Optimized Compilation of Multiset Rewriting with Comprehensions

Edmund S. L. Lam and Iliano Cervesato

June 2014
CMU-CS-14-119
CMU-CS-QTR-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Carnegie Mellon University, Qatar campus.
The author can be reached at sllam@qatar.cmu.edu or 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.

* 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.

Keywords: Multiset Rewriting, Logic Programming, Comprehension, Compilation

Contents

1 Introduction 1

2 Motivating Examples 1
- 2.1 Pivoted Swapping 1
- 2.2 Computing Aggregates from Multisets of Constraints 2
- 2.3 Hyper-Quicksort 3
- 2.4 Distributed Minimal Spanning Tree 4

3 Syntax and Notation 5

4 Operational Semantics of CHR^{cp} 6
- 4.1 Semantics of Matching of CHR^{cp} 7
- 4.2 Rule Body Application and Monotonicity 8
- 4.3 Operational Semantics 8

5 Compiling CHR^{cp} Rules 9
- 5.1 Introducing CHR^{cp} Join Ordering 10
- 5.2 Bootstrapping for Active Comprehension Head Constraints 12
- 5.3 Uniqueness Enforcement 12

6 Representing CHR^{cp} Join Orderings 13

7 Building CHR^{cp} Join Orderings 15

8 Executing Join Orderings 16
- 8.1 Abstract Machine Execution 17
- 8.2 Example of Join Ordering Compilation 19

9 Correctness of the CHR^{cp} Abstract Matching Machine 22
- 9.1 Valid Matching Contexts and States 22
- 9.2 Termination 24
- 9.3 Soundness 25
- 9.4 Completeness 26

10 Operational Semantics with Join Ordering Execution 27

11 Prototype and Preliminary Empirical Results 28

12 Related Work 30

13 Conclusion and Future Works 30

A Proofs 31

B Experiment Program Code 35
- B.1 Pivot Swap 35

B.2 Distributed Minimal Spanning Tree 36
B.3 Hyper-Quicksort 36

List of Figures

2.1 Hyper-Quicksort . . . 3
2.2 GHS Algorithm (Distributed Minimal Spanning Tree) . . . 4
3.1 Abstract Syntax of CHR^{cp} . . . 5
4.1 Semantics of Matching in CHR^{cp}: $\bar{C} \triangleq_{\mathbf{lhs}} St$ $\quad C \triangleq_{\mathbf{lhs}} St$ . . . 6
4.2 Rule Body Application and Unifiability of Comprehension Patterns . . . 7
4.3 Execution States and Auxiliary Meta-operations . . . 9
4.4 Operational Semantics of CHR^{cp} . . . 10
5.1 Optimal Join Ordering for $p_1(E, Z) : 1$ . . . 11
5.2 Optimal Join Ordering for $\wr p_2(Y, C, D) \mid D \dot{\in} Ws, C > D \int_{(C,D) \in Ds} : 2$ . . . 12
5.3 Uniqueness Checks: Optimal Join Ordering for $p(D_0) : 1$ . . . 13
6.1 Indexing Directives . . . 14
7.1 Building Join Ordering from CHR^{cp} Head Constraints . . . 16
7.2 Measuring Cost of Join Ordering . . . 17
8.1 LHS Matching States and Auxiliary Operations . . . 18
8.2 Execution of CHR^{cp} Join Ordering . . . 19
8.3 Join Ordering Comparison for GHS Algorithm, mrg rule . . . 20
9.1 More Auxiliary Operations . . . 22
9.2 State Progress Ranking Function . . . 25
9.3 Example of Incompleteness of Matching . . . 26
10.1 Operational Semantics of CHR^{cp} with Join Ordering Execution . . . 28
11.1 Preliminary Experimental Results . . . 29
B.1 Pivot Swap . . . 36
B.2 GHS Algorithm (Distributed Minimal Spanning Tree) . . . 37
B.3 Hyper-Quicksort with Comprehensions . . . 38
B.4 Hyper-Quicksort with Standard Rules . . . 39

1 Introduction

CHR is a logic constraint programming language based on 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* [2, 9]. 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}.

In previous work [6], 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* [5], determines an optimal sequence of operations to carry out the matching of constraints and guards. This ordering is optimal in that it utilizes the most effective supported indexing methodologies (e.g., 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 optimal 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 [5] 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 report makes the following contributions:

- We define a scheme that compiles CHR^{cp} rules into optimal 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 report 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}. In Section 5, we highlight examples of our compilation scheme, and Section 6 introduces the target of compilation: Join orderings. Section 7 builds optimal join orderings of CHR^{cp} rules. Section 8 defines the abstract state machine and Section 9 establishes correctness results. Section 10 combines join ordering compilation into the operational semantics. In Section 11 we present preliminary empirical results. Section 12 situates CHR^{cp} in the literature and Section 13 outlines directions of future work.

2 Motivating Examples

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

2.1 Pivoted Swapping

Consider the problem of two agents wanting to swap data based on a pivot value. We express an integer datum D belonging to agent X by the constraint $data(X, D)$. The state of this dynamic system is represented by a multiset of constraints, the constraint store. 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:

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

The swap is triggered by the constraint $\mathit{swap}(X, Y, P)$ in the rule head on the left of $\iff$. All of X's data that are greater than or equal to the pivot P are identified by the comprehension pattern $\wr \mathit{data}(X, D) \mid D \geq P \wr_{D \in Xs}$. Similarly, all of Y's data less than P are identified by $\wr \mathit{data}(Y, D) \mid D < P \wr_{D \in Ys}$. 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 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. To this point, 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:

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

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., grabGE, $\mathit{unrollData}$) 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.

2.2 Computing Aggregates from Multisets of 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:

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

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 $\mathit{edge}(X, Y, W)$. The fact $\mathit{remove}(Gs)$ identifies the group Gs whose outgoing edges are the subject of the removal. The minimum weight W_m is computed by collecting all edges

$$find_median \ @ \ \begin{array}{l} \wr data(X,D) \wr_{D \in Ds} \\ findMedian(X) \end{array} \iff median(X, computeMedian(Ds))$$

$$leader_reduce \ @ \ leaderLinks(G) \iff count(G) \leq 1 \ | \ true$$

$$\begin{array}{l} leader_expand \ @ \\ \quad \begin{array}{l} median(X,M) \\ leaderLinks(X,G) \end{array} \iff \begin{array}{l} (Gl, Gg) = split(G) \\ Gg = \wr _, Z \wr \end{array} \left| \begin{array}{l} \wr partnerLink(Y, W, M, X) \wr_{(Y,W) \in zip(Gl,Gg)} \\ spawnLeaders(X, Z, Gl, Gg, count(Gl)) \end{array} \right. \end{array}$$

$$swap \ @ \ \begin{array}{l} partnerLink(X, Y, M, L) \\ \wr data(X,D) \mid D \geq M \wr_{D \in Xs} \\ \wr data(Y,D) \mid D < M \wr_{D \in Ys} \end{array} \iff \begin{array}{l} spawnCounter(L, 1) \\ \wr data(X,D) \wr_{D \in Ys} \\ \wr data(Y,D) \wr_{D \in Xs} \end{array}$$

$$spawn \ @ \ \begin{array}{l} spawnLeaders(X, Z, Gl, Gg, L) \\ \wr spawnCounter(I) \wr_{I \in Cs} \end{array} \iff count(Cs) = L \left| \begin{array}{l} findMedian(X), \ leaderLinks(X, Gl) \\ findMedian(Z), \ leaderLinks(Z, Gg) \end{array} \right.$$

Figure 2.1: Hyper-Quicksort

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

2.3 Hyper-Quicksort

Figure 2.1 shows an implementation of the distributed sorting algorithm, Hyper-Quicksort, in CHR^{cp}. Given that the constraint $data(X, D)$ represents an integer value D located at node X, the Hyper-Quicksort algorithm sorts values D across all nodes such that given any two nodes X and Y can be *globally ordered*. By globally ordered, we mean that values in X are either strictly all less than equal, or strictly more than values in Y. We assume that initially, there are 2^n nodes where n is an integer. Initially, one node (say X) is arbitrarily chosen as the leader with all nodes in the program in G, represented by the constraint $leaderLinks(X, G)$. It is also accompanied by the constraint $find_median(X)$. The first rule $median$ implements the sub-routine of finding the median of all values within a node X. We rely on the function $computemedian$ to compute the actual median value, while the rule itself defines the values Ds to be included in this median. The rule $leader_reduce$ implements the terminal case when a node X is the leader of a singleton group (i.e., $count(G) == 1$). Note that $count(G)$ returns the size of the collection G. The rule $leader_expand$ implements the recursive case of this algorithm. Given we have leader X of group G (i.e., $leaderLink(X, G)$) and the median value $median(X, M)$, this rule does the following: (1) splitting G into two halves (i.e., Gl and Gg), establish swapping links between unique pairs Y and W, across the groups Gl and Gg. This is implemented and represented by constraint $swapLink(Y, W, M, X)$). Median M and the current leader X are kept as auxiliary data whose purpose is discussed later. (2) arbitrarily selecting a node Z in Gg (line 10, via the function $pickone$), seeds a future procedure that spawns nodes X and Z as new leaders of groups Gl and Gg respectively (represented by $spawnLeaders(X, Z, Gl, Gg, count(Gl))$). The rule $swap$ implements the actual swapping of data between two nodes X and Y. This rule is similar to the comprehension-version of the pivot swap code in Figure B.1 with the exception that it adds an auxiliary counter $spawnCounter(L)$. The final rule $spawn$ implements the spawning of new leaders X and Z of groups Gl and Gg, each of size L ($spawnLeaders(X, Z, Gl, Gg, L)$). This rule stages the spawning of the leaders only after all swaps have been executed, by means of counting $spawnCounter$ constraints. Once execution of this program terminates, $data(X, D)$ constraints are swapped to a configuration that satisfies the required global sorting order of nodes.

$$
\mathit{find} \ @ \ \begin{array}{l} \mathit{level}(X, L) \\ \mathit{findMWOE}(X, \mathit{Is}) \\ \{\!\{ \mathit{edge}(I, O, V) \mid I \dot{\in} \mathit{Is} \}\!\}_{(I,O,V) \in \mathit{Is}} \end{array} \iff \begin{array}{l} \mathit{Es} \neq \varnothing \\ (I_m, O_m, V_m) = \mathcal{R} \min\ (\bot, \bot, \infty)\ \mathit{Es} \\ \mathit{Rs} = \{\!\{ (I, O, V) \mid V_m \neq V \}\!\}_{(I,O,V) \in \mathit{Es}} \end{array}
$$

$$
\left| \begin{array}{l} \mathit{foundMWOE}(X, \mathit{Is}) \\ \{\!\{ \mathit{edge}(I, O, V) \}\!\}_{(I,O,V) \in \mathit{Rs}} \\ \mathit{combine}(O_m, X, L, I_m, O_m, V_m) \end{array} \right.
$$

$$
\mathit{cmb1} \ @ \ \begin{array}{l} \mathit{combine}(X, Y, L, O, I, V) \\ \mathit{combine}(Y, X, L, I, O, V) \\ \mathit{level}(X, L) \\ \mathit{level}(Y, L) \end{array} \iff \begin{array}{l} \mathit{merge}(X, Y, I, O, V) \\ \mathit{level}(X, L + 1) \end{array}
$$

$$
\mathit{cmb2} \ @ \ \begin{array}{l} \mathit{level}(X, L_1) \\ \mathit{combine}(X, Y, L_2, I, O, V) \end{array} \iff L_1 > L_2 \ \left| \begin{array}{l} \mathit{level}(X, L_1) \\ \mathit{merge}(X, Y, I, O, V) \end{array} \right.
$$

$$
\mathit{mrg} \ @ \ \begin{array}{l} \mathit{merge}(X, Y, I_m, O_m, V_m) \\ \mathit{foundMWOE}(X, \mathit{Is}_1) \\ \mathit{foundMWOE}(Y, \mathit{Is}_2) \\ \{\!\{ \mathit{edge}(I, O, V) \mid I \dot{\in} \mathit{Is}_1, O \dot{\in} \mathit{Is}_2 \}\!\}_{(I,O,V) \in \mathit{Es}_1} \\ \{\!\{ \mathit{edge}(I, O, V) \mid I \dot{\in} \mathit{Is}_2, O \dot{\in} \mathit{Is}_1 \}\!\}_{(I,O,V) \in \mathit{Es}_2} \end{array} \iff \begin{array}{l} \mathit{findMWOE}(X, \{\!\{ \mathit{Is}_1, \mathit{Is}_2 \}\!\}) \\ \mathit{forward}(Y, X) \\ \mathit{mstEdge}(I_m, O_m, V_m) \\ \mathit{mstEdge}(O_m, I_m, V_m) \end{array}
$$

$$
\mathit{fwd} \ @ \ \begin{array}{l} \mathit{forward}(O_1, O_2) \\ \mathit{combine}(O_1, X, L, I, O, V) \end{array} \iff \begin{array}{l} \mathit{forward}(O_1, O_2) \\ \mathit{combine}(O_2, X, L, I, O, V) \end{array}
$$

Figure 2.2: GHS Algorithm (Distributed Minimal Spanning Tree)

2.4 Distributed Minimal Spanning Tree

Next, we consider a slightly more engaging CHR^{cp} program which faithfully implements the GHS algorithm, a distributed algorithm to compute a minimal spanning tree [4]. An edge of the graph between nodes I and O of weight V is represented by the constraint $edge(I, O, V)$. We assume an undirected weighted graph where all edges have a unique weight, hence we maintain the invariant that each occurrence of $edge(I, O, V)$ must be accompanied by an occurrence of $edge(O, I, V)$.

We begin with a brief and informal description of the GHS algorithm: the algorithm begins with the undirected graph ($edge(O, I, V)$) fully constructed. Each node X is assigned a level initially set to zero ($level(X, 0)$). Each node X is also the set as the leader of the singleton component that consists of itself, represented by $findMWOE(X, \{\!\{X\}\!\})$.

The algorithm proceeds by having each component X find its minimum weighted outgoing edge (MWOE) that connects it to another component Y. Node X will then send a request to combine with Y ($combine(X, Y, L, O, I, V)$), with O, I and V the edge that links X and Y, and L the level of X. If both X and Y send the combine request to each other and are at the same level, X and Y are combined into a new component, X is (arbitrarily) chosen as the new leader, and the level of the new component is incremented by one. If instead, component X receives a combine request from a component Y with a lower level, X will combine with Y with its level retained. In either case, the edge (MWOE) in which the combine request traveled along is marked as an edge of the minimum spanning tree and the new combined component will repeat the steps above. If the original graph is connected, the algorithm reaches quiescence once we have a single component, during which we have assembled the minimal spanning tree.

Figure B.2 illustrates the CHR^{cp} implementation of this algorithm. The *find* rule implements the task of locating the MWOE of a component. This is triggered by $findMWOE(X, Is)$ where X is the leader of component that consists of locations in *Is*, and results in the sending a combine request over the component's MWOE

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

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

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

Figure 3.1: Abstract Syntax of CHR^{cp}

($combine(O_m, X, I_m, O_m, V_m)$). The MWOE of the component is selected by retrieving the minimal of the edges that originate from nodes in Is. Specifically, from $\llbracket edge(I, O, V) \mid I \,\dot{\in}\, Is \rrbracket_{(I,O,V) \in Is}$, we choose I_m, O_m and V_m such that $(I_m, O_m, V_m) = \mathcal{R} \min\ (\bot, \bot, \infty)\ Es$ and $Rs = \llbracket (I, O, V) \mid V_m \neq V \rrbracket_{(I,O,V) \in Es}$. The rules *cmb1* and *cmb2* each implement one of the combine subroutines of the GHS algorithm: *cmb1* implements the case that component leaders X and Y sent a combine request to each other ($combine(X, Y, L, O, I, V), combine(Y, X, L, I, O, V)$) and has the same level L (note that in this case, I, O and V is guaranteed to be the same on both end, since it is the MWOE). X is arbitrarily chosen as the new leader, its level is incremented by one ($level(L + 1)$) and its merging with location Y's component is initiated ($merge(X, Y, I, O, V)$). Rule *cmb2* implements the case that location X receives a combine request ($combine(X, Y, L, I, O, V)$) from a component that has a lower level, during which X will absorb Y's component into its own component ($merge(X, Y, I, O, V)$). Note that since weights are unique and edges are bidirectional, by sending combine messages along MWOEs, the GHS algorithm guarantees progress (proven in [4]), in that no deadlocking cycles of combine messages will occur.

