Constraint Handling Rules with Multiset Comprehension Patterns*

Edmund S. L. Lam and Iliano Cervesato

Carnegie Mellon University
sllam@qatar.cmu.edu and iliano@cmu.edu

Abstract. CHR is a declarative, concurrent and committed choice rule-based constraint programming language. We extend CHR with multiset comprehension patterns, providing the programmer with the ability to write multiset rewriting rules that can match a variable number of constraints in the store. This enables writing more readable, concise and declarative code for algorithms that coordinate large amounts of data or require aggregate operations. We call this extension CHR^{cp}. We give a high-level abstract semantics of CHR^{cp}, followed by a lower-level operational semantics. We then show the soundness of this operational semantics with respect to the abstract semantics.

1 Introduction

CHR is a declarative, concurrent and committed choice rule-based constraint programming language. CHR rules are executed in a pure forward-chaining (data-driven) and committed choice (no backtracking) manner, providing the programmer with a highly expressive programming model to implement complex programs in a concise and declarative manner. Yet, programming in a pure forward-chaining model is not without its shortfalls. Expressive as it is, when faced with algorithms that operate over a dynamic number of constraints (e.g., finding the minimum value or finding *all* constraints in the store matching a particular pattern), a programmer is forced to decompose his/her code over several rules, as a CHR rule can only match a fixed number of constraints. Such an approach is tedious, error-prone and leads to repeated instances of boilerplate codes, suggesting the opportunity for a higher form of abstraction.

This paper explores an extension of CHR with *multiset comprehension patterns* [1,5]. These patterns allow the programmer to write multiset rewriting rules that can match dynamically-sized constraint sets in the store. They enable writing more readable, concise and declarative programs that coordinate large amount of data or use aggregate operations. We call this extension CHR^{cp}.

While defining an abstract semantics that accounts for comprehension patterns is relatively easy, turning it into an efficient model of computation akin to

* This paper was made possible by grant NPRP 09-667-1-100, *Effective Programming for Large Distributed Ensembles*, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

the refined operational semantics of CHR [2] is challenging. The problem is that monotonicity [3], a key requirement for the kind of incremental processing that underlies CHR's refined operational semantics, does not hold in the presence of comprehension patterns. We address this issue by statically identifying CHR^{cp} constraints that are monotonic, and limiting incremental processing to just these constraints. Similarly to [2], this approach yields a sound transformation of the abstract model of computation into an implementable system.

Altogether, this paper makes the following contributions:

- We formally define the abstract syntax and abstract semantics of CHR^{cp}.
- We define a notion of conditional monotonicity for CHR^{cp} programs, and define an operational semantics that exploits it to drive an efficient execution model for CHR^{cp}.
- We prove the soundness of this operational semantics with respect to the abstract semantics.

The rest of the paper is organized as follows: Section 2 introduces CHR^{cp} by examples and Section 3 formalizes its syntax. Section 4 defines the abstract semantics while Section 5 examines monotonicity in CHR^{cp}. In Section 6, we introduce an operational semantics for CHR^{cp} and in Section 7, we prove its soundness with respect to the abstract semantics. Section 8 situates CHR^{cp} in the literature and Section 9 outlines directions of future work.

2 Motivating Examples

In this section, we illustrate the benefits of comprehension patterns in multiset rewriting on some examples. A comprehension pattern $\wr p(\vec{t}) \mid g \int_{\vec{x} \in t}$ represents a multiset of constraints that match the atomic constraint $p(\vec{t})$ and satisfy guard g under the bindings of variables $\vec{x}$ that range over t, known as the *comprehension domain.*

Consider the problem of swapping data among agents based on a pivot value. We express an integer datum D belonging to agent X by the constraint $data(X, D)$. Then, given agents X and Y and pivot value P, we want all of X's data with value greater than or equal to P to be transferred to Y and all of Y's data less than P to be transferred to X. The following CHR^{cp} rule implements this pivot swap procedure:

$$pivotSwap \ @ \begin{array}{l} swap(X, Y, P) \\ \wr data(X, D) \mid D \geq P \int_{D \in Xs} \\ \wr data(Y, D) \mid D < P \int_{D \in Ys} \end{array} \iff \begin{array}{l} \wr data(Y, D) \int_{D \in Xs} \\ \wr data(X, D) \int_{D \in Ys} \end{array}$$

The swap is triggered by the constraint $swap(X, Y, P)$. All of X's data that are greater than or equal to the pivot P are identified by the comprehension pattern $\wr data(X, D) \mid D \geq P \int_{D \in Xs}$. Similarly, all Y's data less than P are identified by $\wr data(Y, D) \mid D < P \int_{D \in Ys}$. The instances of D matched by each comprehension pattern are accumulated in the comprehension domains Xs and Ys, respectively. Finally, these collected bindings are used in the rule body to complete the rewriting by redistributing all of X's selected data to Y and vice versa. The comprehension domains Xs and Ys are treated as output variables in the rule head, since the matches for D are fetched from the store. In the rule body, comprehension ranges are input variables, as we construct the desired multisets of constraints from them. The CHR^{cp} semantics enforces the property that each comprehension pattern captures a *maximal multiset* of constraints in the store, thus guaranteeing that no data that is to be swapped is left behind.

Comprehension patterns allow the programmer to easily write rule patterns that manipulate dynamic numbers of constraints. Now consider how the above program would be written in pure CHR (without comprehension patterns). To do this, we are forced to explicitly implement the operation of collecting a multiset of *data* constraints over several rules. We also need to introduce an accumulator to store bindings for the matched facts as we retrieve them. A possible implementation of this nature is as follows:

$$\begin{array}{rll}
\mathit{init} \ @ \ \mathit{swap}(X, Y, P) & \Longleftrightarrow & \mathit{grabGE}(X, P, Y, [\,]), \mathit{grabLT}(Y, P, X, [\,]) \\
\mathit{ge1} \ @ \ \mathit{grabGE}(X, P, Y, \mathit{Ds}), \mathit{data}(X, D) & \Longleftrightarrow & D \geq P \mid \mathit{grabGE}(X, P, Y, [D \mid \mathit{Ds}]) \\
\mathit{ge2} \ @ \ \mathit{grabGE}(X, P, Y, \mathit{Ds}) & \Longleftrightarrow & \mathit{unrollData}(Y, \mathit{Ds}) \\
\mathit{lt1} \ @ \ \mathit{grabLT}(Y, P, X, \mathit{Ds}), \mathit{data}(Y, D) & \Longleftrightarrow & D < P \mid \mathit{grabLT}(Y, P, X, [D \mid \mathit{Ds}]) \\
\mathit{lt2} \ @ \ \mathit{grabLT}(Y, P, X, \mathit{Ds}) & \Longleftrightarrow & \mathit{unrollData}(X, \mathit{Ds}) \\
\mathit{unroll1} \ @ \ \mathit{unrollData}(L, [D \mid \mathit{Ds}]) & \Longleftrightarrow & \mathit{unrollData}(L, \mathit{Ds}), \mathit{data}(L, D) \\
\mathit{unroll2} \ @ \ \mathit{unrollData}(L, [\,]) & \Longleftrightarrow & \mathit{true}
\end{array}$$

Here, $[\,]$ denotes the empty list and $[D \mid Ds]$ constructs a list with the head element D and the rest from Ds. In a CHR program that consists of several subroutines of this nature, this boilerplate code gets repeated over and over, making the program less concise. Furthermore, the use of list accumulators and auxiliary constraints (e.g., *grabGE*, *unrollData*) makes the implementation less readable and more error-prone. Most importantly, the swap operation as written in CHR^{cp} is *atomic* while the above CHR code involves many rewrites, which could be interspersed by applications of other rules that operate on *data* constraints.

