PARALLEL EXECUTION OF CONSTRAINT HANDLING RULES - THEORY, IMPLEMENTATION AND APPLICATION

LAM SOON LEE EDMUND

(B.Science.(Hons), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING, DEPT OF COMPUTING SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2011

PARALLEL EXECUTION OF CONSTRAINT HANDLING RULES - THEORY, IMPLEMENTATION AND APPLICATION

LAM SOON LEE EDMUND

NATIONAL UNIVERSITY OF SINGAPORE

2011

PARALLEL EXECUTION OF CONSTRAINT HANDLING RULES - THEORY, IMPLEMENTATION AND APPLICATION	LAM SOON LEE EDMUND	2011

Acknowledgements

It has been four years since I embarked on this journey for a PhD degree in Computer Science. I would like to say that it was a journey of epic proportions, of death and destruction, of love and hope, with lots of state-of-the-art C.G work and cinematography which can make Steven Spielberg envious, and even concluded by a climatic final battle of dominance between good and evil. Unfortunately, this journey is of much lesser wonders and some might even consider it to be no more exciting than a walk in the park. That may be so, but it is by no means short of great noble characters who assisted me throughout the course of this fantastic four years and contributed to it's warm fuzzy end. Please allow me a moment, or two page counts, to thank these wonderful people.

Special thanks to my supervisor, Martin. Thank you for all great lessons and advices of how to be a good researcher, the wonderful travel opportunities to research conferences throughout Europe, also for his interesting and motivational analogies and lessons. I would be totally lost and clueless without his guidance. I'll like to thank Dr Dong, for his timely support in my research academic and administrative endeavors. My thanks also goes out to Kenny, who help me in my struggles with working in the Unix environment, among other of my silly trivial questions which a PhD student should silently Google or Wiki about, rather than openly ask others.

I'll like to thank the people who work in the PLS-II Lab, Zhu Ping, Meng, Florin, Corneliu, Hai, David, Beatrice, Cristina, and many others, as well as the people of the software engineering lab, Zhan Xian, Yu Zhang, Lui Yang, Sun Jun and others. They are all wonderful friends, and made my struggles in the lab more pleasant and

less lonely.

Many thanks to Jeremy, Greg, Peter and Tom. Fellow researchers whom visited us during my stay in NUS. Many thanks also to all the people whom reviewed my research papers. Even though some of their reviews were brutal and unnecessarily painful, I believe they have contributed in making me more responsible and humble in the conduct of my research.

My thanks goes out to Prof Chin, Prof Colin Tan and Prof Khoo, whom provided me with useful suggestions and feedback on my research works. I also wish to thank the thesis committee and any external examiner who was made to read my thesis, not by choice, but by the call of duty. Many thanks to all the NUS administration and academic staff. Without their support, conducive research environment and NUS's generous research scholarship, I would not have been able to complete my PhD programme. Thank you ITU Copenhagen for their more than warm welcoming during my research visit, especially the people of the Programming, Logic and Semantics group, Jacob, Anders, Kristian, Arne, Claus, Jeff, Hugo, Carsten, Lars and many others, including the huge office they so graciously handed to me.

Many thanks to my family, Carol, Henry, Andrew, Anakin and Roccat for their unconditional love, support and care all these years. Also thanks to Augustine, Joyce and Thomas for their friendship, wine and philosophical sparing sessions. Thank you, Adeline and family for their support and care.

Last but not least, many thanks to Karen, Jean, Phil, Cas and family, whom seen me through my last days as a PhD student. This work would not have been completed without their love and support.

Summary

Constraint Handling Rules (CHR) is a concurrent committed choice rule based programming language designed specifically for the implementation of incremental constraint solvers. Over the recent years, CHR has become increasingly popular primarily because of it's high-level and declarative nature, allowing a large number of problems to be concisely implemented in CHR.

The abstract CHR semantics essentially involves multi-set rewriting over a multi-set of constraints. This computational model is highly concurrent as theoretically rewriting steps over non-overlapping multi-sets of constraints can execute concurrently. Most intriguingly, this would allow for the possibility for implementations of CHR solvers with highly parallel execution models.

Yet despite of this, to date there is little or no existing research work that investigates into a parallel execution model and implementation of CHR. Further more, parallelism is going mainstream and we can no longer rely on super-scaling with single processors, but must think in terms of parallel programming to scale with symmetric multi-processors (SMP).

In this thesis, we introduce a concurrent goal-based execution model for CHR. Following this, we introduce a parallel implementation of CHR in Haskell, based on this concurrent goal-based execution model. We demonstrate the scalability of this implementation with empirical results. In addition, we illustrate a non-trivial application of our work, known as HaskellJoin, an extension of the popular high-level concurrency abstraction Join Patterns, with CHR guards and propagation.

Contents

List of Tables

List of Figures

List of Symbols

$\in$	Set/Multiset membership
$\subseteq$ / $\subset$	Set/Multiset Subset
$\cup$	Set union
$\cap$	Set intersection
$\uplus$	Multiset union
$\emptyset$	Empty set
$\exists$	Existential quantification
$\models$	Theoretic entailment
$\wedge$	Theoretic conjunction
ϕ	Substitution
x, y, z	Variables
m, n, q	Integers
$\overline{a}$	Set/Multiset of a's
$\rightarrowtail_{\mathcal{A}}$	Abstract derivation step
$\rightarrowtail^*_{\mathcal{A}}$	Abstract derivation steps
$\rightarrowtail_{\mathcal{G}}$	Goal-based derivation step
$\rightarrowtail^*_{\mathcal{G}}$	Goal-based derivation steps
$\rightarrowtail_{\|\|\mathcal{G}}$	Concurrent Goal-based derivation step
$\rightarrowtail^*_{\|\|\mathcal{G}}$	Concurrent Goal-based derivation steps

Chapter 1

Introduction

1.1 Motivation

Rewriting is a powerful discipline to specify the semantics of programming languages and to perform automated deduction. There are numerous flavors of rewriting such as term, graph rewriting etc. Our focus here is on exhaustive, forward chaining, multi-set constraint rewriting as found in Constraint Handling Rules (CHR) [19].

Constraint Handling Rules (CHR) is a concurrent committed choice rule based programming language designed specifically for the implementation of incremental constraint solvers. Over the recent years, CHR has become increasingly popular primarily because of it's high-level and declarative nature, allowing a large number of problems to be concisely implemented in CHR. From typical applications of constraint solving, CHR has been used as a general purpose programming language in many applications from unprecedented fields, from agent programming [56], biological sequence analysis [4] to type inference systems [10].

The abstract CHR semantics essentially involves multi-set rewriting over a multi-set of constraints. This computational model is highly concurrent, as theoretically rewriting steps over non-overlapping multi-sets of constraints can execute concurrently. Most intriguingly, this would allow for the possibility for implementations of

CHR solvers with highly parallel execution models.

Yet despite of the abstract CHR semantics' highly concurrent property, to date there are little or no existing research work that investigates into a concurrent[1] execution model or parallel[2] implementation of CHR. Existing CHR execution models are sequential in nature and often motivated by other implementation issues orthogonal to concurrency or parallelism. For instance, the refined operational semantics of CHR [11] describes a goal-based execution model for CHR programs, where constraints are matched to CHR rules in a fixed sequential order. The rule-priority operational semantics of CHR [33] is similar to the former, but explicitly enforces user-specified rule priorities on goal-based execution of constraints. Nearly all existing CHR systems implement either one of (or variants of) the above execution models, hence can only be executed sequentially.

Further more, parallelism is going mainstream. The development trend of high-performance micro-processor has shifted from the focus on super-scalar architectures to multi-core architectures. This means that we can no longer rely on super-scaling with single processors, but must think in terms of parallel programming to scale with symmetric multi-processors (SMP). We believe that much can be gained from deriving a parallel execution model and parallel implementation of CHR. Specifically, existing applications written as CHR programs can possibly enjoy performance speed ups implicitly when executed on multi-core systems, while applications that deal with asynchronous coordination between multiple agents (threads or processes) can be actually implemented and executed in parallel[3] in CHR (See Section 2.2.3 for examples).

Our last (but not least) motivation for our work lies in a high-level concurrency abstraction, known as *Join Calculus* [18]. *Join Calculus* is a process calculus aimed at providing a simple and intuitive way to coordinate concurrent processes via reaction

[1] **Concurrency** refers to the theoretical property, where CHR derivation is driven by multiple steps of reduction that can occur in arbitrary ordering.

[2] **Parallelism** refers to physically executing CHR programs with multiple CPU cores.

[3] as oppose to be simulated

rules known as Join-Patterns. Interestingly, the execution model of CHR rewriting is very similar to that of Join-Patterns with guard conditions, for which to date no efficient parallel implementation of it's concurrent execution model exists. As such, an understanding of the challenges of implementation a parallel execution model for CHR, would be almost directly applicable to Join Patterns with guards.

These are exactly the goals of this thesis. To summarize, we have four main goals:

- To derive a concurrent execution model that corresponds to the abstract CHR semantics.
- To develop a parallel implementation of CHR that implements this parallel execution model.
- To show that existing CHR applications could benefit from this parallel execution model.
- To demonstrate new concurrent applications can be suitably implemented in our parallel implementation of CHR.

1.2 Contributions

Our main contributions are as follows:

- We derive a parallel goal-based execution model, denoted $\| \mathcal{G}$, that corresponds to the abstract CHR semantics. This execution model is similar to that of in [11] in that it defines execution of CHR rewritings by the execution of CHR constraints (goals), but differs greatly because it allows execution of concurrent goals.
- We prove that $\| \mathcal{G}$ corresponds to the abstract CHR semantics. This is achieved by proving a correspondence between $\| \mathcal{G}$ execution steps and the abstract CHR semantics (denoted $\mathcal{A}$) derivation steps.