The rule *mrg* implements the actual merging of Y's component into X's component ($merge(X, Y, I_m, O_m, V_m)$). Note that the locations of each component are matched from $foundMWOE(X, Is_1)$ and $foundMWOE(Y, Is_2)$, while the multiset of all edges that travel between these two components is captured by the two comprehension patterns of the rule (we omitted the comprehension range binding, since they are not used in the rule). This results in the deletion of these non-outgoing edges of the component, in location X as leader of new component ($findMWOE(X, \llbracket Is_1, Is_2 \rrbracket)$) and in a new minimal spanning tree edge ($mstEdge(I_m, O_m, V_m)$ and $mstEdge(O_m, I_m, V_m)$). Prior to the merging of a location Y into X, node Y may still have combine requests from other components no visible to X. Hence, we have the rule *fwd* which implements a forwarding subroutine on Y to X ($forward(Y, X)$) that forwards any combine requests from the previous leader of Y to the new leader of X and Y.

3 Syntax and Notation

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 $\llbracket \bar{o}_1, \bar{o}_2 \rrbracket$ 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 $\llbracket \bar{o}, o \rrbracket$. Multiset comprehension at the meta-level is denoted by $\llbracket o \mid \Phi(o) \rrbracket$, 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 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 ith 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 $\vec{o}_1 {+}{+} \vec{o}_2$ and given $i, j \in range(\vec{o})$, $\vec{o}[i \ldots j]$ denotes the sublist of $\vec{o}$ consisting of just the elements between i (inclusive) and j (exclusive). We abbreviate a singleton list

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

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

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

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

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

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

Figure 4.1: Semantics of Matching in CHR^{cp}: $\bar{C} \triangleq_{\mathbf{lhs}} St \qquad C \triangleq_{\mathbf{lhs}} St$

containing o as `[o]`. Given a list $\vec{o}$, we write $\lbag \vec{o} \rbag$ to denote the multiset containing all (and only) 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., 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. Composition of substitutions θ and ϕ is denoted $\theta\phi$.

Figure 3.1 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 $\lbag A \mid g \rbag_{\vec{x} \in t}$ 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 implicit α-rename binding variables to avoid capture.

The development of CHR^{cp} is largely agnostic to the language of terms [6]. 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 $\lbag t \mid g \rbag_{x \in m}$ filters multiset m according to g and maps the result as specified by t. An atomic guard is either equality ($t = t'$), multiset membership ($t \dot{\in} t'$) or order comparison ($t\ op\ t'$ where $op \in \{<, \leq, >, \geq\}$).

A CHR head constraint $C : i$ is a constraint C paired with an occurrence index i. As in CHR, a CHR^{cp} rule $r \,@\, \bar{H} \Longleftrightarrow g \mid \bar{B}$ specifies the rewriting of the *head constraints* $\bar{H}$ into the *body* $\bar{B}$ under the conditions that guards g are satisfied.[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. We also require that a rule body be grounded by the head constraints and that guards do not appear in the rule body.

4 Operational Semantics of CHR^{cp}

This section recalls the operational semantics of CHR^{cp} [6]. 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

[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 [6] for a treatment of comprehension patterns in propagation heads.

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

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

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

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

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

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

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

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

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

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

Figure 4.2: Rule Body Application and Unifiability of Comprehension Patterns

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 [3], 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 three steps: Section 4.1 describes the processing of a rule's left-hand side, defining the CHR^{cp} *semantics of matching*. Section 4.2 discusses the execution of its right-hand side, defining rule body applications and the monotonicity property. Section 4.3 combines these components, giving CHR^{cp} its overall operational semantics. A more detailed treatment of the operational semantics of CHR^{cp} can be found in [6].

4.1 Semantics of Matching of CHR^{cp}

The semantics of matching, specified in Figure 4.1, 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 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 $\lbag A \mid g \rbag_{\vec{x} \in ts}$. If the comprehension domain is empty ($x \in \varnothing$), 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.

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 very bottom of Figure 4.1. 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 4.1. Rules ($\mathbf{l}^{\neg}_{mset-*}$) 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 ($\mathbf{l}^{\neg}_{comp-*}$) check that no constraints in St match the comprehension pattern $M = \wr A' \mid g \int_{\vec{x} \in ts}$.

4.2 Rule Body Application and Monotonicity

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. Defined in Figure 4.2, 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. This judgment is similar to the matching judgment (Figure 4.1) except that it skips any element in the comprehension domain that fails the guard (rule $\mathbf{r}_{comp\text{-}2}$).

We showed in [6] that to guarantee the safe incremental goal-based execution of a CHR^{cp} program $\mathcal{P}$, we must determine which rule body constraints are *monotone* (and which are not) and only delay the storage of monotone constraints. 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 $\mathcal{P}$. Thus, to test that a comprehension pattern M has no match 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. We call this property of CHR^{cp} *conditional monotonicity*. We formalize this idea by generalizing the residual non-matching judgment from Figure 4.1. The resulting *residual non-unifiability judgment* is defined in the bottom of Figure 4.2. 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 ($\mathbf{u}^{\neg}_{prog-*}$) 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 $\mathbf{u}^{\neg}_{mset-*}$). Rule ($\mathbf{u}^{\neg}_{atom}$) handles atomic facts, which are valid by default. Rules ($\mathbf{u}^{\neg}_{comp-*}$) 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 4.2. 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 (i.e., $g \rhd M \sqsubseteq_{\mathbf{unf}} B$), then we say that B is *monotone* w.r.t. program $\mathcal{P}$, denoted by $\mathcal{P} \triangleq^{\neg}_{\mathbf{unf}} B$. This relation is can be statically computed to avoid runtime overhead.

4.3 Operational Semantics

In this section, we present the overall operational semantics of CHR^{cp}. Execution states, defined in Figure 4.3, 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` identifies the various types of goals.

Figure 4.3 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 $getIdx(_)$ 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 $\perp$ 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

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

$$\begin{array}{lcll} \text{Goal Stack} & Gs & ::= & \texttt{[]} \mid \texttt{[}G \mid Gs\texttt{]} \\ \text{Labeled Store} & Ls & ::= & \varnothing \mid \{\!|Ls, A\#n|\!\} \\ \text{Execution State} & \sigma & ::= & \langle Gs\ ;\ Ls \rangle \end{array}$$

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

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

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

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

Figure 4.3: Execution States and Auxiliary Meta-operations

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 $\langle \texttt{[init}\ \bar{B}\texttt{]}\ ;\ \varnothing \rangle$ where $\bar{B}$ is the initial multiset of constraints. Figure 4.4 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 4.2). $\bar{B}_l$ 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 only add 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 the goals as '`eager`' goals (eagerly stored). 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 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; the later simply drops the leading goal otherwise. The last three rules deal with leading goals of the form $\texttt{act}\ A\#n\ i$: rule ($act{-}apply$) handles the case where the active constraint $A\#n$ matches the ith 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 4.1). 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. 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 ith 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 [6].

5 Compiling CHR^{cp} Rules

While Figures 4.1–4.4 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.

As described in Section 4, an active constraint $\texttt{act}\ A\#n\ i$ is matched against an occurrence of a 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*

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

Figure 4.4: Operational Semantics of CHR^{cp}

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 optimal ordering of partners that minimizes this intermediate data. Join ordering [5] 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 and balanced trees) that provide the best possible asymptotic time complexity. Our work extends traditional approaches to CHR compilation [5] to handle comprehension head constraints and augments them with optimizations specific to them.

5.1 Introducing CHR^{cp} Join Ordering

The top of Figure 5.1 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 of $p(\vec{t_i})$. 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 5.1 in the textual order given 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 $\lbag p_5(X, P) \mid P \dot{\in} Ws \rbag_{P \in Ps} : 5$) are matched in

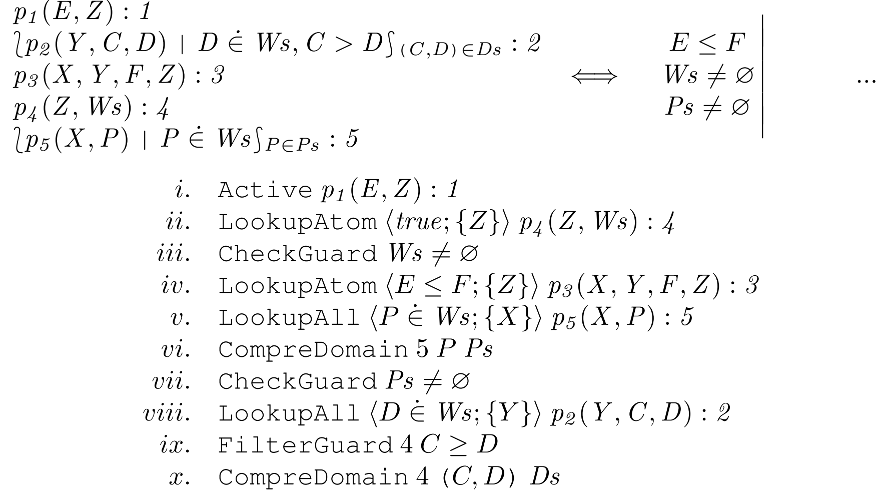

Figure 5.1: Optimal Join Ordering for $p_1(E,Z) : 1$

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

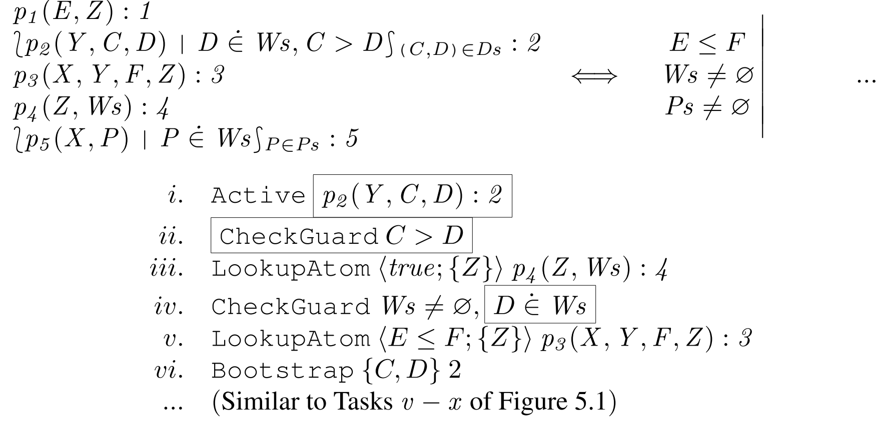

Figure 5.2: Optimal Join Ordering for $\wr p_2(Y,C,D) \mid D \dot{\in} Ws, C > D \int_{(C,D)\in Ds} : 2$

Such optimal 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 7.

5.2 Bootstrapping for Active Comprehension Head Constraints

In the example in Figure 5.1, 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 $H_i = \wr A' \mid g \int_{x\in xs} : i$. 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 5.2 illustrates this compilation for the comprehension head constraint $\wr p_2(Y,C,D) \mid D \dot{\in} Ws, C > D \int_{(C,D)\in Ds} : 2$ from Figure 5.1 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 5.1.

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 this by providing two uniqueness enforcing join tasks: If H_j and H_k are atomic head constraints, join task `NeqHead` j k (figure 6.1) 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.

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

$i.$ `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` $4\ D_2\ Ys$

Figure 5.3: Uniqueness Checks: Optimal Join Ordering for $p(D_0) : 1$

Figure 5.3 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. For a given CHR^{cp} rule, $r \text{ @ } \bar{H} \iff g \mid \bar{B}$, which has two comprehension patterns in $\bar{H}$, $\wr A_1 \mid g_1 \wr_{\vec{x}_1 \in xs_1} : j$ and $\wr A_2 \mid g_2 \wr_{\vec{x}_2 \in xs_2} : k$, a filtering join task `FilterHead` $j\ k$ will only be insert in the join ordering, if we have either $g \wedge g_1 \rhd A_1 \sqsubseteq_{\mathbf{unf}} \wr A_2 \mid g_2 \wr_{\vec{x}_2 \in xs_2}$ or $g \wedge g_2 \rhd A_2 \sqsubseteq_{\mathbf{unf}} \wr A_1 \mid g_1 \wr_{\vec{x}_1 \in xs_1}$. We have implemented a conservative test for the relation $\sqsubseteq_{\mathbf{unf}}$ from our work on reasoning about comprehension patterns in SMT [7].

6 Representing CHR^{cp} Join Orderings

In this section, we formalize the notion of join ordering 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 examine how to chose this sequence of partners. Figure 6.1 defines the constituents 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 3.1, 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 $\bar{g} = \wr g_1, g_2 \wr$, with $\models \bar{g}$ expressing the satisfiability of each guard in $\bar{g}$. We defer a detailed description of indexing directives till later, but for now, note that an indexing directive is a tuple $\langle g; \vec{x} \rangle$ such that g is an indexing guard and $\vec{x}$ are hash variables. Each join task essentially defines a specific sub-routine of the overall multiset match defined by the join-ordering. The following informally describes each type of join task:

- `Active` $A : i$ defines A to be the constraint pattern in which the active constraint must match.
- `LookupAtom` $I\ A : i$ Defines A to be an atomic head constraint pattern that should be matched. This means that this join task is successfully executed if it is matched to exactly one candidate A' in the store. Indexing directive I is to be used to retrieve candidates in the store that match A.
- `LookupAll` $I\ A : i$ defines A to be a comprehension head constraint pattern that should be matched. This means that this join task is executed by matching it to *all* candidates A' in the store that match A. Indexing directive I is to be used to retrieve candidates in the store that match A.
- `Bootstrap` $\vec{x}\ i$ dictates that match to head constraint occurrence i and variable bindings to $\vec{x}$ are to be omitted from this join task.

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

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

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

$$
\mathit{allIdxDirs}\,(\Sigma, A, \bar{g}) \quad ::= \quad \{\!\!\{ \mathit{idxDir}\,(\Sigma, A, g_\alpha) \mid \text{for all } g_\alpha \in \bar{g} \cup \mathit{true} \}\!\!\}
$$

Figure 6.1: Indexing Directives

- `CheckGuard` $\bar{g}$ requires that guard conditions $\bar{g}$ are to be tested in the substitution built by the current match. This join task succeeds only if $\bar{g}$ is satisfiable.
- `FilterGuard` $i\ \bar{g}$ mandates that each candidate accumulated for occurrence i is to be tested on guard conditions $\bar{g}$. Those which are not satisfiable are to be filtered away from the match. Occurrence i must be a comprehension pattern head constraint.
- `NeqHead` $i\ j$ dictates that constraints matching head constraint pattern occurrences i and j are to be compared for referential equality. Indices i and j should be atomic head constraint patterns.
- `FilterHead` $i\ j$ requires that all constraints that were matched to both occurrence i and j head constraints are to be removed from i. Indices i must be a comprehension pattern constraint pattern, while j can be either a comprehension or an atomic constraint.
- `CompreDomain` $i\ \vec{x}\ x$ dictates that we retrieve all constraints in i and project values of variables $\vec{x}$ onto the multiset x that contains the set of all such bindings.

The bottom part of Figure 6.1 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 $\mathit{idxDir}\,(\Sigma, A, g_\alpha)$ returns a valid index directive for a given constraint A, the join context Σ and the atomic guard g_α. 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_α 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_α as part of the index directive, together with the set of variables that appear in both Σ and A. If g_α is a membership relation $t_1 \dot{\in} t_2$, the operation returns g_α only if $\Sigma; A \rhd t_2 \mapsto t_1$. Otherwise, g_α cannot be used as an index, hence the operation returns true. Finally, $\mathit{allIdxDirs}\,(\Sigma, A, \bar{g})$ defines the set of all such indexing directives derivable from $\mathit{idxDir}\,(\Sigma, A, g_\alpha)$ where $g_\alpha \in \bar{g}$.

An indexing directive $\langle g_\alpha ; \vec{x} \rangle$ for a constraint pattern $p(\vec{t})$ determines what type of indexing method can be exploited for the given constraint type. The set of candidates matching $p(\vec{t})$ is retrieved from the store by the following means:

1. $\langle \mathit{true} ; \vec{x} \rangle$, $\vec{x} \neq \emptyset$: constraints $p(\vec{t})$ are stored in a hash multimap that indexes the constraints on argument positions of $\vec{t}$ that variables $\vec{x}$ appear in. It supports amortized $O(1)$ lookups.

2. $\langle x \dot{\in} ts; \vec{x} \rangle$, $\vec{x} \neq \emptyset$: similar to (1), but constraints are indexed by argument position of x as well. But during lookup, we enumerate values of x from ts. It supports amortized $O(1 * m)$ lookups, where m is size of ts.