Comprehension patterns also promote a concise way of coding term-level aggregate computations: using a comprehension pattern's ability to retrieve a dynamic number of constraints, we can compute aggregates with term-level map and reduce operations over multisets of terms. Consider the following CHR^{cp} rule:

$$\begin{array}{l}
\mathit{removeNonMin} \ @ \\
\quad \mathit{remove}(\mathit{Gs}), \wr \mathit{edge}(X, Y, W) \mid X \in \mathit{Gs} \int_{(X,Y,W) \in \mathit{Es}} \\
\quad \Longleftrightarrow \left. \begin{array}{l} \mathit{Es} \neq \emptyset \\ \mathit{Ws} = \wr W \int_{(X,Y,W) \in \mathit{Es}} \\ W_m = \mathcal{R} \ \min \ \infty \ \mathit{Ws} \\ \mathit{Rs} = \wr (X, Y, W) \mid W_m < W \int_{(X,Y,W) \in \mathit{Es}} \end{array} \right| \wr \mathit{edge}(X, Y, W) \int_{(X,Y,W) \in \mathit{Rs}} \\
\text{where} \quad \min = \lambda x.\ \lambda y.\ \text{if } x \leq y \text{ then } x \text{ else } y
\end{array}$$

This CHR^{cp} rule identifies the minimum weight W_m from a group Gs of edges in a directed graph and deletes all edges in that group with weight W_m. Note that there could be several such minimal edges. We represent an edge of weight W between nodes X and Y with the constraint $edge(X, Y, W)$. The fact $remove(Gs)$ identifies the group Gs whose outgoing edges are the subject of the removal. The minimum weight W_m is computed by collecting all edges with origin in a node in Gs (constraint $\wr edge(X, Y, W) \mid X \in Gs \int_{(X,Y,W) \in Es}$), extracting their weight

Variables: x Values: v Predicates: p Rule names: r
Primitive terms: t_α Primitive guards: g_α

$$
\begin{array}{rrcl}
\textit{Terms}: & t & ::= & t_\alpha \mid \bar{t} \mid \wr t \mid g \int_{\vec{x} \in t} \\
\textit{Guards}: & g & ::= & g_\alpha \mid g \wedge g \mid \bigwedge_{\vec{x} \in t} \wr g \int \\
\textit{Atomic Constraints}: & A & ::= & p(\,\vec{t}\,) \\
\textit{Comprehensions}: & M & ::= & \wr A \mid g \int_{\vec{x} \in t} \\
\textit{Rule Constraints}: & C, B & ::= & A \mid M \\
\textit{Rules}: & R & ::= & r \mathbin{@} \bar{C} \setminus \bar{C} \Longleftrightarrow g \mid \bar{C} \\
\textit{Programs}: & \mathcal{P} & ::= & \bar{R}
\end{array}
$$

Fig. 1. Abstract Syntax of *CHR*^{cp}

into the multiset Ws (with $Ws = \wr W \int_{(X,Y,W) \in Es}$) and folding the binary function min over all of Ws by means of the term-level *reduce* operator $\mathcal{R}$ (constraint $W_m = \mathcal{R}$ min ∞ Ws). The multiset Rs collects the edges with weight strictly greater than W_m (constraint $Rs = \wr(X, Y, W) \mid W_m < W \int_{(X,Y,W) \in Es}$).

3 Syntax

In this section, we define the abstract syntax of *CHR*^{cp}. We focus on the core fragment of the *CHR*^{cp} language, on top of which convenient short-hands and a "sugared" concrete syntax can be built.

Figure 1 defines the abstract syntax of *CHR*^{cp}. Throughout this paper, we write $\bar{o}$ for a multiset of syntactic object o, with $\varnothing$ indicating the empty multiset. We write $\wr \bar{o}_1, \bar{o}_2 \int$ for the union of multisets $\bar{o}_1$ and $\bar{o}_2$, omitting the brackets when no ambiguity arises. The extension of multiset $\bar{o}$ with syntactic object o is similarly denoted $\wr \bar{o}, o \int$. We write $\vec{o}$ for a comma-separated tuple of o's.

An atomic constraint $p(\vec{t})$ is a predicate symbol p applied to a tuple $\vec{t}$ of terms. A comprehension pattern $\wr A \mid g \int_{\vec{x} \in t}$ represents a multiset of constraints that match the atomic constraint A and satisfy guard g under the bindings of variables $\vec{x}$ that range over t. We call $\vec{x}$ the *binding variables* and t the *comprehension domain*. The conjunctive comprehension of a multiset of guards of the form g is denoted by $\bigwedge_{\vec{x} \in t} \wr g \int$. It represents a conjunction of all instances of guard g under the bindings of $\vec{x}$ ranging over t. In both forms of comprehension, the variables $\vec{x}$ are locally bound with scope g (and A).

The development of *CHR*^{cp} is largely agnostic with respect to the language of terms. We will assume a base term language $\mathcal{L}_\alpha$, that in examples contains numbers and functions, but may be far richer. We write t_α for a generic term in this base language, g_α for an atomic guard over such terms, and $\models_\alpha$ for the satisfiability relation over ground guards. In addition to $\mathcal{L}_\alpha$, *CHR*^{cp} contains tuples with their standard operators, and a term-level multisets. Multiset constructors include the empty multiset $\varnothing$, singleton multisets $\wr t \int$ for any term t, and multiset union $\wr m_1, m_2 \int$ for multisets m_1 and m_2. Term-level multiset comprehension $\wr t \mid g \int_{x \in m}$ filters multiset m according to g and maps the result as specified by t. The reduce operator $\mathcal{R}\ f\ e\ m$ recursively combines the elements of multiset m pairwise according to f, returning e for the empty multiset.

As in CHR, a CHR^{cp} rule $r\ @\ \bar{C}_p \setminus \bar{C}_s \Longleftrightarrow g \mid \bar{B}$ specifies the rewriting of $\bar{C}_s$ into $\bar{B}$ under the conditions that constraints $\bar{C}_p$ are available and guards g are satisfied. As usual, we refer to $\bar{C}_p$ as the *propagated head*, to $\bar{C}_s$ as the *simplified head* and to $\bar{B}$ as the *body* of the rule. If the propagated head $\bar{C}_p$ is empty or the guard g is always satisfied (i.e., *true*), we omit the respective rule component entirely. Rules with an empty simplified head $\bar{C}_s$ are referred to as *propagation* rules. All free variables in a CHR^{cp} rule are implicitly universally quantified at the head of the rule. We will assume that a rule's body is grounded by the rule heads and that guards (built-in constraint) cannot appear in the rule body. This simplifies the discussion, allowing us to focus on the novelties brought about by comprehension patterns.

4 Abstract Semantics

This section describes the abstract semantics of CHR^{cp}. We first define some meta-notation and operations. The set of the free variables in a syntactic object o is denoted $FV(o)$. We write $[\vec{t}/\vec{x}]o$ for the simultaneous replacement within object o of all occurrences of variable x_i in $\vec{x}$ with the corresponding term t_i in $\vec{t}$. When traversing a binding construct (e.g., comprehension patterns), substitution implicitly α-renames variables to avoid capture. It will be convenient to assume that terms get normalized during (or right after) substitution.

Without loss of generality, we assume that atomic constraints in a CHR^{cp} rule have the form $p(\vec{x})$, including in comprehension patterns. This simplified form pushes complex term expressions and computations into the guard component of the rule or the comprehension pattern. The satisfiability of a ground guard g is modeled by the judgment $\models g$; its negation is written $\not\models g$.

The abstract semantics of CHR^{cp} is modeled by the small-step judgment $\mathcal{P} \rhd St \mapsto_{\alpha} St'$, which applies a rule in CHR^{cp} program $\mathcal{P}$ to constraint store St producing store St'. A constraint store is a multiset of ground atomic constraints. Applying a rule has two phases: we match its heads and guard against the current store, and whenever successful, we replace some of the matched facts with the corresponding instance of this body. We will now describe these two phases in isolation and then come back to rule application.