- We develop an implementation of $\| \mathcal{G}$, known as ParallelCHR , in the functional programming language Haskell. This implementation exploits lessons learnt in [54] and utilizes various concurrency primitives in Haskell to achieve optimal results.
- We derive a parallel execution model for Join-Patterns [14], via adaptions from $\| \mathcal{G}$.
- We extend Join-Patterns with CHR features, specifically, we add guards and propagation.
- We develop an implementation of Join-Patterns with guards and propagation, known as HaskellJoin .
- We provide empirical evidence that our implementations (ParallelCHR and HaskellJoin) scale well with the number of executing shared-memory processors.

1.3 Outline of this Thesis

This thesis is organized as follows.

In Chapter 2, we provide a detailed background of Constraint Handling Rules. We will introduce CHR via examples and illustrate the concurrency of CHR rewritings. This is followed by formal details of it's syntax and abstract semantics.

In Chapter 3, we formally introduce our concurrent goal-based CHR semantics, $\| \mathcal{G}$. Additionally, we provide a proof of it's correspondence with the abstract CHR semantics.

In Chapter 4, we detail our parallel CHR implementation in Haskell. Here, we explain the various technical issues and design decisions that make this implementation non-trivial. We will also highlight our empirical results that shows that this implementation scales with the number of executing processors.

In Chapter 5, we introduce a non-trivial application of our work, Join-Patterns with guards and propagation. We will first briefly introduce Join-Patterns and motivate the case for extending with guards and propagation. Following this, we will illustrate how we use the concurrent CHR goal-based semantics as a computational model to implement Join-Patterns with guards and propagation.

In Chapter 6, we discuss the existing works related to ours, from other similar approaches to parallel programming (eg. parallel production rule systems), existing works that addresses parallelism in CHR, to existing Join-Pattern extensions that shares similarities with ours.

Finally in Chapter 7, we conclude this thesis.

Chapter 2

Background

2.1 Chapter Overview

In this Chapter, we provide a detailed background of Constraint Handling Rules. We will introduce CHR via examples (Section 2.2.1) and illustrate the concurrency of CHR rewritings (Section 2.2.2). This is followed by formal details of it's syntax and abstract semantics (Section 2.2.4). We will also highlight an existing CHR execution model (Section 2.2.5) known as the refined CHR operational semantics, and finally provide brief details of our work (Section 2.3).

Readers already familiar to CHR may choose to skip Section 2.2 of this chapter.

2.2 Constraint Handling Rules

2.2.1 CHR By Examples

Constraint Handling Rules (CHR) is a concurrent committed choice rule based programming language originally designed specifically for the implementation of incremental constraint solvers. The CHR semantics essentially describes exhaustive forward chaining rewritings over a constraint multi-set, known as the *constraint store*. Rewriting steps are specified via CHR rules which replace a multi-set of con-

straints matching the left-hand side of a rule (also known as rule head) by the rule's right-hand side (also known as rule body). The following is an example of a CHR rule:

$$get \ @ \ Get(x), Put(y) \iff x = y$$

This CHR rule models a simple communications buffers. A $Get(x)$ constraint represents a call to retrieve an item from the buffers, while $Put(y)$ represents a call to place an item to the buffers. The symbol *get* is the rule name, used to uniquely identify a CHR rule in the CHR program. $Get(x), Put(y)$ is the rule head, while $x = y$ is the rule body. This CHR rule simply states that any matching occurrence of $Get(x), Put(y)$ can be rewritten to $x = y$, by applying the appropriate substitutions of x and y. A CHR program is defined by a set of CHR rules and an initial constraint store. For instance, treating *get* as a singleton rule CHR program and starting from the initial store $\{Get(m), Get(n), Put(1), Put(8)\}$, we rewrite the store with the *get* rule via the following rewrite steps (also referred to as derivation steps):

Step	**Substitution**		**Constraint Store**
			$\{Get(m), Get(n), Put(1), Put(8)\}$
$D1a$	$\{x = m, y = 1\}$	$\rightarrowtail_{get}$	$\{Get(n), Put(8), m = 1\}$
$D2a$	$\{x = n, y = 8\}$	$\rightarrowtail_{get}$	$\{m = 1, n = 8\}$

Derivation steps are denoted by $\rightarrowtail$ which maps constraint store to constraint store. Each derivation step represents the *firing* of a CHR rule, and are annotated by the name of the respective rule that fired. We will omit rule head annotations if there are no ambiguities caused. Derivation step $D1a$ matches subset $\{Get(m), Put(1)\}$ of the constraint store to the rule head of *get* via the substitution $\{x = m, y = 1\}$. We refer to $\{Get(m), Put(1)\}$ as a *rule head instance* of *get*. Hence it rewrites $\{Get(m), Put(1)\}$ to $\{m = 1\}$. Derivation step $D2a$ does the same for $\{Get(n), Put(8)\}$. The store $\{m = 1, n = 8\}$ is known as a *final store* because no rules in the CHR program can apply on it.

Note that we could have rewritten the same initial store via the following deriva-

tion steps instead:

Step	**Substitution**		**Constraint Store**
			$\{Get(m), Get(n), Put(1), Put(8)\}$
$D1b$	$\{x = m, y = 8\}$	$\rightarrowtail_{get}$	$\{Get(n), Put(1), m = 8\}$
$D2b$	$\{x = n, y = 1\}$	$\rightarrowtail_{get}$	$\{m = 8, n = 1\}$

In this case, derivation step $D1b$ matches the subset $\{Get(m), Put(8)\}$ instead of $\{Get(m), Put(1)\}$. This is followed by derivation step $D2b$, where the remaining pair $\{Get(n), Put(1)\}$ is rewritten to $n = 1$. As a result, a different final store is obtained. The CHR semantics is committed-choice because both sequences of derivation steps leading up to the distinct final stores, are valid computations according to the semantics [1]. As such, the CHR semantics is non-deterministic since an initial CHR store and program can possibility yield multiple final stores, depending on which derivation paths were taken. Interestingly, it is such non-determinism of the semantics which makes it a highly concurrent computational model. We will defer details of CHR and concurrency to Section 2.2.2.

The CHR language also includes guards and propagated constraints in the rule heads. The following shows a CHR program that utilizes such features:

$$gcd1 \ @ \ Gcd(n) \setminus Gcd(m) \iff m \geq n \&\& n > 0 \mid Gcd(m - n)$$
$$gcd2 \ @ \ Gcd(0) \iff True$$

Given an initial constraint store consisting of a set of *Gcd* constraints (each representing a number), this CHR program computes the greatest common divisor, by applying Euclid's algorithm. Rule head of *gcd1* has two components, namely *propagated* and *simplified* heads. Propagated heads ($Gcd(n)$) are to the left of the $\setminus$ symbol, while simplified heads ($Gcd(m)$) are to the right. Semantically, for a CHR rule to fire, propagated heads must be matched with a unique constraint in the

[1] Further more, the semantics does not natively specify any form of backtracking, or search across multiple possibilities of derivations.

store, but that constraint will not be removed from the store when the rule fires. Guard conditions ($m \geq n \&\& n > 0$) are basically boolean conditions with variables bounded by variables in the rule head. Given a CHR rule head instance, the rule guard under the substitution of the rule head must evaluate to true for the CHR rule to fire. We assume that evaluation of CHR guards are based on some built-in theory (For instance, in this example, we assume linear inequality). The following shows a valid derivation step, followed by a non-derivation step of the *gcd1* rule:

$$\{Gcd(1), Gcd(3)\} \rightarrowtail_{gcd1} \{Gcd(1), Gcd(3-1)\} \qquad \{n = 1, m = 3\}$$

$$\{Gcd(0), Gcd(2)\} \not\rightarrowtail_{gcd1} \{Gcd(0), Gcd(2-0)\} \qquad \{n = 0, m = 2\} \text{ and } n \not\geq 0$$

The first rule is valid because $Gcd(1), Gcd(3)$ matches rule heads of *gcd1*, while the rule guard instances evaluates to true. Note that $Gcd(1)$ is propagated (ie. not deleted) The second is not valid because the rule guard instance is not evaluated to true, even if $Gcd(0), Gcd(2)$ matches rule heads of *gcd1*.

The following illustrates the derivation steps that represents the exhaustive application of the rules *gcd1* and *gcd2* over an initial store of *Gcd* constraints. The result is the greatest common divisor of the initial store.

Step	**Substitution**		**Constraint Store**
			$\{Gcd(9), Gcd(6), Gcd(3)\}$
$D1$	$\{n = 6, m = 9\}$	$\rightarrowtail_{gcd1}$	$\{Gcd(3), Gcd(6), Gcd(3)\}$
$D2$	$\{n = 3, m = 6\}$	$\rightarrowtail_{gcd1}$	$\{Gcd(3), Gcd(3), Gcd(3)\}$
$D3$	$\{n = 3, m = 3\}$	$\rightarrowtail_{gcd1}$	$\{Gcd(0), Gcd(3), Gcd(3)\}$
$D4$	$\emptyset$	$\rightarrowtail_{gcd2}$	$\{Gcd(3), Gcd(3)\}$
$D5$	$\{n = 3, m = 3\}$	$\rightarrowtail_{gcd1}$	$\{Gcd(0), Gcd(3)\}$
$D6$	$\emptyset$	$\rightarrowtail_{gcd2}$	$\{Gcd(3)\}$

Derivation step $D1$ illustrates the firing of an instance of rule *gcd1* matching on the constraints $\{Gcd(9), Gcd(6)\}$, where $Gcd(6)$ is propagated and $Gcd(9)$ is simplified.

