_{(2,1)}, out_{(2,2)}\} \\
 \mathcal{T} &= \{T_1, T_2\} = \{(\{in_{(1,1)}\}, \{\}), (\{in_{(2,1)}, in_{(2,2)}\}, \{out_{(2,1)}, out_{(2,2)}\})\} \\
 \mathcal{M} &= \{M_{(2,1,2,1)}, M_{(2,1,2,2)}, M_{(2,2,1,1)}, M_{(2,2,2,1)}, M_{(2,2,2,2)}\} \quad \square
 \end{aligned}$$

A CHR net is a formal and graphical interpretation of a CHR program and is defined as follows.

Definition 19 (CHR net). A CHR net of a CHR program P is a directed bipartite hypergraph $\mathcal{N}_P = \langle \mathcal{In}, \mathcal{Out}, \mathcal{T}, \mathcal{M} \rangle$, where the elements of the tuple are defined in Definition 14, 15, 16 and 17, respectively. \square

A CHR net therefore represents which constraints can become available during the execution of a CHR program and how these result in next available constraints. It describes the flow of constraints in a CHR program and provides for a description of computations in CHR programs. In the next section we relate the CHR net of a program to the different computations of the program.

3 CHR construct

A computation in CHR is traditionally regarded as an alternating sequence of rule applications and CHR states. The following example describes such a computation for the GCD program.

Example 8 (GCD cont.). Given a query $I = [gcd(6), gcd(3)]$, the following sequence represents a computation of P with I .

$$[gcd(6), gcd(3)] \mapsto_{R_2} [gcd(3), gcd(3)] \mapsto_{R_2} [gcd(0), gcd(3)] \mapsto_{R_1} [gcd(3)]$$

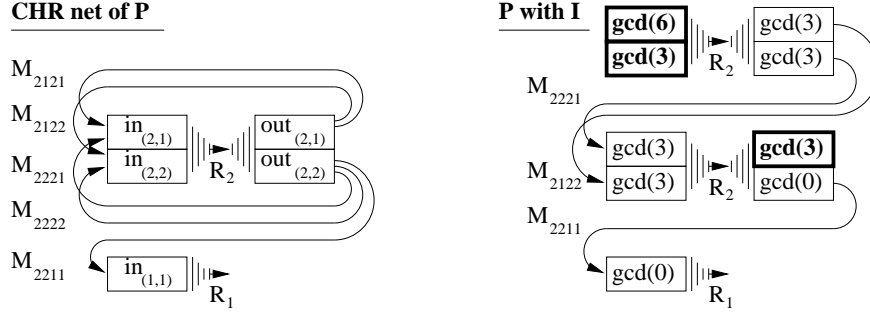


Fig. 4. CHR net of GCD program and a possible computation of the program

A different representation of this computation uses concepts of CHR nets, as is shown in Figure 6. Notice that inputs without an incoming match transition represent constraints required from a query and that outputs without an outgoing match transition represent constraints part of the answer. Computations of CHR programs can therefore be represented using only concepts from CHR nets. That is, a multiset of rules that represents rule applications and a multiset of match transitions that represents dependencies between rule applications. \square

3.1 CHR construct

According to Example 6, we generalize computations by constructs. A construct is a pair of multisets, where the first element represents a multiplicity of rule applications and the other, a multiset of dependencies between rule applications represented by match transitions.

Definition 20 (Construct). *A construct is a pair of multisets $((S, \mu), (M, \mu))$, where (S, μ) is a multiset of rules $S = \{R_1, \dots, R_v\}$ and where (M, μ) is a multiset of match transitions $M = \{M_1, \dots, M_w\}$.* \square

Not every construct can represent an actual computation. After all, some constructs represent computations with generated outputs, used to provide for multiple inputs. Also, some constructs may have more incoming match transitions than required to fire a rule. As both cases do not occur in any computation, we constrain the number of outgoing match transitions from a given output to the number of times its rule occurs in the multiset of rule applications. Similarly, the number of incoming match transitions in a given input may not exceed the number of times its rule occurs in the multiset of rules. Otherwise, according to the pigeon hole principle, at least one constraint has an incoming or outgoing match transition which cannot exist in any computation.

Both properties are expressed by a system of linear inequalities. We obtain therefore the following definition of a CHR construct.