3. $\langle x \dot{\in} ts; \emptyset \rangle$: similar to (2), but index keys are computed solely from argument position of x.

4. $\langle x\ op\ y; \emptyset \rangle$ where $op \in \{<, \leq, >, \geq\}$: constraints $p(\vec{t})$ are stored in a balanced binary tree, sorted by argument position of either x or y (exclusive). Candidates are retrieved by a binary search. It supports $O(log\ n)$ lookups, where n is the number of p constraints.

5. $\langle x\ op\ y; \vec{x} \rangle$, $\vec{x} \neq \emptyset$ where $op \in \{<, \leq, >, \geq\}$: constraints $p(\vec{t})$ are stored in a hash map that indexes the constraints on argument positions of $\vec{t}$ that variables $\vec{x}$ appear in. The contents of this hash map are balanced binary tree that sorts its elements in a manner similar to (4). It supports $O(log\ p)$ lookups, where p is the size of the largest binary tree.

6. $\langle true; \emptyset \rangle$: constraints $p(\vec{t})$ are stored in a linear linked list. It supports $O(n)$ lookups where n is the number of n constraints.

7 Building CHR^{cp} Join Orderings

In this section, we define the construction of valid join orderings from CHR^{cp} rules.

Figure 7.1 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 $r\ @\ \wr \vec{H}_a, \vec{H}_m, H_i \int \iff \bar{g} \mid \vec{B}$ into a valid join ordering for this sequence. The topmost definition of $compileRuleHead$ in Figure 7.1 defines the case when H_i is 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 remembers 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 an atomic pattern $A : j$, 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 (`CheckGuard` $\bar{g}_1$). This subset is computed by the operation $scheduleGrds\ (\Sigma, \bar{g})$ which returns the partition $(\bar{g}_1, \bar{g}_2)$ 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., `LookupAtom` $\langle g_i; \vec{x} \rangle\ A : j$) and uniqueness enforcement join tasks $neqHs\ (A : j, \vec{H}_h)$ 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 ($allIdxDirs\ (\Sigma, A, \bar{g}_2)$). Hence from a given sequence of partners, $compileRuleHead$ 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 task is created in the place of `LookupAtom`; 2) the comprehension guards $\bar{g}_m$ are included as possible indexing guards ($allIdxDirs\ (\Sigma, A, \bar{g}_2 \cup \bar{g}_m)$); 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 ($filterHs\ (C : j, C' : k)$) 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 optimal join ordering for each rule occurrence H_i of a CHR^{cp} program. This heuristic extends the approach in [5] to handle comprehensions. 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 jth partner H_j. Since earlier partners have higher weight, this scoring rewards join orderings with the least expensive partners scheduled early. The join cost w_j for a partner constraint $C : j$ is a pair (v_f, v_l) where v_f is the *degree of freedom* and v_l is the *indexing score*. The 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

$$compileRuleHead\,(A : i, \vec{H}_a, \vec{H}_m, \bar{g}) \quad ::= \quad [\texttt{Active}\ A : i \mid J_a]\texttt{++}J_m\texttt{++}checkGrds\,(\bar{g}'')$$
$$\text{where} \quad (J_a, \Sigma, \bar{g}') \quad = \quad buildJoin\,(\vec{H}_a, FV(A_i), \bar{g}, [\,])$$
$$\text{and} \quad (J_m, \Sigma', \bar{g}'') \quad = \quad buildJoin\,(\vec{H}_m, \Sigma, \bar{g}', \vec{H}_a)$$

$$compileRuleHead\,(\wr A \mid \bar{g}_m \wr_{\vec{x} \in xs} : i, \vec{H}_a, \vec{H}_m, \bar{g})$$
$$::= \quad [\texttt{Active}\ A : i \mid J_a]\texttt{++}[\texttt{Bootstrap}\ FV(A) - FV(\vec{x})\ i \mid J_m]\texttt{++}checkGrds\,(\bar{g}'')$$
$$\text{where} \quad (J_a, \Sigma, \bar{g}') \quad = \quad buildJoin\,(\vec{H}_a, FV(A_i), \bar{g} \cup \bar{g}_m, [\,])$$
$$\text{and} \quad (J_m, \Sigma', \bar{g}'') \quad = \quad buildJoin\,([\wr A_i \mid \bar{g}_m \wr_{\vec{x} \in xs} \mid \vec{H}_m], \Sigma - \vec{x}, \bar{g}', \vec{H}_a)$$

$$buildJoin\,([A : j \mid \vec{H}], \Sigma, \bar{g}, \vec{H}_h)$$
$$::= \quad ([\texttt{CheckGuard}\ \bar{g}_1, \texttt{LookupAtom}\ \langle g_i; \vec{x} \rangle\ A : j]\texttt{++}neqHs\,(A : j, \vec{H}_h)\texttt{++}\vec{J}, \Sigma, \bar{g}_r)$$
$$\text{where} \quad (\bar{g}_1, \bar{g}_2) \quad = \quad scheduleGrds\,(\Sigma, \bar{g})$$
$$\text{and} \quad \langle g_i; \vec{x} \rangle \quad \in \quad allIdxDirs\,(\Sigma, A, \bar{g}_2)$$
$$\text{and} \quad (\vec{J}, \Sigma', \bar{g}_r) \quad = \quad buildJoin\,(\vec{H}, \Sigma \cup FV(A), \bar{g}_2 - g_i, \vec{H}_h\texttt{++}[A : j])$$

$$buildJoin\,([\wr A \mid \bar{g}_m \wr_{\vec{x} \in xs} : j \mid \vec{H}], \Sigma, \bar{g}, \vec{H}_h)$$
$$:= \quad ([\texttt{CheckGuard}\ \bar{g}_1, \texttt{LookupAll}\ \langle g_i; \vec{x}' \rangle\ A : j, \texttt{FilterGuard}\ (\bar{g}_m - \{g_i\})]$$
$$\texttt{++}filterHs\,(\wr A \mid \bar{g}_m \wr_{\vec{x} \in xs} : j, \vec{H}_h)\texttt{++}[\texttt{CompreDomain}\ j\ \vec{x}\ xs \mid \vec{J}], \Sigma, \bar{g}_r)$$
$$\text{where} \quad (\bar{g}_1, \bar{g}_2) \quad = \quad scheduleGrds\,(\Sigma, \bar{g})$$
$$\text{and} \quad \langle g_i; \vec{x}' \rangle \quad \in \quad allIdxDirs\,(\Sigma, A, \bar{g}_2 \cup \bar{g}_m)$$
$$\text{and} \quad (\vec{J}, \Sigma', \bar{g}_r) \quad = \quad buildJoin\,(\bar{H}, \Sigma \cup FV(A), \bar{g}_2 - g_i, \vec{H}_h\texttt{++}[\wr A \mid \bar{g}_m \wr_{\vec{x} \in xs} : j])$$

$$buildJoin\,([\,], \Sigma, \bar{g}, \vec{H}_h) \quad ::= \quad ([\,], \Sigma, \bar{g})$$

$$scheduleGrds\,(\Sigma, \bar{g}) \quad ::= \quad (\{g \mid g \in \bar{g}, FV(g) \subseteq \Sigma\}, \{g \mid g \in \bar{g}, FV(g) \not\subseteq \Sigma\})$$

$$neqHs\,(p(_) : j, p'(_) : k) \quad ::= \quad \text{if } p = p' \text{ then } [\texttt{NeqHead}\ j\ k] \text{ else } [\,]$$

$$filterHs\,(C : j, C' : k) \quad ::= \quad \text{if } true \rhd C' \sqsubseteq_{\mathbf{unf}} C \text{ then } [\texttt{FilterHead}\ j\ k] \text{ else } [\,]$$

Figure 7.1: Building Join Ordering from CHR^{cp} Head Constraints

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. Figure 7.2 defines this heuristic scoring function, denoted by $joScore\,(\vec{J})$.

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 optimal ordering based on this heuristic. Since CHR^{cp} rules typically contain a small number of head constraints, join ordering permutations can be practically computed.

8 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.

$$joScore\,(\vec{J}) \quad ::= \quad joScoreInt\,(\vec{J}, \varnothing, (0,0), (0,0))$$

$$\begin{array}{l}
joScoreInt\,([\texttt{Active}\ A : i \mid \vec{J}], \Sigma, Score, Sum) \\
\qquad ::= \quad joScoreInt\,(\vec{J}, \Sigma \cup FV(A), Score, Sum) \\
joScoreInt\,([\texttt{LookupAtom}\ \langle g_\alpha; \vec{x}_i \rangle\ A : j \mid \vec{J}], \Sigma, Score, Sum) \\
\qquad ::= \quad joScoreInt\,(\vec{J}, \Sigma \cup FV(A), Score + Sum + W_i, Sum + W_i) \\
\qquad\qquad \text{where} \quad W_i = (|FV(A) - \Sigma| + m, -|\vec{x}_i| + m) \text{ and } m = idxMod\,(g_\alpha) \\
joScoreInt\,([\texttt{LookupAll}\ \langle g_\alpha; \vec{x}_i \rangle\ A : j \mid \vec{J}], \Sigma, Score, Sum) \\
\qquad ::= \quad joScoreInt\,(\vec{J}, \Sigma \cup \{xs\}, Score + Sum + W_i, Sum + W_i) \\
\qquad\qquad \text{where} \quad W_i = (|FV(A) - \Sigma| + m, -|\vec{x}_i| + m) \text{ and } m = idxMod\,(g_\alpha) \\
joScoreInt\,([\texttt{CheckGuard}\ \bar{g} \mid \vec{J}], \Sigma, (s_1, s_2), (u_1, u_2)) \\
\qquad ::= \quad joScoreInt\,(\vec{J}, \Sigma, (s_1 - |\bar{g}|, s_2), (u_1 - |\bar{g}|, u_2)) \\
joScoreInt\,([\texttt{FilterGuard}\ \vec{x}\ xs\ \bar{g} \mid \vec{J}], \Sigma, (s_1, s_2), (u_1, u_2)) \\
\qquad ::= \quad joScoreInt\,(\vec{J}, \Sigma, (s_1 + |\bar{g}|, s_2), (u_1 + |\bar{g}|, u_2)) \\
joScoreInt\,([J \mid \vec{J}], \Sigma, Score, Sum) \quad ::= \quad joScoreInt\,(\vec{J}, \Sigma, Score, Sum) \\
\qquad \text{if } J \text{ is either a } \texttt{Bootstrap},\ \texttt{NeqHead},\ \texttt{FilterHead} \text{ or } \texttt{CompreDomain} \text{ join task.} \\
joScoreInt\,([\,], \Sigma, Score, Sum) \quad ::= \quad Score
\end{array}$$

$$idxMod\,(g_\alpha) \quad ::= \quad \text{if } g_\alpha = true \text{ then } 0 \text{ else } -1$$

Figure 7.2: Measuring Cost of Join Ordering

Figure 8.1 defines the constituents of this abstract machine. The input of the machine is a *matching context* $\Theta = \langle A\#n; \vec{J}; Ls \rangle$, which consists of an active constraint $A\#n$, of a join ordering $\vec{J}$ and of the constraint store Ls. A *matching state* $\mathcal{M}$ is a tuple $\langle J; pc; \theta; \vec{Br}; Pm \rangle$ 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 pairs $(\theta, A\#n)$. 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; $lookupCands\,(Ls, A', \langle g; \vec{x} \rangle)$ abstractly retrieves the multiset of candidates $A\#n$ in store Ls that match pattern A' and satisfy g for indexing directive $\langle g; \vec{x} \rangle$.

8.1 Abstract Machine Execution

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 8.2 defines its behavior by means of several transition rules. Most of the rules defines the *positive* (successful match) transitions that transforms a matching state $\mathcal{M}$, implementing a specific subroutine of the overall multiset matching of a CHR^{cp} rule. The exception are rules $(backtrack)$ and $(fail{-}match)$, which implements the collective *negative* (failure) transitions. We will now describe each transition in detail.

- $(active)$ implements the positive behavior of the join task `Active` $occA'i$. Active constraint $A\#n$ is matched with A' (i.e., $\phi = match\,(A, \theta A')$) and the search proceeds under the condition that this match is successful (i.e., $\phi \neq \bot$).
- $(lookup{-}atom)$ implements the positive behavior of the join task `LookupAtom` $\langle g; \vec{x}' \rangle\ A' : j$. This join task represents the task of searching the store Ls for a match with rule head constraint $A' : j$. Constraints in Ls that matches A' under the conditions of the lookup directive $\langle g; \vec{x}' \rangle$ are retrieved ($lookupCands\,(Ls, \theta A', \langle \theta g; \vec{x} \rangle)$). 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.

Matching Context	Θ	$::=$	$\langle A\#n; \vec{J}; Ls\rangle$
Matching State	$\mathcal{M}$	$::=$	$\langle J; pc; \vec{Br}; \theta; Pm\rangle$
Backtrack Branch	Br	$::=$	(pc, θ, Pm)
Candidate Match	U	$::=$	$(\theta, A\#n)$
Partial Match	Pm	$::=$	$Pm, i \mapsto \bar{U} \mid \varnothing$

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

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

Figure 8.1: *LHS* Matching States and Auxiliary Operations

- $(\mathit{check{-}guard})$ implements the positive behavior of the join task `CheckGuard` $\bar{g}$. This join task represents the task of enforcing rule guards $\bar{g}$. If $\bar{g}$ is satisfiable under the substitution of the current state (i.e., $\models \theta\bar{g}$), the search proceeds.

- $(\mathit{lookup{-}all})$ implements the behavior of the join task `LookupAll` $\langle g; \vec{x}\rangle\ A' : j$. It represents the matching of a comprehension pattern that corresponds to head constraint j of the CHR^{cp} rule. In particular, this comprehension pattern mentions the atomic constraint pattern A'. Similarly to $(\mathit{lookup{-}atom})$, this join task retrieves candidates in the store Ls, matching A' and satisfying the indexing directive ($\bar{U} = lookupCands\,(Ls, \theta A', \langle \theta g; \vec{x}\rangle)$). However, rather than branching, the search proceeds by extending the partial match with the set of all candidates (i.e., $j \mapsto \bar{U}$). This effectively implements the maximal comprehension of constraints in Ls that matches with A'. Also, unlike $(\mathit{lookup{-}atom})$, this join task never fails for well-defined join orderings. In fact, the search proceeds even if no candidates are found ($j \mapsto \varnothing$).

- $(\mathit{filter{-}guard})$ implements the behavior of the join task `FilterGuard` $j\ \bar{g}$. It represents the enforcement of the guard conditions $\bar{g}$ of the comprehension pattern head constraint j. This task proceeds by filtering from $Pm\,[j]$ candidates that do not satisfy the guard conditions $\bar{g}$. Hence $Pm\,[j]$ of the successor state will only contain candidates that satisfy the given guard condition. Like $(\mathit{lookup{-}all})$, this join task never fails for well-defined join orderings.

- $(\mathit{neq{-}head})$ implements the positive behavior of the join task `NeqHead` $j\ k$. This join task enforces the uniqueness of candidates matched with head constraints j and k. Specifically, if $A\#m$ and $A\#n$ are constraints in Ls matched to j and k respectively, the search only proceeds if $m \neq n$.

- $(\mathit{filter{-}head})$ implements the behavior of the join task `FilterHead` $j\ k$. Similar to $(\mathit{neq{-}head})$, this join task enforces the uniqueness of candidates matched with head constraints j and k. However, it differs from it in that it enforces uniqueness by filtering from $Pm\,[j]$ any candidates that appear also in $Pm\,[k]$. Unlike $(\mathit{neq{-}head})$, this join task never fails for well-defined join orderings.

- $(\mathit{compre{-}dom})$ implements the behavior of the join task `CompreDomain` $j\ \vec{x}\ xs$. This join task represents the construction of comprehension domain xs of the comprehension pattern head constraint j. It is executed by extracting projections of the variables $\vec{x}$ from each candidate of $Pm\,[j]$. The current substitution is then extended with xs bounded to the multiset containing all such projections of $\vec{x}$ (i.e., $\{\!\!\{\phi'\vec{x} \mid$ for all $(\phi', _) \in \bar{U}\}\!\!\}$). This join task never fails for well-defined join orderings.

