Optimized Compilation of Multiset Rewriting with Comprehensions ⋆

Edmund S. L. Lam and Iliano Cervesato

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

Abstract. We extend the rule-based, multiset rewriting language CHR with multiset comprehension patterns. Multiset comprehension provides the programmer with the ability to write multiset rewriting rules that can match a variable number of entities in the state. This enables implementing algorithms that coordinate large amounts of data or require aggregate operations in a declarative way, and results in code that is more concise and readable than with pure CHR. We call this extension CHR^{cp}. In this paper, we formalize the operational semantics of CHR^{cp} and define a low-level optimizing compilation scheme based on join ordering for the efficient execution of programs. We provide preliminary empirical results that demonstrate the scalability and effectiveness of this approach.

1 Introduction

CHR is a declarative logic constraint programming language based on pure forward-chaining and committed choice multiset rewriting. This provides the user 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 satisfying a property 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 code, suggesting the opportunity for a higher form of abstraction. This paper develops an extension of CHR with *multiset comprehension patterns* [11,2]. 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 amounts of data or use aggregate operations. We call this extension CHR^{cp}.

In previous work [7], we presented an abstract semantics for CHR^{cp} and concretized it into an operational semantics. This paper defines a compilation scheme for CHR^{cp} rules that enables an optimized execution for this operational semantics. This compilation scheme, based on *join ordering* [4,6], determines an optimized sequence of operations to carry out the matching of constraints and guards. This ordering is optimized in that it utilizes the most effective supported indexing methodologies (e.g.,

⋆ 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.

hash map indexing, binary tree search) for each constraint pattern and schedules guard condition eagerly, thereby saving potentially large amounts of computation by pruning unsatisfiable branches as early as possible. The key challenge of this approach is to determine such an optimized ordering and to infer the set of lookup indices required to execute the given CHR^{cp} program with the best possible asymptotic time complexity. Our work augments the approach from [6] to handle comprehension patterns, and we provide a formal definition of this compilation scheme and an abstract machine that implements the resulting compiled CHR^{cp} programs.

Altogether, this paper makes the following contributions: We define a scheme that compiles CHR^{cp} rules into optimized join orderings. We formalize the corresponding CHR^{cp} abstract matching machine. We prove the soundness of this abstract machine with respect to the operational semantics. We provide preliminary empirical results to show that a practical implementation of CHR^{cp} is possible.

The rest of the paper is organized as follows: Section 2 introduces CHR^{cp} by examples and Section 3 gives its syntax. In Section 4, we describe an operational semantics for CHR^{cp} and define our compilation scheme in Section 5. Section 6 builds optimized join orderings of CHR^{cp} rules. Section 7 defines the abstract state machine and Section 8 establishes correctness results. In Section 9 we present preliminary empirical results. Section 10 situates CHR^{cp} in the literature and Section 11 outlines directions of future work.

2 A Motivating Example

In this section, we illustrate the benefits of comprehension patterns in multiset rewriting with an example. 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 the elements of the *comprehension domain* t.

Consider the problem of two agents wanting to swap data that they each possess on the basis of a pivot value P. We express an integer datum I belonging to agent X by the constraint $data(X, I)$. The state of this dynamic system is represented by a multiset of ground constraints, the constraint store. Given agents X and Y and a value P, we want all of X's data with value I less than or equal to P to be transferred to Y and all of Y's data J such that J is greater than or equal to P to be transferred to X. Notice that the value P satisfies the conditions both for I and J. The following CHR^{cp} rule implements this swap procedure:

$$selSwap \;@\; \begin{array}{l} swap(X, Y, P) \\ \wr data(X, I) \mid I \leq P\int_{I\in Xs} \\ \wr data(Y, J) \mid J \geq P\int_{J\in Ys} \end{array} \iff \begin{array}{l} \wr data(Y, I)\int_{I\in Xs} \\ \wr data(X, J)\int_{J\in Ys} \end{array}$$

The swap is triggered by the constraint $swap(X, Y, P)$ in the rule head on the left of $\iff$. All of X's data I such that $I \leq P$ are identified by the comprehension pattern $\wr data(X, I) \mid I \leq P\int_{I\in Xs}$. Similarly, all Y's data J such that $J \geq P$ are identified by $\wr data(Y, J) \mid J \geq P\int_{J\in Ys}$. The instances of I and J 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 on the right of $\iff$ to complete the rewriting by redistributing all of X's selected data to Y and vice versa. 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 rules that manipulate dynamic numbers of constraints. By contrast, 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:

$$
\begin{array}{rl}
\mathit{init} \ @ \ \mathit{swap}(X, Y, P) & \iff \mathit{grab1}(X, P, Y, [\,]), \mathit{grab2}(Y, P, X, [\,]) \\
\mathit{gIter1} \ @ \ \mathit{grab1}(X, P, Y, \mathit{Is}), \mathit{data}(X, I) & \iff I \leq P \mid \mathit{grab1}(X, P, Y, [I \mid \mathit{Is}]) \\
\mathit{gEnd1} \ @ \ \mathit{grab1}(X, P, Y, \mathit{Is}) & \iff \mathit{unrollData}(Y, \mathit{Is}) \\
\mathit{gIter2} \ @ \ \mathit{grab2}(Y, P, X, \mathit{Js}), \mathit{data}(Y, J) & \iff J \geq P \mid \mathit{grab2}(Y, P, X, [J \mid \mathit{Js}]) \\
\mathit{gEnd2} \ @ \ \mathit{grab2}(Y, P, X, \mathit{Js}) & \iff \mathit{unrollData}(X, \mathit{Js}) \\
\mathit{unrollIter} \ @ \ \mathit{unrollData}(L, [D \mid \mathit{Ds}]) & \iff \mathit{unrollData}(L, \mathit{Ds}), \mathit{data}(L, D) \\
\mathit{unrollEnd} \ @ \ \mathit{unrollData}(L, [\,]) & \iff \mathit{true}
\end{array}
$$

In a CHR program with several subroutines of this nature, such boilerplate code gets repeated over and over, making the program verbose. Furthermore, the use of list accumulators and auxiliary constraints (e.g., $grab1$, $grab2$, $unrollData$) makes the code less readable and more prone to errors. 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. Observe also that this pure CHR implementation assumes a priority semantics [3]: rule $gEnd1$ is to be used only when rule $gIter1$ is *not* applicable, and similarly for rules $gEnd2$ and $gIter2$. Rule priority guarantees that all eligible $data$ constraints participate in the swap. We may be tempted to implement the swap procedure as follows in standard CHR:

$$
\begin{array}{rl}
\mathit{swap1} \ @ \ \mathit{swap}(X, Y, I), \mathit{data}(X, I) & \iff I \leq P \mid \mathit{swap}(X, Y, I), \mathit{data}(Y, I) \\
\mathit{swap2} \ @ \ \mathit{swap}(X, Y, J), \mathit{data}(Y, J) & \iff J \geq P \mid \mathit{swap}(X, Y, J), \mathit{data}(X, J) \\
\mathit{swap3} \ @ \ \mathit{swap}(X, Y, D) & \iff \mathit{true}
\end{array}
$$

This, however, does not work in general. This is because the matching conditions of $swap1$ and $swap2$ are potentially overlapping: if we have $data(X, P)$ in the constraint store, applying $swap1$ to it will produce $data(Y, P)$, which will inevitably be reversed by an application of $swap2$, thereby locking the execution in a non-terminating cycle. This code is however correct were the conditions on X's and Y's values to be complementary (e.g., $I < P$ and $J \geq P$). But it is still non-atomic and relies on prioritization as the last rule should be triggered only when neither of the first two is applicable. By contrast, multiset comprehensions in CHR^{cp} provides a high-level abstraction that relinquishes all these technical concerns from the programmer's hands.

3 Syntax and Notations

In this section, we define the abstract syntax of CHR^{cp} and highlight the notations used throughout this paper. We write $\bar{o}$ for a multiset of syntactic objects o, with $\varnothing$ indicating the empty multiset. We write $\wr \bar{o}_1, \bar{o}_2 \wr$ 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 $\wr \bar{o}, o \wr$. Multiset comprehension at the meta-level is denoted by

Variables: x Predicates: p Rule names: r Primitive terms: t_α Occurrence index: i

$$\begin{array}{rrcl} \textit{Terms}: & t & ::= & t_\alpha \mid \bar{t} \mid \wr t \mid g \int_{\vec{x} \in t} \\ \textit{Guards}: & g & ::= & t = t \mid t \doteq t \mid t < t \mid t \leq t \mid t > t \mid t \geq t \mid g \wedge g \end{array}$$

$$\begin{array}{rrclrrcl} \textit{Atomic Constraints}: & A & ::= & p(\vec{t}) & \textit{Head Constraints}: & H & ::= & C : i \\ \textit{Comprehensions}: & M & ::= & \wr A \mid g \int_{\vec{x} \in t} & \textit{Rules}: & R & ::= & r \mathbin{@} \bar{H} \iff g \mid \bar{B} \\ \textit{Rule Constraints}: & C, B & ::= & A \mid M & \textit{Programs}: & \mathcal{P} & ::= & \bar{R} \end{array}$$

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

$\wr o \mid \Phi(o) \int$, where o a meta object and $\Phi(o)$ is a logical statement on o. We write $\vec{o}$ for a comma-separated tuple of o's. A list of objects o is also denoted by $\vec{o}$ and given o, we write $[o \mid \vec{o}]$ for the list with head o and tail $\vec{o}$. The empty list is denoted by $[]$. We will explicitly disambiguate lists from tuples where necessary. Given a list $\vec{o}$, we write $\vec{o}[i]$ for the i^{th} element of $\vec{o}$, with $\vec{o}[i] = \bot$ if i is not a valid index in $\vec{o}$. We write $o \in \vec{o}$ if $\vec{o}[i] \neq \bot$ for some i. The set of valid indices of the list $\vec{o}$ is denoted $range(\vec{o})$. The concatenation of list $\vec{o}_1$ with $\vec{o}_2$ is denoted $\vec{o}_1 +\!+ \vec{o}_2$. We abbreviate a singleton list containing o as $[o]$. Given a list $\vec{o}$, we write $\wr \vec{o} \int$ to denote the multiset containing all (and only) the elements of $\vec{o}$. 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., a comprehension pattern), substitution implicitly α-renames variables to avoid capture. It will be convenient to assume that terms get normalized during substitution. The composition of substitutions θ and ϕ is denoted $\theta\phi$.