Definition 21 (CHR construct). Let $\mathcal{N}_S = \langle \mathcal{In}_S, \mathcal{Out}_S, \mathcal{T}_S, \mathcal{M}_S \rangle$ be the CHR net of a set of rules S . Then, a CHR construct is a construct $((S, \mu), (M_S, \mu))$, such that:

$$\begin{aligned} \forall in_{(i,j)} \in \mathcal{In}_S : & \sum_{\forall M_{(k,l,i,j)} \in \mathcal{M}_S} \mu(M_{(k,l,i,j)}) \leq \mu(R_i) \\ \forall out_{(i,j)} \in \mathcal{Out}_S : & \sum_{\forall M_{(i,j,k,l)} \in \mathcal{M}_S} \mu(M_{(i,j,k,l)}) \leq \mu(R_i) \end{aligned}$$

We say that a CHR construct is traversed, if every rule in it was applied once according to the match transitions of the construct. \square

Revisiting Example 11, we obtain as a representation of the computation, the CHR construct $([R_1, R_2, R_2], [M_{(2,2,1,1)}, M_{(2,2,2,1)}, M_{(2,1,2,2)}])$. Notice that a CHR construct can represent multiple computations and that not all CHR constructs represent actual computations of the program.

3.2 Minimality of CHR constructs

The set of all possible CHR constructs is represented by a system of linear inequalities derived from the definition of a CHR construct. That is,

$$\begin{aligned} \forall in_{(i,j)} \in \mathcal{In}_C : & \sum_{\forall M_{(k,l,i,j)} \in \mathcal{M}_C} m_{(k,l,i,j)} \leq m_{R_i} \\ \forall out_{(i,j)} \in \mathcal{Out}_C : & \sum_{\forall M_{(i,j,k,l)} \in \mathcal{M}_C} m_{(i,j,k,l)} \leq m_{R_i} \end{aligned}$$

Where, every $m_{(i,j,k,l)}$ and m_{R_i} are variables that take positive integer values that represent the multiplicity $\mu(M_{(i,j,k,l)})$ of match transition $M_{(i,j,k,l)}$ and the number of applications $\mu(R_i)$ of a rule R_i in the CHR construct, respectively. Therefore, any positive integer solution describes a CHR construct in the CHR program.

It has been proven that such systems have a unique computable basis of solutions, called a Hilbert basis, from which any solution to the system can be computed by positive linear combinations of the solution vectors. As such, a concept of minimality is introduced as any CHR construct can be represented by joining minimal CHR constructs. Here, we mean by joining the CHR constructs $(\mathcal{S}', \mathcal{M}'_S)$ and $(\mathcal{S}'', \mathcal{M}''_S)$, joining their corresponding elements together: $(\mathcal{S}' \uplus \mathcal{S}'', \mathcal{M}'_S \uplus \mathcal{M}''_S)$, which corresponds to adding solution vectors.

Example 9 (GCD cont.). We revisit the GCD program and obtain the following system of inequalities, describing its CHR constructs.

$$\begin{aligned} m_{(2,2,1,1)} &\leq m_{R_1} & m_{(2,1,2,1)} + m_{(2,1,2,2)} &\leq m_{R_2} \\ m_{(2,1,2,1)} + m_{(2,2,2,1)} &\leq m_{R_2} & m_{(2,2,2,1)} + m_{(2,2,2,2)} &\leq m_{R_2} \\ m_{(2,1,2,2)} + m_{(2,2,2,2)} &\leq m_{R_2} \end{aligned}$$

By adding slack variables we obtain a system of equations with positive integer solutions, from which we can compute the following Hilber basis. The basis consists of a set of eleven minimal CHR constructs.

$$\begin{array}{ll}
([R_1], []) & ([R_2], [M_{(2,1,2,2)}, M_{(2,2,2,1)}]) \\
([R_2], []) & ([R_2], [M_{(2,1,2,1)}, M_{(2,2,2,2)}]) \\
([R_2], [M_{(2,1,2,1)}]) & ([R_1, R_2], [M_{(2,2,1,1)}]) \\
([R_2], [M_{(2,1,2,2)}]) & ([R_1, R_2], [M_{(2,2,1,1)}, M_{(2,1,2,1)}]) \\
([R_2], [M_{(2,2,2,1)}]) & ([R_1, R_2], [M_{(2,2,1,1)}, M_{(2,1,2,2)}]) \\
([R_2], [M_{(2,2,2,2)}]) &
\end{array}$$

By joining elements from the basis, we can obtain any CHR construct. Therefore, we can represent every computation of the gcd program by minimal CHR constructs only. \square

3.3 Cyclic CHR constructs

In CHR, there are two different kinds of CHR constructs. Either a CHR construct is *cyclic* or it is not. Cyclic constructs replace the removed constraints in computations, while acyclic constructs do not.

To express this property of a CHR construct, we require that for every input in the construct, exactly one incoming match transition exists, resulting in a more strict version of the requirement on inputs as in our definition of a CHR construct. A cyclic CHR construct is therefore defined as follows.