- $(\mathit{bootstrap})$ implements the behavior of the join task `Bootstrap` $\vec{x}\ j$. This join task represents the administrative task of removing variable mappings and candidate bindings imposed by the active constraint. Specifically, mappings of $\vec{x}$ from the current substitution and candidate binding $(j \mapsto (\phi, A\#n)) \in Pm$ are removed from the current state to facilitate the boot strapping process (discussed in Section 5.2) of matching the active constraint to head constraint j which is a comprehension pattern. This join task never fails for well-defined join orderings.

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

where $\Theta = \langle A\#n; \vec{J}; Ls\rangle$

Figure 8.2: Execution of CHR^{cp} Join Ordering

- $(backtrack)$ implements the backtracking of failed matching states. Specifically, it implements the negative behavior of join tasks `LookupAtom`, `CheckGuard` and `NeqHead`, where backtracking may possibly occur because the respective conditions of the rules $(lookup{-}atom)$, $(check{-}guard)$ or $(neq{-}head)$ is not satisfiable. 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'.

- $(fail{-}match)$ implements the overall failure to find a match in the store Ls. Specifically, it implements the negative behavior of $(active)$ and $(backtrack)$, during which there are no backtracking options available (i.e., $\vec{Br} = \emptyset$).

The execution of this abstract machine implicitly terminates when pc reaches an index outside the join ordering (i.e., $\vec{J}\,[pc] = \bot$). Once this happens, the match Pm of the ending state contains a complete match for the CHR^{cp} rule represented by the join ordering. The failure case is modeled by rule $(fail{-}match)$ which causes a reachable state to transition to the failure state $\bot$.

8.2 Example of Join Ordering Compilation

In this section, we consider a more complex example of join ordering compilation and execution. This example demonstrates how performance is greatly affected by the ordering of join tasks, hence highlighting the importance of

$$mrg \ @ \begin{array}{l} merge(X, Y, I_m, O_m, V_m) : 1 \\ foundMWOE(X, Is_1) : 2 \\ foundMWOE(Y, Is_2) : 3 \\ \wr edge(I, O, V) \mid I \dot{\in} Is_1, O \dot{\in} Is_2 \int_{(I,O,V) \in Es_1} : 4 \\ \wr edge(I, O, V) \mid I \dot{\in} Is_2, O \dot{\in} Is_1 \int_{(I,O,V) \in Es_2} : 5 \end{array} \iff \begin{array}{l} findMWOE(X, \wr Is_1, Is_2 \int) \\ forward(Y, X) \\ mstEdge(I_m, O_m, V_m) \\ mstEdge(O_m, I_m, V_m) \end{array}$$

A: Optimal Ordering and Indexing

Task No.	Join Task	# Comparisons/Operations
$A.i$	`Active` $merge(X, Y, I_m, O_m, V_m) : 1$	1
$A.ii$	`LookupAtom` $\langle true; \{X\} \rangle$ $foundMWOE(X, Is_1) : 2$	1
$A.iii$	`LookupAtom` $\langle true; \{Y\} \rangle$ $foundMWOE(Y, Is_2) : 3$	1
$A.iv$	`LookupAll` $\langle I \dot{\in} Is_1; \varnothing \rangle$ $edge(I, O, V) : 4$	$\|Is_1\|$
$A.v$	`FilterGuard` $4\ O \dot{\in} Is_2$	$\|\bar{U}_4\| * \|Is_2\|$
$A.vi$	`CompreDomain` $4\ (I, O, V)\ Es_1$	$\|\bar{U}_4\|$
$A.vii$	`LookupAll` $\langle I \dot{\in} Is_2; \varnothing \rangle$ $edge(I, O, V) : 5$	$\|Is_2\|$
$A.viii$	`FilterGuard` $5\ O \dot{\in} Is_1$	$\|\bar{U}_5\| * \|Is_1\|$
$A.ix$	`FilterHead` $5\ 4$	$\|\bar{U}_5\| * \|\bar{U}_4\|$
$A.x$	`CompreDomain` $5\ (I, O, V)\ Es_2$	$\|\bar{U}_5\|$

B: Worst-case Ordering with no Indexing

Task No.	Join Task	# Comparisons/Operations
$B.i$	`Active` $merge(X, Y, I_m, O_m, V_m) : 1$	1
$B.ii$	`LookupAll` $\langle true; \varnothing \rangle$ $edge(I, O, V) : 4$	m
$B.iii$	`LookupAll` $\langle true; \varnothing \rangle$ $edge(I, O, V) : 5$	m
$B.iv$	`LookupAtom` $\langle true; \varnothing \rangle$ $foundMWOE(X, Is_1) : 2$	n
$B.v$	`LookupAtom` $\langle true; \varnothing \rangle$ $foundMWOE(Y, Is_2) : 3$	n
$B.vi$	`FilterGuard` $4\ I \dot{\in} Is_1$	$m * \|Is_1\|$
$B.vii$	`FilterGuard` $4\ O \dot{\in} Is_2$	$\|\bar{U}_4\| * \|Is_2\|$
$B.viii$	`CompreDomain` $4\ (I, O, V)\ Es_1$	$\|\bar{U}_4\|$
$B.ix$	`FilterGuard` $5\ I \dot{\in} Is_2$	$m * \|Is_2\|$
$B.x$	`FilterGuard` $5\ O \dot{\in} Is_1$	$\|\bar{U}_5\| * \|Is_1\|$
$B.xi$	`FilterHead` $5\ 4$	$\|\bar{U}_5\| * \|\bar{U}_4\|$
$B.xii$	`CompreDomain` $5\ (I, O, V)\ Es_2$	$\|\bar{U}_5\|$

where m/n are numbers of $edge/foundMWOE$ constraints and $\bar{U}_4/\bar{U}_5$ are candidates for head constraints $4/5$.

Figure 8.3: Join Ordering Comparison for GHS Algorithm, *mrg* rule

the compilation techniques defined in Section 7. We consider the join ordering compilation of the mrg rule of the CHR^{cp} implementation of the GHS algorithm, introduced in Section 2.4. Figure 8.2 illustrates two join orderings for the mrg rule, with $merge(X, Y, I_m, O_m, V_m) : 1$ as the active head constraint pattern. These two join orderings shows two distinct execution sequences for the multiset matching of this rule: Join ordering A is optimal and is computed from the operations $compileRuleHead$ and $joScore$ defined in Section 7. Instead, join ordering B is the worst possible ordering, which may emerge without preprocessing. Furthermore, to illustrate the importance of computing sophisticated indexing (Figure 6.1), we have omitted the use of all but the most trivial indexing directives in join ordering B.

Optimal join ordering A executes the multiset matching of the head constraints of rule mrg in the sequence 2–3–4–5. This ordering schedules atomic head constraint patterns (2 and 3) before comprehension patterns (4 and 5), and exploits hash map indexing directives (join tasks $A.ii$ and $A.iii$) and membership guard indexing directives (join tasks $A.iv$ and $A.vii$). Note that since our current implementation does not support indexing directives with multiple guards, for tasks $A.iv$ and $A.vii$, we arbitrarily selected a possible membership guard indexing directive out of the two membership guards of each comprehension pattern ($I \dot{\in} Is_1$ for $A.iv$ and $I \dot{\in} Is_2$ for $A.vii$) while the remainder are scheduled as post-comprehension guards ($O \dot{\in} Is_2$ in $A.v$ and $O \dot{\in} Is_1$ in $A.viii$).

Join ordering B executes the multiset matching of the head constraints of rule mrg in the sequence 4–5–2–3. Furthermore, the only form of indexing directive used is trivial linear iteration (i.e., $\langle true; \varnothing \rangle$). Comprehension patterns 4 and 5 are scheduled before the atomic constraint patterns, and all comprehension guards are enforced as post-comprehension guards (i.e., join tasks $B.vi$, $B.vii$, $B.ix$ and $B.x$).

For each join task, Figure 8.2 also presents an approximate count on the number of comparisons (or operations) executed by the join task. For optimal join ordering A, tasks $A.ii$ and $A.iii$ execute in amortized constant time because candidates are retrieved by using the hash map indexing directives $\langle true; \{X\} \rangle$ and $\langle true; \{Y\} \rangle$. Join task $A.iv$, which retrieves candidates for the comprehension pattern 4, executes in time $|Is_1|$, since it uses the indexing directive $\langle I \dot{\in} Is_1; \varnothing \rangle$ that does $|Is_1|$ queries on the hash map containing $edge$ constraints. This produces $\bar{U}_4$, the multiset of candidates containing all $edge$ constraints retrieved as the match to head constraint pattern 4. Task $A.v$ costs $|\bar{U}_4| * |Is_2|$ comparisons because for each candidate c in $\bar{U}_4$, in general we conduct $|Is_2|$ comparisons on c with each candidate in Is_2. For task $A.vi$, each candidate in $\bar{U}_4$ is projected onto Es_1, hence requiring $|\bar{U}_4|$ operations. The cost of join tasks $A.vii$ to $A.x$ are largely similar to the join tasks for the comprehension pattern 4, with the exception of task $A.ix$: it is executed in time proportional to $|\bar{U}_5| * |\bar{U}_4|$, because, in general, we need to compare each candidate in $\bar{U}_5$ with each candidate in $\bar{U}_4$.

Join ordering B, on the other hand, executes more operations: $B.ii$ and $B.iii$ executes in time proportional to m, the number of $edge$ constraints in the store. $B.iv$ and $B.v$ executes in time proportional to n, the number of $foundMWOE$ constraints in the store. Comparing these with the respective lookup for atomic head constraints 2 and 3 in the optimal join ordering (i.e., $A.ii$ and $A.iii$), the optimal join ordering performs much better, since lookups take amortized constant time. We next compare the optimal task $A.iv$ with join task $B.ii$, both of which deal with the initial enumeration of candidates for comprehension head constraint 2. For task $A.iv$, the cost of execution is proportional to $|Is_1|$ (thanks to the lookup directive) while for $B.ii$ executes in time proportional to m. Note that $|Is_1|$ is certainly less than m, since Is_1 is a subset of the m $edge$ constraints in the store. Even though $|Is_1|$ and m are similar in terms of asymptotic comparison, their difference imposes a significant impact on performances, especially given that it is incurred in every application of the rule. The situation is similar for join tasks $A.vii$ and $B.iii$: the performance of task $A.vii$, which is $|Is_2|$, dwarfs that of $B.iii$, that is m. Another handicap of join ordering B is its reliance on *unbounded post-comprehension filtering*: the first post-comprehension filtering tasks for comprehension patterns 4 and 5 ($B.vi$ and $B.ix$ respectively) filter from the multiset of all $edge$ constraints in the store (each the size of m). Hence $B.vi$ and $B.ix$ costs $m * |Is_1|$ and $m * |Is_2|$. Comparing this against the first filtering of join-ordering A ($A.v$ and $A.viii$) whose cost is $|\bar{U}_4| * |Is_2|$ and $|\bar{U}_5| * |Is_1|$, the latter values are guaranteed to be lower, since $\bar{U}_4$ and $\bar{U}_5$ are subsets of m $edge$ constraints in the store.

In terms of raw counts of comparison operations, the optimal join ordering A clearly performs better than join ordering B (empirical results can be found in Section 11). Additionally, scheduling atomic constraint patterns (2 and 3) before comprehension patterns (4 and 5) has an important advantage: In optimal join ordering A, since atomic constraints 2 and 3 are scheduled before comprehension patterns 4 and 5, we do not incur overheads of building

$$constr\ (Pm, i) \quad ::= \quad \begin{cases} \wr A\#n \mid \text{for all } (_, A\#n) \in \bar{U} \wr & \text{if } Pm\,[i] = \bar{U} \text{ and } \bar{U} \neq \bot \\ \emptyset & \text{otherwise} \end{cases}$$

$$jtRuleHeads\ (J) \quad ::= \quad \begin{cases} \wr H \wr & \text{if } J = \texttt{Active}\ H,\ J = \texttt{LookupAtom}\ _\ H \text{ or } J = \texttt{LookupAll}\ _\ H \\ \emptyset & \text{otherwise} \end{cases}$$

$$jtGuards\ (J) \quad ::= \quad \begin{cases} \wr g \wr & \begin{cases} \text{if } J = \texttt{LookupAtom}\ \langle g; _\rangle\ _,\ J = \texttt{CheckGuard}\ g, \\ J = \texttt{LookupAll}\ \langle g; _\rangle\ _ \text{ or } J = \texttt{FilterGuard}\ _\ g \end{cases} \\ \emptyset & \text{otherwise} \end{cases}$$

$$jtOccs\ (J) \quad ::= \quad \begin{cases} \wr i \wr & \text{if } J = \texttt{Bootstrap}\ _\ i \text{ or } J = \texttt{CompreDomain}\ i\ _\ _ \\ \wr i, j \wr & \text{if } J = \texttt{NeqHead}\ i\ j \text{ or } J = \texttt{FilterHead}\ i\ j \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 9.1: More Auxiliary Operations

comprehensions 4 and 5 if we cannot find matches to either 2 or 3. For join ordering B however, note that if either head constraints 2 or 3 fail to be matched (by $B.iv$ and $B.v$ respectively), join tasks $B.ii$ and $B.iii$ that costs m operations each would effectively been wasted.

9 Correctness of the 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. Refer to Appendix A for the proofs of the lemmas and theorems presented in this section.

Figure 9.1 introduces a few more auxiliary operations we will use in this section. The operation $constr\ (Pm, i)$ returns the multiset of all constraints in partial match Pm mapped by i. We inductively extend $constr\ (Pm, _)$ to sets of occurrence indices $\bar{i}$. Operations $jtRuleHeads\ (J)$, $jtGuards\ (J)$ and $jtOccs\ (J)$ returns rule heads, guards and occurrence indices that appear in J respectively, or the empty set if none exist. We inductively extend their inputs to lists of join tasks $\vec{J}$. We denote the exhaustive transition of the CHR^{cp} abstract matching machine as $\Theta \rhd \mathcal{M} \rightarrowtail^{*}_{lhs} \mathcal{M}'$ where $\mathcal{M}'$ is a *terminal* state of the form $\langle \bot; _; _; _; _\rangle$ with $_$ denoting an arbitrary value. The program counter is $\bot$ since it has gone past the last index of $\vec{J}$. An *initial* state has the form $\langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing\rangle$, while all other states in between (including initial and terminal) are called *reachable* states. Finally, we denote s transitions of the CHR^{cp} abstract matching machine (not necessary exhaustive) as $\Theta \rhd \mathcal{M} \rightarrowtail^{s}_{lhs} \mathcal{M}'$.

9.1 Valid Matching Contexts and States

We define a notion of validity for our abstract machine. Furthermore, we show that given valid inputs (matching contexts and initial matching states), the abstract machine transition operation derives valid states.