Figure 2 defines the matching phase of CHR^{cp}. It relies on two forms of judgments, each with a variant operating on a multiset of constraint patterns $\bar{C}$ and a variant on an individual pattern C. The first matching judgment, $\bar{C} \triangleq_{\mathbf{lhs}} St$, holds when the constraints in the store fragment St match *completely* the multiset of constraint patterns $\bar{C}$. It will always be the case that $\bar{C}$ is closed (i.e., $FV(\bar{C}) = \emptyset$). Rules ($\mathbf{l}_{mset\text{-}*}$) iterate rules ($\mathbf{l}_{atom}$) and ($\mathbf{l}_{comp\text{-}*}$) on St, thereby partitioning it into fragments matched by these rules. Rule ($\mathbf{l}_{atom}$) matches an atomic constraint A to the singleton store A. Rules ($\mathbf{l}_{comp\text{-}*}$) match a comprehension pattern $\wr A \mid g \wr_{\vec{x} \in ts}$. If the comprehension domain is empty ($x \in \emptyset$), the store must be empty (rule $\mathbf{l}_{comp\text{-}2}$). Otherwise, rule ($\mathbf{l}_{comp\text{-}1}$) binds $\vec{x}$ to an element $\vec{t}$ of the comprehension domain ts, matches the instance $[\vec{t}/\vec{x}]A$ of the pattern A with a constraint A' in the store if the corresponding guard instance $[\vec{t}/\vec{x}]g$ is satisfiable, and continues with the rest of the comprehension domain.

$$\boxed{\text{Matching:} \quad \bar{C} \triangleq_{\mathbf{lhs}} St \qquad C \triangleq_{\mathbf{lhs}} St}$$

$$\frac{\bar{C} \triangleq_{\mathbf{lhs}} St \quad C \triangleq_{\mathbf{lhs}} St'}{\wr \bar{C}, C \int \triangleq_{\mathbf{lhs}} \wr St, St' \int} (\mathbf{l}_{mset\text{-}1}) \qquad \frac{}{\emptyset \triangleq_{\mathbf{lhs}} \emptyset} (\mathbf{l}_{mset\text{-}2}) \qquad \frac{}{A \triangleq_{\mathbf{lhs}} A} (\mathbf{l}_{atom})$$

$$\frac{[\vec{t}/\vec{x}]A \triangleq_{\mathbf{lhs}} A' \quad \models [\vec{t}/\vec{x}]g \quad \wr A \mid g \int_{\vec{x} \in ts} \triangleq_{\mathbf{lhs}} St}{\wr A \mid g \int_{\vec{x} \in \wr ts, \vec{t} \int} \triangleq_{\mathbf{lhs}} \wr St, A' \int} (\mathbf{l}_{comp\text{-}1}) \qquad \frac{}{\wr A \mid g \int_{\vec{x} \in \emptyset} \triangleq_{\mathbf{lhs}} \emptyset} (\mathbf{l}_{comp\text{-}2})$$

$$\boxed{\text{Residual Non-matching:} \quad \bar{C} \triangleq^{\neg}_{\mathbf{lhs}} St \qquad C \triangleq^{\neg}_{\mathbf{lhs}} St}$$

$$\frac{\bar{C} \triangleq^{\neg}_{\mathbf{lhs}} St \quad C \triangleq^{\neg}_{\mathbf{lhs}} St}{\wr \bar{C}, C \int \triangleq^{\neg}_{\mathbf{lhs}} St} (\mathbf{l}^{\neg}_{mset\text{-}1}) \qquad \frac{}{\emptyset \triangleq^{\neg}_{\mathbf{lhs}} St} (\mathbf{l}^{\neg}_{mset\text{-}2})$$

$$\frac{}{A \triangleq^{\neg}_{\mathbf{lhs}} St} (\mathbf{l}^{\neg}_{atom}) \qquad \frac{A \not\sqsubseteq_{\mathbf{lhs}} M \quad M \triangleq^{\neg}_{\mathbf{lhs}} St}{M \triangleq^{\neg}_{\mathbf{lhs}} \wr St, A \int} (\mathbf{l}^{\neg}_{comp\text{-}1}) \qquad \frac{}{M \triangleq^{\neg}_{\mathbf{lhs}} \emptyset} (\mathbf{l}^{\neg}_{comp\text{-}2})$$

Subsumption: $A \sqsubseteq_{\mathbf{lhs}} \wr A' \mid g \int_{\vec{x} \in ts}$ iff $A = \theta A'$ and $\models \theta g$ for some $\theta = [\vec{t}/\vec{x}]$

Fig. 2. Semantics of Matching in CHR^{cp}

To guarantee the maximality of comprehension patterns, we test a store for *residual matchings*. This relies on the matching subsumption relation $A \sqsubseteq_{\mathbf{lhs}} \wr A' \mid g \int_{\vec{x} \in ts}$, defined at the bottom of Figure 2. This relation holds if A can be absorbed into the comprehension pattern $\wr A' \mid g \int_{\vec{x} \in ts}$. Note that it ignores the available bindings in ts: t need not be an element of the comprehension domain. Its negation is denoted by $A \not\sqsubseteq_{\mathbf{lhs}} \wr A' \mid g \int_{\vec{x} \in ts}$. We test a store for residual matchings using the *residual non-matching judgment* $\bar{C} \triangleq^{\neg}_{\mathbf{lhs}} St$. Informally, for each comprehension pattern $\wr A' \mid g \int_{\vec{x} \in ts}$ in $\bar{C}$, this judgment checks that no constraints in St matches A' satisfying g. This judgment is defined in the middle section of Figure 2. Rules $(\mathbf{l}^{\neg}_{mset\text{-}*})$ apply the remaining rules to each constraint patterns C in $\bar{C}$. Observe that each pattern C is ultimately matched against the entire store St. Rule $(\mathbf{l}^{\neg}_{atom})$ asserts that atoms have no residual matches. Rules $(\mathbf{l}^{\neg}_{comp\text{-}*})$ check that no constraints in St match the comprehension pattern $M = \wr A' \mid g \int_{\vec{x} \in ts}$.

If an instance of a CHR^{cp} rule passes the matching phase, we need to *unfold* the comprehension patterns in its body into a multiset of atomic constraints. The judgment $\bar{C} \ggg_{\mathbf{rhs}} St$, defined in Figure 3, does this unfolding. This judgment is similar to the matching judgment (Figure 2) except that it skips any element in the comprehension domain that fails the guard (rule $\mathbf{r}_{comp\text{-}2}$).

We now have all the pieces to define the application of a CHR^{cp} rule. The judgment $\mathcal{P} \rhd St \mapsto_{\alpha} St'$ describes a state transition from stores St to St' triggered by applying a rule instance in CHR^{cp} program $\mathcal{P}$. This judgment is defined by the rule at the bottom of Figure 3. A CHR^{cp} rule $r \ @ \ \bar{C}_p \setminus \bar{C}_s \Longleftrightarrow g \mid \bar{B} \in \mathcal{P}$ is applicable in St if there is a substitution θ that makes the guard satisfiable (i.e., $\models \theta g$) and if there are fragments St_p and St_s of the store that match the