Definition 22 (Cyclic CHR construct). *Let $C = \{R_1, \dots, R_n\}$ be a set of rules and $\mathcal{N}_C = \langle \mathcal{I}n_C, \mathcal{O}ut_C, \mathcal{T}_C, \mathcal{M}_C \rangle$ its CHR net. Then, a CHR construct $((C, \mu), (\mathcal{M}_C, \mu))$ is cyclic iff:*

$$\forall in_{(i,j)} \in \mathcal{I}n_C : \sum_{\forall M_{(k,l,i,j)} \in \mathcal{M}_C} \mu(M_{(k,l,i,j)}) = \mu(R_i)$$

We call cyclic constructs also self-sustainable constructs. \square

Notice that according to the definition of a cyclic construct, every constraint is provided for by generated constraints after the application of every rule in the cyclic construct. Analogously, we can compose a system which describes cyclic CHR constructs in a CHR program.

$$\begin{array}{l}
\forall in_{(i,j)} \in \mathcal{I}n_C : \sum_{\forall M_{(k,l,i,j)} \in \mathcal{M}_C} m_{(k,l,i,j)} = m_{R_i} \\
\forall out_{(i,j)} \in \mathcal{O}ut_C : \sum_{\forall M_{(i,j,k,l)} \in \mathcal{M}_C} m_{(i,j,k,l)} \leq m_{R_i}
\end{array}$$

Where, every $m_{(i,j,k,l)}$ and m_{R_i} are variables that take positive integer values that represent the multiplicity $\mu(M_{(i,j,k,l)})$ of match transition $M_{(i,j,k,l)}$ and the number of applications $\mu(R_i)$ of a rule R_i in the CHR construct, respectively.

Therefore, any positive integer solution describes a CHR construct in the CHR program.

Obviously, a concept of minimality exists for cyclic constructs as well. From these minimal cyclic constructs, any cyclic construct in the program can be computed, using positive linear combinations only.

Example 10 (GCD cont.). For the GCD program, we obtain the following system of inequalities, describing the cyclic CHR constructs of the program.

$$\begin{aligned} m_{(2,2,1,1)} &= m_{R_1} & m_{(2,1,2,1)} + m_{(2,1,2,2)} &\leq m_{R_2} \\ m_{(2,1,2,1)} + m_{(2,2,2,1)} &= m_{R_2} & m_{(2,2,2,1)} + m_{(2,2,2,2)} &\leq m_{R_2} \\ m_{(2,1,2,2)} + m_{(2,2,2,2)} &= m_{R_2} \end{aligned}$$

By adding slack variables we obtain a system of equations with positive integer solutions, from which we can compute the following Hilber basis.

$$([R_2], [M_{(2,1,2,2)}, M_{(2,2,2,1)}]) \quad ([R_2], [M_{(2,1,2,1)}, M_{(2,2,2,2)}])$$

By joining elements from the basis, we can obtain every cyclic construct of a CHR program. Notice that minimal CHR cycles are a subset of minimal CHR constructs. \square

Notice furthermore that if the basis is empty, that the construct cannot be a cycle and therefore cannot lead to an infinite computation. In termination analysis, these constructs can be neglected.

4 Termination of CHR programs

Traditionally, computations in CHR are regarded as alternating sequences of rule applications and CHR states. In the following example such a computation for a finite query is given for the GCD program.

Example 11 (GCD cont.). Given a query $I = [gcd(6), gcd(3)]$, the following sequence represents a computation for P with I .

$$[gcd(6), gcd(3)] \mapsto_{R_2} [gcd(3), gcd(3)] \mapsto_{R_2} [gcd(0), gcd(3)] \mapsto_{R_1} [gcd(3)]$$

Such a representation of computations does not encode information regarding dependencies between rules. E.g. in the subcomputation $[gcd(3), gcd(3)] \mapsto_{R_2} [gcd(0), gcd(3)]$, we do not know which $gcd(3)$ constraint is removed. \square

The applicability of rules in CHR depends on the presence of multiple constraints in the constraint store. Therefore, multiple rules may be applicable on different combinations of constraints of which required constraints may partially overlap. This is in contrast to single-headed languages, where applicability depends on disjunct elements of the state. This makes termination analysis of CHR programs a more difficult problem to tackle. After all, true cyclic behavior in CHR isn't as apparent as is the case in single-headed languages. It requires in general the replacement of multiple data elements in stead of only one.

Example 12 (Termination of multi-headed languages). Consider the program:

$$R_1 @ a, a \Leftrightarrow b, b. \quad R_2 @ b \Leftrightarrow a.$$

The program is non-terminating for queries with at least two constraints a or b . One can immediately observe a difference between cyclic behavior in single-headed programs and multi-headed programs. In this particular case, the second rule has to fire twice in order to compensate for the removal of constraints by a single application of the first rule. As can be observed, termination in CHR not only depends on decreases in interpretation of constraints, but depends as well on decreases in number of constraints of a certain type in the constraint store. The latter is referred to as exhaustion of some constraint. \square

A further point of consideration in termination analysis of CHR is its multiset semantics. After all, multiple identical constraints may be involved in different kinds of cyclic behavior. To illustrate this we show in Figure 5 cyclic behavior for the GCD program in Example 11. Both diagrams show part of a computation for identical queries, where we only represent applications of the second rule.