Figure 1 defines the abstract syntax of CHR^{cp}. 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 variables $\vec{x}$ are locally bound with scope A and g. We implicitly α-rename binding variables to avoid capture. The development of CHR^{cp} is largely agnostic to the language of terms [7]. In this paper however, we assume for simplicity that t_α are arithmetic terms (e.g., 10, $x + 4$). We also include tuples and multisets of such terms. Term-level multiset comprehension $\wr t \mid g \int_{x \in m}$ filters multiset m according to guard g and maps the result as specified by t.

A CHR head constraint $C : i$ is a constraint C paired with an occurrence index i. As in CHR, a CHR^{cp} rule $r \mathbin{@} \bar{H} \iff g \mid \bar{B}$ specifies the rewriting of the *head constraints* $\bar{H}$ into the *body* $\bar{B}$ under the conditions that guards g are satisfied; r is the name of the rule.[1] If the guard g is always satisfied (i.e., $true$), we drop that rule component entirely. All free variables in a CHR^{cp} rule are implicitly universally quantified at the head of the rule. A CHR program is a set of CHR rules and we require that each head constraint has a unique occurrence index i. For simplicity, we assume that a rule body is grounded by the head constraints and that guards do not appear in the rule body.

[1] CHR rules traditionally have a fourth component, the propagation head, which we omit in the interest of space as it does not fundamentally impact the compilation process or our abstract machine. See [7] for a treatment of comprehension patterns in propagation heads.

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

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

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

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

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

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

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}

4 Operational Semantics of CHR^{cp}

This section recalls the operational semantics of CHR^{cp} [7]. Without loss of generality, we assume that atomic constraints in a 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 $\not\models g$.

Similarly to [5], this operational semantics defines a goal-based execution of a CHR^{cp} program $\mathcal{P}$ that incrementally processes store constraints against rule instances in $\mathcal{P}$. By "incrementally", we mean that goal constraints are added to the store one by one, as we process each for potential match with the head constraints of rules in $\mathcal{P}$. We present the operational semantics in two sub-sections: Section 4.1 describes in isolation, the processing of a rule's left-hand side (*semantics of matching*) and right-hand-side execution. Section 4.2 presents the overall operational semantics. We assume that the constraint store contains only ground facts, a property that is maintained during execution. This entail that matching (as opposed to unification) suffices to guarantee the completeness of rule application.

4.1 Semantics of Matching and Rule Body Execution

The semantics of matching, specified in Figure 2, identifies applicable rules in a CHR^{cp} program by matching their head with the constraint store. The 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 ground (i.e., $FV(\bar{C}) = \emptyset$). Rules $(\mathbf{l}_{mset-*})$ iterate rules $(\mathbf{l}_{atom})$ and $(\mathbf{l}_{comp-*})$ 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 $(\mathbf{l}_{comp-*})$ match a comprehension pattern $\wr A \mid g \int_{\vec{x} \in ts}$. If the comprehension domain is empty ($x \in \varnothing$), the store must be empty

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

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

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

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

Residual Non-unifiability: $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} \bar{B} \quad g \rhd \bar{H} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}$

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

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

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

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

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

Fig. 3. Rule Body Application and Unifiability of Comprehension Patterns

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. To guarantee the maximality of comprehension patterns, we test a store for *residual matchings* using the residual non-matching judgment $\bar{C} \triangleq^{\neg}_{\mathbf{lhs}} St$ (Also shown in Figure 2). 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.

Once a CHR^{cp} rule instance has been identified, we need to *unfold* the comprehension patterns in its body into a multiset of atomic constraints that will be added to the store. The judgment $\bar{C} \ggg_{\mathbf{rhs}} St$ does this unfolding: given $\bar{C}$, this judgment holds if and only if St is the multiset of all (and only) constraints found in $\bar{C}$, after comprehension patterns in $\bar{C}$ have been unfolded. Figure 3 defines this judgment.

An important property of CHR is *monotonicity*: if a rule instance r transforms store Ls to Ls', then r transforms $\wr Ls, Ls''\int$ to $\wr Ls', Ls''\int$ for any Ls''. This property allows for incremental processing of constraints ([5]) that is sound w.r.t. the abstract semantics of CHR. Monotonicity does not hold in CHR^{cp}. We showed in [7] that to guarantee the sound incremental goal-based execution of a CHR^{cp} program $\mathcal{P}$, we must identify those rule body constraints are *monotone*, and only incrementally store monotone constraints, while non-monotone constraints are immediately stored. A monotone constraint in program $\mathcal{P}$ is a constraint A that can never be matched by a comprehension head constraint of any rule in $\mathcal{P}$. To test that a comprehension pattern M has no match

$$\text{Goal Constraint} \quad G ::= \texttt{init}\ \bar{B} \mid \texttt{lazy}\ A \mid \texttt{eager}\ A\#n \mid \texttt{act}\ A\#n\ i$$

$$\text{Goal Stack} \quad Gs ::= [] \mid [G \mid Gs] \qquad \text{Store} \quad Ls ::= \varnothing \mid \wr Ls, A\#n \wr \qquad \text{State} \quad \sigma ::= \langle Gs\ ;\ Ls \rangle$$

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

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

$$\mathcal{P}[i] \quad ::= \text{if } R \in \mathcal{P} \text{ and } i \in getIdx(R) \text{ then } R \text{ else } \bot$$

Fig. 4. Execution States and Auxiliary Meta-operations

in a store Ls (i.e., $M \triangleq^{\neg}_{\mathbf{lhs}} Ls$), it suffices to test M against the subset of Ls containing just its non-monotone constraints (see [7] for proofs). We call this property of CHR^{cp} *conditional monotonicity*. Given a CHR^{cp} program $\mathcal{P}$, for each rule body constraint B in $\mathcal{P}$, if for every head constraint comprehension pattern $M : j$ and rule guard g in $\mathcal{P}$, B is not unifiable with M while satisfying g (denoted $g \rhd M \sqsubseteq_{\mathbf{unf}} B$), then we say that B is *monotone* w.r.t. program $\mathcal{P}$, denoted by $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} B$. These judgments are defined in the bottom half of Figure 3.

4.2 Operational Semantics

In this section, we define the overall operational semantics of CHR^{cp}. This semantics explicitly supports partial incremental processing of constraints that are monotone to a given CHR^{cp} program. Execution states, defined in Figure 4, are pairs $\sigma = \langle Gs\ ;\ Ls \rangle$ where Gs is the *goal stack* and Ls is the *labeled store*. Store labels n allow us to distinguish between copies of the same constraint in the store and to uniquely associate a goal constraint with a specific stored constraint. Each goal in a goal stack Gs represents a unit of execution and Gs itself is a list of goals to be executed. Goal labels `init`, `lazy`, `eager` and `act` identify the various types of goals.

Figure 4 defines several auxiliary operations that either retrieve or drop occurrence of indices and store labels: $dropIdx(H)$ and $getIdx(H)$ deal with indices, $dropLabels(\_)$ and $getLabels(\_)$ with labels. We inductively extend $getIdx(\_)$ to multisets of head constraints 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 program $\mathcal{P}$ and occurrence index i, $\mathcal{P}[i]$ denotes the rule $R \in \mathcal{P}$ in which i occurs, or $\bot$ if i does not occur in any of $\mathcal{P}$'s rules. We implicitly extend the matching judgment ($\triangleq_{\mathbf{lhs}}$) and residual non-matching judgment ($\triangleq^{\neg}_{\mathbf{lhs}}$) to annotated entities.

The operational semantics of CHR^{cp} is defined by the judgment $\mathcal{P} \rhd \sigma \ \mapsto_{\omega}\ \sigma'$, where $\mathcal{P}$ is a CHR^{cp} program and σ, σ' are execution states. It describes the goal-oriented execution of the CHR^{cp} program $\mathcal{P}$. Execution starts in an *initial* execution state σ of the form $\langle [\texttt{init}\ \bar{B}]\ ;\ \varnothing \rangle$ where $\bar{B}$ is the initial multiset of constraints. Figure 5 shows the transition rules for this judgment. 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 (via rule body application, Section 4.1). $\bar{B}_l$ contains the multiset of constraints which are monotone w.r.t. to $\mathcal{P}$ (i.e., $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}_l$). These constraints are *not* added to the store immediately, rather we incrementally process them by only adding them into the goal as '`lazy`' goals (lazily stored). Constraints $\bar{B}_e$ are not monotone w.r.t. to $\mathcal{P}$, hence they are immediately added to the store and added to