Rule Body: $\bar{C} \ggg_{\mathbf{rhs}} St \quad C \ggg_{\mathbf{rhs}} St$

$$\frac{\bar{C} \ggg_{\mathbf{rhs}} St \quad C \ggg_{\mathbf{rhs}} St'}{\lbag \bar{C}, C \rbag \ggg_{\mathbf{rhs}} \lbag St, St' \rbag}\ (\mathbf{r}_{mset\text{-}1}) \qquad \frac{}{\emptyset \ggg_{\mathbf{rhs}} \emptyset}\ (\mathbf{r}_{mset\text{-}2}) \qquad \frac{}{A \ggg_{\mathbf{rhs}} A}\ (\mathbf{r}_{atom})$$

$$\frac{\models [\vec{t}/\vec{x}]g \quad [t/\vec{x}]A \ggg_{\mathbf{rhs}} A' \quad \lbag A \mid g \rbag_{\vec{x} \in ts} \ggg_{\mathbf{rhs}} A'}{\lbag A \mid g \rbag_{\vec{x} \in \lbag ts, \vec{t} \rbag} \ggg_{\mathbf{rhs}} \lbag St, A' \rbag}\ (\mathbf{r}_{comp\text{-}1})$$

$$\frac{\not\models [\vec{t}/\vec{x}]g \quad \lbag A \mid g \rbag_{\vec{x} \in ts} \ggg_{\mathbf{rhs}} St}{\lbag A \mid g \rbag_{\vec{x} \in \lbag ts, \vec{t} \rbag} \ggg_{\mathbf{rhs}} St}\ (\mathbf{r}_{comp\text{-}2}) \qquad \frac{}{\lbag A \mid g \rbag_{\vec{x} \in \emptyset} \ggg_{\mathbf{rhs}} \emptyset}\ (\mathbf{r}_{comp\text{-}3})$$

Rule Application: $\mathcal{P} \rhd St \mapsto_{\alpha} St$

$$\frac{(r \ @ \ \bar{C}_p \setminus \bar{C}_s \Longleftrightarrow g \mid \bar{B}) \in \mathcal{P} \quad \models \theta g \qquad \theta \bar{C}_p \triangleq_{\mathbf{lhs}} St_p \quad \theta \bar{C}_s \triangleq_{\mathbf{lhs}} St_s \quad \theta \lbag \bar{C}_p, \bar{C}_s \rbag \triangleq^{\neg}_{\mathbf{lhs}} St \quad \theta \bar{B} \ggg_{\mathbf{rhs}} St_b}{\mathcal{P} \rhd \lbag St_p, St_s, St \rbag \mapsto_{\alpha} \lbag St_p, St_b, St \rbag}$$

Fig. 3. Abstract Semantics of *CHR*^{cp}

corresponding instance of the propagated and simplified heads ($\theta \bar{C}_p \triangleq_{\mathbf{lhs}} St_p$ and $\theta \bar{C}_s \triangleq_{\mathbf{lhs}} St_s$) and that are maximal in St (i.e., $\theta \lbag \bar{C}_p, \bar{C}_s \rbag \triangleq^{\neg}_{\mathbf{lhs}} St$). We then apply this rule by replacing the store fragment St_s that matches the simplified head instance with the unfolded rule body instance ($\theta \bar{B} \ggg_{\mathbf{rhs}} St_b$). We write $\mathcal{P} \rhd St \ \mapsto^{*}_{\alpha} \ St'$ for zero to more applications of this rule.

5 Monotonicity

In this section, we analyze the impact that comprehension patterns have on monotonicity in *CHR*^{cp}. Specifically, we show that *CHR*^{cp} enjoys a *conditional* form of monotonicity, that we will exploit in Section 6 to define an operational semantics for *CHR*^{cp} based on (partial) incremental processing of constraints.

In CHR, monotonicity [3] is an important property. Informally, monotonicity ensures that if a transition between two CHR states (stores) is possible, it is also possible in any larger store. This property underlies many efficient implementation techniques for CHR. For instance, the incremental processing of constraints in CHR's refined operational semantics [2] is sound because of the monotonicity property. When parallelizing CHR execution [6], the soundness of composing concurrent rule application also depends on monotonicity. In *CHR*^{cp} however, monotonicity is not guaranteed in its standard form:

$$\text{if } \mathcal{P} \rhd St \mapsto_{\alpha} St', \text{ then } \mathcal{P} \rhd \lbag St, St'' \rbag \mapsto_{\alpha} \lbag St', St'' \rbag \text{ for any } St''$$

This is not surprising, since the maximality requirement of comprehension patterns could be violated if we add a constraint $A \in St''$. Consider the following example, where we extend the store with a constraint $a(3)$ which can be matched by a comprehension pattern in program $\mathcal{P}$:

$$\mathcal{P} \equiv r @ \lbag a(X) \rbag_{X \in Xs} \Longleftrightarrow \lbag b(X) \rbag_{X \in Xs}$$

$$\mathcal{P} \rhd \lbag a(1), a(2) \rbag \mapsto_{\alpha} \lbag b(1), b(2) \rbag \text{ but } \mathcal{P} \rhd \lbag a(1), a(2), a(3) \rbag \not\mapsto_{\alpha} \lbag b(1), b(2), a(3) \rbag$$

$$\frac{g \rhd \{\bar{C}_p, \bar{C}_s\} \triangleq^{\neg}_{\mathbf{unf}} \bar{B} \quad \mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}}{\mathcal{P}, (r \text{ @ } \bar{C}_p \setminus \bar{C}_s \Longleftrightarrow g \mid \bar{C}_b) \triangleq^{\neg}_{\mathbf{unf}} \bar{B}} (\mathbf{u}^{\neg}_{prog\text{-}1}) \qquad \frac{}{\emptyset \triangleq^{\neg}_{\mathbf{unf}} \bar{B}} (\mathbf{u}^{\neg}_{prog\text{-}2})$$

$$\frac{g \rhd \bar{C} \triangleq^{\neg}_{\mathbf{unf}} \bar{B} \quad g \rhd C \triangleq^{\neg}_{\mathbf{unf}} \bar{B}}{g \rhd \{\bar{C}, C\} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}} (\mathbf{u}^{\neg}_{mset\text{-}1}) \qquad \frac{}{g \rhd \emptyset \triangleq^{\neg}_{\mathbf{unf}} \bar{B}} (\mathbf{u}^{\neg}_{mset\text{-}2}) \qquad \frac{}{g \rhd A \triangleq^{\neg}_{\mathbf{unf}} \bar{B}} (\mathbf{u}^{\neg}_{atom})$$

$$\frac{g \rhd B \not\sqsubseteq_{\mathbf{unf}} M \quad g \rhd M \triangleq^{\neg}_{\mathbf{unf}} \bar{B}}{g \rhd M \triangleq^{\neg}_{\mathbf{unf}} \{\bar{B}, B\}} (\mathbf{u}^{\neg}_{comp\text{-}1}) \qquad \frac{}{g \rhd M \triangleq^{\neg}_{\mathbf{unf}} \emptyset} (\mathbf{u}^{\neg}_{comp\text{-}2})$$

$$g \rhd A \sqsubseteq_{\mathbf{unf}} \{A' \mid g'\}_{\vec{x} \in ts} \text{ iff } \theta A \equiv \theta A', \models \theta g', \models \theta g \text{ for some } \theta$$

$$g'' \rhd \{A \mid g\}_{\vec{x} \in ts} \sqsubseteq_{\mathbf{unf}} \{A' \mid g'\}_{\vec{x}' \in ts'} \text{ iff } \theta A \equiv \theta A', \models \theta g'', \models \theta g', \models \theta g \text{ for some } \theta$$

Fig. 4. Residual Non-Unifiability

In this example, extending the store with $a(3)$ violates the maximality of comprehension pattern $\{a(X)\}_{X \in Xs}$. Hence, the derivation under the larger store is not valid with respect to the abstract semantics. Yet all is not lost: if we can guarantee that St'' only contains constraints that *never* match any comprehension pattern in the head of any rule in $\mathcal{P}$, we recover monotonicity, albeit in a restricted form. For instance, extending the store in the above example with constraint $c(3)$ does not violate monotonicity.