The left diagram consists of cyclic behavior wrt. only two constraints. The right diagram on the other hand combines such "simple" cycles and results in cyclic behavior that cannot be represented by cyclic behavior for two constraints. This would not be a problem if it were not for a fire-once policy on labeled constraints. After all, as identical constraints are labeled to prevent them from firing propagation rules infinitely many times, the constraint store behaves as a set of labels wrt. propagation rules.

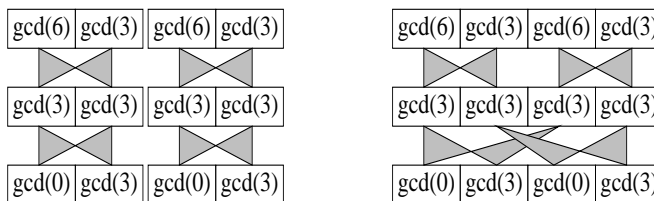


Fig. 5. Different cyclic behavior for two constraints as for four constraints

In the following subsections we first discuss a more expressive description of CHR computations. Instead of representing computations by sequences, we represent them as directed hypergraphs. Based on this new representation, we define a dependency relation. Then, we define termination wrt. our new representation for computations and redefine the RC for general CHR programs in the context of dependencies in computations. We show that the RC is composable according to composition of dependencies. Afterwards, we define an abstraction for dependencies in computations using concepts of a CHR net and extend this abstraction by satisfaction of the RC. Finally, we refine our abstraction and propose a modular approach to prove termination of CHR programs.

4.1 Computations of CHR programs

As stated earlier, the applicability of rules depends on partially overlapping subsets of the state. Furthermore, dependencies between rules may be partial as well. We require therefore a more refined description of computations in CHR. That is, one in which dependencies between rule applications are made explicit. In further contrast to single-headed languages, we do not represent all possible computations originating in a query. After all, rule applications are a committed choice. In stead, we represent single computations for a query. In Figure 6 we show a computation for Example 11.

The hypergraph representation of a computation of a CHR program is cycle-free. Nodes without an incoming hyperarc represent constraints required from a query and nodes without an outgoing arc represent constraints part of the answer. Both answer and query nodes are marked in bold in Figure 6. All other nodes are called *intermediate* nodes and represent constraints that have both an incoming and an outgoing arc. These nodes represent constraints made available by the application of some rule and used afterwards in the application of some other rule. The hyperarcs represent rule applications in the computation.

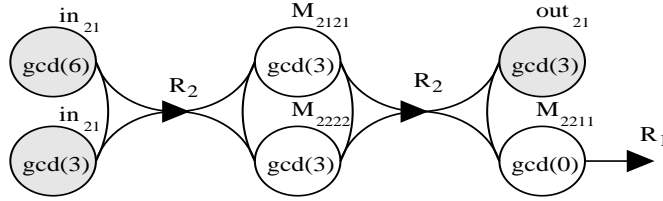


Fig. 6. A computation for the GCD program

By such computations, we represent the state in a distributed way. A query is represented by query nodes, one for every constraint in the query. A next state is obtained by replacing nodes according to the hyperarcs in the computation. Answer nodes represent the answer in a distributed way and cannot contain any combination of constraints for which a rule in the program is applicable. Intermediate nodes are part of some state during the computation.

Notice that a computation does not impose a strict order between applications of rules in the program. For some state, that is a set of nodes in the computation, there may be several rules applicable on disjunct subsets of nodes. This can be seen in Figure 5. There, the second rule is applicable simultaneously on two disjunct subsets of the query.

As stated earlier, the position of identical constraints matters. Therefore, we obtain the following definition for computations of CHR programs.

Definition 23 (CHR computation). *Let P be a CHR program and I a finite query. Then a computation for P with I is a cycle-free directed hypergraph $\mathcal{P}_I =$*

(N, E) , where $N = \{N_1, \dots, N_n\}$ is the set of nodes of the hypergraph and where $E = \{E_1, \dots, E_e\}$ is the set of hyperarcs, for which:

1. The nodes N_i of \mathcal{P}_I are labeled with an abstract CHR constraint and a substitution $((C, B), \phi)$, such that each node represents a constraint $C\phi$ part of the computation. We require that $C\phi \in \wp((C, B))$ as the set of built-ins must be satisfied wrt. ϕ . We define a function $\aleph : ((C, B), \phi) \mapsto C\phi$ from labels to actual constraints. Note that \aleph is only defined for $C\phi \in \wp((C, B))$.
2. For each constraint C in I there exists a query node, labeled with an abstract input constraint $in_{(i,j)} = (C_{(i,j)}, G_i)$ and a substitution σ , such that $\aleph((C_{(i,j)}, G_i), \sigma) = C$. Query nodes can only have an outgoing arc.
3. Answer nodes represent constraints C part of the answer and are labeled with abstract output constraints $out_{(i,j)} = (C_{(i,j)}, G_i)$ and a substitution $\sigma\theta$, such that $\aleph((C_{(i,j)}, G_i), \sigma\theta) = C$. Answer nodes can only have an incoming arc and there may be no combination of answer nodes representing actual constraints that can be used to fire a rule in the program.
4. All other nodes in computations are called intermediate nodes and have one incoming and one outgoing arc. These nodes represent constraints C that became available after the application of some rule and are used afterwards in the application of some other rule in the program. Intermediate nodes are labeled with abstract constraints $M_{(i,j,k,l)} = \wp^{-1}(\wp(out_{(i,j)}) \cap \wp(in_{(k,l)}))$ at the intersection of an abstract input and output constraint and a substitution $\sigma_i\theta_i\sigma_k$, such that $\aleph(M_{(i,j,k,l)}, \sigma_i\theta_i\sigma_k) = C$.
5. The hyperarcs of \mathcal{P}_I are labeled with a substitution $\sigma_i\theta_i$ and a fresh variant of the rule R_i represented by it. The hyperarc connects nodes labeled with constraints that are up to a substitution σ_i variants of the head constraints of the rule, to nodes labeled with constraints that are up to a substitution $\sigma\theta$ variants of the kept and added body constraints of the rule, such that $CT \models G\sigma\theta$. Here, we refer to σ as the match substitution and to θ as the computed answer substitution of a rule application for a match substitution σ and we annotate the hyperarcs with these substitutions. \square

The advantage of such a representation for CHR computations is that we can verify dependencies between individual constraints in the program, something which is not possible for computations represented as sequences of rule applications. As such we can verify cyclic behaviour of CHR programs at the level of individual constraints. Reasons for termination of programs become as such more clear and interpretations for constraints can be found much more easy. This is discussed in the following.

4.2 Dependency relation in CHR computations

We can define a dependency relation between nodes of CHR computation according to their connectedness by hyperarcs representing rule applications.

Definition 24 (Dependency relation). Consider two nodes $N_i = (in_{(i,j)}, \sigma_i)$ and $N_j = (out_{(i,k)}, \sigma_i\theta_i)$ that are connected by a hyperarc $E_k = (R_i, \sigma_i\theta_i)$ in a

computation of a CHR program P with a finite query I . Then we say that N_j depends directly on N_i . We define the dependency relation as the irreflexive transitive closure of the direct dependency relation. We call a path between two nodes in the computation, such that one depends on the other, a dependency path and represent it by an alternating sequence of nodes and rule applications:

$$[(in_{(i,j)}, \sigma_1) \rightarrow_{R_1 \sigma_1 \theta_1} (M_{(i,u,v,w)}, \sigma_1 \theta_1 \sigma_2) \rightarrow_{R_2 \sigma_2 \theta_2} \cdots \rightarrow_{R_n \sigma_n \theta_n} (out_{(k,l)}, \sigma_n \theta_n)]$$

We call the root of the dependency path a query node and represent it by an abstract input. The tail is called an answer node and is represented by an abstract output constraint. In both cases the nodes may appear in a computation for the program as intermediate nodes. We represent them in the context of dependency paths however as abstract inputs and outputs. Finally, any other node in the dependency path in between the root and the tail are intermediate nodes and are represented as such. Note that square bracket in our representation are merely present for improving readability. \square

There is some structure in the set of dependency paths. If we have two dependency paths, one going from N_1 to N_2 and one going from N_3 to N_4 , we can if $N_2 = N_3$ define their composition, resulting in the dependency path from N_1 to N_4 . For the set of dependency paths in a computation, composition is a partial associative operation $* : S \times S \mapsto S$ and this set is therefore, according to the nomenclature in algebra, a semi-groupoid. We reformulate composition based on our representation of dependency paths. Let $(out_{(i,k)}, \sigma_i \theta_i) = (in_{(u,v)}, \sigma_u)$. Then,

$$[(in_{(i,j)}, \sigma_i) \rightarrow_{R_i \sigma_i \theta_i} (out_{(i,k)}, \sigma_i \theta_i)] * [(in_{(u,v)}, \sigma_u) \rightarrow_{R_u \sigma_u \theta_u} (out_{(u,w)}, \sigma_u \theta_u)] = [(in_{(i,j)}, \sigma_i) \rightarrow_{R_i \sigma_i \theta_i} (M_{(i,k,u,v)}, \sigma_i \theta_i \sigma_u) \rightarrow_{R_u \sigma_u \theta_u} (out_{(u,w)}, \sigma_u \theta_u)]$$

Note that the intermediate node, labeled with $(M_{(i,k,u,v)}, \sigma_i \theta_i \sigma_u)$, is obtained through composition, where that the tail of the first path corresponds to the abstract output $(out_{(i,k)}, \sigma_i \theta_i)$ and where the root of the second path corresponds to the abstract input $(in_{(u,v)}, \sigma_u)$, such that $M_{(i,k,u,v)} = \wp^{-1}(\wp(out_{(i,k)}) \cap \wp(in_{(u,v)}))$.