$(init)$	$\mathcal{P} \rhd \langle$ `[init` $\wr \bar{B}_l, \bar{B}_e \int$ `|` Gs `]` ; $Ls\rangle \mapsto_\omega \langle lazy(St_l)$++$eager(Ls_e)$++$Gs$; $\wr Ls, Ls_e \int\rangle$ such that $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} \bar{B}_l \quad \bar{B}_e \ggg_{\mathbf{rhs}} St_e \quad \bar{B}_l \ggg_{\mathbf{rhs}} St_l \quad Ls_e = newLabels(Ls, St_e)$ where $eager(\wr Ls, A\#n \int) ::=$ `[eager` $A\#n$ `|` $eager(Ls)$`]` $\quad eager(\varnothing) ::=$ `[]` $lazy(\wr St_m, A \int) ::=$ `[lazy` A `|` $lazy(St_m)$`]` $\quad lazy(\varnothing) ::=$ `[]`
$(lazy{-}act)$	$\mathcal{P} \rhd \langle$ `[lazy` A `|` Gs `]` ; $Ls\rangle \mapsto_\omega \langle$ `[act` $A\#n\ 1$ `|` Gs `]` ; $\wr Ls, A\#n \int\rangle$ such that $\wr A\#n \int = newLabels(Ls, \wr A \int)$
$(eager{-}act)$	$\mathcal{P} \rhd \langle$ `[eager` $A\#n$ `|` Gs `]` ; $\wr Ls, A\#n \int\rangle \mapsto_\omega \langle$ `[act` $A\#n\ 1$ `|` Gs `]` ; $\wr Ls, A\#n \int\rangle$
$(eager{-}drop)$	$\mathcal{P} \rhd \langle$ `[eager` $A\#n$ `|` Gs `]` ; $Ls\rangle \mapsto_\omega \langle Gs$; $Ls\rangle$ if $A\#n \notin Ls$
$(act{-}apply)$	$\mathcal{P} \rhd \langle$ `[act` $A\#n\ i$ `|` Gs `]` ; $\wr Ls, Ls_h, Ls_a, A\#n \int\rangle \mapsto_\omega \langle$ `[init` $\theta\bar{B}$ `|` Gs `]` ; $Ls\rangle$ if $\mathcal{P}[i] = (r \mathbin{@} \wr \bar{H}_h, C : i \int \Longleftrightarrow g \mid \bar{B})$, there exists some θ such that $\models \theta g \quad \theta C \triangleq_{\mathbf{lhs}} \wr Ls_a, A\#n \int \quad \theta\bar{H}_h \triangleq_{\mathbf{lhs}} Ls_h \quad \theta\bar{H}_h \triangleq^{\neg}_{\mathbf{lhs}} Ls \quad \theta C \triangleq^{\neg}_{\mathbf{lhs}} Ls$
$(act{-}next)$	$\mathcal{P} \rhd \langle$ `[act` $A\#n\ i$ `|` Gs `]` ; $Ls\rangle \mapsto_\omega \langle$ `[act` $A\#n\ (i+1)$ `|` Gs `]` ; $Ls\rangle$ if $(act{-}apply)$ does not applies.
$(act{-}drop)$	$\mathcal{P} \rhd \langle$ `[act` $A\#n\ i$ `|` Gs `]` ; $Ls\rangle \mapsto_\omega \langle Gs$; $Ls\rangle$ if $\mathcal{P}[i] = \bot$

Fig. 5. Operational Semantics of CHR^{cp}

the goals as '`eager`' goals (eagerly stored). Rule $(lazy{-}act)$ handles goals of the form `lazy` A: we initiate active matching on A by adding it to the store and adding the new goal `act` $A\#n\ 1$. Rules $(eager{-}act)$ and $(eager{-}drop)$ deal with goals of the form `eager` $A\#n$. The former adds the goal '`act` $A\#n\ 1$' if $A\#n$ is still present in the store; the later simply drops the leading goal otherwise. The last three rules deal with leading goals of the form `act` $A\#n\ i$: rule $(act{-}apply)$ handles the case where the active constraint $A\#n$ matches the i^{th} head constraint occurrence of $\mathcal{P}$. If this match satisfies the rule guard, matching partners exist in the store and the comprehension maximality condition is satisfied, we apply the corresponding rule instance. These matching conditions are defined by the semantics of matching of CHR^{cp} (Figure 2). Note that the rule body instance $\theta\bar{B}$ is added as the new goal `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. 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} head constraint. Finally, rule $(act{-}drop)$ drops the leading goal if occurrence index i does not exist in $\mathcal{P}$. The correctness of this operational semantics w.r.t. a more abstract semantics for CHR^{cp} is proven in [7].

5 Compiling CHR^{cp} Rules

While Figures 2–5 provide a formal operational description of the overall multiset rewriting semantics of CHR^{cp}, they are high-level in that they keep multiset matching abstract. Specifically, the use of judgments $\triangleq_{\mathbf{lhs}}$ and $\triangleq^{\neg}_{\mathbf{lhs}}$ in rule $(act{-}apply)$ hides away crucial details of how a practical implementation is to conduct these expensive operations. In this section, we describe a scheme that compiles CHR^{cp} head constraints into a lower-level representation optimized for efficient execution, without using $\triangleq_{\mathbf{lhs}}$ or $\triangleq^{\neg}_{\mathbf{lhs}}$. This compilation focuses on CHR^{cp} head constraints (left-hand side), where the bulk of execution time (and thus most optimization opportunities) comes from.

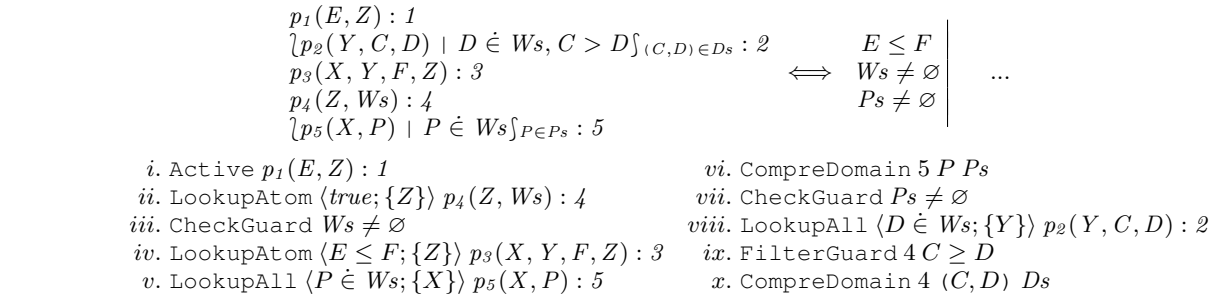

Fig. 6. Optimized Join Ordering for $p_1(E, Z) : 1$

As described in Section 4, an active constraint $\texttt{act}\ A\#n\ i$ is matched against an occurrence of head constraint H_i in a rule r, and all other head constraints H_k in r are matched against distinct constraints in the store. We call H_i the *active head constraint* and the other H_k *partner head constraints* (or simply, *active pattern* and *partners* respectively). Computing complete matches for the multiset of constraint patterns is a combinatorial search problem. In general, any ordering of partners leads to the computation of intermediate data that may ultimately be discarded, resulting in redundant storage and processing time. Therefore, we want to determine an optimized ordering of partners that minimizes this intermediate data. Join ordering [4,6] leverages the dependencies among rule heads and rule guards to do precisely this. This allows pruning search branches early and utilizing lookup methods (e.g., indexing on hash maps, balanced trees) that provide the best possible asymptotic time complexity. Our work extends traditional approaches to CHR compilation [6] to handle comprehension head constraints and augments them with optimizations specific to them. In particular, our approach is an extension of *static* join-ordering techniques (e.g., [4]) that relies on a heuristic cost model to determine optimized orderings at compile-time.

5.1 Introducing CHR^{cp} Join Ordering

The top of Figure 6 shows an example rule with five head constraints. In this example, all predicates are different, hence each head constraint will always match distinct constraints from the store (in Section 5.3, we discuss the case where different rule heads match the same constraint). To better appreciate the benefits of join ordering, consider an example constraint store Ls of the form:

$$p_1(t_{E1}, t_{Z1}),\ \biguplus_{i=1}^{n_2} p_2(t_{Yi}, t_{Ci}, t_{Di}),\ \biguplus_{i=1}^{n_3} p_3(t_{Xi}, t_{Yi}, t_{Fi}, t_{Zi}),\ \biguplus_{i=1}^{n_4} p_4(t_{Zi}, t_{Wsk}),\ \biguplus_{i=1}^{n_5} p_5(t_{Xi}, t_{Pi})$$

where $\biguplus_{i=1}^{n} p(\vec{t_i})$ denotes a store fragment containing n ground constraints of the form $p(\vec{t_i})$. Hence n_2, n_3, n_4 and n_5 are the number of constraints in the store for the predicates p_2, p_3, p_4 and p_5, respectively. As we carry out this analysis, we optimistically assume that each of the n_2 instances of p_2 has a different term t_{Yi} in its first argument, and similarly for each argument position and predicate.

Consider a naive execution of the rule in Figure 6 in the textual order with active constraint $\texttt{act}\ p_1(t_{E1}, t_{Z1})\#n\ i$ for some n and i, so that $p_1(E, Z) : 1$ is the active pattern. This binds variables E and Z to terms t_{E1} and t_{Z1} respectively. Next, we identify all constraints $p_2(t_{Yi}, t_{Ci}, t_{Di})$ such that $C > D$, and for each bindings t_{Yi} for Y,

we build the comprehension range Ds from the t_{Ci}'s and t_{Di}'s. Since this pattern shares no common variables with the active pattern and variable Ws is not ground, to build the above match we have no choice but examining all n_2 constraints for p_2 in the store. Furthermore, the guard $D \in Ws$ would have to be enforced at a later stage, after $p_4(Z, Ws)$ is matched, as a post comprehension filter. We next seek a match for $p_3(X, Y, F, Z) : 3$. Because it shares variables Y and Z with patterns 1 and 2, we can find matching candidates in constant time, if we have the appropriate indexing support ($p_3(\_, Y, \_, Z)$). The next two patterns ($p_4(Z, Ws) : 4$ and $\wr p_5(X, P) \mid P \dot{\in} Ws \wr_{P \in Ps} : 5$) are matched in a similar manner and finally $Ps \neq \varnothing$ is checked at the very end. This naive execution has two main weaknesses: first, scheduling partner 2 first forces the lower bound of the cost of processing this rule to be $O(n_2)$, even if we find matches to partners 3 and 4 in constant time. Second, suppose we fail to find a match for partner 5 such that $Ps \neq \varnothing$, then the execution time spent computing Ds of partner 2, including the time to search for candidates for partners 3 and 4, was wasted.