We formalize this idea by generalizing the residual non-matching judgment from Figure 2. The resulting *residual non-unifiability judgment* is defined in Figure 4. Given a program $\mathcal{P}$ and a multiset of constraint patterns $\bar{B}$, the judgment $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}$ holds if no constraint that matches any pattern in $\bar{B}$ can be unified with any comprehension pattern in any rule heads of $\mathcal{P}$. Rules $(\mathbf{u}^{\neg}_{prog\text{-}*})$ iterate over each *CHR*^{cp} rule in $\mathcal{P}$. For each rule, the judgment $g \rhd \bar{C} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}$ tests each rule pattern in $\bar{C}$ against all the patterns $\bar{B}$ (rules $\mathbf{u}^{\neg}_{mset\text{-}*}$). Rule $(\mathbf{u}^{\neg}_{atom})$ handles atomic facts, which are valid by default. Rules $(\mathbf{u}^{\neg}_{comp\text{-}*})$ check that no body pattern $\bar{B}$ is unifiable with any rule head pattern $\bar{C}$ under the guard g. It does so on the basis of the relations at the bottom of Figure 4.

A constraint (atom or comprehension pattern) C is *monotone* w.r.t. program $\mathcal{P}$ if $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} C$ is derivable. With the residual non-unifiability judgment, we can ensure the conditional monotonicity of *CHR*^{cp}.

Theorem 1 (Conditional Monotonicity). *Given a program $\mathcal{P}$ and stores St and St', if $\mathcal{P} \rhd St \mapsto^*_\alpha St'$, then for any store fragment St'' such that $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} St''$, we have that $\mathcal{P} \rhd \{St, St''\} \mapsto^*_\alpha \{St', St''\}$.*

Proof. The proof proceeds by induction on the derivation $\mathcal{P} \rhd St \mapsto^*_\alpha St'$. The monotonicity property holds trivially in the base case where we make zero steps. In the inductive case, we rely on the fact that if $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} St''$, then, for any instance of a comprehension pattern M occurring in a rule head $\mathcal{P}$, we are guaranteed to have $M \triangleq^{\neg}_{\mathbf{lhs}} St''$. ◻

This theorem allows us to enlarge the context of any derivation $\mathcal{P} \rhd St \mapsto^*_\alpha St'$ with St'', if we have the guarantee that all constraints in St'' are monotone with respect to $\mathcal{P}$.

$$
\begin{array}{c}
\text{Occurrence Index } i \\
\text{Store Label } n
\end{array}
\qquad
\begin{array}{llll}
\text{Rule Head } H & & ::= & C : i \\
\text{CHR Rule } R_\omega & & ::= & r \ @ \ \bar{H} \setminus \bar{H} \Longleftrightarrow g \mid \bar{B} \\
\text{Program} & \mathcal{P}_\omega & ::= & \bar{R_\omega}
\end{array}
$$

$$
\begin{array}{llll}
\text{Matching History} & \Theta & ::= & (\bar{\theta}, \bar{n}) \\
\text{Goal Constraint} & G & ::= & \texttt{init } \bar{B} \mid \texttt{lazy } A \mid \texttt{eager } A\#n \mid \texttt{act } A\#n \ i \mid \texttt{prop } A\#n \ i \ \Theta \\
\text{Goal Stack} & Gs & ::= & \epsilon \mid [G \mid Gs] \\
\text{Labeled Store} & Ls & ::= & \emptyset \mid \lbag Ls, A\#n \rbag \\
\text{Execution State} & \sigma & ::= & \langle Gs \ ; \ Ls \rangle
\end{array}
$$

$$
\mathit{dropIdx}(C : i) ::= C \qquad \mathit{getIdx}(C : i) ::= \{i\} \qquad \mathit{dropLabels}(A\#n) ::= A \qquad \mathit{getLabels}(A\#n) ::= \{n\}
$$

$$
\mathit{newLabels}(Ls, A) ::= A\#n \quad \text{such that } n \notin \mathit{getLabels}(Ls)
$$

$$
\mathcal{P}_\omega[i] \quad ::= \begin{cases} R_\omega & \text{if } R_\omega \in \mathcal{P}_\omega \text{ and } i \in \mathit{getIdx}(R_\omega) \\ \bot & \text{otherwise} \end{cases}
$$

$$
\frac{\mathit{dropIdx}(\bar{H}) \triangleq_{\mathbf{lhs}} \mathit{dropLabels}(Ls)}{\bar{H} \triangleq_{\mathbf{lhs}} Ls}
\qquad
\frac{\mathit{dropIdx}(\bar{H}) \triangleq^{\neg}_{\mathbf{lhs}} \mathit{dropLabels}(Ls)}{\bar{H} \triangleq^{\neg}_{\mathbf{lhs}} Ls}
\qquad
\frac{\mathit{dropIdx}(\mathcal{P}) \triangleq^{\neg}_{\mathbf{unf}} \bar{C}}{\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} \bar{C}}
$$

Fig. 5. Annotated Programs, Execution States and Auxiliary Meta-operations

6 Operational Semantics

In this section, we define a lower-level operational semantics for CHR^{cp}. Similarly to [2], this operational semantics determines a goal-based execution of CHR^{cp} programs that utilizes monotonicity (conditional, in our case) to incrementally process constraints. By "incrementally", we mean that goal constraints are added to the store one by one, as we process each for potential match to the rule heads. The main difference with [2] is that a goal constraint C that is not monotone w.r.t. the program $\mathcal{P}$ (i.e., such that $\mathcal{P} \not\triangleq^{\neg}_{\mathbf{unf}} C$) is stored immediately before any other rule application is attempted. Similarly to other operational semantics for CHR, our semantics also handles *saturation*, enforcing the invariant that a propagation rule instance is only applied once for each matching rule head instance in the store. Hence, programs with propagation rules are not necessarily non-terminating. This makes our operational semantics incomplete w.r.t. the abstract semantics, but saturation is generally viewed as desirable.

Figure 5 defines the execution states of CHR^{cp} programs in this operational semantics and some auxiliary notions. We annotate a CHR^{cp} program $\mathcal{P}$ with *rule head occurrence indices*. The result is denoted $\mathcal{P}_\omega$. Specifically, each rule head pattern C of $\mathcal{P}$ is annotated with a unique integer i starting from 1, and is written $C : i$ in $\mathcal{P}_\omega$. This represents the order in which rule heads are matched against a constraint. Execution states are pairs $\sigma = \langle Gs \ ; \ Ls \rangle$ where Gs is the *goal stack* and Ls is the *labeled store*. The latter is a constraint store with each constraint annotated with a unique label n. This label allows us to distinguish between copies of the same constraint in the store and to uniquely associate a goal constraint with a specific stored constraint. Labels also support saturation for propagation rules (see below). Each goal in a goal stack Gs represents a unit of execution and Gs itself is a sequence of goals to be executed. A non-empty goal stack has the form $[G \mid Gs]$, where G is the goal at the top of the stack and Gs the rest of the stack. The empty stack is denoted ϵ. We abbreviate the