This introduces the constraint $Gcd(9-6)$. Note that for brevity, we omit built-in solving (reduction) steps, thus $Gcd(9-6)$ is treated immediately as $Gcd(3)$. Note that we cannot match the two constraints by propagating $Gcd(9)$ and simplifying $Gcd(6)$ because this requires the substitution $\{n = 9, m = 6\}$ which will make the rule guard inconsistent (ie. $m \not\geq n$).

Derivation steps $D2$ and $D3$ similarly shows the firing of instances of the *gcd1* rule, $\{Gcd(3), Gcd(6)\}$ and $\{Gcd(3), Gcd(3)\}$ respectively. In derivation step $D4$, $\{Gcd(0)\}$ in the store matches the rule head of *gcd2* hence rewrites to the *True* constraint, which we shall omit. Finally, derivation steps $D5$ and $D6$ follows in similar ways.

We will denote transitive derivation steps with $\rightarrowtail^*$. In other words, we can use: $\{Gcd(9), Gcd(6), Gcd(3)\} \rightarrowtail^* \{Gcd(3)\}$ To summarize the derivation sequence $D1 - D6$.

2.2.2 CHR and Concurrency

As demonstrated in Section 2.2.1, the abstract CHR semantics is non-deterministic and highly concurrent. Rule instances can be applied concurrently as long as they do not interfere. By interfere, we mean that they simplify (delete) distinct constraints in a store. In other words, they do not content for the same resources by attempting to simplify the same constraints.

Figure 2.1 illustrates this concurrency property via our earlier examples, communication buffer and greatest common divisor. We indicate concurrent derivations via the symbol $\|$. Given derivation steps $\{Gcd(m), Put(1)\} \rightarrowtail_{get} \{m = 1\}$ and $\{Gcd(n), Put(8)\} \rightarrowtail_{get} \{n = 8\}$, we can straightforwardly combine both derivations which leads to the final store $\{m = 1, n = 8\}$. The gcd example shows a more complex parallel composition: we combine the derivations $\{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\}$ and $\{Gcd(3), Gcd(18)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(15)\}$ in a way that they share only propagated components (ie. $Gcd(3)$). The resultant parallel deriva-

Communication channel:

$$get \ @ \ Get(x), Put(y) \Longleftrightarrow x = y$$

$$\frac{\{Get(m), Put(1)\} \rightarrowtail_{get} \{m = 1\} \parallel \{Get(n), Put(8)\} \rightarrowtail_{get} \{n = 8\}}{\{Get(m), Put(1), Get(n), Put(8)\} \rightarrowtail^* \{m = 1, n = 8\}}$$

Greatest common divisor:

$$gcd1 \ @ \ Gcd(0) \Longleftrightarrow True$$
$$gcd2 \ @ \ Gcd(n) \backslash Gcd(m) \Longleftrightarrow m >= n \&\& n > 0 \mid Gcd(m - n)$$

$$\frac{\begin{array}{l} \{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\} \\ \parallel \\ \{Gcd(3), Gcd(18)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(15)\} \end{array}}{\begin{array}{lll} \{Gcd(3), Gcd(9), Gcd(18)\} & \rightarrowtail_{gcd2,gcd2} & \{Gcd(3), Gcd(6), Gcd(15)\} \\ & \rightarrowtail^* & \{Gcd(3)\} \end{array}}$$
$$\{Gcd(3), Gcd(9), Gcd(18)\} \rightarrowtail^* \{Gcd(3)\}$$

Figure 2.1: Communication channel and greatest common divisor

tion is consistent since the propagated components are not deleted.

Recall in Section 2.2.1, for the communication buffer example, we have another possible final store $\{n = 1, m = 8\}$, that can be derived from the initial store $\{Get(m), Put(1), Get(n), Put(8)\}$. The abstract CHR semantics is non-deterministic and can possibly yield more than one results for a particular domain. The Gcd example on the other hand, is an example of a domain which is *confluent*. This means that rewritings over overlapping constraint sets are always joinable, thus a unique final store can be guaranteed. The communications buffer on the other hand is an example of a non-confluent CHR program. In general, (non)confluence of a CHR program is left to the programmer if desired. We will address issues of confluence with more details in Chapter 3.

Our approach extends from the abstract CHR semantics [19] (formally defined later in Section 2.2.4) which is inherently indeterministic. Rewrite rules can be applied in any order and thus CHR enjoy a high degree of concurrency.

An important property in the CHR abstract semantics is *monotonicity*. Illus-

$$\textbf{(Concurrency)} \quad \frac{S \uplus S_1 \rightarrowtail^* S \uplus S_2 \quad S \uplus S_3 \rightarrowtail^* S \uplus S_4}{S \uplus S_1 \uplus S_3 \rightarrowtail^* S \uplus S_2 \uplus S_4}$$

Figure 2.2: Concurrency of CHR Abstract Semantics

trated in Theorem 1, monotonicity of CHR execution guarantees that derivations of the CHR abstract semantics remain valid if we include a larger context (eg. $A \rightarrowtail^* B$ is valid under the additional context of constraints S, hence $A \uplus S \rightarrowtail^* B \uplus S$). This has been formally verified in [47].

Theorem 1 (Monotonicity of CHR) *For any sets of CHR constraints A,B and S, if* $A \rightarrowtail^* B$ *then* $A \uplus S \rightarrowtail^* B \uplus S$

An immediate consequence of monotonicity is that concurrent CHR executions are sound in the sense that their effect can be reproduced using an appropriate sequential sequence of execution steps. Thus, we can immediately derive the concurrency rule, illustrated in Figure 2.2. This rule essentially states that CHR derivations which affect different parts of the constraint store can be composable (ie. joined as though that occur concurrently). In [20], the above is referred to as "Strong Parallelism of CHR". However, we prefer to use the term "concurrency" instead of "parallelism". In the CHR context, concurrency means to run a CHR program (i.e. a set of CHR rules) by using concurrent execution threads.

Our last example in Figure 2.3 is a CHR encoding of the well-known merge sort algorithm. To sort a sequence of (distinct) elements $e_1, ..., e_m$ where m is a power of 2, we apply the rules to the initial constraint store $Merge(1, e_1), ..., Merge(1, e_m)$ Constraint $Merge(n, e)$ refers to a sorted sequence of numbers at level n whose smallest element is e. Constraint $Leq(a, b)$ denotes that a is less than b. Rule *merge2* initiates the merging of two sorted lists and creates a new sorted list at the next level. The actual merging is performed by rule *merge1*. Sorting of sublists belonging to different mergers can be performed simultaneously. See the example

$$merge1 \ @ \ Leq(x,a) \setminus Leq(x,b) \iff a < b \mid Leq(a,b)$$
$$merge2 \ @ \ Merge(n,a), Merge(n,b) \iff a < b \mid Leq(a,b), Merge(n+1,a)$$

Shorthands: $L = Leq$ and $M = Merge$

$$\begin{array}{rl}
 & M(1,a), M(1,c), M(1,e), M(1,g) \\
\rightarrowtail_{merge2} & M(2,a), M(1,c), M(1,e), L(a,g) \\
\rightarrowtail_{merge2} & M(2,a), M(2,c), L(a,g), L(c,e) \\
\rightarrowtail_{merge2} & M(3,a), L(a,g), L(c,e), L(a,c) \\
\rightarrowtail_{merge1} & M(3,a), L(a,c), L(c,g), L(c,e) \\
\rightarrowtail_{merge1} & M(3,a), L(a,c), L(c,e), L(e,g) \\
\multicolumn{2}{c}{\|} \\
 & M(1,b), M(1,d), M(1,f), M(1,h) \\
\rightarrowtail^* & M(3,b), L(b,d), L(d,f), L(f,h)
\end{array}$$

$$\begin{array}{rl}
 & M(3,a), L(a,c), L(c,e), L(e,g), M(3,b), L(b,d), L(d,f), L(f,h) \\
\rightarrowtail_{merge2} & M(4,a), L(a,c), L(a,b), L(c,e), L(e,g), L(b,d), L(d,f), L(f,h) \\
\rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,e), L(e,g), L(b,d), L(d,f), L(f,h) \\
\rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(c,e), L(e,g), L(d,f), L(f,h) \\
\rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,g), L(d,f), L(f,h) \\
\rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(e,g), L(f,h) \\
\rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(f,g), L(f,h) \\
\rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(f,g), L(g,h)
\end{array}$$

$$\begin{array}{rl}
 & M(1,a), M(1,c), M(1,e), M(1,g), M(1,b), M(1,d), M(1,f), M(1,h) \\
\rightarrowtail^* & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(f,g), L(g,h)
\end{array}$$

Figure 2.3: Merge sort

derivation in Figure 2.3 where we simultaneously sort the characters a, c, e, g and b, d, f, h.

2.2.3 Parallel Programming in CHR

In this thesis, we will focus on promoting CHR as a high-level concurrency abstraction for parallel programming. In Section 2.2.2, we have demonstrated CHR as a general programming language to solve general programming problems[2]. For

[2]This is as oppose to using CHR for constraint solving problems, it's traditional application.

instance, CHR solutions for greatest common divisor and communication buffers were presented in Figure 2.1 and merge-sort in Figure 2.3. CHR implementations of general programming problems such as the above are immediately parallel implementations as well, assuming that we have an implementation of a CHR solver which allows parallel rule execution.