Now consider the join ordering for the active pattern $p_1(E, Z) : 1$ shown in Figure 6. It is an optimized ordering of the partner constraints in this instance: Task (i) announces that $p_1(E, Z) : 1$ is the constraint pattern that the active constraint must match. Task (ii) dictates that we look up the constraint $p_4(Z, Ws)$. This join task maintains a set of possible constraints that match partner 4 and the search proceeds by exploring each constraint as a match to partner 4 until it finds a successful match or fails; the *indexing directive* $I = \langle true; \{Z\}\rangle$ mandates a hash multimap lookup for p_4 constraints with first argument value of Z (i.e., $p_4(Z, \_)$). This allows the retrieval of all matching candidate constraints from Ls in amortized constant time (as oppose to linear $O(n_4)$). Task (iii) checks the guard condition $Ws \neq \varnothing$: if no such $p_4(Z, Ws)$ exists, execution of this join ordering can terminate *immediately* at this point (a stark improvement from the naive execution). Task (iv) triggers the search for $p_3(X, Y, F, Z)$ with the indexing directive $\langle E \leq F; \{Z\}\rangle$. This directive specifies that candidates of partner 3 are retrieved by utilizing a two-tiered indexing structure: a hash table that maps p_3 constraints in their fourth argument (i.e., $p_3(\_, \_, \_, Z)$) to a binary balance tree that stores constraints in sorted order of the third argument (i.e., $p_3(\_, \_, F, \_)$, $E \leq F$). The rule guard $E \leq F$ can then be omitted from the join ordering, since its satisfiability is guaranteed by this indexing operation. Task (v) initiates a lookup for constraints matching $p_5(X, P) : 5$ which is a comprehension. It differs from Tasks (ii) and (iv) in that rather than branching for each candidate match to $p_5(X, P) : 5$, we collect the set of all candidates as matches for partner 5. The multiset of constraints matching this partner is efficiently retrieved by the indexing directive $\langle P \dot{\in} Ws; \{X\}\rangle$. Task (vi) computes the comprehension domain Ps by projecting the multiset of instances of P from the candidates of partner 5. The guard $Ps \neq \varnothing$ is scheduled at Task (vii), pruning the current search immediately if Ps is empty. Tasks $(viii - x)$ represent the best execution option for partner 2, given that composite indexing ($D \dot{\in} Ws$ and $C \leq D$) is not yet supported in our implementation: Task $(viii)$ retrieves candidates matching $p_2(Y, C, D) : 2$ via the indexing directive $\langle D \dot{\in} Ws; \{Y\}\rangle$, which specifies that we retrieve candidates from a hash multimap that indexes p_2 constraints on the first and third argument (i.e., $p_2(Y, \_, D)$); values of D are enumerated from Ws. Task (ix) does a post-comprehension filter, removing candidates of partner 2 that do not satisfy $C \leq D$. Finally, task (x) computes the comprehension domain Ds. While we still conduct a post comprehension filtering (Task (ix)), this filters from a small set of candidates (i.e.,

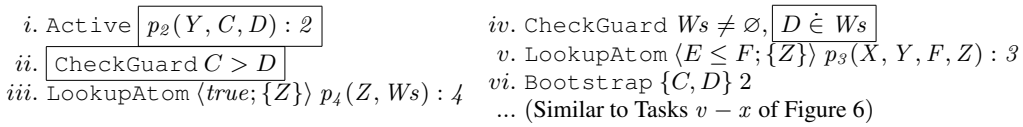

Fig. 7. Optimized Join Ordering for $\lbag p_2(Y, C, D) \mid D \dot{\in} Ws, C > D \rbag_{(C,D) \in Ds} : 2$

$p_2(Y, \_, D)$ where $D \dot{\in} Ws$) and hence is likely more efficient than linear enumeration and filtering on the store (i.e., $O(\mid Ws \mid)$ *vs* $O(n_2)$).

Such optimized join orderings are statically computed by our compiler and the constraint store is compiled to support the set of all indexing directives that appears in the join orderings. In general, our implementation always produces join orderings that schedule comprehension partners after all atom partners. This is because comprehension lookups (`LookupAll`) never fail and hence do not offer any opportunity for early pruning. However, orderings within each of the partner categories (atom or comprehension) are deliberate. For instance, $p_4(Z, Ws) : 4$ was scheduled before $p_3(X, Y, F, Z) : 3$ since it is more constrained: it has fewer free variables and $Ws \neq \varnothing$ restricts it. Comprehension partner 5 was scheduled before 2 because of guard $Ps \neq \varnothing$ and also that 2 is considered more expensive because of the post lookup filtering (Task (ix)). Cost heuristics are discussed in Section 6.

5.2 Bootstrapping for Active Comprehension Head Constraints

In the example in Figure 6, the active pattern is an atomic constraint. Our next example illustrates the case where the active pattern H_i is a comprehension. In this case, the active constraint $A\#n$ must be part of a match with the comprehension rule head $H_i = \lbag A' \mid g \rbag_{x \in xs} : i$. While the join ordering should allow early detection of failure to match A with A' or to satisfy comprehension guard g, it must also avoid scheduling comprehension rule head H_i before atomic partner constraints are identified. Our implementation uses *bootstrapping* to achieve this balance: Figure 7 illustrates this compilation for the comprehension head constraint $\lbag p_2(Y, C, D) \mid D \dot{\in} Ws, C > D \rbag_{(C,D) \in Ds} : 2$ from Figure 6 playing the role of the active pattern. The key components of bootstrapping are highlighted in boxes: Task (i) identifies $p_2(Y, C, D)$ as the active pattern, treating it as an atom. The match for atom partners proceeds as in the previous case (Section 5.1) with the difference that the comprehension guards of partner 2 ($D \dot{\in} Ws, C > D$) are included in the guard pool. This allows us to schedule them early ($C > D$ in Task (ii) and $D \dot{\in} Ws$ in Task (iv)) or even as part of an indexing directive to identify compatible partner atom constraints that support the current partial match. Once all atomic partners are matched, at Task (vi), `Bootstrap` $\{C, D\}$ 5, clears the bindings imposed by the active constraint, while the rest of the join ordering executes the actual matching of the comprehension head constraint similarly to Figure 6.

5.3 Uniqueness Enforcement

In general, a CHR^{cp} rule r may have overlapping head constraints, i.e., there may be a store constraint $A\#n$ that matches both H_j and H_k in r's head. Matching two head constraints to the same object in the store is not valid in CHR^{cp}. We guard against

$r \mathbin{@} p(D_0) : 1,\ q(P) : 2,\ \wr p(D_1) \mid D_1 > P \int_{D_1 \in Xs} : 3,\ \wr p(D_2) \mid D_2 \leq P \int_{D_2 \in Ys} : 4 \iff \ldots$

i. `Active` $p(D_0) : 1$
ii. `LookupAtom` $\langle true; \emptyset \rangle$ $q(P) : 2$
iii. `LookupAll` $\langle D_1 > P; \varnothing \rangle$ $p(D_1) : 3$
iv. `FilterHead` 3 1
v. `CompreDomain` 3 D_1 Xs
vi. `LookupAll` $\langle D_2 \leq P; \varnothing \rangle$ $p(D_2) : 4$
vii. `FilterHead` 4 1
viii. `FilterHead` 4 3
ix. `CompreDomain` 4 D_2 Ys

Fig. 8. Uniqueness Checks: Optimized Join Ordering for $p(D_0) : 1$

this by providing two uniqueness enforcing join tasks: If H_j and H_k are atomic head constraints, join task `NeqHead` j k (figure 9) checks that constraints $A\#m$ and $A\#p$ matching H_j and H_k respectively are distinct (i.e., $m \neq p$). If either H_j or H_k (or both) is a comprehension, the join ordering must include a `FilterHead` join task.

Figure 8 illustrates filtering for active pattern $p(D_0) : 1$. Task (iv) `FilterHead` 3 1 states that we must filter constraint(s) matched by rule head 1 away from constraints matched by partner 3. For partner 4, we must filter from 1 and 3 (Tasks $(vii - viii)$). Notice that partner 2 does not participate in any such filtering, since its constraint has a different predicate symbol and filtering is obviously not required. However, it is less obvious that task $(viii)$, highlighted, is in fact not required as well: because of the comprehension guards $D_1 > P$ and $D_2 \leq P$, partners 3 and 4 always match distinct sets of p constraints. Our implementation uses a more precise check for non-unifiability of head constraints ($\sqsubseteq_{\mathbf{unf}}$) to determine when uniqueness enforcement is required.