Example 13 (GCD cont.). In the computation as shown in Figure 6, $gcd(0)$ depends on $gcd(3)$ directly and depends on both query nodes $gcd(6)$ and $gcd(3)$ indirectly. However, $gcd(6)$ does not depend on any other constraint in the computation. \square

It is straightforward that if an infinite dependency path exists in some computation that an infinite sequence of nodes must exist in the computation, such that each node depends on a previous one in the considered sequence.

Lemma 1. *If there is an infinite dependency path in a computation corresponding to a program and a query then there is an infinite sequence of nodes N_1, N_2, \dots such that for each i , N_{i+1} depends on N_i .*

Proof. Straightforward. \square

4.3 Termination of CHR programs

Infinite computations correspond to an infinite directed graph, with a finite number of query nodes, an infinite number of intermediate nodes and a possibly infinite number of answer nodes. We say that a CHR program terminates if there exist no infinite computations given a finite query. We call this kind of termination universal termination of a program and define it as follows.

Definition 25 (Universal termination). *A CHR program P terminates for a finite query I iff all computations for P with I are finite.* \square

We reformulate termination of CHR programs by considering infinite dependency paths. The following lemma is used to prove that a CHR program without an infinite dependency path terminates for all finite queries. The prove demonstrates that in any infinite computation at least one infinite dependency path must exist. As such, we relate termination of a CHR program to the existence of infinite dependency paths in computations.

Lemma 2. *An infinite computation contains an infinite dependency path.*

Proof. In an infinite computation there are an infinite number of rule applications. Therefore, given a finite query, there are an infinite number of head constraints required that must become available after the application of some other rule in the computation. This implies the existence of an infinite number of intermediate nodes in our representation of a computation in CHR. Therefore, either there exists an infinite dependency path or there exists an infinite number of finite dependency paths. The latter cannot be the case as an infinite number of finite dependency paths requires an infinite query. \square

Therefore, if we prove non-existence of infinite dependency paths in computations of a program with finite queries, we prove that a program must be terminating.

Theorem 1 (Termination). *A CHR program P terminates for a finite query I iff no infinite dependency paths exist in computations for P with I .*

Proof. \Rightarrow If a program for a finite query is terminating, then only a finite number of rules is applied and thus only a finite number of dependency paths may exist. \Leftarrow If there may not exist an infinite dependency path, then given Lemma 2 there cannot exist an infinite computation and therefore according to Definition 25, the program must terminate. \square

It's now sufficient to define conditions that if satisfied guarantee non-existence of infinite dependency paths in computations of the program with finite queries. In the next subsection, we redefine the RC for general CHR programs to a RC for dependency paths and show that this RC is composable. Next, we discuss how to abstract CHR computations by using concepts from CHR nets combined with the RC for dependency paths.

4.4 The RC for dependency paths

In this subsection we redefine the RC for general CHR programs in the context of dependency paths. As dependency paths may consist of multiple rules, we reformulate the existing conditions.

First we define the call set for a program P with a finite query I as the set $Call(P, I)$ of all constraints which may enter the constraint store during some computation of P with I . A ranking function $|\cdot|: Call(P, I) \mapsto \mathbb{N}$ is then a monotonic mapping from elements of the call set to the set of natural numbers.

Definition 26 (RC for dependency paths). *Let B be a dependency path in a computation of a CHR program P with a finite query I . Then B is of the following form:*

$$(in_{(i,j)}, \sigma_1) \rightarrow_{R_1 \sigma_1 \theta_1} (M_{(i,u,v,w)}, \sigma_1 \theta_1 \sigma_2) \rightarrow_{R_2 \sigma_2 \theta_2} \cdots \rightarrow_{R_n \sigma_n \theta_n} (out_{(k,l)}, \sigma_n \theta_n)$$

Let \mathcal{N}_a represent the set of added constraint in B and \mathcal{N}_r the removed constraints in B . Let max be a mapping of a set of constraints to the max rank in the set, given some ranking function and \sharp a mapping of a set of constraints to the number of elements of rank max , wrt. some ranking function. Then B satisfies the RC for general CHR programs, wrt. a ranking function $|\cdot|$ for $Call(P, I)$, iff