The concurrent nature of the CHR semantics makes parallel programming in CHR straight-forward and intuitive. This means that we can naturally use CHR as a high-level concurrency abstraction which allow us to focus on programming the synchronization of concurrent resources and processes, rather than on micro-managing the concurrent accesses of shared memory. For example, consider the following CHR rules implementing a concurrent dictionary, which concurrent *lookup* and *set* operations can occur in parallel as long as the operated keys are non-overlapping (theoretically, of course[3]):

$$
\begin{array}{lcl}
lookup & @ & Entry(k1, v) \backslash Lookup(k2, x) \iff k1 == k2 \mid x = v \\
set & @ & Set(k1, v), Entry(k2, \_) \iff k1 == k2 \mid Entry(k2, v) \\
new & @ & NewEntry(k, v) \iff Entry(k, v)
\end{array}
$$

Constraint $Entry(k, v)$ represents a dictionary mapping of key k to value v. The CHR rule *lookup* models the action of looking up a key $k2$ in the dictionary, and assigning it's value to v. Similarly, the CHR rule *set* represents the action of setting a new value v to the dictionary key k, while *new* creates new entries in the dictionary. Note that constraints $Lookup(k, x)$, $Set(k, v)$ and $newEntry(k, v)$ represents triggers to the respective actions. The following derivation illustrates non-overlapping dictionary operations:

[3]In practice, we rely on the implementation of the CHR system to make this possible

$$\{Lookup('a', x1), Entry('a', 1)\} \rightarrowtail \{x1 = 1, Entry('a', 1)\}$$

$$||$$

$$\{Lookup('b', x2), Entry('b', 2)\} \rightarrowtail \{x2 = 2, Entry('b', 2)\}$$

$$||$$

$$\{Set('c', 10), Entry('c', 3)\} \rightarrowtail \{Entry('c', 10)\}$$

$$\{Lookup('a', x1), Lookup('b', x2), Set('c', 10), Entry('a', 1), Entry('b', 2), Entry('c', 3)\}$$

$$\rightarrowtail^* \quad \{x1 = 1, x2 = 2, Entry('a', 1), Entry('b', 2), Entry('c', 10)\}$$

Let's consider another example, implementing the parallel programming framework *map-reduce* in CHR:

$$\begin{array}{rcl} map1 & @ & Map((x : xs), m, r) \Longleftrightarrow Work(x, m, r), Map(xs, m, r) \\ map2 & @ & Map([\,], \_, \_) \Longleftrightarrow True \\ work & @ & Work(x, m, r) \Longleftrightarrow Reduce([m(x)], r) \\ reduce & @ & Reduce(xs1, r), Reduce(xs2, \_) \Longleftrightarrow Reduce(r(xs1, xs2), r) \end{array}$$

We assume that m and r are higher-order functions representing the abstract *map* and *reduce* functions. The constraint $Map(xs, m, r)$ initiates the *map1* rule which maps the function m onto each element in xs. Each application of m is represented by $Work(x, m, r)$ and the actual application $m(x)$ is implemented by the rule $work$, producing the results $Reduce(xs, r)$. The rule $reduce$ models the reduce step, combining the results in the manner specified by reduce function r[4] When CHR rewritings are exhaustively applied, the store will have a single $Reduce(xs, r)$ constraint where xs is the final result. Note that the concurrent CHR semantics

[4]For simplicity, we assume a simple setting, where the ordering of elements need not be preserved.

models the parallelism of the map reduce framework: multiple $Work(x, m, r)$ constraints are free to be applied to the *work* rule concurrently, while non-overlapping pairs of $Reduce(xs, r)$ can be combined by the *reduce* rule concurrently.

Note that in the examples above, the CHR rules here declaratively defines the synchronization patterns of the constraints representing concurrent processes, while the concurrent CHR semantics abstracts away the actual details of the synchronization. To execute such programs to scale with multi-core systems, we will require an implementation of the CHR concurrent semantics that actually executes multiple CHR rewritings in parallel. We will provide details of such an implementation in Chapter 4.

2.2.4 Syntax and Abstract Semantics

Figure 2.4 reviews the essentials of the abstract CHR semantics [19]. The general form of CHR rules contains propagated heads H_P and simplified heads H_S as well as a guard t_g

$$r \ @ \ H_P \backslash H_S \iff t_g \mid B$$

In CHR terminology, a rule with simplified heads only (H_P is empty) is referred to as a *simplification* rule, a rule with propagated heads only (H_S is empty) is referred to as a *propagation* rule. The general form is referred to as a *simpagation* rule.

CHR rules manipulate a global constraint store which is a multi-set of constraints. We execute CHRs by exhaustive rewriting of constraints in the store with respect to the given CHR program (a finite set of CHR rules), via the derivations $\rightarrowtail$. To avoid ambiguities, we annotate derivations of the abstract semantics with $\mathcal{A}$.

Rule (Rewrite) describes application of a CHR rule r at some instance ϕ. We simplify (remove from the store) the matching copies of $\phi(H_S)$ and propagate (keep in the store) the matching copies of $\phi(H_P)$. But this only happens if the instantiated guard $\phi(t_g)$ is entailed by the equations present in the store S, written $Eqs(S) \models$

CHR Syntax:

Functions	$f ::= + \mid > \mid \&\& \mid ...$
Constants	$v ::= 1 \mid true \mid ...$
Terms	$t ::= x \mid f\ \overline{t}$
Predicates	$p ::= Get \mid Put \mid ...$
Equations	$e ::= t = t$
CHR constraints	$c ::= p(\overline{t})$
Constraints	$b ::= e \mid c$
CHR Guards	$t_g ::= t$
CHR Heads	$H ::= \overline{c}$
CHR Body	$B ::= \overline{b}$
CHR Rule	$R ::= r\ @\ H \setminus H \iff t_g \mid B$
CHR Store	$S ::= \overline{b}$
CHR Program	$\mathcal{P} ::= \overline{R}$

Abstract Semantics Rules: $\boxed{Store \rightarrowtail_{\mathcal{A}} Store}$

$$\textbf{(Rewrite)} \qquad \frac{\begin{array}{c}(r\ @\ H_P \backslash H_S \iff t_g \mid B) \in \mathcal{P} \text{ such that} \\ \exists \phi \quad Eqs(S) \models \phi \wedge t_g \quad \phi(H_P \uplus H_S) = H'_P \uplus H'_S\end{array}}{H'_P \uplus H'_S \uplus S \rightarrowtail_{\mathcal{A}} H'_P \uplus \phi(B) \uplus S}$$

$$\textbf{(Concurrency)} \qquad \frac{S \uplus S_1 \rightarrowtail^*_{\mathcal{A}} S \uplus S_2 \quad S \uplus S_3 \rightarrowtail^*_{\mathcal{A}} S \uplus S_4}{S \uplus S_1 \uplus S_3 \rightarrowtail^*_{\mathcal{A}} S \uplus S_2 \uplus S_4}$$

$$\textbf{(Closure)} \qquad \frac{S \rightarrowtail_{\mathcal{A}} S'}{S \rightarrowtail^*_{\mathcal{A}} S'} \qquad \frac{S \rightarrowtail_{\mathcal{A}} S' \quad S' \rightarrowtail^*_{\mathcal{A}} S''}{S \rightarrowtail^*_{\mathcal{A}} S''}$$

where $Eqs(S) = \{e \mid e \in S, e \text{ is an equation}\}$

Figure 2.4: Abstract CHR semantics

$\phi(t_g)$. In case of a propagation rule we need to avoid infinite re-propagation. We refer to [1, 9] for details. Rule (Concurrency), introduced in [20], states that rules can be applied concurrently as long as they simplify on non-overlapping parts of the store.

Definition 1 (Non-overlapping Rule Application) *Two applications of the rule instances* $r\ @\ H_P \backslash H_S \iff t_g \mid B$ *and* $r'\ @\ H'_P \backslash H'_S \iff t'_g \mid B'$ *in store* S *are said to be non-overlapping if and only if they simplify unique parts of* S *(ie.* $H_S, H'_S \subseteq S$ *and* $H_S \cap H'_S = \emptyset$*).*

The two last (Closure) rules simply specify the transitive application of CHR rules. A final store of a given CHR program (Definition 2) is a constraint store where no rules from the CHR program can be applied.

Definition 2 (Final Store) *A store S is known as a final store, denoted $Final_{\mathcal{A}}(S)$ if and only if no more CHR rules applies on it (ie. $\neg\exists S'$ such that $S \rightarrowtail_{\mathcal{A}} S'$).*

CHR programs may not necessary be terminating of course. A CHR program is said to be terminating (with respect to the abstract CHR semantics, $\mathcal{A}$) if and only if it contains no infinite computation paths (derivation sequences).

Definition 3 (Terminating CHR Programs) *A CHR program $\mathcal{P}$ is said to be terminating, if and only if for any CHR store S, there exists no infinite derivation paths from S, via the program $\mathcal{P}$.*

2.2.5 CHR Execution Models

The abstract CHR semantics discussed in Section 2.2.4 sufficiently describes the behaviour of CHR programs. However, it does not explain how CHR programs are practically executed. As a result, existing CHR systems often implement more systematic execution models to performing CHR rewritings, while the concise behaviour of such execution models are largely not captured by the abstract CHR semantics.

For this reason, works in [9, 11, 33] aims to fill this theoretical 'gap' between the abstract CHR semantics and actual execution models implemented by most existing CHR systems. In this section, we will highlight the refined CHR operational semantics as found in [9], since among the three works mentioned here, it is the most general.