6 Building Join Orderings

In this section, we formalize join orderings for CHR^{cp}, as illustrated in the previous section. We first construct a valid join ordering for a CHR^{cp} rule r given a chosen sequencing of partners of r and later discuss how this sequence of partners is chosen. Figure 9 defines the elements of join orderings, join tasks and indexing directives. A list of join tasks $\vec{J}$ forms a join ordering. A join context Σ is a set of variables. Atomic guards are as in Figure 1, however we omit equality guards and assume that equality constraints are enforced as non-linear variable patterns in the head constraints. For simplicity, we assume that conjunctions of guards $g_1 \wedge g_2$ are unrolled into a multiset of guards $\bar{g} = \wr g_1, g_2 \int$, with $\models \bar{g}$ expressing the satisfiability of each guard in $\bar{g}$. An indexing directive is a tuple $\langle g; \vec{x} \rangle$ such that g is an indexing guard and $\vec{x}$ are hash variables. The bottom part of Figure 9 defines how valid index directives are constructed. The relation $\Sigma; A \rhd t \mapsto x$ states that from the join context Σ, term t connects to atomic constraint A via variable x. Term t must be either a constant or a variable that appears in Σ and $x \in FV(A)$. The operation $idxDir(\Sigma, A, g)$ returns a valid index directive for a given constraint A, the join context Σ and the atomic guard g. This operation requires that Σ be the set of all variables that have appeared in a prefix of a join ordering. It is defined as follows: If g is an instance of an order relation and it acts as a connection between Σ and A (i.e., $\Sigma; A \rhd t_i \mapsto t_j$ where t_i and t_j are its arguments), then the operation returns g as part of the index directive, together with the set of variables that appear in both Σ and A. If g is a membership relation $t_1 \mathrel{\dot{\in}} t_2$, the operation returns g only if $\Sigma; A \rhd t_2 \mapsto t_1$. Otherwise, g cannot be used as an index, hence the operation

$$
\begin{array}{llll}
\text{Join Context } \Sigma & ::= & \vec{x} & \text{Index Directive } \quad I \quad ::= \quad \langle g; \vec{x} \rangle \\
\text{Join Task } \quad J & ::= & \texttt{Active}\ H \mid \texttt{LookupAtom}\ I\ H \mid \texttt{LookupAll}\ I\ H & \\
& \mid & \texttt{Bootstrap}\ \vec{x}\ i \mid \texttt{CheckGuard}\ \bar{g} \mid \texttt{FilterGuard}\ i\ \bar{g} & \\
& \mid & \texttt{NeqHead}\ i\ i \mid \texttt{FilterHead}\ i\ i \mid \texttt{CompreDomain}\ i\ \vec{x}\ x &
\end{array}
$$

$\Sigma; A \rhd t \mapsto x$ iff t is a constant or t is a variable such that $t \in \Sigma$ and $x \in FV(A)$

$$
idxDir(\Sigma, A, g) \quad ::= \quad \begin{cases} \langle g; \Sigma \cap FV(A) \rangle & \begin{cases} \text{if } g = t_1\ op\ t_2 \text{ and } op \in \{\leq, <, \geq, >\} \\ \text{and } \Sigma; A \rhd t_i \mapsto t_j \text{ for } \{i, j\} = \{1, 2\} \end{cases} \\ \langle g; \Sigma \cap FV(A) \rangle & \text{if } g = t_1 \ \dot{\in}\ t_2 \text{ and } \Sigma; A \rhd t_2 \mapsto t_1 \\ \langle true; \Sigma \cap FV(A) \rangle & \text{otherwise} \end{cases}
$$

$$
allIdxDirs(\Sigma, A, \bar{g}) \quad ::= \quad \wr idxDir(\Sigma, A, g) \mid \text{for all } g \in \bar{g} \cup true \int
$$

Fig. 9. Join Tasks and Indexing Directives

returns $true$. Finally, $allIdxDirs(\Sigma, A, \bar{g})$ defines the set of all such indexing derivable from $idxDir(\Sigma, A, g)$ where $g \in \bar{g}$.

An indexing directive $\langle g; \vec{x} \rangle$ for a constraint pattern $p(\vec{t})$ determines what type of indexing method can be exploited for the given constraint type. For example, $\langle true; \vec{x} \rangle$ where $\vec{x} \neq \emptyset$ states that we can store constraints $p(\vec{t})$ in a hash multimap that indexes the constraints on argument positions of $\vec{t}$ where variables $\vec{x}$ appear, supporting amortized $O(1)$ lookups. For $\langle x \ \dot{\in}\ ts; \vec{x} \rangle$, we store $p(\vec{t})$ in the same manner, but during lookup we enumerate the values of x from ts, hence we get amortized $O(m)$ lookups, where m is size of ts. Directive $\langle x\ op\ y; \emptyset \rangle$ specifies binary tree storage and binary search lookups, while $\langle x\ op\ y; \vec{x} \rangle$ specifies a composite structure: a hash map with binary trees as contents. The default indexing directive is $\langle true; \emptyset \rangle$, that corresponds to a linear iteration lookup on $p(\vec{t})$. For full details, refer to [8].

Figure 10 defines the operation $compileRuleHead(H_i, \vec{H}_a, \vec{H}_m, \bar{g})$ which compiles an active pattern H_i, a particular sequencing of partners, and rule guards of a CHR^{cp} rule (i.e., $r \ @\ \wr \vec{H}_a, \vec{H}_m, H_i \int \iff \bar{g} \mid \bar{B}$) into a *valid* join ordering for this sequence. A join-ordering $\vec{J}$ is valid w.r.t. to a CHR rule r if and only if it possesses certain well-formedness properties (See [8] for details of these properties) that allows for its sound execution of the abstract matching machine (Section 7). The topmost definition of $compileRuleHead$ in Figure 10 defines the case for H_i being an atomic constraint, while the second definition handles the case for a comprehension. The auxiliary operation $buildJoin(\vec{H}, \Sigma, \bar{g}, \vec{H}_h)$ iteratively builds a list of join tasks from a list of head constraints $\vec{H}$, the join context Σ and a multiset of guards $\bar{g}$, the *guard pool*, with a list of head constraints $\vec{H}_h$, the *prefix head constraints*. The join context contains the variables that appear in the prefix head constraints, while the guard pool contains guards g that are available for either scheduling as tests or as indexing guards. The prefix head constraints contain the list of atomic constraint patterns observed thus far in the computation. If the head of $\vec{H}$ is atomic $A : j$, the join ordering is constructed as follows: the subset $\bar{g}_1$ of $\bar{g}$ that are grounded by Σ are scheduled at the front of the ordering ($\texttt{CheckGuard}\ \bar{g}_1$). This subset is computed by the operation $scheduleGrds(\Sigma, \bar{g})$ which returns the partition of $\bar{g}$ such that $\bar{g}_1$ contains guards grounded by Σ and $\bar{g}_2$ contains all other guards. This is followed by the lookup join task for atom $A : j$ (i.e., $\texttt{LookupAtom}\ \langle g_i; \vec{x} \rangle\ A : j$) and uniqueness enforcement join tasks $neqHs(A : j, \vec{H}_h)$

$$
\begin{array}{l}
\mathit{compileRuleHead}(A:i,\vec{H}_a,\vec{H}_m,\bar{g}) \quad ::= \quad \texttt{[Active}\ A:i \mid J_a\texttt{]}\texttt{++}J_m\texttt{++}\mathit{checkGrds}(\bar{g}'') \\
\text{where } (J_a,\Sigma,\bar{g}') \ = \ \mathit{buildJoin}(\vec{H}_a,FV(A_i),\bar{g},\texttt{[]}) \text{ and } (J_m,\Sigma',\bar{g}'') \ = \ \mathit{buildJoin}(\vec{H}_m,\Sigma,\bar{g}',\vec{H}_a) \\
\mathit{compileRuleHead}(\wr A \mid \bar{g}_m\int_{\vec{x}\in xs}:i,\vec{H}_a,\vec{H}_m,\bar{g}) \\
\qquad ::= \quad \texttt{[Active}\ A:i \mid J_a\texttt{]}\texttt{++[Bootstrap}\ FV(A)-FV(\vec{x})\ i \mid J_m\texttt{]}\texttt{++}\mathit{checkGrds}(\bar{g}'') \\
\qquad \text{where } (J_a,\Sigma,\bar{g}') \quad = \mathit{buildJoin}(\vec{H}_a,FV(A_i),\bar{g}\cup\bar{g}_m,\texttt{[]}) \\
\qquad\qquad (J_m,\Sigma',\bar{g}'') = \mathit{buildJoin}(\texttt{[}\wr A_i \mid \bar{g}_m\int_{\vec{x}\in xs} \mid \vec{H}_m\texttt{]},\Sigma-\vec{x},\bar{g}',\vec{H}_a) \\
\mathit{buildJoin}(\texttt{[}A:j \mid \vec{H}\texttt{]},\Sigma,\bar{g},\vec{H}_h) \\
\qquad ::= \quad (\texttt{[CheckGuard}\ \bar{g}_1,\texttt{LookupAtom}\ \langle g_i;\vec{x}\rangle\ A:j\texttt{]}\texttt{++}\mathit{neqHs}(A:j,\vec{H}_h)\texttt{++}\vec{J},\Sigma,\bar{g}_r) \\
\qquad \text{where } (\bar{g}_1,\bar{g}_2) = \mathit{scheduleGrds}(\Sigma,\bar{g}) \text{ and } \langle g_i;\vec{x}\rangle \in \mathit{allIdxDirs}(\Sigma,A,\bar{g}_2) \\
\qquad\qquad (\vec{J},\Sigma',\bar{g}_r) = \mathit{buildJoin}(\vec{H},\Sigma\cup FV(A),\bar{g}_2-g_i,\vec{H}_h\texttt{++[}A:j\texttt{]}) \\
\mathit{buildJoin}(\texttt{[}\wr A \mid \bar{g}_m\int_{\vec{x}\in xs}:j \mid \vec{H}\texttt{]},\Sigma,\bar{g},\vec{H}_h) \\
\qquad := \quad (\texttt{[CheckGuard}\ \bar{g}_1,\texttt{LookupAll}\ \langle g_i;\vec{x}'\rangle\ A:j,\texttt{FilterGuard}\ (\bar{g}_m-\{g_i\})\texttt{]} \\
\qquad\quad \texttt{++}\mathit{filterHs}(\wr A \mid \bar{g}_m\int_{\vec{x}\in xs}:j,\vec{H}_h)\texttt{++[CompreDomain}\ j\ \vec{x}\ xs \mid \vec{J}\texttt{]},\Sigma,\bar{g}_r) \\
\qquad \text{where } (\bar{g}_1,\bar{g}_2) = \mathit{scheduleGrds}(\Sigma,\bar{g}) \text{ and } \langle g_i;\vec{x}'\rangle \in \mathit{allIdxDirs}(\Sigma,A,\bar{g}_2\cup\bar{g}_m) \\
\qquad\qquad (\vec{J},\Sigma',\bar{g}_r) = \mathit{buildJoin}(\vec{H},\Sigma\cup FV(A),\bar{g}_2-g_i,\vec{H}_h\texttt{++[}\wr A \mid \bar{g}_m\int_{\vec{x}\in xs}:j\texttt{]}) \\
\mathit{buildJoin}(\texttt{[]},\Sigma,\bar{g},\vec{H}_h) \quad ::= \quad (\texttt{[]},\Sigma,\bar{g}) \\
\mathit{scheduleGrds}(\Sigma,\bar{g}) \quad ::= \quad (\{g \mid g\in\bar{g},FV(g)\subseteq\Sigma\},\{g \mid g\in\bar{g},FV(g)\not\subseteq\Sigma\}) \\
\mathit{neqHs}(p(\_):j,p'(\_):k) \quad ::= \quad \text{if } p=p' \text{ then } \texttt{[NeqHead}\ j\ k\texttt{]} \text{ else } \texttt{[]} \\
\mathit{filterHs}(C:j,C':k) \quad ::= \quad \text{if } \mathit{true}\rhd C' \sqsubseteq_{\mathbf{unf}} C \text{ then } \texttt{[FilterHead}\ j\ k\texttt{]} \text{ else } \texttt{[]}
\end{array}
$$