$(init)$	$\mathcal{P}_\omega \rhd \langle [\texttt{init}\ \lbag \bar{B}_l, \bar{B}_e \rbag \mid Gs]\ ;\ Ls \rangle \mapsto_\omega \langle lazyGs(St_l) + eagerGs(Ls_e) + Gs\ ;\ \lbag Ls, Ls_e \rbag \rangle$ such that $\mathcal{P}_\omega \triangleq^{\neg}_{\mathbf{unf}} \bar{B}_l \quad \bar{B}_e \ggg_{\mathbf{rhs}} St_e \quad \bar{B}_l \ggg_{\mathbf{rhs}} St_l \quad Ls_e = newLabels(Ls, St_e)$ where $eagerGs(\lbag Ls, A\#n \rbag) ::= [\texttt{eager}\ A\#n \mid eagerGs(Ls)] \qquad eagerGs(\emptyset) ::= \epsilon$ $lazyGs(\lbag St_m, A \rbag) ::= [\texttt{lazy}\ A \mid lazyGs(St_m)] \qquad lazyGs(\emptyset) ::= \epsilon$
$(lazy\text{-}act)$	$\mathcal{P}_\omega \rhd \langle [\texttt{lazy}\ A \mid Gs]\ ;\ Ls \rangle \mapsto_\omega \langle [\texttt{act}\ A\#n\ 1 \mid Gs]\ ;\ \lbag Ls, A\#n \rbag \rangle$ such that $\lbag A\#n \rbag = newLabels(Ls, \lbag A \rbag)$
$(eager\text{-}act)$	$\mathcal{P}_\omega \rhd \langle [\texttt{eager}\ A\#n \mid Gs]\ ;\ \lbag Ls, A\#n \rbag \rangle \mapsto_\omega \langle [\texttt{act}\ A\#n\ 1 \mid Gs]\ ;\ \lbag Ls, A\#n \rbag \rangle$
$(eager\text{-}drop)$	$\mathcal{P}_\omega \rhd \langle [\texttt{eager}\ A\#n \mid Gs]\ ;\ Ls \rangle \mapsto_\omega \langle Gs\ ;\ Ls \rangle$ if $A\#n \notin Ls$
$(act\text{-}simpa\text{-}1)$	$\mathcal{P}_\omega \rhd \langle [\texttt{act}\ A\#n\ i \mid Gs]\ ;\ \lbag Ls, Ls_p, Ls_s, Ls_a, A\#n \rbag \rangle \mapsto_\omega \langle [\texttt{init}\ \theta\bar{B} \mid Gs]\ ;\ \lbag Ls, Ls_p \rbag \rangle$ if $\mathcal{P}_\omega[i] = (r\ @\ \bar{H}_p \setminus \lbag \bar{H}_s, C : i \rbag \iff g \mid \bar{B})$, there exists some θ such that $- \models \theta g \quad \theta C \triangleq_{\mathbf{lhs}} \lbag Ls_a, A\#n \rbag$ (Guard Satisfied and Active Match) $- \theta\bar{H}_p \triangleq_{\mathbf{lhs}} Ls_p \quad \theta\bar{H}_s \triangleq_{\mathbf{lhs}} Ls_s$ (Partners Match) $- \theta\bar{H}_p \triangleq^{\neg}_{\mathbf{lhs}} Ls \quad \theta\bar{H}_s \triangleq^{\neg}_{\mathbf{lhs}} Ls \quad \theta C \triangleq^{\neg}_{\mathbf{lhs}} Ls$ (Maximal Comprehension)
$(act\text{-}simpa\text{-}2)$	$\mathcal{P}_\omega \rhd \langle [\texttt{act}\ A\#n\ i \mid Gs]\ ;\ \lbag Ls, Ls_p, Ls_s, Ls_a, A\#n \rbag \rangle$ $\mapsto_\omega \langle [\texttt{init}\ \theta\bar{B}] + [\texttt{act}\ A\#n\ i \mid Gs]\ ;\ \lbag Ls, Ls_p, Ls_a, A\#n \rbag \rangle$ if $\mathcal{P}_\omega[i] = (r\ @\ \lbag \bar{H}_p, C : i \rbag \setminus \bar{H}_s \iff g \mid \bar{B})$ and $\bar{H}_s \neq \emptyset$, there exists some θ such that $- \models \theta g \quad \theta C \triangleq_{\mathbf{lhs}} \lbag Ls_a, A\#n \rbag$ (Guard Satisfied and Active Match) $- \theta\bar{H}_p \triangleq_{\mathbf{lhs}} Ls_p \quad \theta\bar{H}_s \triangleq_{\mathbf{lhs}} Ls_s$ (Partners Match) $- \theta\bar{H}_p \triangleq^{\neg}_{\mathbf{lhs}} Ls \quad \theta\bar{H}_s \triangleq^{\neg}_{\mathbf{lhs}} Ls \quad \theta C \triangleq^{\neg}_{\mathbf{lhs}} Ls$ (Maximal Comprehension)
$(act\text{-}next)$	$\mathcal{P}_\omega \rhd \langle [\texttt{act}\ A\#n\ i \mid Gs]\ ;\ Ls \rangle \mapsto_\omega \langle [\texttt{act}\ A\#n\ (i+1) \mid Gs]\ ;\ Ls \rangle$ if neither $(act\text{-}simpa\text{-}1)$ nor $(act\text{-}simpa\text{-}2)$ applies.
$(act\text{-}drop)$	$\mathcal{P}_\omega \rhd \langle [\texttt{act}\ A\#n\ i \mid Gs]\ ;\ Ls \rangle \mapsto_\omega \langle Gs\ ;\ Ls \rangle$ if $\mathcal{P}_\omega[i] = \bot$

Fig. 6. Operational Semantics of CHR^{cp} (Core-Set)

singleton stack containing G as $[G]$. Given two stacks Gs_1 and Gs_2 we denote their concatenation as $Gs_1 + Gs_2$. We write $G \in Gs$ to denote that G occurs in some position of Gs. Unlike [2], we attach a label to each goal. These labels are `init`, `lazy`, `eager`, `act` and `prop`. We will explain the purpose of each as we describe the semantics.

Figure 5 defines several auxiliary operations that either retrieve or drop occurrence indices and store labels: $dropIdx(H)$ and $getIdx(H)$ deal with indices, $dropLabels(\_)$ and $getLabels(\_)$ with labels. We inductively extend $dropIdx(\_)$ to multisets of rule heads and annotated programs, each returning the respective syntactic construct with occurrence indices removed. Likewise, we extend $getIdx(\_)$ to multisets of rule heads and CHR^{cp} rules, to return the set of all occurrence indices that appear in them. We similarly extend $dropLabels(\_)$ and $getLabels(\_)$ to be applicable with labeled stores. As a means of generating new labels, we also define the operation $newLabels(Ls, A)$ that returns $A\#n$ such that n does not occur in Ls. Given annotated program $\mathcal{P}_\omega$ and occurrence index i, $\mathcal{P}_\omega[i]$ denotes the rule $R_\omega \in \mathcal{P}_\omega$ in which i occurs, or $\bot$ if i does not occur in any of $\mathcal{P}_\omega$'s rules. The bottom of Figure 5 also defines extensions to the match, residual non-matching and residual unifiability judgment to annotated entities. Applied to the respective occurrence indexed or labeled syntactic constructs, these judgments simply strip away occurrence indices or labels.