The refined CHR operational semantics (denoted ω_r) describes a goal-based execution model of CHR rules. The idea is to treat each newly added constraint in the global constraint store as a *goal* constraint. Goal constraints are simply constraints which are waiting to be executed. When a goal is executed, it is first added to the

$get \ @ \ Get(x)_1, Put(y)_2 \Longleftrightarrow x = y$

Transition Step		**Constraint Store**	**Explanation**
		$\langle [Get(m), Put(1)] \mid \emptyset \rangle$	
(D1: Activate)	$\rightarrowtail$	$\langle [Get(m)\#1 : 1, Put(1)] \mid \{Get(m)\#1\} \rangle$	`Add` $Get(m)$ `to store.`
(D2: Default)	$\rightarrowtail$	$\langle [Get(m)\#1 : 2, Put(1)] \mid \{Get(m)\#1\} \rangle$	`Try match` $Get(m)$ `on occ 2`
(D3: Default)	$\rightarrowtail$	$\langle [Get(m)\#1 : 3, Put(1)] \mid \{Get(m)\#1\} \rangle$	`Try match` $Get(m)$ `on occ 3`
(D4: Drop)	$\rightarrowtail$	$\langle [Put(1)] \mid \{Get(m)\#1\} \rangle$	`Drop` $Get(m)$ `from goals.`
(D5: Activate)	$\rightarrowtail$	$\langle [Put(1)\#2 : 1] \mid \{Get(m)\#1, Put(1)\#2\} \rangle$	`Add` $Put(1)$ `to store.`
(D6: Default)	$\rightarrowtail$	$\langle [Put(1)\#2 : 2] \mid \{Get(m)\#1, Put(1)\#2\} \rangle$	`Try match` $Put(1)$ `on occ 2`
(D7: Fire *get*)	$\rightarrowtail$	$\langle [m = 1] \mid \emptyset \rangle$	`Fire` *get* `rule on` $Put(1)$
(D8: Solve)	$\rightarrowtail$	$\langle [\,] \mid \{m = 1\} \rangle$	`Add constraint m=1 to store`

Figure 2.5: Example of Refined Operational Semantics, ω_r

constraint store, then followed by a search routine: search for matching partner constraints in the store that together with the goal, forms a complete rule head match. We will omit formal details of the refined operational semantics, but will illustrate it's intuition by example. Figure 2.5 illustrates ω_r derivations of our communication buffer example introduced in Section 2.2.1. Firstly, note that ω_r derivations map from CHR states to CHR states, namely tuples $\langle G \mid S \rangle$, where G (Goals) is a list (sequence) of goal constraints and S (Store) is a multiset of constraints. There are three types of goal constraints: active goal constraints ($c(\bar{x})\#n : m$), numbered goal constraints ($c(\bar{x})\#n$) and new goal constraints ($c(\bar{x})$). Constraint store now contains only numbered constraints ($c(\bar{x})\#n$), which are uniquely identified by their numbers. Also note that unlike derivations in the abstract semantics, the refined operational semantics contain derivation steps of various transition types other than firing (Fire) of rules (eg. Activate, Default, etc..). Finally, notice that the CHR rule heads are annotated by a unique integer, known as *occurrence numbers*. These occurrence numbers are used to identify which rule head an active goal constraint is matching with. For presentation purpose here, we label the x^{th} derivation step of the sequence of derivations as Dx.

Informally, the ω_r derivations work as follows: We consider the refined CHR

semantics derivation illustrated in Figure 2.5. The store is initially empty. All constraints are 'new', hence are new goal constraints. Derivation step $D1$ *activates* the head of the list, $Get(m)$. This replaces $Get(m)$ with the active goal constraint $Get(m)\#1 : 1$ and also adds the numbered constraint $Get(m)\#1$. Intuitively, the active constraint $Get(m)\#1 : 1$ simply extends the original goal constraint with additional book keeping information. An active goal constraint $Get(m)\#n : p$ represents a goal constraint associated with the constraint in the store ($Get(m)\#n$) and is to be matched with the p^{th} rule head occurrence. Indeed $Get(m)$ matches with rule head occurrence 1 (ie. $Get(x)$, under substitution $m = x$), but no matching partner constraint (ie. $Put(y)$) exists in the store to complete the rule match for *get*. Hence for derivation step $D2$, we take a *default* transition which increments the active constraint occurrence number by one, essentially advancing the matching of constraint $Get(m)\#1$ with the next rule head occurrence (ie. 2). Since $Get(m)$ obviously does not match with rule head occurrence 2, we take another *default* step in $D3$. For derivation $D4$, we have tried matching $Get(m)\#1$ with all rule head occurrences and have reached an occurrence number which does not exist (ie. 3), thus we can *drop* the active constraint and can move on to the next goal. Derivation step $D5$ activates the next goal (ie. $Put(1)$). Similar to $D1$, it assigns the goal a new unique identifier and sets it's occurrence number to 1 (hence we have $Put(1)\#2 : 1$). Derivation $D6$ is another (Default) step since $Put(1)$ does not match with $Get(m)$. Finally, in derivation $D7$, $Put(1)\#2$ matches $Put(y)$ of the *get* rule, and we have a matching partner $Get(m)\#1$ in the store. Thus we *fire* the *get* rule instance $\{Get(m)\#1, Put(1)\#2\}$ in the store. Note that new constraints (ie. $m = 1$) are added to the goals for future execution. The final step $D8$ denoted $Solve$ simply adds the built-in constraint $m = 1$ to the store. When goals are empty, derivation terminates. Correspondence results in [9] show that a reachable state with empty goals have a constraint store which correspond to a final store derivable by the CHR abstract semantics. Hence the refined operational semantics is sound, with respect

to the CHR abstract semantics.

While the refined operational semantics seems to be much more complex than the abstract CHR semantics, it provides a more concise description of how CHR programs are executed in a systematic manner. Further more, goals are executed in stack order (executed in left-to-right order, while new goals added to left) and rule head occurrences are tried in a fixed sequence. For this reason, the refined operational semantics more deterministic than the abstract CHR semantics. The refined operational semantics also exhibits better confluence results in that CHR programs can be confluent under the refined operational semantics but not the abstract CHR semantics. In essence, the refined operational semantics offers a theoretical model which much more closely describes how existing CHR systems are implemented (compared to the abstract CHR semantics).

2.3 Our Work

2.3.1 Concurrent Goal-based CHR semantics

The CHR refined operational semantics discussed in Section 2.2.5 describes an inherently single threaded computation model. The semantics implicitly impose the limitation that reachable CHR states contain at most one active goal constraint, essentially describing a computation model with exactly one thread of computation. As such, it would seem that the concurrency exhibited by the CHR abstract semantics (as discussed in Section 2.2.2) is not observable in the refined operational semantics. We wish to develop a new execution model of CHR which allows concurrent execution of multiple CHR goals. It would be tempting to directly lift concurrency results of the CHR abstract semantics (Figure 2.2) to the refined operational semantics to allow multiple active goal constraints, thus obtaining a concurrent execution model for CHR rewriting.

Figure 2.6 shows an attempt to extend the refined operational semantics with a

Erroneous concurrency Rule for ω_r

$$\frac{\langle G_1 \mid S \uplus S_1\rangle \rightarrowtail^* \langle G_2 \mid S \uplus S_2\rangle \quad \langle G_3 \mid S \uplus S_3\rangle \rightarrowtail^* \langle G_4 \mid S \uplus S_4\rangle}{\langle G_1 + + G_3 \mid S \uplus S_1 \uplus S_3\rangle \rightarrowtail^* \langle G_2 + + G_4 \mid S \uplus S_2 \uplus S_4\rangle}$$

Counter Example:

$$get \ @ \ Get(x)_1, Put(y)_2 \Longleftrightarrow x = y$$

$$\begin{array}{lll|lll}
 & \langle [Get(m)] \mid \emptyset\rangle & & & \langle [Put(1)] \mid \emptyset\rangle & \\
\rightarrowtail & \langle [Get(m)\#1:1] \mid \{Get(m)\#1\}\rangle & & \rightarrowtail & \langle [Put(1)\#2:1] \mid \{Put(1)\#2\}\rangle & \\
\rightarrowtail & \langle [Get(m)\#1:2] \mid \{Get(m)\#1\}\rangle & \| & \rightarrowtail & \langle [Put(1)\#2:2] \mid \{Put(1)\#2\}\rangle & \\
\rightarrowtail & \langle [Get(m)\#1:3] \mid \{Get(m)\#1\}\rangle & & \rightarrowtail & \langle [Put(1)\#2:3] \mid \{Put(1)\#2\}\rangle & \\
\rightarrowtail & \langle [\,] \mid \{Get(m)\#1\}\rangle & & \rightarrowtail & \langle [\,] \mid \{Put(1)\#2\}\rangle &
\end{array}$$

$$\langle [Get(m), Put(1)] \mid \emptyset\rangle \rightarrowtail^* \langle [\,] \mid \{Get(m)\#1, Put(1)\#2\}\rangle$$

Figure 2.6: An example of inconsistency in concurrent execution

concurrency rule. This derivation rule is directly lifted from the concurrency rule of the CHR abstract semantics (Figure 2.2). Figure 2.6 also illustrates a counter example against this derivation rule. We consider the communication buffer example (Section 2.2.1). The premise of this rule instance shows the concurrent execution of a $Get(m)$ and a $Put(1)$ goal constraint. Let's consider the derivation steps on the left (Execution of $Get(m)$). $Get(m)$ is first activated. Since we do not have a matching $Put(y)$ constraint in the store, we take a default derivation. Next derivation is another default since $Get(m)$ cannot match with rule head occurrence 2. Finally the goal $Get(m)\#1:3$ is dropped, since rule head occurrence 3 does not exist. Similarly, derivations steps on the right (execution of $Put(1)$) activates $Put(1)$ and drops it eventually without triggering any rule instances. We compose the two derivations and find that we arrive at a non-final CHR state ($\{Get(m)\#1, Put(1)\#2\}$ is a rule instance of get). The problem is that both derivation steps are taken in isolation and do not observe each other's modification (addition of new constraints $Get(m)\#1$ and $Put(1)\#2$ respectively), thus both goals are dropped without triggering the

rule instance. Dropping the goals are consistent in their respective local contexts (constraint stores) but inconsistent in the global context, thus rule instances can be missed.