Fig. 10. Building Join Ordering from CHR^{cp} Head Constraints

which returns a join tasks `NeqHead` j k for each occurrence in $\vec{H}_h$ that has the same predicate symbol as A. The rest of the join ordering $\vec{J}$ is computed from the tail of $\vec{H}$. Note that the operation picks *one* indexing directive $\langle g_i;\vec{x}\rangle$ from the set of all available indexing directives ($\mathit{allIdxDirs}(\Sigma,A,\bar{g}_2)$). Hence from a given sequence of partners, $\mathit{compileRuleHead}$ defines a family of join orderings for the same inputs, modulo indexing directives. If the head of $\vec{H}$ is a comprehension, the join ordering is constructed similarly, with the following differences: 1) a `LookupAll` join tasks in created in the place of `LookupAtom`; 2) the comprehension guards $\bar{g}_m$ are included as possible indexing guards ($\mathit{allIdxDirs}(\Sigma,A,\bar{g}_2\cup\bar{g}_m)$); 3) immediately after the lookup join task, we schedule the remaining of comprehension guards as filtering guards (i.e., `FilterGuard` $\bar{g}_m-g_i$); 4) `FilterHead` uniqueness enforcement join tasks are deployed ($\mathit{filterHs}(C:j,C':k)$) as described in Section 5.3; 5) We conclude the comprehension partner with `CompreDomain` $\vec{x}$ xs.

We briefly highlight the heuristic scoring function we have implemented to determine an optimized join ordering for each rule occurrence H_i of a CHR^{cp} program (refer to [8] for more details). This heuristic augments [6] to handle comprehensions. While we do not claim that such heuristics always produce optimal join-orderings, in practice it produces join-orderings that perform generally better than arbitrary ordering (see Section 9). Given a join ordering, we calculate a numeric score for the cost of executing $\vec{J}$: a weighted sum value $(n-1)w_1+(n-2)w_2+...+w_n$ for a join ordering with n partners, such that w_j is the join cost of the j^{th} partner H_j. Since earlier partners have higher weight, this scoring rewards join orderings with the least expensive partners scheduled earlier. The join cost w_j for a partner constraint $C:j$ is a tuple (v_f,v_l) where v_f is the *degree of freedom* and v_l is the *indexing score*. The

$$\begin{array}{ll} \text{Matching Context } \Theta & ::= \langle A\#n; \vec{J}; Ls\rangle \\ \text{Matching State} & \mathcal{M} ::= \langle J; pc; \vec{Br}; \theta; Pm\rangle \end{array} \qquad \begin{array}{ll} \text{Backtrack Branch } Br & ::= (pc, \theta, Pm) \\ \text{Candidate Match } U & ::= (\theta, A\#n) \\ \text{Partial Match} & Pm ::= Pm, i \mapsto \bar{U} \mid \emptyset \end{array}$$

$match(A, A') ::=$ if exists ϕ such that $\phi A = A'$ then ϕ else $\bot$

$lookupCands(Ls, A', \langle g; \vec{x}'\rangle) ::= \wr (\phi, A\#n) \mid$ for all $A\#n \in Ls$ s.t. $match(A, A') = \phi$ and $\phi \neq \bot$ and $\models g\int$

Fig. 11. *LHS* Matching States and Auxiliary Operations

degree of freedom v_f counts the number of new variables introduced by C, while the indexing score v_l is the negative of the number of common variables between C and all other partners matched before it. In general, we want to minimize v_f since a higher value indicates larger numbers of candidates matching C, hence larger branching factor for `LookupAtom` join tasks, and larger comprehension multisets for `LookupAll` join tasks. Our heuristics also accounts for indexing guards and early scheduled guards: a lookup join tasks for $C : j$ receives a bonus modifier to w_j if it utilizes an indexing directive $\langle g_\alpha; \_\rangle$ where $g_\alpha \neq true$ and for each guard (`CheckGuard` g) scheduled immediately after it. This rewards join orderings that heavily utilizes indexing guards and schedules guards earlier. The filtering guards of comprehensions (`FilterGuard`) are treated as penalties instead, since they do not prune the search tree.

For each rule occurrence H_i and partner atomic constraints and comprehensions $\bar{H}_a$ and $\bar{H}_c$ and guards $\bar{g}$, we compute join orderings from all permutations of sequences of $\bar{H}_a$ and $\bar{H}_c$. For each such join ordering, we compute the weighted sum score and select an optimized ordering based on this heuristic. Since CHR^{cp} rules typically contain a small number of constraints, join ordering permutations can be practically computed.

7 Executing Join Orderings

In this section, we define the execution of join orderings by means of an abstract state machine. The CHR^{cp} *abstract matching machine* takes an active constraint $A\#n$, the constraint store Ls and a valid join ordering $\vec{J}$ for a CHR^{cp} rule r, and computes an instance of a head constraint match for r in Ls.

Figure 11 defines the elements of this abstract machine. The inputs of the machine are the *matching context* Θ, $A\#n$, a join ordering $\vec{J}$ and the constraint store Ls. A *matching state* $\mathcal{M}$ is a tuple consisting of the current join task J, a program counter pc, a list of backtracking branches $\vec{Br}$, the current substitution θ and the current partial match Pm. A partial match is a map from occurrence indices i to multisets of candidates U, which are tuples $(\theta, A\#n)$. We denote the empty map as $\varnothing$ and the extension of Pm with $i \mapsto U$ as $(Pm, i \mapsto U)$. We extend the list indexing notation $Pm[j]$ to retrieve the candidates that Pm maps j to. We define two auxiliary meta-operations: $match(A, A')$ returns a substitution ϕ such that $\phi A = A'$ if it exists and $\bot$ otherwise; $lookupCands(Ls, A', \langle g; \vec{x}\rangle)$ retrieves the multiset of candidates $A\#n$ in store Ls that match pattern A' and satisfy g for indexing directive $\langle g; \vec{x}\rangle$.

Given an execution context $\Theta = \langle A\#n; \vec{J}; Ls\rangle$, the state transition operation, denoted $\Theta \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, defines a transition step of this abstract machine. Figure 12 defines its transition rules: rule $(active)$ executes `Active` $A' : i$ by matching the active constraint $A\#n$ with A' ($\phi = match(A, \theta A')$). If this match is successful ($\phi \neq$