The operational semantics of CHR^{cp} is defined by the judgment $\mathcal{P}_\omega \rhd \sigma \;\mapsto_\omega\; \sigma'$, where $\mathcal{P}_\omega$ is an annotated CHR^{cp} program and σ, σ' are execution states. It describes the goal-orientated execution of the CHR^{cp} program $\mathcal{P}_\omega$. We write $\mathcal{P}_\omega \rhd \sigma \;\mapsto_\omega^*\; \sigma'$ for zero or more such derivation steps. The operational semantics introduces administrative derivation steps that describe the incremental processing of constraints, as well as the saturation of propagation rule applications (see below). Execution starts from an *initial* execution state σ of the form $\langle \texttt{[init } \bar{B}\texttt{]} \;;\; \emptyset \rangle$ where $\bar{B}$ is the initial multiset of constraints. Figure 6 shows the core set of rules for this judgment. They handle all cases except those for propagation rules. Rule (*init*) applies when the leading goal has the form $\texttt{init } \bar{B}$. It partitions $\bar{B}$ into $\bar{B}_l$ and $\bar{B}_e$, both of which are unfolded into St_l and St_e respectively. $\bar{B}_l$ contains the multiset of constraints which are monotone w.r.t. to $\mathcal{P}_\omega$ (i.e., $\mathcal{P}_\omega \triangleq^{\neg}_{\mathbf{unf}} \bar{B}_l$). These constraints are *not* added to the store immediately, rather we only add them into the goal as '`lazy`' goals (lazily stored). Constraints $\bar{B}_e$ are not monotone w.r.t. to $\mathcal{P}_\omega$, hence they are immediately added to the store and added to the goals as '`eager`' goals (eagerly stored). This is key to preserving the soundness of the operational semantics w.r.t. the abstract semantics. Rule (*lazy-act*) handles goals of the form $\texttt{lazy } A$: we initiate active matching on A by adding it to the store and adding the new goal $\texttt{act } A\#n\ 1$. Rules (*eager-act*) and (*eager-drop*) deal with the cases of goals of the form $\texttt{eager } A\#n$. The former adds the goal '$\texttt{act } A\#n\ 1$' if $A\#n$ is still present in the store, while the later simply drops the leading goal otherwise. The last four rules handle case for a leading goal of the form $\texttt{act } A\#n\ i$: rules (*act-simpa*-1) and (*act-simpa*-2) handle the cases where the active constraint $A\#n$ matches the i^{th} rule head occurrence of $\mathcal{P}_\omega$, which is a simplified or propagated head respectively. If this match satisfies the rule guard condition, matching partners exist in the store and the comprehension maximality condition is satisfied, we apply the corresponding rule instance. To define these matching conditions, we use the auxiliary judgments defined by the abstract semantics (Figure 3). Note that the rule body instance $\theta\bar{B}$ is added as the new goal $\texttt{init } \bar{B}$. This is because it potentially contains non-monotone constraints: we will employ rule (*init*) to determine the storage policy of each constraint. For rule (*act-simpa*-2), we have the additional condition that the simplified head of the rule be not empty, hence this case does not apply for propagation rules. Rule (*act-next*) applies when the previous two rules do not, hence we cannot apply any instance of the rule with $A\#n$ matching the i^{th} rule head. Finally, rule (*act-drop*) drops the leading goal if occurrence index i does not exist in $\mathcal{P}_\omega$. Since the occurrence index is incremented by (*act-next*) starting with the activation of the goal at index 1, this indicates that we have exhaustively matched the constraint $A\#n$ against all rule head occurrences.

Figure 7 defines the rules that handle propagation rules. Propagation rules need to be managed specially to avoid non-termination. Rule (*act-prop*) defines the case where the active goal $\texttt{act } A\#n\ i$ is such that the rule head occurrence index i is found in a propagation rule, then we replace the leading goal with $\texttt{prop } A\#n\ i\ \emptyset$. Rule (*prop-prop*) applies an instance of this propagation rule that has not been applied before: the application history is tracked by Θ, which contains a set of pairs $(\theta, \bar{n})$. Finally, (*prop-sat*) handles the case where (*prop-prop*) no longer applies, hence *saturation* has been achieved. Since we

$(act\text{-}prop)$	$\mathcal{P} \rhd \langle \texttt{[act } A\#n\ i \texttt{ \| } Gs\texttt{]}\ ;\ Ls\rangle \mapsto_\omega \langle \texttt{[prop } A\#n\ i\ \emptyset \texttt{ \| } Gs\texttt{]}\ ;\ Ls\rangle$ if $\mathcal{P}_\omega[i] = (r\ @\ \bar{H}_p \setminus \emptyset \Longleftrightarrow g \mid \bar{B})$
$(prop{-}prop)$	$\mathcal{P}_\omega\ \rhd \langle \texttt{[prop } A\#n\ i\ \Theta \texttt{ \| } Gs\texttt{]}\ ;\ \wr Ls, Ls_p, Ls_a, A\#n \wr\rangle$ $\mapsto_\omega\ \langle \texttt{[init } \theta\bar{B}\texttt{]} + \texttt{[prop } A\#n\ i\ (\Theta \cup (\theta, \bar{n})) \texttt{ \| } Gs\texttt{]}\ ;\ \wr Ls, Ls_p, Ls_a, A\#n \wr\rangle$ if $\mathcal{P}_\omega[i] = (r\ @\ \wr \bar{H}_p, C : i \wr \setminus \emptyset \Longleftrightarrow g \mid \bar{B})$, there exists some θ such that – $\bar{n} \equiv getLabels(\wr Ls_p, Ls_a, A\#n \wr)$ $(\theta, \bar{n}) \notin \Theta$ (Unique Instance) – $\models \theta g$ $\theta C \triangleq_{\mathbf{lhs}} \wr Ls_a, A\#n \wr$ (Guard Satisfied and Active Match) – $\theta \bar{H}_p \triangleq_{\mathbf{lhs}} Ls_p$ (Partners Match) – $\theta \bar{H}_p \triangleq^{\neg}_{\mathbf{lhs}} Ls$ $\theta C \triangleq^{\neg}_{\mathbf{lhs}} Ls$ (Maximal Comprehension)
$(prop\text{-}sat)$	$\mathcal{P} \rhd \langle \texttt{[prop } A\#n\ i\ \Theta \texttt{ \| } Gs\texttt{]}\ ;\ Ls\rangle \mapsto_\omega \langle \texttt{[act } A\#n\ (i+1) \texttt{ \| } Gs\texttt{]}\ ;\ Ls\rangle$ if $(prop{-}prop)$ does not apply.

Fig. 7. Operational Semantics of CHR^{cp} (Propagation-Set)

uniquely identify an instance of the propagation rule by the pair $(\theta, \bar{n})$, saturation is based on unique permutations of constraints in the store that match the rule heads.

7 Correspondence with the Abstract Semantics

In this section, we relate the operational semantics shown in Section 6 with the abstract semantics (Section 4). Specifically, we prove the soundness of the operational semantics w.r.t. the abstract semantics.

Figure 8 defines a correspondence relation between meta-objects of the operational semantics and those of the abstract semantics. Given an object o_ω of the operational semantics, $o_\alpha = \llbracket o_\omega \rrbracket$ is the corresponding abstract semantic object. For instance, $\llbracket \mathcal{P}_\omega \rrbracket$ strips occurrence indices away from $\mathcal{P}_\omega$. Instead, the abstract constraint store $\llbracket \langle Gs\ ;\ Ls\rangle \rrbracket$ contains constraints in Ls with labels removed, and the multiset union of constraints found in '`init`' and '`lazy`' goals of Gs.

We also need to define several invariants and prove that they are preserved throughout the derivations of the operational semantics of CHR^{cp}. An execution state $\langle Gs\ ;\ Ls\rangle$ is *valid* for program $\mathcal{P}_\omega$ if:

- $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} A$ for any `lazy` $A \in Gs$.
- If $Gs = \texttt{[}G \texttt{ | } Gs'\texttt{]}$, then `init` $\bar{B} \notin Gs'$ for any $\bar{B}$.

Initial states of the form $\langle \texttt{[init } \bar{B}\texttt{]}\ ;\ \emptyset\rangle$ are trivially valid states. Lemma 1 proves that derivation steps $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega \sigma'$ preserve validity during execution.