A matching context $\Theta = \langle A\#n; \vec{J}; Ls\rangle$ is *valid* if and only if $A\#n \in Ls$ and $\vec{J}$ is a valid join ordering. A valid join ordering $\vec{J}$ of a CHR^{cp} rule $r\ @\ \bar{H} \Longleftrightarrow \bar{g} \mid \bar{B}$, have the following properties:

- *Active Head*: $\vec{J}\,[0] = \texttt{Active}\ H$ for some rule head H.
- *Unique Rule Heads*: For any $A : j$ and $A' : k$ such that $\llparenthesis A : j, A' : k \rrparenthesis \subseteq \mathit{jtRuleHeads}\,(\vec{J})$, we have $j \neq k$.
- *Uniqueness Enforcement*: For any $p(_) : j, p'(_) : k \in \mathit{jtRuleHeads}\,(\vec{J})$ such that $j \neq k$, then:
  - for the case where j and k are both atomic head constraints, i.e., $p_j(_) : j, p_k(_) : k \in \bar{H}$, if $p_j = p_k$ then there exists some pc such that $\vec{J}\,[pc] = \texttt{NeqHead}\ j\ k$
  - for the case where j is a comprehension pattern, i.e., $M : j, C : k \in \bar{H}$, if $\mathit{true} \rhd C \sqsubseteq_{\mathbf{unf}} M$, then $\vec{J}\,[pc] = \texttt{FilterHead}\ j\ k$
- *Guard Representation*: Guards in rule r are presented in $\vec{J}$. Specifically, the $\vec{J}$ has the following properties:
  - *Rule Guards*: $\bar{g} \subseteq \mathit{jtGuards}\,(\vec{J})$
  - *Comprehension Guards*: For each $\llparenthesis A \mid \bar{g}' \rrparenthesis_{\vec{x} \in xs} : j \in \bar{H}, \bar{g}' \subseteq \mathit{jtGuards}\,(\vec{J})$
- *Guard Scope Coverage*: For any $pc \in \mathit{range}\,(\vec{J})$ such that $\llparenthesis g \rrparenthesis = \mathit{jtGuards}\,(\vec{J}\,[pc])$, we have:

$$FV(g) \subseteq FV(\mathit{jtRuleHeads}\,(\vec{J}\,[0 \ldots pc{+}1]))$$

- *Rule Head Constraint Representation*: For each head constraint $C : i \in \bar{H}$,
  - If C is an atomic constraint, then there exists some $\texttt{LookupAtom}\ _\ C : i \in \vec{J}$.
  - If C is a comprehension pattern, i.e., $C = \llparenthesis A \mid \bar{g}' \rrparenthesis_{\vec{x} \in xs}$, then there exists some $\texttt{LookupAll}\ _\ A : i \in \vec{J}$, $\texttt{FilterGuard}\ i\ \bar{g}' \in \vec{J}$ and $\texttt{CompreDomain}\ i\ \vec{x}\ xs \in \vec{J}$.
- *Occurrence Scope Coverage*: For any $pc \in \mathit{range}\,(\vec{J})$, we have:

$$\mathit{jtOccs}\,(\vec{J}\,[pc]) \subseteq \llparenthesis j \mid \text{for all } A : j \in \mathit{jtRuleHeads}\,(\vec{J}\,[0 \ldots pc]) \rrparenthesis$$

Active head states that the leading join tasks must be of the form $\texttt{Active}\ _$. *Unique rule heads* specifies that each appearance of a rule head $A : j$ is unique in $\vec{J}$. *Uniqueness Enforcement* asserts that for any two rule heads i and j with the same predicate symbol, there must be a join task in $\vec{J}$ that enforces uniqueness between i and j. *Guard scope coverage* specifies that for each g that appears in $\vec{J}$ has its free variables appearing in rule heads that appear before it in $\vec{J}$. Finally, *occurrence scope coverage* states that every occurrence index j in $\vec{J}$ appears after rule head $A : j$. Lemma 1 states that given any rule heads of a CHR^{cp} rule, the operation *compileRuleHead* (Section 7) computes a valid join ordering of that rule head. Hence this proves that the operation *compileRuleHead* provides us the means of obtaining valid join orderings.

Lemma 1 (Building Valid Join Ordering) *For any rule heads of a CHR^{cp} rule, $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$, if $\vec{J} = \mathit{compileRuleHead}\ (C : i, \vec{H}_a, \vec{H}_m, \bar{g})$, then $\vec{J}$ is valid.*

We also define a notion of valid matching states: a matching state $\langle J; pc; \vec{Br}; \theta; Pm \rangle$ is *valid* with respect to a matching context $\langle A\#n; \vec{J}; Ls \rangle$ of a CHR^{cp} rule $r\ @\ \bar{H} \Longleftrightarrow \bar{g} \mid \bar{B}$ if and only if we have the following:

- *Valid Program Counter*: if $pc \in \mathit{range}\,(\vec{J})$ then $J = \vec{J}\,[pc - 1]$, otherwise $J = \bot$ and $pc - 1 \in \mathit{range}\,(\vec{J})$
- *Valid Partial Match*: For each $(i \mapsto \bar{U}) \in Pm$, the following two properties are satisfied:
  - *Valid Indices*: either
    - there exists some $\texttt{LookupAtom}\ _\ A : j \in \vec{J}\,[0 \ldots pc]$ such that $i = j$ and $\theta A \triangleq_{\mathbf{lhs}} \llparenthesis A\#n \mid \text{for all } (_, A\#n) \in \bar{U} \rrparenthesis$, or

- $*$ there exists some $\texttt{LookupAll}\ _\ A : j \in \vec{J}[0 \ldots pc]$ such that $i = j$ and $\theta A \triangleq_{\mathbf{lhs}} \wr A\#n \mid \text{for all } (_, A\#n) \in \bar{U} \wr$. Furthermore, for comprehension patterns (i.e., $M : i \in \bar{H}$), we have $M \triangleq^{\neg}_{\mathbf{lhs}} Ls - E$ such that $E = \wr A\#n \mid \text{for all } (j \mapsto \bar{U}) \in Pm \text{ s.t. } (_, A\#n) \in \bar{U} \wr$, or
  - $*$ if none of the above, then $\texttt{Active}\ C : i,\ \texttt{Bootstrap}\ _\ i \in \vec{J}[0 \ldots pc]$.
- *Valid Candidates*: For each $(i \mapsto \bar{U}) \in Pm$, we have $\wr A\#n \mid \text{for all } (_, A\#n) \in \bar{U} \wr \subseteq Ls$.

- *Valid Backtracking Branches*: For each $(pc', \theta', Pm') \in \vec{Br}$, a backtracked state $\langle \vec{J}[pc']; pc'+1; \varnothing; \theta'; Pm' \rangle$ must also be a valid state, with respect to the matching context $\langle A\#n; \vec{J}; Ls \rangle$.

Valid program counter asserts that the program counter pc of a valid state is always synchronized with the current join task J of that state. In particular, J is the pc^{th} element of the join ordering $\vec{J}$. *Valid partial match* defines two properties on the partial match Pm of a valid state: 1) Valid indices: Pm must contain mappings $j \mapsto \bar{U}$ for all head constraints j that appears before the current program counter (i.e., $\vec{J}[0 \ldots pc]$), unless j is a bootstrapped active pattern (i.e., $(\texttt{Bootstrap}\ _\ i) \in \vec{J}[0 \ldots pc]$). If j corresponds to a comprehension pattern, we additionally must have the property that candidates $\bar{U}$ contains the maximal set of candidates from Ls. 2) Valid candidates: candidates appearing in Pm (i.e., $(_ \mapsto \bar{U}) \in Pm$) must contain only constraints that appear in the store Ls. Finally, *valid backtracking branches* states that backtracking branches $\vec{Br}$ in a valid state only points to valid states.

Lemma 2 states that the transition operation $\Theta \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$ preserves the validity of states: given that Θ and $\mathcal{M}$ are valid, then $\mathcal{M}'$ is valid as well.

Lemma 2 (Preservation of Valid States) *For any valid matching context $\Theta = \langle A\#n; \vec{J}; Ls \rangle$ and a valid state (w.r.t. Θ) $\mathcal{M}$, for any reachable state $\mathcal{M}'$ such that $\Theta \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, $\mathcal{M}'$ must be valid.*

9.2 Termination

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, the execution either terminates with some terminal state (i.e., $\langle \bot; _; _; _; _ \rangle$), or returns $\bot$. This means that for any valid context Θ and state $\mathcal{M}$, we must have either $\Theta \rhd \mathcal{M} \rightarrowtail^{*}_{lhs} \mathcal{M}'$ for some terminal state $\mathcal{M}'$ or $\Theta \rhd \mathcal{M} \rightarrowtail^{*}_{lhs} \bot$.

To show this, we must first demonstrate that the transition operation $\rightarrowtail_{lhs}$ provides some form of progress guarantees. To measure progress, we define the function $progress(\Theta, \mathcal{M})$: Given a valid context $\langle _; \vec{J}; _ \rangle$, a valid state $\mathcal{M} = \langle _; pc; \vec{Br}; _; _ \rangle$, this function returns an $n+1$-tuple containing non-negative integers, known as the *progress ranking* of $\mathcal{M}$. For the tuple positions 1 to n, known as the *lookup progress values*, indicates the number of backtracking branches in $\vec{Br}$ that are created by n^{th} atomic partner constraint of $\vec{J}$. Position 0 (leftmost), known as the *active progress value*, indicates whether the current program counter pc has moved passed at least one $\texttt{LookupAtom}$ join task in $\vec{J}$. Finally, the rightmost tuple position ($n+1$), known as *program counter progress*, indicates the number of join tasks between pc and the end of $\vec{J}$. Figure 9.2 defines this function.

By using lexicographical order comparison between state progress (i.e., $progress(\Theta, \mathcal{M}) > progress(\Theta, \mathcal{M}')$), we show that the abstract matching machine transition operation monotonically decreases progress ranking of matching states. Lemma 3 defines this progress guarantee.

Lemma 3 (Monotonic Progress of Abstract Matching Machine) *For any valid context Θ and valid states $\mathcal{M}$, $\mathcal{M}'$, if $\Theta \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, then we have $progress(\Theta, \mathcal{M}) > progress(\Theta, \mathcal{M}')$.*

Theorem 4 is the main termination result of the abstract matching machine. For any valid matching context and matching state, the abstract matching machine, either evaluates to a terminal state ($\mathcal{M} = \langle \bot; _; _; \theta; Pm \rangle$) or exhaustively searched the store, resulting to $\bot$. The proof of this theorem relies on Lemma 3: because progress rank of matching states monotonically decreases, exhaustive derivations of abstract matching transitions are finite and must eventually terminate.

Theorem 4 (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 $\mathcal{M} = \bot$.*

$$
\begin{array}{lll}
\mathit{progress}\,(\vec{J}, \langle _; pc; \vec{Br}; _; _\rangle) & ::= & (\mathit{act_progress}, \mathit{lookup_progress}', \mathit{pc_progress}) \\
\quad \text{where} \quad \mathit{act_progress} & = & \text{if not exists any } \texttt{LookupAtom}\ _\ _ \in \vec{J}\,[0 \ldots pc] \text{ then } 1 \text{ else } 0 \\
\quad \mathit{lookup_progress} & = & \mathit{btProgress}\,(\vec{Br}, \mathit{btIndices}\,(\vec{J}, 0), 0) \\
\quad \mathit{lookup_progress}' & = & \mathit{pcModifier}\,(\mathit{lookup_progress}, \mathit{btIndices}\,(\vec{J}, 0), pc, 1) \\
\quad \mathit{pc_progress} & = & |\vec{J}| - pc
\end{array}
$$

$$
\begin{array}{lll}
\mathit{btIndices}\,([J \mid \vec{J}], pc) & ::= & \begin{cases} \mathit{btIndices}\,(\vec{J}, pc{+}1) \mathbin{++} [pc] & \text{if } \texttt{LookupAtom}\ _\ _ \\ \mathit{btIndices}\,(\vec{J}, pc{+}1) & \text{otherwise} \end{cases} \\
\mathit{btIndices}\,([\,], pc) & ::= & [\,]
\end{array}
$$

$$
\begin{array}{lll}
\mathit{btProgress}\,([(pc, \theta, Pm) \mid \vec{Br}], [pc' \mid \vec{pc}], s) & ::= & \begin{cases} \mathit{btProgress}\,(\vec{Br}, [pc' \mid \vec{pc}], s{+}1) & \text{if } pc = pc' \\ (\mathit{btProgress}\,([(pc, \theta, Pm) \mid \vec{Br}], \vec{pc}, 0), s) & \text{otherwise} \end{cases} \\
\mathit{btProgress}\,([\,], [pc \mid \vec{pc}], _) & ::= & (\mathit{btProgress}\,([\,], \vec{pc}, 0), 0) \\
\mathit{btProgress}\,(_, [\,], _) & ::= & 0 \\
\mathit{pcModifier}\,((\mathit{rest}, \mathit{curr}), [pc \mid \vec{pc}], pc', \mathit{mod}) & ::= & \begin{cases} (\mathit{pcModifier}\,(\mathit{rest}, \vec{pc}, pc', 0), \mathit{curr} + \mathit{mod}) & \text{if } pc < pc' \\ (\mathit{pcModifier}\,(\mathit{rest}, \vec{pc}, pc', \mathit{mod}), \mathit{curr}) & \text{otherwise} \end{cases} \\
\mathit{pcModifier}\,(_, [\,], _, \mathit{mod}) & ::= & \mathit{mod}
\end{array}
$$

Figure 9.2: State Progress Ranking Function

9.3 Soundness

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 correspond to head constraint match as specified by the semantics of matching of CHR^{cp} (Figure 4.1). The operation $\mathit{constr}\,(Pm, i)$ returns the multiset of all constraints in partial match Pm mapped by i.

Lemma 5 states that each reachable state $\mathcal{M}$ of a abstract matching machine execution of the matching context $\Theta = \langle A\#n; \vec{J}; Ls\rangle$ contains a partial match to rule head constraints and guards that appears in the prefix sequence of join tasks of $\vec{J}$ up to $\mathcal{M}$ current program counter.

Lemma 5 (Soundness of Abstract Matching Machine Transition Step) *For any CHR^{cp} head constraints $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$, such that $\vec{J} = \mathit{compileRuleHead}\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$, given a constraint store Ls and an active constraint $A\#n$, all reachable states $\mathcal{M} = \langle J; pc; \vec{Br}; \theta; Pm\rangle$, satisfies the following:*

1. Satisfied Guards*: $\models \theta \mathit{jtGuards}\,(\vec{J}\,[0 \ldots pc])$*
2. Partial Match*: For each $A : i \in \mathit{jtRuleHeads}\,(\vec{J}\,[0 \ldots pc])$, with corresponding rule head constraint $C : i \in \lbag \vec{H}_a, \vec{H}_m \rbag$, we have $C : i \triangleq_{\mathbf{lhs}} \mathit{constr}\,(Pm, i)$*
3. Maximality of Comprehension*: For each $A : i \in \mathit{jtRuleHeads}\,(\vec{J}\,[0 \ldots pc])$, such that there exists $\texttt{LookupAll}\ _\ A : i \in \vec{J}\,[0 \ldots pc]$, with corresponding rule head constraint $M : i \in \vec{H}_m$, we have $M \triangleq^{\neg}_{\mathbf{lhs}} Ls - E$ such that $E = \lbag A\#n \mid \text{for all } (j \mapsto \bar{U}) \in Pm\,[0 \ldots pc] \text{ s.t. } (_, A\#n) \in \bar{U} \rbag$*

Theorem 6 asserts the soundness result. Its proof relies on Lemma 1 to guarantee that any join ordering $\vec{J}$ constructed from *compileRuleHead* is valid, and on Lemma 2 and Lemma 5 to guarantee that reachable states are valid. Hence substitutions θ and partial matches Pm of reachable states fulfill the soundness requirements with respect to the semantics of matching of CHR^{cp}.

Theorem 6 (Soundness of Abstract Matching Machine) *For any CHR^{cp} head constraints $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$, such that $\vec{J} = \mathit{compileRuleHead}\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$, given a constraint store Ls and an active constraint $A\#n$,*

$$r\ @\ \begin{array}{l} p(D_0) : 1 \\ \lbag p(D_1) \rbag_{D_1 \in Xs} : 2 \\ \lbag p(D_2) \rbag_{D_2 \in Ys} : 3 \end{array} \iff \begin{array}{l} D_0 \dot{\in} Xs \\ D_0 \dot{\in} Ys \end{array} \mid \ldots$$

Optimal Join Ordering for Occurrence $p(D_0) : 1$

Task No.	Join Task
$i.$	`Active` $p(D_0) : 1$
$ii.$	`LookupAll` $\langle true; \emptyset \rangle\ p(D_1) : 2$
$iii.$	`FilterHead` 2 1
$iv.$	`CompreDomain` 2 D_1 Xs
$v.$	`CheckGuard` $D_0 \dot{\in} Xs$
$vi.$	`LookupAll` $\langle true; \emptyset \rangle\ p(D_2) : 3$
$vii.$	`FilterHead` 3 1
$viii.$	`FilterHead` 3 2
$ix.$	`CompreDomain` 3 D_2 Ys
$x.$	`CheckGuard` $D_0 \dot{\in} Ys$

Figure 9.3: Example of Incompleteness of Matching

if $\langle A\#n; \vec{J}; Ls \rangle \ \triangleright\ \langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing \rangle \ \rightarrowtail^*_{lhs}\ \langle \bot; _; _; \theta; Pm \rangle$*, then for some* Ls_{act}*,* Ls_{part}*,* Ls_{rest} *such that* $Ls = \lbag Ls_{act}, Ls_{part}, Ls_{rest} \rbag$ *and* $Ls_{act} = constr\ (Pm, i)$ *and* $Ls_{part} = constr\ (Pm, getIdx(\lbag \vec{H}_a, \vec{H}_m \rbag))$*, we have:*

1. Satisfied Guards*:* $\models \theta g$*,*
2. Active Pattern Match*:* $C : i \triangleq_{\mathbf{lhs}} Ls_{act}$*,*
3. Partners Match*:* $\theta \lbag \vec{H}_a, \vec{H}_m \rbag \triangleq_{\mathbf{lhs}} Ls_{part}$*, and*
4. Maximality of Comprehension*:* $\theta \lbag \vec{H}_a, \vec{H}_m, C : i \rbag \triangleq^{\neg}_{\mathbf{lhs}} Ls_{rest}$*.*

9.4 Completeness

We also wish to provide a completeness result for our abstract machine. Specifically, that for a CHR^{cp} rule r with a rule compilation $\vec{J}$:

$$\langle A\#n; \vec{J}; Ls \rangle \ \triangleright\ \langle \vec{J}\,[0]; 1; \emptyset; \cdot; \emptyset \rangle \ \rightarrowtail^*_{lhs}\ \bot$$

implies that no possible match of r with $A\#n$ as the active constraint exists in Ls.