$(active)$	$\Theta \rhd \langle \texttt{Active}\ A' : i; pc; Br; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta\phi; Pm, i \mapsto (\phi, A\#n)\rangle$ if $\phi = match(A, \theta A')$ and $\phi \neq \bot$
$(lookup{-}atom)$	$\Theta \rhd \langle \texttt{LookupAtom}\ \langle g; \vec{x}\rangle\ A' : j; pc; Br; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br'{+}{+}Br; \theta\phi; Pm, j \mapsto (\phi, A''\#m)\rangle$ if $\wr \bar{U}, (\phi, A''\#m) \int = lookupCands(Ls, \theta A', \langle \theta g; \vec{x}\rangle)$ $Br' = \wr (pc, \theta\phi, Pm, j \mapsto (\phi, A''\#m)) \mid$ for all $(\phi, A''\#m) \in \bar{U} \int$
$(check{-}guard)$	$\Theta \rhd \langle \texttt{CheckGuard}\ \bar{g}; pc; Br; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta; Pm\rangle$ if $\models \theta\bar{g}$
$(lookup{-}all)$	$\Theta \rhd \langle \texttt{LookupAll}\ \langle g; \vec{x}\rangle\ A' : j; pc; Br; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta; Pm, j \mapsto \bar{U}\rangle$ where $\bar{U} = lookupCands(Ls, \theta A', \langle \theta g; \vec{x}\rangle)$
$(filter{-}guard)$	$\Theta \rhd \langle \texttt{FilterGuard}\ j\ \bar{g}; pc; Br; \theta; Pm, j \mapsto \bar{U}\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta; Pm, j \mapsto \bar{U}'\rangle$ where $\bar{U}' = \wr (\phi', C) \mid$ for all $(\phi', C) \in \bar{U}$ s.t. $\models \theta\phi'\bar{g} \int$
$(neq{-}head)$	$\Theta \rhd \langle \texttt{NeqHead}\ j\ k; pc; Br; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta; Pm\rangle$ if $Pm\,\texttt{[}j\texttt{]} = (\_, A'\#m)$ and $Pm\,\texttt{[}k\texttt{]} = (\_, A'\#n)$ such that $m \neq n$
$(filter{-}head)$	$\Theta \rhd \langle \texttt{FilterHead}\ j\ k; pc; Br; \theta; Pm, j \mapsto \bar{U}, k \mapsto \bar{U}'\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta; Pm, j \mapsto \bar{U}'', k \mapsto \bar{U}'\rangle$ where $\bar{U}'' = \wr (\phi, A''\#m) \mid$ for all $(\phi, A''\#m) \in \bar{U}$ s.t. $\neg\exists(\_, A''\#m) \in \bar{U}' \int$
$(compre{-}dom)$	$\Theta \rhd \langle \texttt{CompreDomain}\ j\ \vec{x}\ xs; pc; Br; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta\phi; Pm\rangle$ where $Pm\,\texttt{[}j\texttt{]}$ and $\phi = [\wr \phi'\vec{x} \mid$ for all $(\phi', \_) \in \bar{U} \int / xs]$
$(bootstrap)$	$\Theta \rhd \langle \texttt{Bootstrap}\ \vec{x}\ j; pc; Br; \theta[\_/\vec{x}]; Pm, j \mapsto \_\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc\texttt{]}; pc{+}1; Br; \theta; Pm\rangle$
$(backtrack)$	$\Theta \rhd \langle \_; pc; \texttt{[}(pc', \theta', Pm')\ \texttt{|}\ Br\texttt{]}; \theta; Pm\rangle \rightarrowtail_{lhs} \langle \vec{J}\,\texttt{[}pc'\texttt{]}; pc'{+}1; Br; \theta'; Pm'\rangle$ if neither $(lookup{-}atom)$, $(check{-}guard)$ nor $(neq{-}head)$ applies.
$(fail{-}match)$	$\Theta \rhd \langle \_; pc; \varnothing; \theta; Pm\rangle \rightarrowtail_{lhs} \bot$ if neither $(active)$, $(lookup{-}atom)$, $(check{-}guard)$, $(neq{-}head)$ nor $(backtrack)$ applies.

Fig. 12. Execution of CHR^{cp} Join Ordering

$\bot$), the search proceeds. Rule $(lookup{-}atom)$ executes $\texttt{LookupAtom}\ \langle g; \vec{x}'\rangle\ A' : j$ by retrieving ($lookupCands(Ls, \theta A, \langle \theta g; \vec{x}\rangle)$) constraints in Ls that match $A' : j$. If there is at least one such candidate $(\phi, A''\#m)$, the search proceeds with it as the match to partner j and all other candidates as possible backtracking branches (Br'). This is the only type of join task where the search branches. Rule $(check{-}guard)$ executes $\texttt{CheckGuard}\ \bar{g}$ by continuing the search only if all guards $\bar{g}$ are satisfiable under the current substitution ($\models \theta\bar{g}$). Rule $(lookup{-}all)$ defines the case for $\texttt{LookupAll}\ \langle g; \vec{x}\rangle\ A' : j$, during which candidates matching A' are retrieved ($\bar{U} = lookupCands(Ls, \theta A, \langle \theta g; \vec{x}\rangle)$). But rather than branching, the search proceeds by extending the partial match with all candidates (i.e., $j \mapsto \bar{U}$). Rule $(filter{-}guard)$ defines the case for $\texttt{FilterGuard}\ j\ \bar{g}$, in which the search proceeds by filtering from $Pm\,\texttt{[}j\texttt{]}$ candidates that do not satisfy the guard conditions $\bar{g}$. Rule $(neq{-}head)$ defines the case for $\texttt{NeqHead}\ j\ k$: if $Pm\,\texttt{[}j\texttt{]}$ and $Pm\,\texttt{[}k\texttt{]}$ maps to unique constraints, the search proceeds. Rule $(filter{-}head)$ executes $\texttt{FilterHead}\ j\ k$ by filtering from $Pm\,\texttt{[}j\texttt{]}$ any candidates that appear also in $Pm\,\texttt{[}k\texttt{]}$. Rule $(compre{-}dom)$ executes $\texttt{CompreDomain}\ j\ \vec{x}\ xs$ by extending the current substitution θ with $\phi = [ps/xs]$ where ps is the multiset of projections of $\vec{x}$ extracted from each candidate of $Pm\,\texttt{[}j\texttt{]}$. Rule $(bootstrap)$ executes $\texttt{Bootstrap}\ \vec{x}\ j$ by removing mappings of j from cur-

rent partial matches and mappings of $\vec{x}$ from the current substitution. Rule $(backtrack)$ backtracks when rules $(lookup{-}atom)$, $(check{-}guard)$ and $(neq{-}head)$ are not applicable. Backtracking is achieved by accessing the head of the backtracking branches (pc', θ', Pm'), and restoring the execution state to that particular state: the current join task becomes $\vec{J}\,[\,pc'\,]$, the program counter $pc' + 1$, the current substitution θ' and the partial matches Pm'. If there are no more backtracking options, rule $(fail{-}match)$ declares failure to find a match. Execution of this machine implicitly terminates when pc reaches an index outside the join ordering (i.e., $\vec{J}\,[\,pc\,] = \bot$).

8 Correctness of CHR^{cp} Abstract Matching Machine

In this section, we highlight the correctness results of the CHR^{cp} abstract matching machine. Specifically, we show that our abstract machine always terminates for a valid matching context $\langle A\#n; \vec{J}; Ls\rangle$. By valid, we mean that Ls is finite, that $A\#n \in Ls$, and that $\vec{J}$ is a join ordering constructed by $compileRuleHead$. We also show that it produces sound results w.r.t. the CHR^{cp} operational semantics. Finally, we show that it is complete for a class of CHR^{cp} rules that are not *selective* on comprehension patterns. We assume that matching ($match(A, A')$) and guard satisfiability tests ($\models g$) are decidable procedures. Proofs and details for these results can be found in [8].

We denote the exhaustive transition of the CHR^{cp} abstract matching machine as $\Theta \rhd \mathcal{M} \rightarrowtail^*_{lhs} \mathcal{M}'$. There, $\mathcal{M}'$ is a *terminal* state of the form $\langle \bot; \_; \_; \_; \_\rangle$: $\bot$ since the program counter has gone past the last index of $\vec{J}$. An *initial* state has the form $\langle \vec{J}\,[\,0\,]; 1; \varnothing; \cdot; \varnothing\rangle$. For our CHR^{cp} abstract matching machine to be effective, we need some guarantees that if we run it on a valid join ordering $\vec{J}$ and a finite constraint store Ls, execution either terminates at some terminal state (i.e., $\langle \bot; \_; \_; \_; \_\rangle$), or returns $\bot$.

Theorem 1 (Termination of the CHR^{cp} Abstract Matching Machine). *For any valid $\Theta = \langle A\#n; \vec{J}; Ls\rangle$, we have $\Theta \rhd \langle \vec{J}\,[\,0\,]; 1; \varnothing; \cdot; \varnothing\rangle \rightarrowtail^*_{lhs} \mathcal{M}$ such that either $\mathcal{M} = \langle \bot; \_; \_; \theta; Pm\rangle$ or $\mathcal{M} = \bot$.*

The CHR^{cp} abstract matching machine is also sound w.r.t. the semantics of matching of CHR^{cp}: in the final state of a valid execution, θ and Pm corresponds to head constraint match as specified by the semantics of matching of CHR^{cp} (Figure 2). The operation $constr(Pm, i)$ returns the multiset of all constraints in partial match Pm mapped by i.

Theorem 2 (Soundness of the CHR^{cp} Abstract Matching Machine). *For any CHR^{cp} head constraints $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$, such that $\vec{J} = compileRuleHead(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$, given a constraint store Ls and an active constraint $A\#n$, if $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle \vec{J}\,[\,0\,]; 1; \varnothing; \cdot; \varnothing\rangle \rightarrowtail^*_{lhs} \langle \_; \_; \_; \theta; Pm\rangle$, then for some Ls_{act}, Ls_{part}, Ls_{rest} such that $Ls = \wr Ls_{act}, Ls_{part}, Ls_{rest}\int$ and $Ls_{act} = constr(Pm, i)$ and $Ls_{part} = constr(Pm, getIdx(\wr \vec{H}_a, \vec{H}_m\int))$, we have 1) $\models \theta g$, 2) $C : i \triangleq_{\mathbf{lhs}} Ls_{act}$, 3) $\theta\wr \vec{H}_a, \vec{H}_m\int \triangleq_{\mathbf{lhs}} Ls_{part}$, and 4) $\theta\wr \vec{H}_a, \vec{H}_m, C : i\int \triangleq^{\neg}_{\mathbf{lhs}} Ls_{rest}$.*

However, our CHR^{cp} abstract matching machine is not complete in general. Incompleteness stems from the fact that it *greedily* matches comprehension patterns: comprehensions that are scheduled early consume all matching constraints in the store Ls.