This counter example, illustrates that deriving a concurrent CHR execution model is a non-trivial task and is not a simple extension from the refined CHR operational semantics. Concurrent derivation steps are not naively composable and clearly they require some form of synchronization through the constraint store.

The first part of our work (presented in Section 3) formalizes a concurrent goal-based CHR semantics, denoted $\| \mathcal{G}$ semantics. $\| \mathcal{G}$ is a goal-based CHR operational semantics, similar to the refined operational semantics, but it additionally defines concurrent derivations of CHR goal constraints on a shared constraint store. We will detail how we deal with problems of maintaining consistency of concurrent CHR rewritings, such as that illustrated in Figure 2.6. We also provide a proof of correspondence to show that $\| \mathcal{G}$ is sound with respect to the CHR abstract semantics.

2.3.2 Parallel CHR Implementation in Haskell (GHC)

Moving ahead from our formalization of the $\| \mathcal{G}$ concurrent goal-based CHR semantics, the next part of our work focuses on the technical details of a practical implementation of a parallel CHR system, based on the $\| \mathcal{G}$ semantics. As the most computationally intensive routine of CHR goal execution is the search for matching constraints, much can be gained by implementing a CHR system which can execute search routines (for matching constraints) of multiple CHR goals in parallel, over a shared constraint store. While the $\| \mathcal{G}$ semantics formally describes how CHR goals can be executed concurrently over a shared constraint store, it provides little details on how we can implement this in a practical and scalable manner. In other words, the technical concerns of how to implement scalable CHR rewritings are not observable in the formal semantics.

$$r \ @ \ A(1,x), B(x,y), C(z) \Longleftrightarrow y > z \mid D(x,y,z)$$

```
1  execGoal ⟨G | Sn⟩ A(1,x)#n {
2        lock Sn
3        ms1 = match Sn B(x,_)
4        for B(x,y)#m in ms1 {
5             ms2 = match Sn C(_)
6             for C(z)#p in ms2 {
7                  if(y > z) {
8                       deleteFromStore Sn [B(x,y)#m,C(z)#p]
9                       addToGoals G [A(1,x)#n,D(x,y,z)]
10                      unlock Sn
11                      return true
12                 }
13            }
14       }
15       unlock Sn
16       return false
17 }
```

Table 2.1: A coarse-grained locking implementation of concurrent CHR goal-based rewritings

We illustrate this point by considering a straight-forward implementation of concurrent CHR goal-based rewriting. Table 2.1 shows a traditional locking approach to implement concurrent execution of a goal constraint. Specifically, the procedure `execGoal` [5] implements the execution of the goal $A(x)\#n$ in the context of the CHR program consisting only of rule r, given the components of the current CHR state (goals G and store Sn). We assume that we have several APIs that behaves in the following way:

- `match` Sn c - Where Sn is the CHR constraint store, and c is a CHR constraint pattern. Returns an iteration of constraints matching c.
- `deleteFromStore` Sn cs - Where Sn is the CHR constraint store and cs is a list of stored constraints in Sn, deletes all stored constraints in cs from Sn.

[5] Note that we will use pseudo code of an imperative style language to introduce the general ideas of implementing CHR rewritings. In Chapter 4, we will detail our actual implementation in the functional programming language, Haskell.

- `addToGoals` G cs - Where G is the goals and cs is a list of CHR constraints, add all CHR constraints in cs into the goals G.

- `lock` Sn and `unlock` Sn - Lock or unlock constraint store Sn respectively. The former blocks if Sn is already locked.

Line 2 locks the store Sn so that the current thread of computation has exclusive access to the store. Line 3 creates an iteration ($ms1$) of constraints in the store Sn that matches the pattern $B(x, \_)$, where the $\_$ symbol represents the 'any' pattern. The 'For' loop of lines $4 - 14$ tries matching constraints in $ms1$ with the rest of the search procedure. Similar to Line 3, Line 5 creates an iteration of constraints matching $C(\_)$. This is followed by the inner 'For' loop of Lines $6 - 13$ which iterates through constraints in $ms2$. Line 7 checks the rule guard which only executes rewriting (Lines $8-11$) for constraint sets satisfying $y > z$. CHR rewriting is modeled by the following: Line 8 removes the constraints $B(x, y)\#m$ and $C(z)\#p$ which matched the simplified heads of the rule. Line 9 adds the rule body $D(x, y, z)$ and the propagated goal constraint $A(1, x)\#n$ into the CHR goals G as new goal(s) to be executed later. Line 9 simply unlocks the store Sn when the rewriting procedure is complete, while Line 10 exits the procedure with success (true). Finally, Lines $15 - 16$ implements the 'failure' case, where no rule head match is found, and the goal constraint is dropped, during which the store Sn is unlocked and the procedure is exited with failure (false).

This implementation uses a coarse-grained locking scheme[6]. This guarantees consistency of concurrent execution of goal execution functions (like `execGoal`) simply by 'wrapping' the matching and rewriting routines of goal execution between the `lock` and `unlock` calls, allowing them to execute in an uninterrupted and uninterleaving manner. Yet while consistency is naively guaranteed, this implementation is

[6]By *coarse-grained locking scheme*, we refer to a simple synchronization protocol where shared variables are accessed via a single (or minimal number of) high-level lock that possibly locks multiple shared objects

unlikely to scale well. This is because at most one executing thread can access the shared store at a time, making concurrent execution multiple CHR goals effectively sequential. Parallelism in this approach requires fine-grained locking implementation, which will require non-trivial modifications and to maintain completeness and correctness of CHR rewriting. For instance, APIs like `match`, `deleteFromStore` and `addToGoals` must be heavily modified with micro-management of fine-grained locking protocols to allow consistent interleaving concurrent executions. In Chapter 4, we also show another approach in Software Transaction Memory (STM) which like this, is extremely simple but will not scale well, emphasizing that there are no 'free lunch' in parallel programming and implementing a scalable parallel CHR system is non-trivial.

We develop a concrete implementation of the $\| \mathcal{G}$ semantics, a parallel CHR system in the functional language Haskell (GHC), known as ParallelCHR . ParallelCHR is a library extension of Haskell that act as an interpreter for CHR programs. It implements CHR rewritings over a shared constraint store, utilizing fine-grained manipulation of existing concurrency primitives (eg. Software Transactional Memory and IO References). We will illustrate that our implementation of ParallelCHR is scalable through empirical results presented in Section 4.6.

2.3.3 Join-Patterns with Guards and Propagation

The next step of our work is to identify and study a non-trivial application of parallel CHR rewritings. For this, we focus on a promising high-level concurrency model, known as Join-Calculus [18]. Join-Calculus is a process calculus that introduces an expressive high-level concurrency model, aimed at providing a simple and intuitive way to coordinate concurrent processes via reaction rules known as **Join-Patterns**.

We review the basic idea of Join-Patterns with a classic example to model a concurrent buffer. In Table 2.2, we introduce two events to consume (`Get`) and produce (`Put`) buffer elements. To distinguish join process calls from standard function calls,

```
event Put(Async Int)
event Get(Sync Int)

Get(x) & Put(y) = x := y

t1 = do { Put(3)              t2 = do { Put(5)
        ; Put(4)                       ; x2 <- newSync
        ; x1 <- newSync                ; Get(x2)
        ; Get(x1)                      ; v2 <- readSync x2
        ; v1 <- readSync x1            ; print v2 }
        ; print v1 }
```

Table 2.2: Get-Put Communication Buffer in Join-Patterns

join process start with upper-case letters, while standard function calls start with lower-case letters. Events are stored in a global multiset, referred to as event store (or store for short). Via the Join-Pattern `Get(x) & Put(y)` we look for matching consumer/producer events. If present, the matching events are removed and the join body `x := y` is executed, modeling the retrieval of a buffered item. In general, the join body is simply a call back function executed when the matching events specified by the Join-Pattern are present.

Events are essentially called like function calls. For instance, in Table 2.2 operation `t1` and `t2` make calls to `Get` and `Put`. Arguments of events can either be asynchronous (ground input values), synchronous (output variables). Synchronous arguments, generated via the `newSync` primitive, serve to transmit buffer elements. We can access the transmitted values via primitive `readSync` which blocks until the variable is bound to a value. Synchronous variables are written into via `:=`. We assume that `print` is a primitive function that prints it's argument on the shell terminal.

Suppose we execute the two threads executing `t1` and `t2` respectively. Events are non-blocking, they will be recorded in the store and we proceed until we hit a blocking operation. Hence, both threads potentially block once we reach their first

`readSync` statement. At this point, the following events are in the store

```
{Get(x1), Get(x2), Put(3), Put(4), Put(5)}
```

The combinations `Get(x1) & Put(3)` and `Get(x2) & Put(4)` match the join pattern `Get(x) & Put(y)`, hence two join-pattern instances will be triggered and the join bodies `x1 := 3` and `x2 := 4` will be executed, unblocking both `readSync` calls of `t1` and `t2`. Eventually, `3` and `4` will be printed on the shell terminal. Note there are other combinations which lead to a different result. This is no surprise given that concurrent join semantics is indeterministic.

What we have described so far is the basic idea of Join-Patterns. In [5], the idea of Join-Patterns with guards is briefly discussed, proposing an extension of Join-Patterns with guard conditions that allows more complex and convenient synchronization patterns. For instance, suppose we want to implement a buffer access function that only retrieves items less than a specified value. With guards, this can be easily implemented by the following:

```
GetLess(x,v) & Put(y) | y < v = x := y
```

While this seems to be a simple syntactic extension, it imposes significant technical challenge for existing Join-Pattern compilation techniques, as pointed out in [5]. For this reason, existing Join-Pattern systems [29, 5, 38] do not consider guards. The challenge for an implementation is that we now need to *search* for matching events which satisfy a guard condition, as oppose to straight-forward pairing of event symbols. Further more, on a multi-core architecture, we wish exploit parallelism by executing such matching routines in parallel to increase the performance (parallel matching). Most interestingly, our work in implementing parallel CHR rewriting (Chapter 4) can be adapted to implement the matching of Join-Pattern events with guards, since the parallel matching routines we have developed already performs such parallel search for matching constraints (events).