However, the 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. This matching scheme is incomplete for CHR^{cp} rules that are *selective* on comprehension head constraints. A CHR^{cp} rule r with head constraints $\bar{H}_a$, $\bar{H}_m$ and $\bar{g}$ is selective on its comprehension head constraints if for any comprehension head constraint $\lbag A \mid \bar{g}_m \rbag_{x \in xs} : j \in \bar{H}_m$, we have the following:

- There exists $C : k \in \lbag \bar{H}_a, \bar{H}_m \rbag$ and $k \neq j$ such that $\bar{g} \triangleright C \not\triangleq^{\neg}_{\mathbf{unf}} \lbag A \mid \bar{g}_m \rbag_{x \in xs}$
- There exists $g \in \bar{g}$ such that $xs \in FV(g)$

The first condition implies that comprehension j possibly "competes" with head constraint k for constraints in the store Ls (their atomic constraint patterns are unifiable). The second condition states that comprehension domain of j (i.e., xs) appears as an argument of g. Hence satisfiability of g possible depends on a specific partitioning of constraints between head constraints j and k.

Figure 9.3 shows an example of this incompleteness: Consider the execution of this join ordering with the constraint store $Ls = \lbag p(2)\#n, p(2)\#m, p(2)\#h \rbag$ and the active constraint $p(2)\#n$. Ls indeed contains a match to this rule, namely $Pm = 1 \mapsto p(2)\#n, 2 \mapsto p(2)\#m, 3 \mapsto p(2)\#h$, since we have $Xs = \lbag 2 \rbag$ and $Ys = \lbag 2 \rbag$, and

hence $D_0 \dot{\in} Xs$ and $D_0 \dot{\in} Ys$. However, the abstract machine will not be able to compute this match because it implements a *greedy* comprehension matching scheme: by task v, occurrence 2 has been computed and finalized with $Pm\,[2] = \lbag p(2)\#m, p(2)\#h \rbag$, leaving none for occurrence 3 (i.e., $Pm\,[3] = \emptyset$) after task $(viii)$, `FilterHead` 3 2 is executed. In general, if we insist on guaranteeing completeness, given a join ordering where a multiset of constraints $\bar{A}$ can match with two unique comprehension patterns, H_j and H_k, our abstract machine must permute all possible partitions on $\bar{A}$ that splits between i and j. Such an execution scheme would have severe impact on performance. The 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 *greedy* comprehension matching scheme implemented by the abstract matching machine is, however, complete for a class of CHR^{cp} rules that are not *selective* on comprehensions. Lemma 7 asserts that if the abstract machine transition operation computes to $\bot$ (failure to find match), computation must have exhaustively backtracked and tested all possible matchings to atomic head constraints.

Lemma 7 (Exhaustive Backtracking For Atomic Head Constraints) *Let r be any CHR^{cp} rule and its head constraints be $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$ with $\vec{J} = compileRuleHead\ (C : i, \vec{H}_a, \vec{H}_m, \bar{g})$. Given a constraint store Ls and an active constraint $A\#n$, for any reachable state $\mathcal{M} = \langle J; pc; \vec{Br}; \theta; Pm\rangle$, we have:*

- *For some $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle$ such that $\langle A\#n; \vec{J}; Ls\rangle \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, if $J = \texttt{LookupAtom}\,_\,A : j$, then for all $A'\#n \in Ls$ such that $match\ (A', \theta A) = \phi\ (\phi \neq \bot)$, either we have $j \mapsto (\phi, A'\#n) \in Pm'$ or $(pc, \theta\phi, (Pm, j \mapsto (\phi, A'\#n))) \in \vec{Br}'$.*
- *If $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle J; pc; \vec{Br}; \theta; Pm\rangle \rightarrowtail^{*}_{lhs} \bot$, then for each $(pc', \theta', Pm') \in \vec{Br}$, there exists some $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle$ such that $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle J; pc; \vec{Br}; \theta; Pm\rangle \rightarrowtail^{s_1}_{lhs} \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle \rightarrowtail^{s_2}_{lhs} \bot$*

The first part of Lemma 7 asserts that the execution of an atomic head constraint $A : j$ (i.e., $\texttt{LookupAtom}\,_\,A : j$) must result to one matching constraint in the store $A'\#n \in Ls$ extended as the partial match of the successor state $\mathcal{M}'$, while all other matching constraints in the store are included as possible backtracking points. The second part asserts that if the overall exhaustive execution results in $\bot$ from some reachable state $\langle J; pc; \vec{Br}; \theta; Pm\rangle$, then each backtracking branch $(pc', \theta', Pm') \in \vec{Br}$ must have been restored as an intermediate state at some point of the exhaustive execution.

Theorem 8 expresses our completeness result for CHR^{cp} rule that is non-selective on comprehension rule heads.

Theorem 8 (Completeness of 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 $\vec{J} = compileRuleHead\ (C : i, \vec{H}_a, \vec{H}_m, \bar{g})$. Given a constraint store Ls and an active constraint $A\#n$, if $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing\rangle \rightarrowtail^{*}_{lhs} \bot$ then there exists no applicable rule instance of r from Ls.*

10 Operational Semantics with Join Ordering Execution

In this section, we integrate the join ordering execution defined in Section 8 with the operational semantics of CHR^{cp}. Specifically, we redefine the operational semantics of CHR^{cp} by means of the transition operation $\mathcal{P} \rhd \sigma \mapsto_{\omega+} \sigma'$. This operation is similar to the transition operation $\mapsto_{\omega}$ (Section 4.3) except that it defines the $(act\text{-}apply)$ and $(act\text{-}next)$ transitions in terms of compiled join orderings and execution. The result is an operational semantics that more closely resembles an actual implementation of the CHR^{cp} runtime.

Figure 10.1 defines this operational semantics. Note that rules other than $(act\text{-}apply)$ and $(act\text{-}next)$ are similar to those found in Figure 4.4. The rules $(act\text{-}apply)$ defines a successful match and application of the CHR^{cp} rule instance with active constraint $A\#n$ matching the ith head constraint of program $\mathcal{P}$ ($\mathcal{P}[i] = r\ @\ \lbag \vec{H}_a, \vec{H}_m, C : i \rbag \Longleftrightarrow \bar{g} \mid \bar{B}$). Rule $(act\text{-}next)$ handles the unsuccessful case, where a match is not possible. Both transitions are defined in terms of join ordering compilation and execution: the match to rule head constraints of CHR^{cp} rule r is computed by first compiling the relevant rule head constraints into the join ordering $\vec{J}$ ($\vec{J} = compileRuleHead\ (C : i, \vec{H}_a, \vec{H}_m, \bar{g})$).

$(init)$	$\mathcal{P} \rhd \langle$ `[init` $\lbag \bar{B}_l, \bar{B}_e \rbag$ `|` Gs`]` ; $Ls\rangle \mapsto_{\omega+} \langle lazy(St_l){+\!+}eager(Ls_e){+\!+}Gs$; $\lbag Ls, Ls_e \rbag\rangle$ 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 $eager(\lbag Ls, A\#n \rbag)$ ::= `[eager` $A\#n$ `|` $eager(Ls)$`]` $\quad eager(\varnothing)$ ::= `[]` $lazy(\lbag St_m, A \rbag)$ ::= `[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`]` ; $\lbag Ls, A\#n \rbag\rangle$ such that $\lbag A\#n \rbag = newLabels(Ls, \lbag A \rbag)$
$(eager{-}act)$	$\mathcal{P} \rhd \langle$ `[eager` $A\#n$ `|` Gs`]` ; $\lbag Ls, A\#n \rbag\rangle \mapsto_{\omega+} \langle$ `[act` $A\#n\,1$ `|` Gs`]` ; $\lbag Ls, A\#n \rbag\rangle$
$(eager{-}drop)$	$\mathcal{P} \rhd \langle$ `[eager` $A\#n$ `|` Gs`]` ; $Ls\rangle \mapsto_{\omega+} \langle Gs\,;\,Ls\rangle$ $\quad$ if $A\#n \notin Ls$
$(act{-}apply)$	$\mathcal{P} \rhd \langle$ `[act` $A\#n\,i$ `|` Gs`]` ; $\lbag Ls, Ls_h, Ls_a, A\#n \rbag\rangle \mapsto_{\omega+} \langle$ `[init` $\theta\bar{B}$ `|` Gs`]` ; $Ls\rangle$ For $\mathcal{P}[i] = (r \;@\; \lbag \vec{H}_a, \vec{H}_m, C : i \rbag \Longleftrightarrow \bar{g} \mid \bar{B})$ and $\vec{J} = compileRuleHead\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$ if $\langle A\#n; \vec{J}; \lbag Ls, Ls_h, Ls_a, A\#n \rbag\rangle \;\rhd\; \langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing\rangle \;\rightarrowtail^*_{lhs}\; \langle _; _; _; \theta; Pm\rangle$ such that $constr\,(Pm, i) = \lbag Ls_a, A\#n \rbag$ and $constr\,(Pm, getIdx(\lbag \vec{H}_a, \vec{H}_m \rbag)) = Ls_h$
$(act{-}next)$	$\mathcal{P} \rhd \langle$ `[act` $A\#n\,i$ `|` Gs`]` ; $Ls\rangle \mapsto_{\omega+} \langle$ `[act` $A\#n\,(i+1)$ `|` Gs`]` ; $Ls\rangle$ For $\mathcal{P}[i] = (r \;@\; \lbag \vec{H}_a, \vec{H}_m, C : i \rbag \Longleftrightarrow \bar{g} \mid \bar{B})$ and $\vec{J} = compileRuleHead\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$ if $\langle A\#n; \vec{J}; \lbag Ls, Ls_h, Ls_a, A\#n \rbag\rangle \;\rhd\; \langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing\rangle \;\rightarrowtail^*_{lhs}\; \bot$
$(act{-}drop)$	$\mathcal{P} \rhd \langle$ `[act` $A\#n\,i$ `|` Gs`]` ; $Ls\rangle \mapsto_{\omega+} \langle Gs\,;\,Ls\rangle$ $\quad$ if $\mathcal{P}[i] = \bot$

Figure 10.1: Operational Semantics of CHR^{cp} with Join Ordering Execution

This join ordering $\vec{J}$, together with the active constraint $A\#n$ and the current store $\lbag Ls, Ls_h, Ls_a, A\#n \rbag$ is used to compute a rule instance via the abstract matching machine transition (Section 8). The transition rule $(act{-}apply)$ states that if $\langle A\#n; \vec{J}; \lbag Ls, Ls_h, Ls_a, A\#n \rbag\rangle \;\rhd\; \langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing\rangle \;\rightarrowtail^*_{lhs}\; \langle _; _; _; \theta; Pm\rangle$, we have a successful match such that θ corresponds to the matching substitution and matching constraints are extracted from Pm. Otherwise ($\langle A\#n; \vec{J}; \lbag Ls, Ls_h, Ls_a, A\#n \rbag\rangle \;\rhd\; \langle \vec{J}\,[0]; 1; \varnothing; \cdot; \varnothing\rangle \;\rightarrowtail^*_{lhs}\; \bot$) and we apply the rule $(act{-}next)$ that iterates the search to the next rule head constraint (i.e., `act` $A\#n\,(i+1)$).

Corollary 9 defines the one-to-one correspondence between the two operational semantics. It directly follows from our results in Section 9 (i.e., Theorem 4, 6 and 8).

Corollary 9 (Correctness of Join Ordering) *Let $\mathcal{P}$ be a CHR^{cp} program such that all rules in $\mathcal{P}$ are not selective on comprehension head constraints. For any valid states σ and σ', $\mathcal{P} \;\rhd\; \sigma \mapsto^*_{\omega} \sigma'$ if and only if $\mathcal{P} \;\rhd\; \sigma \mapsto^*_{\omega+} \sigma'$.*

11 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 4.2) and uniqueness enforcement optimization (Section 5.3), we have deployed a conservative implementation of the relation test routine $\sqsubseteq_{\mathbf{unf}}$, discussed in [7].

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:

1. (OJO) Optimal Join Ordering (Section 7).
2. (Bt) Bootstrapping of active comprehension head constraints (Section 5.2).
3. $(Mono)$ Incremental Storage for monotone constraints (Section 4.2).

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.

Figure 11.1: Preliminary Experimental Results

4. $(Uniq)$ Non-unifiability test for uniqueness enforcement (Section 5.3).

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 Appendix B for actual code):

- **Swap** (Figure B.1): Swapping data, as describe in Section 2. Input size (s, d), where s is number of swaps and d is number of data constraints.
- **GHS** (Figure B.2): Distributed GHS algorithm [4] for finding minimal spanning tree. Input sizes (v, e) where v is number of vertices and e is number of edges.
- **HQSort** (Figures B.3 and B.4): Simulation of Hyper-Quicksort distributed sorting algorithm. Input sizes (n, i) where n is number of nodes and i number of integers in each node.

Figure 11.1 displays our experimental results. The 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 over 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 optimal 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 $+(OJO)$ come 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. We have not yet fully analyzed the reasons for this speedup.

12 Related Work

Compilation optimization for CHR has received a lot of attention. Efficient implementations are available in Prolog, HAL [5], Java [11] and even in hardware (via FPGA) [10]. 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[12], while an extension that allows computation of limited form of aggregates is discussed in [9]. 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 Meld [2] offers rich features including aggregates and a limited form of comprehension patterns. To the best of our knowledge, a low-level semantics for an efficient implementation of Meld has not yet been explored.

13 Conclusion and Future Works

In this report, 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 [5, 11, 10]. 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 [8] and develop a decentralized multiset rewriting language.

References

[1] D. Batory. The LEAPS Algorithm. Technical report, University of Texas at Austin, 1994.

[2] F. Cruz, M. P. Ashley-Rollman, S. C. Goldstein, Ricardo Rocha, and F. Pfenning. Bottom-Up Logic Programming for Multicores. In *DAMP'12*, January 2012.

[3] 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.

[4] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[5] 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.

[6] E. S. L. Lam and I. Cervesato. Constraint Handling Rules with Multiset Comprehension Patterns. In *CHR'14*, 2014.

[7] E. S. L. Lam and I. Cervesato. Reasoning about Set Comprehension. In *SMT'14*, 2014.

[8] E.S.L. Lam and I. Cervesato. Decentralized Execution of Constraint Handling Rules for Ensembles. In *PPDP'13*, pages 205–216, Madrid, Spain, 2013.

[9] J. Sneyers, P. V. Weert, T. Schrijvers, and B. Demoen. Aggregates in Constraint Handling Rules. In *ICLP'07*, pages 446–448, 2007.

[10] A. Triossi, S. Orlando, A. Raffaetà, and T. W. Frühwirth. Compiling CHR to parallel hardware. In *PPDP'12*, pages 173–184, 2012.

[11] 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.

[12] P. V. Weert, J. Sneyers, T. Schrijvers, and B. Demoen. Extending CHR with Negation as Absence. In *CHR'06*, pages 125–140, 2006.

A Proofs

Lemma 1 (Building Valid Join Ordering) For any rule heads of a CHR^{cp} rule, $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$, if $\vec{J} = \mathit{compileRuleHead}\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$, then $\vec{J}$ is valid.