1. there is a hyperarc in the dependency path representing a propagation rule $\rightarrow_{R_i \sigma_i \theta_i}$ with kept constraints $in_{(i,1)} \sigma_i, \dots, in_{(i,m_i)} \sigma_i$ ranked strictly higher by $|\cdot|$ than any of the added constraints in \mathcal{N}_a . That is, $\forall in_{(i,j)} \sigma_i : |in_{(i,j)} \sigma_i| > max(\mathcal{N}_a)$
2. there is a decrease in number of maximally ranked constraints wrt. $|\cdot|$, where we disregard the kept constraints of the rules in the dependency path. Therefore, either $max(\mathcal{N}_r) > max(\mathcal{N}_a)$ or $max(\mathcal{N}_r) = max(\mathcal{N}_a)$ and $\sharp(\mathcal{N}_r) > \sharp(\mathcal{N}_a)$.

In verifying the RC, we do not consider intermediate nodes in the dependency path corresponding to both an added constraint and a removed constraint. These constraints are only temporarily present and do not fire any other rule except for the rule which removes the constraint.

Next we prove that if two dependency paths satisfying the RC are composed it implies that their composition satisfies the RC as well.

Theorem 2 (Composability of the ranking condition). *Two dependency paths which are composable, each satisfying the RC for dependency paths, compose into a dependency path satisfying the RC for dependency paths.*

Proof. Assume two dependency paths that are composable which both satisfy the RC for CHR. Then such a composition can occur in three different settings:

There are two dependency paths satisfying the first condition. Then there are only constraints added of rank strictly lower than the kept constraints of some propagation rule in both dependency paths. One propagation rule contains the lowest ranked constraints in its head. The other has therefore kept constraints ranked strictly higher than all added constraints in their composition.

There are two dependency paths satisfying the second condition. Then there are two separate cases. The maximally ranked constraint is either determined by one or by both dependency paths. In either case, their composition results in a decrease of some maximally ranked constraint.

There are two dependency paths each satisfying a different condition. Then there are two cases. Either the dependency path satisfying the first condition contains only added constraints ranked strictly lower than the max ranked constraint in the path satisfying the second condition and therefore the second condition applies on their composition as well. Either this is not the case and the kept constraints in the propagation rule are ranked strictly higher than all added constraints in the composition. \square

We can therefore define an abstraction from dependency paths to satisfaction of the RC. The abstract domain to which we map is composed of three elements. Either it is unknown whether the RC is satisfied \boxtimes , either it is satisfied \boxplus , either it is not \boxminus . Composition of the property is defined as follows:

$$RC * \boxtimes = \boxtimes \quad RC * \boxminus = \boxtimes \quad \boxplus * \boxplus = \boxplus$$

Therefore if all direct dependency paths satisfy the *RC*, it implies that all dependency paths satisfy the *RC*. In the next subsection, we define an abstraction from dependency paths to abstract dependency paths, using concepts from CHR nets extended by satisfaction of the RC.

4.5 Abstract dependency paths

We abstract computations of CHR programs by concepts of CHR nets. We represent query nodes of a computation by abstract input constraints. Answer nodes in computations are represented by abstract output constraints. Intermediate constraints represent constraints that are made available by some rule and are used afterwards in the application of some other rule. These constraints are abstracted by the intersection of an abstract input and abstract output constraint. In the CHR net such abstractions are represented by match transitions. We define abstract dependencies using the CHR net.

Definition 27 (Abstract dependency relation). *Consider an abstract output node N_j connected to an abstract input node N_i in the CHR net of a CHR program P , then we say that N_j depends on N_i . We call a path between two nodes in the CHR net, such that one depends on the other, an abstract dependency path and represent it by an alternating sequence of abstract constraints and rule transitions:*

$$[in_{(i,j)} \rightarrow_{T_i} M_{(i,k,u,v)} \rightarrow_{T_u} \cdots \rightarrow_{T_v} out_{(v,w)}]$$

Notice that the dependency relation is only defined between an output and an input and is denoted by $in_{(i,j)} \rightarrow out_{(v,w)}$. \square

There is some structure in the set of abstract dependencies. We get that two abstract dependency paths, one going from N_1 to N_2 and one going from N_3 to N_4 , are composable if a match transition between N_2 and N_3 exists in the CHR net. Composition of abstract dependency paths is also a partial associative operation $* : S \times S \mapsto S$.

$$\begin{aligned} [in_{(i,j)} \xrightarrow{T_i} out_{(i,k)}] * [in_{(u,v)} \xrightarrow{T_u} out_{(u,w)}] = \\ [in_{(i,j)} \xrightarrow{T_i} M_{(i,k,u,v)} \xrightarrow{T_u} out_{(u,w)}] \end{aligned}$$

We abstract the dependency paths in a computation by the abstract dependency relation.