Program	Standard rules only			With comprehensions			Code reduction (lines)
Swap	5 preds	7 rules	21 lines	2 preds	1 rule	10 lines	110%
GHS	13 preds	13 rules	47 lines	8 preds	5 rules	35 lines	34%
HQSort	10 preds	15 rules	53 lines	7 preds	5 rules	38 lines	39%

Program	Input Size	Orig	$+OJO$	$+OJO$ $+Bt$	$+OJO$ $+Mono$	$+OJO$ $+Uniq$	All	Speedup
Swap	$(40, 100)$	241 *vs* 290	121 *vs* 104	*vs* 104	*vs* 103	*vs* 92	*vs* 91	33%
	$(200, 500)$	1813 *vs* 2451	714 *vs* 681	*vs* 670	*vs* 685	*vs* 621	*vs* 597	20%
	$(1000, 2500)$	8921 *vs* 10731	3272 *vs* 2810	*vs* 2651	*vs* 2789	*vs* 2554	*vs* 2502	31%
GHS	$(100, 200)$	814 *vs* 1124	452 *vs* 461	*vs* 443	*vs* 458	*vs* 437	*vs* 432	5%
	$(500, 1000)$	7725 *vs* 8122	3188 *vs* 3391	*vs* 3061	*vs* 3290	*vs* 3109	*vs* 3005	6%
	$(2500, 5000)$	54763 *vs* 71650	15528 *vs* 16202	*vs* 15433	*vs* 16097	*vs* 15835	*vs* 15214	2%
HQSort	$(8, 50)$	1275 *vs* 1332	1117 *vs* 1151	*vs* 1099	*vs* 1151	*vs* 1081	*vs* 1013	10%
	$(16, 100)$	5783 *vs* 6211	3054 *vs* 2980	*vs* 2877	*vs* 2916	*vs* 2702	*vs* 2661	15%
	$(32, 150)$	13579 *vs* 14228	9218 *vs* 8745	*vs* 8256	*vs* 8617	*vs* 8107	*vs* 8013	15%

Execution times (ms) for various optimizations on programs with increasing input size.

Fig. 13. Preliminary Experimental Results

Consider a rule r with guard g, a comprehension head constraint $M : i$ and another head constraint $C : j$ with i and j unifiable. If guards g is satisfiable only for some particular partitions of i and j, we call r a *comprehension selective* rule. Our abstract machine will not necessary be able to identify this partitioning: suppose that a join ordering executes j before i, then the join task `FilterHead` i j always forces all constraints that can match either with i or j to be in j. The abstract matching machine is complete for CHR^{cp} rules that are non-selective on comprehensions.

Theorem 3 (Completeness of the CHR^{cp} Abstract Matching Machine). *Let r be any CHR^{cp} rule that is non-selective on comprehension rule heads. Let its head constraints be $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$ with $\vec{J} = compileRuleHead(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$. If $\langle A\#n; \vec{J}; Ls\rangle \triangleright \langle \vec{J}\,[0\,]; 1; \varnothing; \cdot; \varnothing\rangle \rightarrowtail^*_{lhs} \bot$ for a constraint store Ls and an active constraint $A\#n$, then there exists no applicable rule instance of r from Ls.*

9 Prototype and Preliminary Empirical Results

In this section, we report preliminary experimental results of our CHR^{cp} implementation. We have implemented a prototype (available for download at `https://github.com/sllam/chrcp`) that utilizes a source-to-source compilation of CHR^{cp} programs: our compiler is written in Python and translates CHR^{cp} programs into a sequence of join orderings. Then, it generates C++ code that implements multiset rewriting as specified by the operational semantics of CHR^{cp}. To support unifiability analysis for constraint monotonicity (Section 4.1), we have deployed a conservative implementation of the relation test routine $\sqsubseteq_{\mathbf{unf}}$, discussed in [9].

We have conducted preliminary experiments aimed at assessing the performance of standard CHR programs (without comprehension patterns), CHR^{cp} programs with comprehension patterns and also to investigate the effects of the optimizations described in this paper: OJO optimized join ordering (Section 6), Bt bootstrapping of active comprehension head constraints (Section 5.2), $Mono$ incremental storage for monotone constraints (Section 4.1) and $Uniq$ non-unifiability test for uniqueness enforcement

(Section 5.3). When OJO is omitted, join ordering are of arbitrary matching ordering (e.g., textual order). When Bt is omitted, an active comprehension pattern aggressively collects all matching constraints and filters non-matches away in later stages of the join ordering execution. When $Mono$ is omitted, all goals are treated as eager goals, hence eagerly stored and forsaking any opportunity of incremental processing. Finally, when $Uniq$ is omitted, join ordering produced conservatively (exhaustively) include uniqueness enforcement tasks for each pairs of rule head constraints. Optimization OJO is not specific to comprehension patterns: we use it to investigate the performance gains for programs with comprehension patterns relative to standard CHR variants. All other optimizations are specific to comprehension patterns, and hence we do not anticipate any performance gains for standard CHR programs. We have analyzed performance on three CHR^{cp} programs of varying sizes (refer to [8] for codes): *swap* is the swapping data example (Section 2) with input size (s, d) where s is number of swaps and d is number of data constraints. ***GHS*** is a simulation of the GHS distributed minimal spanning tree algorithm with input sizes (v, e) where v is number of vertices and e is number of edges. Finally, ***HQSort*** is a simulation of the hyper-quicksort algorithm with input sizes (n, i) where n is number of nodes and i number of integers in each node.

Figure 13 displays our experimental results. All experiments were conducted on an Intel $i7$ quad-core processor with 2.20 GHz CPUs and 4 Gb of memory. All execution times are averages from ten runs of the same experiments. The column $Orig$ contains results for runs with all optimizations turned off, while All contains results with all optimizations. In between, we have results for runs with optimized join ordering and at least one optimization specific to comprehension patterns. For $Orig$ and $+OJO$, we show two values, n *vs* m, where n is the execution time for the program implemented with standard rules and m for code using comprehension patterns. Relative gains demonstrated in $Orig$ and $+OJO$ comes at no surprise: join ordering and indexing benefit both forms of programs. For the $Swap$ example, optimization $+Uniq$ yields the largest gains, with $+Bt$ for GHS. $+Mono$ yields the least gains across the board and we believe that this is because, for programs in this benchmark, constraints exclusively appear as atomic constraint patterns or in comprehension patterns. The last column shows the speedup of the CHR^{cp} code with all optimizations turned on w.r.t. the standard CHR code with join ordering. Our experiments, although preliminary, show very promising results: comprehensions not only provide a common abstraction by reducing code size, but, maybe more surprisingly, we get significant performance gains over CHR.

10 Related Work

Compilation optimization for CHR has received a lot of attention. Efficient implementations are available in Prolog, HAL [6], Java [13] and even in hardware (via FPGA) [12]. Join-ordering in pure CHR are extensively studied in [4,6]. The multiset matching technique implemented in these systems are based on the LEAPS algorithm [1]. Our work implements a variant of this algorithm, augmented to handle matching of comprehension patterns. These systems utilize optimization techniques (e.g., join ordering, index selection) that resemble query optimization in databases. The main difference is that in the multiset rewriting context we are interested in finding *one* match, while relational queries return *all* matches. Two related extensions to CHR have been proposed: negated head constraints allows encoding of a class of comprehension

patterns[14], while an extension that allows computation of limited form of aggregates is discussed in [11]. Like the present work, both extensions introduce non-monotonicity into the semantics. By contrast, we directly address the issue of incrementally processing of constraints in the presence of non-monotonicity introduced by comprehension patterns. The logic programming language LM (Linear Meld) [2] offers features like aggregates and comprehension patterns, that are very similar to our work here. By contrast, comprehension patterns discussed here are more generalized: aggregates in LM can be expressed in CHR^{cp} as term-level comprehension and reduce operations.

11 Conclusion and Future Works

In this paper, we introduced CHR^{cp}, an extension of CHR with multiset comprehension patterns. We highlighted an operational semantics for CHR^{cp}, followed by a lower-level compilation scheme into join orderings. We defined an abstract machine that executes these join orderings, and proved its soundness with respect to the operational semantics. We have implemented a prototype CHR^{cp} system and have demonstrated promising results in preliminary experimentation. In future work, we intend to further develop our prototype implementation of CHR^{cp} by investigating the possibility of adapting other orthogonal optimization techniques found in [6,13,12]. Next, we intend to expand on our empirical results, testing our prototype with a larger benchmark and also testing its performance against other programming frameworks. We also intend to extend CHR^{cp} with some result form prior work in [10].

References

1. D. Batory. The LEAPS Algorithm. Technical report, University of Texas at Austin, 1994.
2. F. Cruz, R. Rocha, S. Copen Goldstein, and F. Pfenning. A linear logic programming language for concurrent programming over graph structures. *CoRR*, abs/1405.3556, 2014.
3. L. De Koninck, T. Schrijvers, and B. Demoen. User-definable rule priorities for chr. In *PPDP'07*, PPDP '07, pages 25–36, New York, NY, USA, 2007. ACM.
4. L. De Koninck and J. Sneyers. Join ordering for constraint handling rules. In *CHR*, 2007.
5. G. J. Duck, P. J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04*, pages 90–104. Springer, 2004.
6. C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of constraint handling rules in HAL. *CoRR*, cs.PL/0408025, 2004.
7. E. S. L. Lam and I. Cervesato. Constraint Handling Rules with Multiset Comprehension Patterns. In *CHR'14*, 2014.
8. E. S. L. Lam and I. Cervesato. Optimized Compilation of Multiset Rewriting with Comprehensions (Full-Version). Technical Report CMU-CS-14-119, Carnegie Mellon, June 2014.
9. E. S. L. Lam and I. Cervesato. Reasoning about Set Comprehension. In *SMT'14*, 2014.
10. E.S.L. Lam and I. Cervesato. Decentralized Execution of Constraint Handling Rules for Ensembles. In *PPDP'13*, pages 205–216, Madrid, Spain, 2013.
11. J. Sneyers, P. V. Weert, T. Schrijvers, and B. Demoen. Aggregates in Constraint Handling Rules. In *ICLP'07*, pages 446–448, 2007.
12. A. Triossi, S. Orlando, A. Raffaetà, and T. W. Frühwirth. Compiling CHR to parallel hardware. In *PPDP'12*, pages 173–184, 2012.
13. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *CHR'05*, pages 47–62, 2005.
14. P. V. Weert, J. Sneyers, T. Schrijvers, and B. Demoen. Extending CHR with Negation as Absence. In *CHR'06*, pages 125–140, 2006.