Proof: We proof for any output $\vec{J}$, of the operation $\mathit{compileRuleHead}\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$ (Figure 7.1), each of the following properties:

- *Active Head*: Both cases of *compileRuleHead* constructs join ordering that begins with `Active _`. Hence this is proven.
- *Unique Rule Head*: The proof proceeds by structural induction on $\mathit{buildJoin}\,(\vec{H}, \Sigma, \bar{g}, \vec{H}_h)$ in *compileRuleHead*: with base case such that $\vec{H}$ which outputs the empty list (i.e., $\vec{J} = [\,]$) hence by default proven. We show that each inductive case exclusively constructs exactly one `LookupAtom` or `LookupAll` join task for each constraint in $\vec{H}_a$ and $\vec{H}_m$. Since $\vec{H}_a$ and $\vec{H}_m$ are valid CHR^{cp} rule head constraints, thus each must have unique occurrence indices. Hence this is proven.
- *Uniqueness Enforcement*: The proof proceeds by structural induction on $\mathit{buildJoin}\,(\vec{H}, \Sigma, \bar{g}, \vec{H}_h)$, with base case such that $\vec{H}$ which outputs the empty list (i.e., $\vec{J} = [\,]$), in which the property trivially holds. For inductive cases, if current head constraint is an atom (i.e., $\mathit{buildJoin}\,([A : j \mid \vec{H}], \Sigma, \bar{g}, \vec{H}_h)$) output join ordering $\vec{J}$ consist of $\mathit{neqHs}\,(A : j, \vec{H}_h)$ that constructs all `NeqHead` join tasks essential to enforce uniqueness of partner j. Similarly, if current head constraint is a comprehension (i.e., $\mathit{buildJoin}\,([\wr A \mid \bar{g}_m \int_{\vec{x} \in xs} : j \mid \vec{H}], \Sigma, \bar{g}, \vec{H}_h)$), output join ordering $\vec{J}$ consist of $\mathit{filterHs}\,(\wr A \mid \bar{g}_m \int_{\vec{x} \in xs} : j, \vec{H}_h)$ that constructs all `FilterHead` join tasks essential for enforcing uniqueness of partner j. Hence this is proven.
- *Guard Representation*: Similar to the above, by definition of the *buildJoin* operation, each guard condition of the rule is embedded in either a check guard join task (`CheckGuard`), filter guard join task (`FilterGuard`) or as an indexing directive (i.e., `LookupAtom` and `LookupAll`).
- *Guard Scope Coverage*: Similarly, the proof proceeds by structural induction on $\mathit{buildJoin}\,(\vec{H}, \Sigma, \bar{g}, \vec{H}_h)$. For inductive case, guards scheduled $\bar{g}_1$ (i.e., `CheckGuard` $\bar{g}_1$) are such that $(\bar{g}_1, _) = \mathit{scheduleGrds}\,(\Sigma, \bar{g})$. Since Σ contains the set of variables that appear in rule heads before the current, hence this is proven.
- *Rule Head Constraint Representation*: Similar to the above, by definition of the *buildJoin* operation, each atomic head constraint is represented as a `LookupAtom` join task, while each comprehension pattern is represented by a `LookupAll` join task, a series of `FilterGuard` join tasks and a `CompreDomain` join task.

- *Occurrence Scope Coverage*: For `Bootstrap` join task, we show that by definition of *compileRuleHead* for comprehensions, a `Bootstrap` join task is only created for head constraint j such that active pattern is j, hence `Bootstrap` $_\ j$ only appears after `Active` $A : j$. The rest of the proof, proves the other cases: it proceeds by structural induction on $buildJoin\ (\vec{H}, \Sigma, \bar{g}, \vec{H}_h)$. For inductive case, join tasks `NeqHead`, `FilterHead` and `CompreDomain` that contain current partner j appears after `LookupAtom` or `LookupAll` join task of j. For `NeqHead` and `FilterHead` created in this manner, only extracts partner indices from $\vec{H}_h$, which contains partners that appear before the current. Hence this is proven.

□

Lemma 2 (Preservation of Valid States) For any valid matching context $\Theta = \langle A\#n; \vec{J}; Ls \rangle$ and a valid state (w.r.t. Θ) $\mathcal{M}$, for any reachable state $\mathcal{M}'$ such that $\Theta \ \rhd\ \mathcal{M} \ \rightarrowtail_{lhs}\ \mathcal{M}'$, $\mathcal{M}'$ must be valid.

Proof: We show that given that $\Theta = \langle A\#n; \vec{J}; Ls \rangle$ and $\mathcal{M} = \langle J; pc; \vec{Br}; \theta; Pm \rangle$ are valid matching context and valid matching state, then we prove that $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm' \rangle$ satisfies each of properties of valid matching states. The proof are as follows, each proceeding by structural induction on the all derivation rule (except for $(fail\text{-}match)$ rule, since it does not qualify in the premise) of the transition operation (Figure 8.2):

- *Valid Program Counter*: For each derivation rule, except for $(backtrack)$, we increment program counter by one (i.e., $pc + 1$), hence we have two cases: 1) if $pc \in range\ (\vec{J})$, then for each of the applicable rules (all except $(fail\text{-}match)$ and $(backtrack)$) J is defined as $\vec{J}\,[pc]$, hence this is proven. 2) if $pc \notin range\ (\vec{J})$, then since we only increment pc, then we have proven $pc - 1 \in range\ (\vec{J})$ and $J = \vec{J}\,[pc] = \bot$. For $(backtrack)$ rule, since previous state $\mathcal{M}$ satisfies '*valid backtracking branches*' property, backtracked state satisfies this property by definition. Hence this is proven.

- *Valid Partial Match*: For rules $(check\text{-}guard)$, $(neq\text{-}head)$ and $(compre\text{-}dom)$, $Pm = Pm'$ hence this is trivially proven for those cases. For $(bootstrap)$ with the current join task `Bootstrap` $_\ j$, since partial match of $\mathcal{M}$, i.e., '$Pm, j \mapsto _$', is valid, we can safely remove $j \mapsto _$ from the partial match. Hence Pm is a valid partial match in $\mathcal{M}'$ and this is proven. $(active)$ extends current partial match Pm with $i \mapsto (\phi, A\#n)$ and increments pc by 1. Since $A\#n \in Ls$ and that current join task $J =$ `Active` $_ : i$, hence this is proven. For rules $(lookup\text{-}atom)$ and $(lookup\text{-}all)$ with current join task `LookupAtom` $_\ _ : i$ and `LookupAll` $_\ _ : i$ respectively, extends the partial match Pm with $i \mapsto \bar{U}$ such that $\bar{U}$ are candidates extracted from Ls, hence this is proven. Rules $(filter\text{-}guard)$ and $(filter\text{-}head)$ only modifies partial match by filtering candidates from an index i, hence resultant partial match is still valid. For i corresponding to a comprehension pattern (i.e., `LookupAll` $_\ A : i$ and exists some $M : i \in \bar{H}$), we additionally show that partial match $i \mapsto \bar{U} \in Pm$ is maximal, with respect to the fragment of store Ls not appearing in Pm (i.e., $Ls - E$ such that $E = \wr A\#n \mid$ for all $(j \mapsto \bar{U}) \in Pm$ s.t. $(_, A\#n) \in \bar{U}$ and $i \neq j \wr$): if current state is execution of a join task of form `LookupAll` $_\ A : i$, by definition of $(lookup\text{-}all)$ rule, $(i \mapsto \bar{U}) \in Pm$ and $\bar{U}$ will contain all constraints in Ls that matches with A. $(filter\text{-}guard)$ and $(filter\text{-}head)$ are the only other rules that may modify $(i \mapsto \bar{U}) \in Pm$, however for $(filter\text{-}guard)$, we only filter away matches that do not satisfy the comprehension guard, while for $(filter\text{-}head)$ we only filter away matches that appear else where in Pm, hence this is proven. Finally for $(backtrack)$, backtracked partial match Pm' must be valid, since $\mathcal{M}$ satisfies the valid backtracking branches property, hence this is proven.

- *Valid Backtracking Branches*: For rules $(active)$, $(check\text{-}guard)$, $(lookup\text{-}all)$, $(filter\text{-}guard)$, $(neq\text{-}head)$, $(filter\text{-}head)$, $(compre\text{-}dom)$ and $(bootstrap)$, backtrack branches $\vec{Br}$ are not modified, hence $\vec{Br} = \vec{Br}'$ and this is trivially proven. $(lookup\text{-}atom)$ extends the backtrack branches $\vec{Br}$ with $\vec{Br}'$. For each branch $(pc, \theta\phi, Pm, j \mapsto (\phi, A'' : m)) \in \vec{Br}$, we show that $\langle \vec{J}\,[pc]; pc + 1; \varnothing; \theta\phi; Pm, j \mapsto (\phi, A'' : m) \rangle$ is a valid state: the proofs of these are instances of the proof for valid program counter and valid partial match for the $(lookup\text{-}atom)$ rule (valid backtracking branches is omitted, since backtrack branch is empty). Hence, this is proven for $(lookup\text{-}atom)$. For $(backtrack)$, backtrack branches in $\mathcal{M}$ (i.e., $[_ \mid \vec{Br}]$) is reduced to $\vec{Br}$. Since $\vec{Br}$ must be valid backtrack branches, hence this is proven.

□

Lemma 3 (Monotonic Progress of Abstract Matching Machine) For any valid context Θ and valid states $\mathcal{M}$, $\mathcal{M}'$, if $\Theta \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, then we have $progress\,(\Theta, \mathcal{M}) > progress\,(\Theta, \mathcal{M}')$.

Proof: The proof proceeds by structural induction on the types of transition rules of the abstract matching machine transition operations: For valid context $\Theta = \langle A\#n; \vec{J}; Ls \rangle$ and valid state $\mathcal{M} = \langle J; pc; \vec{Br}; \theta; Pm \rangle$, we show that for $\Theta \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, we have $progress\,(\Theta, \mathcal{M}) > progress\,(\Theta, \mathcal{M}')$ in each of the following inductive cases:

- Rules $(active)$, $(check{-}guard)$, $(lookup{-}all)$, $(filter{-}guard)$, $(neq{-}head)$, $(filter{-}head)$, $(compre{-}dom)$ and $(bootstrap)$: Since for these cases, program counter is incremented, rightmost component of progress ranking of $progress\,(\Theta, \mathcal{M}')$ is decremented by 1, relative to $progress\,(\Theta, \mathcal{M})$ (i.e., $pc' = pc + 1$, hence $|\vec{J}| - pc > |\vec{J}| - pc'$). Furthermore, since $\vec{Br}$ remains unchanged (i.e., $\vec{Br} = \vec{Br}'$), hence middle values of progress rankings (i.e., $lookup_progress$ in Figure 9.2) must remain the same. For leftmost value ($act_progress$), that value for $progress\,(\Theta, \mathcal{M}')$ must be less than or equal to that of $progress\,(\Theta, \mathcal{M})$, by definition of the progress ranking function. Therefore we must have $progress\,(\Theta, \mathcal{M}) > progress\,(\Theta, \mathcal{M}')$.
- $(lookup{-}atom)$: Current join task $J = \texttt{LookupAtom}\ _\ A\#j$. Backtracking branch is expanded with $\vec{Br}'$ that contains candidates in Ls that matched with $A\#j$. Since $\vec{Br}'$ must be finite in size n, and each backtracking branches points to pc (i.e., $(pc, _, _) \in \vec{Br}'$), progress rank value corresponding to pc of $progress\,(\Theta, \mathcal{M})$ must be n less then successor state ($progress\,(\Theta, \mathcal{M}')$) However, progress rank value left of pc in successor state $progress\,(\Theta, \mathcal{M}')$ must be 1 less than $progress\,(\Theta, \mathcal{M})$, hence by lexicographical ordering, we still have $progress\,(\Theta, \mathcal{M}) > progress\,(\Theta, \mathcal{M}')$.
- $(backtrack)$: Backtrack branch $(pc', _, _)$ is removed, hence progress rank value corresponding to pc' of $progress\,(\Theta, \mathcal{M})$ is 1 more than the successor state $\mathcal{M}'$. Even though rightmost progress rank component (i.e., $pc_progress$) is relatively higher (i.e. $|\vec{J}| - pc < |\vec{J}| - pc'$), by lexicographical ordering, we still have $progress\,(\Theta, \mathcal{M}) > progress\,(\Theta, \mathcal{M}')$.
- $(fail{-}match)$: Not applicable, since premise of the lemma is not satisfied (i.e., $\mathcal{M}' \neq \bot$).

Hence we have proven this in all applicable cases. □

Theorem 4 (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 $\mathcal{M} = \bot$.

Proof: The proof for this relies on Lemma 3: progress ranking of initial state is finite, and monotonicity decremented as successor states are applied to the abstract matching machine transition operation. Since increments to individual progress ranking component values (during application of rule $(lookup{-}atom)$) must be finite (bound by size of store Ls), we can only decrement progress ranking by finite number of steps. Hence we must always eventually reach a terminal state $\mathcal{M} = \langle \bot; _; _; \theta; Pm \rangle$, or failed state $\mathcal{M} = \bot$. □

Lemma 5 (Soundness of Abstract Matching Machine Transition Step) 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 $A\#n$, all reachable states $\mathcal{M} = \langle J; pc; \vec{Br}; \theta; Pm \rangle$, satisfies the following:

1. *Satisfied Guards*: $\models \theta jtGuards\,(\vec{J}\,[0 \ldots pc])$
2. *Partial Match*: For each $A : i \in jtRuleHeads\,(\vec{J}\,[0 \ldots pc])$, with corresponding rule head constraint $C : i \in \lbag \vec{H}_a, \vec{H}_m \rbag$, we have $C : i \triangleq_{\mathbf{lhs}} constr\,(Pm, i)$

3. *Maximality of Comprehension*: For each $A : i \in \mathit{jtRuleHeads}\,(\vec{J}\,[0 \ldots pc]\,)$, such that exists $\texttt{LookupAll}\;_\;A : i \in \vec{J}\,[0 \ldots pc]$, with corresponding rule head constraint $M : i \in \vec{H}_m$, we have $M \triangleq^{\neg}_{\mathbf{lhs}} Ls - E$ such that $E = \wr A\#n \mid \text{for all } (j \mapsto \bar{U}) \in Pm\,[0 \ldots pc] \text{ s.t. } (_, A\#n) \in \bar{U} \int$

Proof: This Lemma is a rephrasing of Lemma 2 that focuses on the incremental construction of partial matches by reachable states. Its proof subsumed by the proof of Lemma 2. Hence we omit its full details, but provide the proof sketch: The proof proceeds by structural induction on the types of transition rules of the abstract matching machine transition operations. We show that the abstract machine transition operation preserves the required properties of over partial matches Pm. These properties can be directly inferred from properties of reachable states, asserted by Lemma 2. □

Theorem 6 (Soundness of Abstract Matching Machine) For any CHR^{cp} head constraints $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$, such that $\vec{J} = \mathit{compileRuleHead}\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$, given a constraint store Ls and an active constraint $A\#n$, if $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle \vec{J}\,[0]\,; 1; \varnothing; \cdot; \varnothing\rangle \rightarrowtail^{*}_{lhs} \langle \bot; _; _; \theta; Pm\rangle$, then for some Ls_{act}, Ls_{part}, Ls_{rest} such that $Ls = \wr Ls_{act}, Ls_{part}, Ls_{rest} \int$ and $Ls_{act} = \mathit{constr}\,(Pm, i)$ and $Ls_{part} = \mathit{constr}\,(Pm, \mathit{getIdx}(\wr \vec{H}_a, \vec{H}_m \int))$, we have:

1. *Satisfied Guards*: $\models \theta g$,
2. *Active Pattern Match*: $C : i \triangleq_{\mathbf{lhs}} Ls_{act}$,
3. *Partners Match*: $\theta \wr \vec{H}_a, \vec{H}_m \int \triangleq_{\mathbf{lhs}} Ls_{part}$, and
4. *Maximality of Comprehension*: $\theta \wr \vec{H}_a, \vec{H}_m, C : i \int \triangleq^{\neg}_{\mathbf{lhs}} Ls_{rest}$.