We have introduced a novel approach to implement Join-Patterns with guards, essentially exploiting the similarities of parallel CHR rewriting and the executing of Join-Patterns. We have implemented this system in Haskell(GHC), utilizing our parallel CHR matching routines highlighted in Chapter 4. We also demonstrate that other features like propagation can be included to the Join-Pattern world almost for free. Finally, empirical results in Section 5.5 shows the scalability of this approach to implement Join-Patterns with guards.

Chapter 3

Concurrent Goal-Based CHR Semantics

3.1 Chapter Overview

In this Chapter, we formally introduce our concurrent goal-based CHR semantics, $\| \mathcal{G}$ and provide a proof of it's correspondence with the abstract CHR semantics. Specifically, we first review the goal-based refined CHR operational semantics, which in essence leads to highly efficient implementations but relies on a single-threaded execution model (Section 3.2). Next, We devise a concurrent goal-based semantics ($\| \mathcal{G}$ semantics) which forms the basis for an efficient parallel CHR implementation (Section 3.3). Section 3.4 highlights several subtle issues of the $\| \mathcal{G}$ semantics, while Section 3.5 presents the correspondence results.

3.2 Goal-Based CHR Semantics

In this Section, we introduce the goal-based CHR semantics $\mathcal{G}$, which is essentially a generalization of the refined CHR operational semantics ω_r.

Most existing CHR implementation (Eg. JCHR System [30], CHR in HAL [27]

, SWI-Prolog CHR [55]) employ a more systematic CHR execution model where rules are triggered based on a set of available goals. The idea behind a goal-based CHR execution model is to separate the constraint store into two components: a set of goal constraints (constraints yet to be executed) and the actual constraint store (constraints that were executed). In the abstract semantics, transitions $\rightarrowtail_{\mathcal{A}}$ maps between CHR stores S, whereas in the goal-based semantics we find now transitions $\rightarrowtail_{\mathcal{G}}$ maps between CHR states of the form $\langle G \mid S \rangle$ where G is the CHR goals and S is the CHR store. Only goal constraints (in G) can trigger CHR rewriting by first searching for matching constraints in the store S to build a complete match of a rule head, then applying the rewriting specified by the CHR rule.

Below, we give a goal-based execution of the earlier communication buffer example.

$$get \ @ \ Get(x), Put(y) \iff x = y$$

Step	**Transition Type**		**Constraint Store**
			$\langle \{Get(x_1), Get(x_2), Put(1), Put(2)\} \mid \{\} \rangle$
$D1$	(Activate)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Get(x_1)\#1, Get(x_2), Put(1), Put(2)\} \mid \{Get(x_1)\#1\} \rangle$
$D2$	(Drop)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Get(x_2), Put(1), Put(2)\} \mid \{Get(x_1)\#1\} \rangle$
$D3$	(Activate)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Get(x_2)\#2, Put(1), Put(2)\} \mid \{Get(x_1)\#1, Get(x_2)\#2\} \rangle$
$D4$	(Drop)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Put(1), Put(2)\} \mid \{Get(x_1)\#1, Get(x_2)\#2\} \rangle$
$D5$	(Activate)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Put(1)\#3, Put(2)\} \mid \{Get(x_1)\#1, Get(x_2)\#2, Put(1)\#3\} \rangle$
$D6$	(Fire)get	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Put(2), x_1 = 1\} \mid \{Get(x_2)\#2\} \rangle$
$D7$	(Activate)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{Put(2)\#3, x_1 = 1\} \mid \{Get(x_2)\#2, Put(2)\#3\} \rangle$
$D8$	(Fire)get	$\rightarrowtail_{\mathcal{G}}$	$\langle \{x_1 = 1, x_2 = 2\} \mid \{\} \rangle$
$D9$	(Solve)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{x_2 = 2\} \mid \{x_1 = 1\} \rangle$
$D10$	(Solve)	$\rightarrowtail_{\mathcal{G}}$	$\langle \{\} \mid \{x_1 = 1, x_2 = 2\} \rangle$

We label the x^{th} derivation step by a label Dx. Let's walk through each of the individual goal-based execution steps. Initially, all constraints are kept in the set

of goals. At this point, all of the goals are inactive. Execution of goals proceeds in two stages: (1) Activation and (2a) rule execution, or (2b) dropping of goals. In the first stage, we activate a goal. In general, the order in which goals are activated is arbitrary. For concreteness, we assume a left-to-right activation order.

Hence, we first activate $Get(x_1)$ in derivation step $D1$. Active goals carry a unique identifier, a distinct integer number. Besides assigning numbers to active goals, we also put them into the store. For instance, after activating $Get(x_1)$, we have $Get(x_1)\#1$ in both the goals and the store. [1]

Active goals like $Get(x_1)\#1$ are executed by trying to build a complete match for a rule head with matching partner constraints in the store. Since there are no other constraints in the store, we cannot match $Get(x_1)\#1$ with the *get* rule. Therefore we drop $Get(x_1)\#1$ in step $D2$. Dropping of a goal means the goal is removed from the set of goals but of course the (now inactive) goal is still present in the store. Step $D3$ and $D4$ are similar but executed on goal $Get(x_2)$. Then, we activate $Get(x_2)$ and find that $Get(x_2)\#2$ cannot build a complete match of the *get* rule, thus it is dropped too.

Next, we activate $Put(1)$ (Step $D5$). Constraint $Put(1)\#3$ can match with either $Get(x_1)\#1$ or $Get(x_2)\#2$ to form a complete instance of rule head of *get*. We pick $Get(x_1)\#1$ and fire the rule *get*, see step $D6$. Step $D7$ and $D8$ perform similar execution steps on $Put(2)$ and the remaining stored constraint $Get(x_2)\#2$. Finally, we add the equations $x_1 = 1$ and $x_2 = 2$ into the store in steps $D9$ and $D10$. Exhaustive application of this goal-based execution strategy then leads to a state with no goals and a final store.

What we have described so far is essentially the execution scheme employed in all major CHR implementations. The semantics of these implementations assume a more deterministic goal activation policy. For instance [9] assumes that CHR goals

[1] Numbered constraints also disambiguate multiple copies in the store but this is rather a side-effect. The main purpose of numbering constraints is to indicate activation (only active goals are numbered) and retain the link between active goal constraints and their stored copy (each active goal corresponds to a stored constraint and they share the same number).

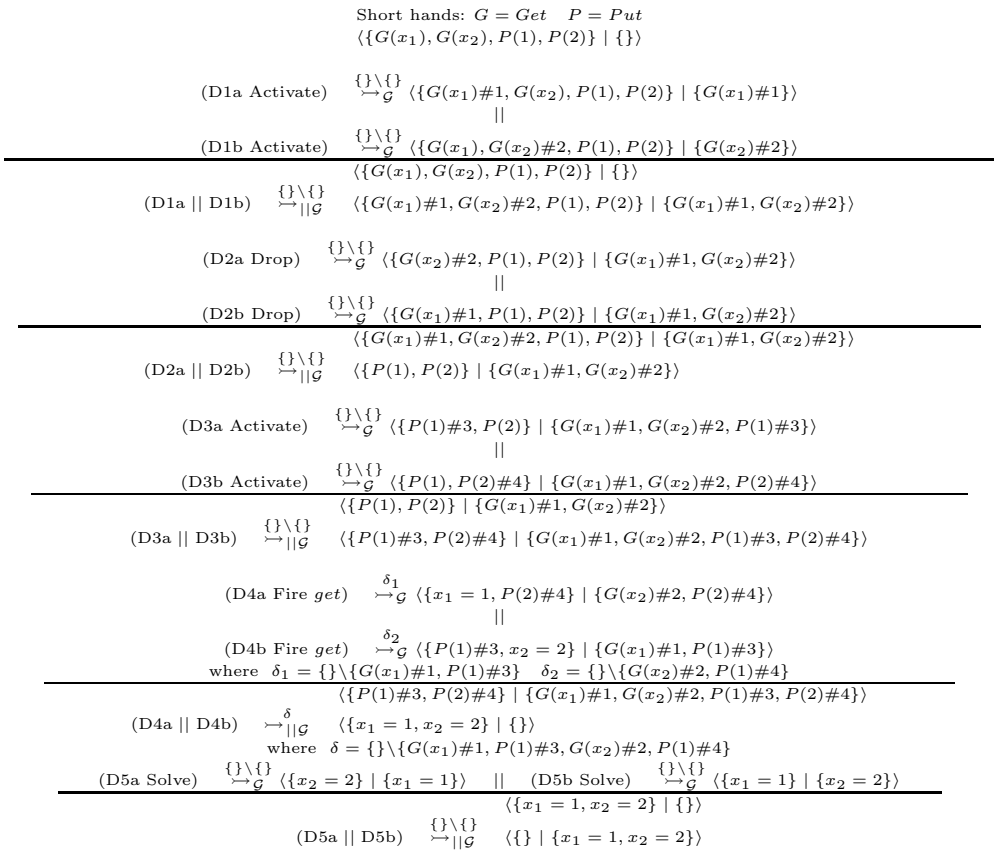

Figure 3.1: Example of concurrent goal-based CHR derivation

are kept in a stack, while [31] uses a priority queue. While imposing such ordering of goals offers better confluence results and thus allowing use of more programming idioms and perhaps convenience, this of course comes at a cost of a strictly sequential execution scheme.