Definition 28 (Abstraction function α). *Let $(in_{(i,j)}\sigma_i) \rightarrow (out_{(u,v)}, \sigma_u\theta_u)$ be a dependency in some computation of a CHR program P with a finite query I and consider its corresponding dependency path. Then, we define α as a mapping of the dependency to an abstract dependency in the CHR net of a program P and as a mapping of the corresponding dependency path to whether it satisfies the RC or not. A dependency between nodes in a computation $(in_{(i,j)}\sigma_i) \rightarrow (out_{(u,v)}, \sigma_u\theta_u)$ is therefore mapped to an abstract dependency $in_{(i,j)} \rightarrow out_{(u,v)}$ and to an element $\{\boxtimes \boxminus \boxplus\}$ corresponding to the dependency path's satisfaction of the RC.*

There are only a finite number of abstract dependency relations even though an infinite number of abstract dependency paths exist in a CHR net. This finite set of abstractions for dependency paths in computations implies the existence of a circular element in case there exists an infinite dependency path.

Theorem 3. *Consider an infinite computation for a CHR program P and a finite query I . Let α be as defined in the previous Definition. Then there is an infinite sequence of nodes M_1, M_2, \dots and an abstract dependency path A in the set of abstract dependency paths \mathcal{A} , such that for each i , M_{i+1} depends on M_i , and for each j, k the dependency path from M_j to M_k is mapped by α to A .*

Proof. By Lemma 1, there is an infinite sequence of nodes N_1, N_2, \dots such that for each i , N_{i+1} depends on N_i . To each dependency path from N_i to N_j the mapping α assigns an abstract dependency path in the CHR net of P . By Ramsey's theorem we get that there is a subsequence N_{k_1}, N_{k_2}, \dots such that for each i, j the mapping α assigns to the dependency path from N_{k_i} to N_{k_j} the same element. \square

As our mapping α is a homomorphism, that is a mapping for which holds that $\alpha(B_1 * B_2) = \alpha(B_1) * \alpha(B_2)$, it is sufficient to consider only abstractions for dependency paths between nodes which directly depend on each other in the CHR net. This is the reason for the name. The element A whose existence is proved in the theorem is an element that is idempotent. We call such an element a circular element of the CHR net.

TODO: Why only considering circular elements in the CHR net wrt. satisfaction of the RC. The meaning of SCCs in a modular framework for termination

analysis Adapting the principles of the dependency framework to termination analysis for CHR. That is, how to prove termination by composing a lexicographical ordering. Here, we want to compose an interpretation such that the RC is satisfied wrt. all circular elements in the CHR net.

4.6 Typing abstract constraints in the CHR net

As the CHR net of a program represents the CHR constraints of rules and their dependencies, we can group all abstract constraints that are related by considering the match transitions of the CHR net. After all, these represent non-empty intersections of abstract constraints and thus connect abstract constraints of the same type.

When considering the set of match transitions, some may overlap in that they connect different abstract outputs to the same input or connect the same output to different inputs. These match transitions therefore determine constraints of the same type and can be grouped accordingly. We obtain therefore a partitioning in the set of match transitions. For every partition, we know that the abstract inputs and output are of the same type and these abstract inputs and outputs should therefore be handled as such. Notice that we do not type constraints based on their arguments, as there may exist different types of constraints, using identical types for their arguments. Consider for example:

$$a(N) \Leftrightarrow N > 50 \mid a(N - 1). \quad a(N) \Leftrightarrow N < 50 \mid a(N + 1).$$

The example consists of two different types of constraints. Those with an argument $N > 50$ and those with an argument $N < 50$. Both rules are unrelated. That is, these can only produce matching constraints for their own input. Both rules can therefore be analyzed separately. This would correspond to a renaming of constraints.

4.7 Strongly connected components in the CHR net

A strongly connected component in the CHR net of a CHR program relates to a subgraph for which a dependency path exists between every two nodes in the CHR net. There may be several SCCs in a CHR program. However, some may represent a subgraph of some other SCC. Therefore, we regard only the maximal SCCs of a CHR program. These consist of disjunct sets of rule in the program, which we call the modules of the program. If all such modules, when considered separate, universally terminate for every finite query, then it implies that any program composed of these modules must terminate as well for a finite query.

Consider a program consisting of a finite number of modules, which all are universally terminating. Assume now that there exists a finite query for which the program is non-terminating. Then there must exist an infinite dependency path and given our abstraction, there must exist some element which is repeated infinitely many times. None of the circular elements as defined by the SCCs can give rise to such an infinite dependency path. Therefore, a circular element must

exist which was not represented by the SCCs themselves. As such a circular element would imply a combination of considered SCCs there must exist a more maximal SCC. This cannot be the case by assumption.