Proof: The proof relies on Lemma 1 which provides the guarantees that any join ordering $\vec{J}$ constructed by valid inputs to $\mathit{compileRuleHead}$ is valid, thus execution from such matching contexts are well behaved. Lemma 5 provides the property fulfilled by reachable states which incrementally computes substitutions θ and partial match Pm. Finally, we show that terminal states $\langle \bot; _; _; \theta; Pm\rangle$ are such that Pm contains all matches to the rule heads of rule r, while satisfying guard constraints and maximality of comprehensions (thanks to Lemma 5). Hence is this proven. □

Lemma 7 (Exhaustive Backtracking For Atomic Head Constraints) Let r be any CHR^{cp} rule and its head constraints be $C : i$, $\vec{H}_a$, $\vec{H}_m$ and $\bar{g}$ with $\vec{J} = \mathit{compileRuleHead}\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$. Given a constraint store Ls and an active constraint $A\#n$, for any reachable state $\mathcal{M} = \langle J; pc; \vec{Br}; \theta; Pm\rangle$, we have:

- For some $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle$ such that $\langle A\#n; \vec{J}; Ls\rangle \rhd \mathcal{M} \rightarrowtail_{lhs} \mathcal{M}'$, if $J = \texttt{LookupAtom}\;_\;A : j$, then for all $A'\#n \in Ls$ such that $\mathit{match}\,(A', \theta A) = \phi$ $(\phi \neq \bot)$, either we have $j \mapsto (\phi, A'\#n) \in Pm'$ or $(pc, \theta\phi, (Pm, j \mapsto (\phi, A'\#n))) \in \vec{Br}'$.
- If $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle J; pc; \vec{Br}; \theta; Pm\rangle \rightarrowtail^{*}_{lhs} \bot$, then for each $(pc', \theta', Pm') \in \vec{Br}$, there exists some $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle$ such that $\langle A\#n; \vec{J}; Ls\rangle \rhd \langle J; pc; \vec{Br}; \theta; Pm\rangle \rightarrowtail^{s_1}_{lhs} \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle \rightarrowtail^{s_2}_{lhs} \bot$

Proof: The proof proceeds as follows:

- The first property immediately follows by the definition of the $(\mathit{lookup{-}atom})$. Specifically, successor state $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle$ must be such that partial matches Pm' is extended with one possible match to $A : j$, while all others are included as backtracking branches in $\vec{Br}'$. Hence this is proven.
- We proof the second property by negation, suppose that there exists some backtrack branch $(pc', \theta', Pm') \in \vec{Br}$ that was never restored, hence the execution never computes the intermediate state $\langle J'; pc'; \vec{Br}'; \theta'; Pm'\rangle$.

However, the abstract matching machine is defined such that the only transition rule that removes backtracking branches is $(backtrack)$ and $\langle J'; pc'; \vec{Br}'; \theta'; Pm' \rangle$ must inevitably be computed as an intermediate state in order for execution to result to $\bot$. This is because the only transition rule that results to $\bot$ is $(fail{-}match)$, which requires backtracking branches to be empty (i.e., $\varnothing$) in order to qualify. Hence it must be the case that $(pc', \theta', Pm') \in \vec{Br}$ that was restored. Hence this is proven.

$\square$

Theorem 8 (Completeness of 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 $\vec{J} = compileRuleHead\,(C : i, \vec{H}_a, \vec{H}_m, \bar{g})$. Given a constraint store Ls and an active constraint $A\#n$, if $\langle A\#n; \vec{J}; Ls \rangle \rhd \langle \vec{J}[0]; 1; \varnothing; \cdot; \varnothing \rangle \rightarrowtail^*_{lhs} \bot$ then there exists no applicable rule instance of r from Ls.

Proof: We proof by negation: Assume that we have $\langle A\#n; \vec{J}; Ls \rangle \rhd \langle \vec{J}[0]; 1; \varnothing; \cdot; \varnothing \rangle \rightarrowtail^*_{lhs} \bot$, yet there exists a rule instance of r that is applicable in Ls. Lemma 7 states that for all constraint $A'\#m \in Ls$ that can match with some head constraint $\texttt{LookupAtom}\ _\ A'' : j \in \vec{J}$, must at some intermediate state $\mathcal{M}' = \langle J'; pc'; \vec{Br}'; \theta'; Pm' \rangle$ participated as in the partial match Pm' (i.e., $\theta A'' = A'$ and $j \mapsto (_, A'\#m) \in Pm'$). Lemma 1 ensures that join ordering $\vec{J}$ is valid with respect to the given CHR^{cp} rule, hence all head constraints and rule guards must be appropriately represented in $\vec{J}$ (*Rule head constraint representation* and *Guard representation*), and all combination of atomic head constraint matches must have been tried. The only possibility that execution has reached failed state $\bot$ is that all partial matched failed at one of the following: $(active)$ if active constraint did not match the active pattern, $(check{-}guard)$ by failing guard test $\models \theta\bar{g}$ or $(lookup{-}atom)$ if $lookCands$ retrieves an empty set of candidates. This means that all combinations of matches to atomic head constraints are not valid matches to rule r, deriving a contradiction to our original assumption. Hence it must be the case that there are no applicable rule instances of r. $\square$

B Experiment Program Code

In this section, we show the code of the three experiment programs discussed in Section 11. Because we use their concrete syntax in our prototype implementation, we first give an informal introduction to the concrete syntax of CHR^{cp}, highlighting the main difference with the abstract syntax. A comprehension $\lbag p(\vec{t}) \mid \bar{g} \rbag_{\vec{x} \in t}$ is written as $\{\ \texttt{p}(\vec{t})\ \mid\ \vec{x}\ \texttt{<-}\ t.\ \ \bar{g}\ \}$ in our concrete syntax. Comprehension guard (i.e., $\bar{g}$) is optional and will be omitted if they are not required. A CHR^{cp} rule $r\ @\ \bar{H} \Longleftrightarrow \bar{g} \mid \bar{B}$ is written as $\texttt{rule}\ r\ \texttt{::}\ \ \bar{H}\ \mid\ \bar{g}'\ \texttt{<==>}\ \bar{B}\ \texttt{where}\ \bar{g}''$, such that $\bar{g} = \lbag \bar{g}', \bar{g}'' \rbag$. Specifically, $\bar{g}'$ and $\bar{g}''$ represents a partition of $\bar{g}$ such that $\bar{g}''$ contains guards of the form $x = t$ such that x does not appear in $\bar{H}$, while $\bar{g}'$ contains all other guards of the rule. The intuition is that $\bar{g}''$ are essentially assignment bindings, akin to let bindings in functional programming languages. Both guard and '`where`' components of a rule are optional. Our concrete syntax include *propagation* head constraints $\bar{H}_p$, which are an optional component of CHR^{cp} rules. For instance, written as $\texttt{rule}\ r\ \texttt{::}\ \ \bar{H}_p\ \backslash \bar{H}_s\ \texttt{<==>}\ \bar{B}$, this syntax represents the CHR^{cp} rule $r @ \lbag \bar{H}_p, \bar{H}_s \rbag \Leftrightarrow \lbag \bar{H}_p, \bar{B} \rbag$ (although they are treated slightly differently from an operational perspective).

B.1 Pivot Swap

Figure B.1 shows the concrete implementation of the pivot swap example from Section 2.1, with and without comprehension rule head constraints. The program on top concisely implements pivot swap with a single CHR^{cp} rule that exploits comprehension patterns. The code at the bottom uses several rules that only references atomic constraint patterns. This involves using auxiliary constraints to accumulate intermediate matches (i.e., $grabGE$, $grabLT$ and $unrollData$) and implementing these boilerplate variable-sized constraint matching patterns by means of several rules.

Pivot Swap with Comprehensions:

```
1  rule pivotSwap :: swap(X,Y,P),
2                    { data(X,D) | D <- Xs. D >= P },
3                    { data(Y,D) | D <- Ys. D < P }
4                             <==> { data(Y,D) | D <- Xs }, { data(Y,D) | D <- Ys }.
```

Pivot Swap with Standard Rules

```
1  rule ini :: swap(X,Y,P) <==> grabGE(X,P,Y,[]), grabLT(Y,P,X,[]).
2
3  rule ge1 :: grabGE(X,P,Y,Ds), data(X,D) | D >= P <==> grabGE(X,P,Y,[D|Ds]).
4  rule ge2 :: grabGE(X,P,Y,Ds) <==> unrollData(Y,Ds).
5
6  rule lt1 :: grabLT(Y,P,X,Ds), data(Y,D) | D < P <==> grabLT(Y,P,X,[D|Ds]).
7  rule lt2 :: grabLT(Y,P,X,Ds) <==> unrollData(X,Ds).
8
9  rule unroll1 :: unrollData(L,[D|Ds]) <==> unrollData(L,Ds),data(L,D).
10 rule unroll2 :: unrollData(L,[]) <==> 1.
```

Figure B.1: Pivot Swap

B.2 Distributed Minimal Spanning Tree

The top part of Figure B.2 shows the concrete syntax for the distributed minimal spanning tree implementation, highlight in Section 2.4. This implementation relies on two auxiliary operations, namely `reduce_min` (line 5) and `union` (line 16), which are assumed to be side-effect free C++ functions: `reduce_min(Es)` returns the minimum value within `Es` while `union(Is1,Is2)` returns the union of the two collections `Is1` and `Is2`.

The bottom portion of Figure B.2 shows the concrete syntax for distributed minimal spanning tree with standard CHR^{cp} rules. Similar to the standard code in Figure B.1, this implementation uses auxiliary constraints (e.g., `seekMWOE` and `deleteEdges`) and multiset accumulator rules (e.g., `del1`, `del2`, etc..) in place of comprehension patterns.

B.3 Hyper-Quicksort

Figure B.3 shows the concrete syntax of an implementation of Hyper-Quicksort in CHR^{cp} (discussed in Section 2.3). Figure B.4 illustrates the implementation of Hyper-Quicksort in CHR^{cp} without the use of comprehension patterns. Similarly to the standard CHR code in Figures B.1 and B.2, this implementation utilizes auxiliary constraints and accumulator matching rules to implement the behavior of comprehension patterns.

GHS Algorithm with Comprehensions

```
1  rule find :: level(X,L) \ findMWOE(X,Is),
2                 { edge(I,O,V) | (I,O,V) <- Es. I in Is }
3                      | Es != {} <==> foundMWOE(X,Is), { edge(I,O,V) | (I,O,V) <- Rs },
4                                      combine(Om,X,L,Im,Om,Vm)
5                                      where (Im,Om,Vm) = reduce_min(Es),
6                                            Rs = { (I,O,V) | (I,O,V) <- Es. V != Vm }.
7
8  rule cmb1 :: combine(X,Y,L,O,I,V), combine(Y,X,L,I,O,V), level(X,L), level(Y,L)
9                 <==> merge(X,Y,I,O,V), level(X,L+1).
10
11 rule cmb2 :: level(X,L1) \ combine(X,Y,L2,I,O,V) | L1 > L2 <==> merge(X,Y,I,O,V).
12
13 rule mrg :: merge(X,Y,Im,Om,Vm), foundMWOE(X,Is1), foundMWOE(X,Is2),
14               { edge(I,O,V) | (I,O,V) <- Es1. I in Is1, O in Is2 },
15               { edge(I,O,V) | (I,O,V) <- Es2. I in Is2, O in Is1 }
16                  <==> findMWOE(X,union(Is1,Is2)), forward(Y,X),
17                       mstEdge(Im,Om,Vm), mstEdge(Om,Im,Vm).
18
19 rule fwd :: forward(O1,O2) \ combine(O1,X,L,I,O,V) <==> combine(O2,X,L,I,O,V).
```

GHS Algorithm with Standard Rules

```
1  rule start_find :: level(X,L) \ findMWOE(X,Is), edge(I,O,V) | I in Is
2                          <==> seekMWOE(X,Is,L,I,O,V,[]).
3  rule iter_find1 :: seekMWOE(X,Is,L,I1,O1,V1,Hs), edge(I2,O2,V2) | I2 in Is, V2 < V1
4                          <==> seekMWOE(X,Is,L,I2,O2,V2,[(I1,O1,V1)|Hs]).
5  rule iter_find2 :: seekMWOE(X,Is,L,I1,O1,V1,Hs), edge(I2,O2,V2) | I2 in Is, V2 >= V1
6                          <==> seekMWOE(X,Is,L,I1,O1,V1,[(I2,O2,V2)|Hs]).
7  rule end_find   :: seekMWOE(X,Is,L,Im,Om,Vm,Hs) <==> completeMWOE(X,Is,L,Im,Om,Vm,Hs).
8  rule comp_find1 :: completeMWOE(X,Is,L,Im,Om,Vm,[(I,O,V)|Hs])
9                          <==> edge(I,O,V),completeMWOE(X,Is,L,Im,Om,Vm,Hs).
10 rule comp_find2 :: completeMWOE(X,Is,L,Im,Om,Vm,[])
11                         <==> foundMWOE(X,Is), combine(Om,X,L,Im,Om,Vm).
12
13 rule cmb1 :: combine(X,Y,L,O,I,V), combine(Y,X,L,I,O,V), level(X,L), level(Y,L)
14                <==> merge(X,Y,I,O,V), level(X,L+1).
15
16 rule cmb2 :: level(X,L1) \ combine(X,Y,L2,I,O,V) | L1 > L2 <==> merge(X,Y,I,O,V).
17
18 rule mrg :: merge(X,Y,Im,Om,Vm), foundMWOE(X,Is1), foundMWOE(Y,Is2)
19                <==> deleteEdges(X,Y,Is1,Is2), mstEdge(Im,Om,Vm), mstEdge(Om,Im,Vm).
20 rule del1 :: deleteEdges(X,Y,Is1,Is2) \ edge(I,O,V) | I in Is1, O in Is2 <==> 1.
21 rule del2 :: deleteEdges(X,Y,Is1,Is2) \ edge(I,O,V) | I in Is2, O in Is1 <==> 1.
22 rule del3 :: deleteEdges(X,Y,Is1,Is2)
23                <==> [X]findMWOE(union(Is1,Is2)), [Y]forward(X).
24
25 rule fwd :: forward(O1,O2) \ combine(O1,X,L,I,O,V) <==> combine(O2,X,L,I,O,V).
```

Figure B.2: GHS Algorithm (Distributed Minimal Spanning Tree)

```
1  rule median :: { data(X,D) | D <- Ds } \ find_median(X)
2                      <==> median(X,computemedian(Ds)).
3
4  rule leader_reduce :: leaderLinks(X,G) | (count(G)) <= 1 <==> 1.
5
6  rule leader_expand :: median(X,M), leaderLinks(X,G)
7                            <==> { swapLink(Y,W,M,X) | (Y,W) <- zip(Gl,Gg) },
8                                 spawnLeaders(X,Z,Gl,Gg,count(Gl))
9                                 where (Gl,Gg) = split(G),
10                                       Z  = pickone(Gg).
11
12 rule swap :: swapLink(X,Y,M,L),
13              { data(X,D) | D <- Xs. D >= M },
14              { data(Y,D) | D <- Ys. D < M }
15                   <==> { data(X,D) | D <- Ys }, { data(Y,D) | D <- Xs },
16                        spawnCounter(L).
17
18 rule spawn :: spawnLeaders(X,Z,Gl,Gg,L), { spawnCounter(L) | 1 <- Cs }
19                 | (count(Cs)) == L <==> find_median(X), leaderLinks(X,Gl),
20                                         find_median(Z), leaderLinks(Z,Gg).
```

Figure B.3: Hyper-Quicksort with Comprehensions

```
rule find_median1 :: data(X,D), find_median(X,Ds) <==> find_median(X,[D|Ds]).
rule find_median2 :: find_median(X,Ds)
                         <==> ret_data(X,Ds), median(X,computemedian(Ds)).

rule acc1 :: ret_data(X,[D|Ds]) <==> ret_data(X,Ds), data(X,D).
rule acc2 :: ret_data(X,[]) <==> 1.

rule leader_reduce :: leaderLinks(X,G) | (len(G)) <= 1 <==> 1.

rule leader_expand :: median(X,M), leaderLinks(X,G)
                          <==> distPartnerLinks(X,M,zip(Gl,Gg)),
                               spawnLeaders(X,Z,Gl,Gg,len(Gl))
                               where (Gl,Gg) = split(G),
                                     Z = pickone(Gg).

rule partner1 :: distPartnerLinks(X,M,[(Y,W)|Gs])
                    <==> distPartnerLinks(X,M,Gs), partnerLink(Y,W,M,X,[],[]).
rule partner2 :: distPartnerLinks(X,M,[]) <==> 1.

rule swap1 :: partnerLink(X,W,M,L,Xs,Ys), data(X,D) | D >= M
                <==> partnerLink(X,W,M,L,[D|Xs],Ys).
rule swap2 :: partnerLink(X,W,M,L,Xs,Ys), data(Y,D) | D < M
                <==> partnerLink(X,W,M,L,Xs,[D|Ys]).
rule swap3 :: partnerLink(X,W,M,L,Xs,Ys)
                <==> ret_data2(X,Ys), ret_data2(W,Xs), spawnCounter(L,1).

rule acc3 :: ret_data2(X,[D|Ds]) <==> ret_data2(X,Ds), data(X,D).
rule acc4 :: ret_data2(X,[]) <==> 1.

rule spawn1 :: spawnCounter(L,X), spawnCounter(L,Y) <==> spawnCounter(L,X+Y).
rule spawn2 :: spawnLeaders(X,Z,Gl,Gg,L), spawnCounter(X,I)
                 | I == L <==> find_median(X,[]), leaderLinks(X,Gl),
                               find_median(Z,[]), leaderLinks(Z,Gg).
```

Figure B.4: Hyper-Quicksort with Standard Rules