Lemma 1 (Preservation). *For any program $\mathcal{P}_\omega$, given any valid state σ and any state σ', if $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega \sigma'$, then σ' must be a valid state.*

Proof. The proof proceeds by structural induction on all possible forms of derivation $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega \sigma'$. It is easy to show that each transition preserves validity. □

Lemma 2 states that any derivation step of the operational semantics $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega \sigma'$ is either silent in the abstract semantics (i.e., $\llbracket \sigma \rrbracket \equiv \llbracket \sigma' \rrbracket$) or corresponds to a valid derivation step (i.e., $\llbracket \mathcal{P}_\omega \rrbracket \rhd \llbracket \sigma \rrbracket \mapsto_\alpha \llbracket \sigma' \rrbracket$).

$$\text{Multisets} \begin{cases} \llbracket \lbag \bar{o}, o \rbag \rrbracket & ::= \lbag \llbracket \bar{o} \rrbracket, \llbracket o \rrbracket \rbag \\ \llbracket \emptyset \rrbracket & ::= \emptyset \end{cases} \qquad \text{Rule Head} \begin{cases} \llbracket\ C : i\ \rrbracket & ::= C \end{cases}$$

$$\text{Rule} \begin{cases} \llbracket\ r \ @\ \bar{H}_p \setminus \bar{H}_s \Longleftrightarrow g \mid \bar{B}\ \rrbracket & ::= r\ @\ \llbracket \bar{H}_p \rrbracket \setminus \llbracket \bar{H}_s \rrbracket \Longleftrightarrow g \mid \bar{B} \end{cases}$$

$$\text{State} \begin{cases} \llbracket\ \langle Gs\ ;\ Ls \rangle\ \rrbracket & ::= \lbag \llbracket Gs \rrbracket, \llbracket Ls \rrbracket \rbag \end{cases}$$

$$\text{Constraint} \begin{cases} \llbracket\ A\#n\ \rrbracket & ::= A \end{cases}$$

$$\text{Goals} \begin{cases} \llbracket \texttt{[}G \mid Gs\texttt{]} \rrbracket & ::= \lbag \llbracket G \rrbracket, \llbracket Gs \rrbracket \rbag \\ \llbracket \epsilon \rrbracket & ::= \emptyset \end{cases}$$

$$\text{Goal} \begin{cases} \llbracket\ \texttt{init}\ \bar{B}\ \rrbracket & ::= St \quad \text{s.t.} \quad \bar{B} \ggg_{\mathbf{rhs}} St \\ \llbracket\ \texttt{lazy}\ A\ \rrbracket & ::= \lbag A \rbag \\ \llbracket\ \texttt{eager}\ A\#n\ \rrbracket & ::= \emptyset \\ \llbracket\ \texttt{act}\ A\#n\ i\ \rrbracket & ::= \emptyset \\ \llbracket\ \texttt{prop}\ A\#n\ i\ \Theta\ \rrbracket & ::= \emptyset \end{cases}$$

Fig. 8. Correspondence Relation

Lemma 2 (Correspondence Step). *For any program $\mathcal{P}_\omega$ and valid execution states σ, σ', if $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega \sigma'$, then either $\llbracket \sigma \rrbracket \equiv \llbracket \sigma' \rrbracket$ or $\llbracket \mathcal{P}_\omega \rrbracket \rhd \llbracket \sigma \rrbracket \mapsto_\alpha \llbracket \sigma' \rrbracket$.*

Proof. The proof proceeds by structural induction on all possible forms of derivation $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega \sigma'$. Rules (*act*-*simpa*-1), (*act*-*simpa*-2) and (*prop*-*prop*) correspond to abstract steps. For them, we exploit conditional monotonicity in Theorem 1 and preservation in Lemma 1 to guarantee the validity of corresponding derivation step in the abstract semantics. All other rules are silent. □

Theorem 2 (Soundness). *For any program $\mathcal{P}_\omega$ and valid execution states σ and σ', if $\mathcal{P}_\omega \rhd \sigma \mapsto_\omega^* \sigma'$, then $\llbracket \mathcal{P}_\omega \rrbracket \rhd \llbracket \sigma \rrbracket \mapsto_\alpha^* \llbracket \sigma' \rrbracket$.*

Proof. The proof proceeds by induction on derivation steps. The inductive case is proved using Lemmas 2 and 1. □

While the operational semantics is sound, completeness w.r.t. the abstract semantics does not hold. There are two aspects of the operational semantics that contributes to this: first the saturation behavior of propagation rules (Figure 7) is not modeled in the abstract semantics. This means that while a program $\mathcal{P}_\omega$ with a propagation rule terminates in the operational semantics (thanks to saturation), $\llbracket \mathcal{P}_\omega \rrbracket$ may diverge in the abstract semantics. Second, although we can model negation with comprehension patterns, we cannot guarantee completeness when we do so. For instance, consider the rule $\lbag a(X) \rbag_{X \in Xs} \Longleftrightarrow Xs = \emptyset \mid noA$, which adds noA to the constraint store if there are no occurrences of $a(X)$ for any value of X. The application of this rule solely depends on the absence of any occurrences $a(X)$ in the store. Yet, in our operational semantics, rule application is triggered only by the presence of constraints. The idea of negated active constraint can be borrowed from [7] to rectify this incompleteness, but space limitations prevent us from discussing the details of this conservative extension to our operational semantics.

8 Related Work

An extension of CHR with aggregates is proposed in [5]. This extension allows the programmer to write CHR rules with aggregate constraints that incrementally maintains term-level aggregate computations. Differently from our comprehension patterns, these aggregate constraints are only allowed to appear as *propagated* rule heads. The authors of [5] also suggested extending the refined CHR operational semantics [2] with aggregates, in a manner analogous to their previous work on CHR with negated constraints [7]. While both extensions (aggregates and negated constraints) introduce non-monotonicity in the respective CHR semantics, the observable symptoms (from an end-user's perspective) of this non-monotonicity are described as "unexpected behaviors" in [7], serving only as caveats for the programmers. No clear solution is proposed at the level of the semantics. By contrast, our work here directly addresses the issue of incrementally processing of constraints in the presence of non-monotonicity introduced by comprehension patterns.

The logic programming language Meld [1], originally designed to program cyber-physical systems, offers a rich range of features including aggregates and a limited form of comprehension patterns. To the best of our knowledge, a low-level semantics on which to base an efficient implementation of Meld has not yet been explored. By contrast, our work explores comprehension patterns in multiset rewriting rules in detail and defines an operational semantics that is amenable to an incremental strategy for processing constraints.

9 Conclusion and Future Works

In this paper, we introduced CHR^{cp}, an extension of CHR with multiset comprehension patterns. We defined an abstract semantics for CHR^{cp}, followed by an operational semantics and proved its soundness with respect to the abstract semantics. We are currently developing a prototype implementation based on the operational semantics discussed here.

In future work, we intend to further develop our prototype implementation of CHR^{cp} by investigating efficient compilation schemes for comprehension patterns. We also wish to explore alternatives to our current greedy comprehension pattern matching scheme. We also intend to extend CHR^{cp} with some result form prior work in [4] and develop a decentralized multiset rewriting language. Finally, we believe that the operational semantics of CHR^{cp} can be generalized to other non-monotonic features along the lines of [5,7].

References

1. F. Cruz, M. P. Ashley-Rollman, S. C. Goldstein, Ricardo Rocha, and F. Pfenning. Bottom-Up Logic Programming for Multicores. In Vítor Santos Costa, editor, *Proc. of DAMP 2012*. ACM Digital Library, January 2012.
2. G. J. Duck, P. J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *In 20th Int. Conf. on Logic Programming ICLP'04*, pages 90–104. Springer, 2004.
3. T. Frühwirth. Constraint Handling Rules. In *Constraint Prog.*, pages 90–107, 1994.
4. E.S.L. Lam and I. Cervesato. Decentralized Execution of Constraint Handling Rules for Ensembles. In *Proc. PPDP'13*, pages 205–216, Madrid, Spain, 2013.
5. J. Sneyers, P. V. Weert, T. Schrijvers, and B. Demoen. Aggregates in Constraint Handling Rules. In *ICLP*, pages 446–448, 2007.
6. A. Triossi, S. Orlando, A. Raffaetà, and T. W. Frühwirth. Compiling CHR to parallel hardware. In *PPDP*, pages 173–184, 2012.
7. P. V. Weert, J. Sneyers, T. Schrijvers, and B. Demoen. Extending CHR with Negation as Absence. In *CHR Workshop*, pages 125–140, 2006.