To obtain a systematic, yet concurrent, CHR execution scheme we adapt the goal-based CHR semantics as follows. Several active goal constraints can simultaneously seek for partner constraints in the store to fire a rule instance. In the extreme case, all goal constraints could be activated and executed at once. However, we generally assume that the number of active goals are bounded by some value n, where n represents some practical limitation of system resources (eg. number of available processors). Interestingly, the concept of CHR goals in our concurrent context resembles that of thread pooling in parallel programming. We will defer a

discussion on this until Section 3.4.4.

Figure 3.1 shows a sample concurrent goal-based CHR derivation. We assume two concurrent threads, referred to as a and b, each thread executes the standard goal-based derivation steps. The novelty is that each goal-based derivation step $\overset{\delta}{\rightarrowtail}_{\mathcal{G}}$ now records its effect on the store. The effect δ represents the sets of constraints in the store which were propagated or simplified. Goal-based derivation steps can be executed concurrently if their effects are not in conflict.

$$\textbf{(Goal-Concurrency)} \quad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} \langle G_1' \mid H_{S2} \cup S\rangle \\ \langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} \langle G_2' \mid H_{S1} \cup S\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{c}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \quad \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

The (Goal-Concurrency) rule, abbreviated ($\| \mathcal{G}$), states that two goal-derivations are not in conflict if their simplification effects are disjoint and the propagated effects are present in the joint store. We will provide more explanations later. Let's continue with our example.

Each thread activates one of the two *Get* goals (Steps $D1a$ and $D1b$). Since both steps involve no rule application, side-effects are empty ($\{\}\backslash\{\}$). Both steps are executed concurrently denoted by the concurrent derivation step ($D1a \mid\mid D2a$) $\overset{\{\}\backslash\{\}}{\rightarrowtail}_{||\mathcal{G}}$. Concurrent goal-based execution threads operate on a shared store and their effects will be immediately made visible to other threads. This is important to guarantee exhaustive rule firings.

In the second step ($D2a \mid\mid D2b$), both active goals are dropped because there is no complete match for any rule head yet. Next, steps $D3a$ and $D3b$ activate the last two

goal constraints, $Put(1)$ and $Put(2)$. Each active constraint can match with either of the two Get constraints in the store. We assume that active constraint $Put(1)\#3$ in step $D4a$ matches with $Get(x_1)\#1$, while $Put(2)\#4$ in step $D4b$ matches with $Get(x_2)\#2$, corresponding to the side-effects δ_1 and δ_2. This guarantees that steps $D4a$ and $D4b$ operates on different (non-conflicting) parts of the store. Thus, we can execute them concurrently which yields step ($D4a$ || $D4b$). Their side-effects are combined as δ. Finally, in step ($D5a$ || $D5b$) we concurrently solve the two remaining equations by adding them into the store and we are done.

The correctness of our concurrent goal-based semantics is established by showing that all concurrent derivations can be replicated by sequential goal-based executions. We also prove that there is a correspondence between our goal-based CHR semantics with the abstract CHR semantics. This proof generalizes from [9] which shows a correspondence between the refined CHR operational semantics and abstract semantics. There are a number of subtle points we came across when developing the concurrent variant of the goal-based semantics. We will postpone a discussion of these issues until Section 3.4. Next, we formally introduce the details of the abstract CHR semantics.

3.3 Concurrent Goal-Based CHR Semantics

We present the formal details of the concurrent goal-based CHR semantics. Figure 3.2 describes the necessary syntactic extensions. Because constraints in the store now have unique identifiers, we treat the store as a set (as opposed to a multiset) and use set union $\cup$. Goals are still treated as multi-sets because they can contain multiple copies of (un-numbered) CHR constraints. Please note that we will use lower-case identifiers for variables, upper-case identifiers for constants[2] Note that we will only

[2]Advocates of logic constraint programming should have noticed this "abnormally". We sincerely apologize, but insist that this is for consistency with our Haskell formulation of CHR in Chapter 4

Notations:

$\uplus$	Multi-set union
$\cup$	Set union
$\models$	Theoretic entailment
ϕ	Substitution
$\overline{a}$	Set/List of a's

CHR Syntax:

Functions	$f ::= + \mid >\mid \&\& \mid ...$
Constants	$v ::= 1 \mid true \mid ...$
Terms	$t ::= x \mid f\ \overline{t}$
Predicates	$p ::= Get \mid Put \mid ...$
Equations	$e ::= t = t$
CHR Constraints	$c ::= p(\overline{t})$
Constraints	$b ::= e \mid c$
CHR Guards	$t_g ::= t$
CHR Heads	$H ::= \overline{c}$
CHR Body	$B ::= \overline{b}$
CHR Rule	$R ::= r\ @\ H \setminus H \Longleftrightarrow t_g \mid B$
CHR Program	$\mathcal{P} ::= \overline{R}$
Num Constraint	$nc ::= c\#i$
Goal Constraint	$g ::= c \mid e \mid nc$
Stored Constraint	$sc ::= nc \mid e$
CHR Num Store	$Sn ::= \overline{sc}$
CHR Goals	$G ::= \overline{g}$
CHR State	$\sigma ::= \langle G, Sn \rangle$
Side Effects	$\delta ::= Sn \setminus Sn$

Figure 3.2: CHR Goal-based Syntax

consider CHR rules with non-empty simplification heads (i.e. no pure propagation rules). The actual semantics is given in two parts. Figure 3.3 describes the single-step execution part whereas Figure 3.4 introduces the concurrent execution part. The first part is a generalization of an earlier goal-based description [9] whereas the second (concurrent) part is novel.

We first discuss the single-step derivation steps in Figure 3.3. A derivation step $\sigma \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \sigma'$ maps the CHR state σ to σ' with some side-effect δ. δ represents the constraints that where propagated or simplified during rule application. Hence derivation steps that do not involve rule application ((Activate) and (Drop)) contains no side-effects (i.e. $\{\}\setminus\{\}$). We will omit side-effects δ as and when it is not relevant

$$\textbf{(Solve)} \qquad \frac{W = WakeUp(e, Sn)}{\langle \{e\} \uplus G \mid Sn\rangle \overset{W\setminus\{\}}{\rightarrowtail_{\mathcal{G}}} \langle W \uplus G \mid \{e\} \cup Sn\rangle}$$

$$\textbf{(Activate)} \qquad \frac{i \text{ is a fresh identifier}}{\langle \{c\} \uplus G \mid Sn\rangle \overset{\{\}\setminus\{\}}{\rightarrowtail_{\mathcal{G}}} \langle \{c\#i\} \uplus G \mid \{c\#i\} \cup Sn\rangle}$$

$$\textbf{(Simplify)} \qquad \frac{\begin{array}{c}(r \ @\ H'_P \backslash H'_S \Longleftrightarrow t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists\phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H'_P) = DropIds(H_P) \\ \phi(H'_S) = \phi(\{c\} \uplus DropIds(H_S)) \quad \delta = H_P \backslash \{c\#j\} \cup H_S\end{array}}{\begin{array}{c}\langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup H_S \cup Sn\rangle \\ \overset{\delta}{\rightarrowtail_{\mathcal{G}}} \quad \langle \phi(B') \uplus G \mid H_S \cup Sn\rangle\end{array}}$$

$$\textbf{(Propagate)} \qquad \frac{\begin{array}{c}(r \ @\ H'_P \backslash H'_S \Longleftrightarrow t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists\phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H'_S) = DropIds(H_S) \\ \phi(H'_P) = \phi(\{c\} \uplus DropIds(H_P)) \quad \delta = \{c\#j\} \cup H_P \backslash H_S\end{array}}{\begin{array}{c}\langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup H_S \cup Sn\rangle \\ \overset{\delta}{\rightarrowtail_{\mathcal{G}}} \quad \langle \phi(B') \uplus \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup Sn\rangle\end{array}}$$

$$\textbf{(Drop)} \qquad \frac{\text{(Simplify) and (Propagate) does not apply on } c\#j \text{ in } Sn}{\langle \{c\#j\} \uplus G \mid Sn\rangle \overset{\{\}\setminus\{\}}{\rightarrowtail_{\mathcal{G}}} \langle G \mid Sn\rangle}$$

where
$$\begin{array}{lcl} Eqs(S) & = & \{e \mid e \in S, e \text{ is an equation}\} \\ DropIds(Sn) & = & \{c \mid c\#i \in Sn\} \uplus \{e \mid e \in Sn, e \text{ is an equation}\} \\ WakeUp(e, Sn) & = & \{c\#i \mid c\#i \in Sn \wedge \phi \text{ m.g.u. of } Eqs(Sn) \wedge \\ & & \theta \text{ m.g.u. of } Eqs(Sn \cup \{e\}) \wedge \phi(c) \neq \theta(c)\} \end{array}$$

Figure 3.3: Goal-Based CHR Semantics (Single-Step Execution $\overset{\delta}{\rightarrowtail_{\mathcal{G}}}$)

to our discussions. We ignore the (Solve) step for the moment. In (Activate), we activate a goal CHR constraint by assigning it a fresh unique identifier and adding it to the store. Rewrite rules are executed in steps (Simplify) and (Propagate). We distinguish if the rewrite rule is executed on a simplified or propagated active (goal) constraint $c\#i$. For both cases, we seek for the missing partner constraints in the store for some matching substitution ϕ. The auxiliary function $DropIds$ ignores the unique identifiers of numbered constraints. They don't matter when finding a rule head match. The guard t_g must be entailed by the primitive (here equations) store

constraints under the substitution ϕ.

In case of a simplified goal, step (Simplify), we apply the rule instance of r by deleting all simplified matching constraints H_S and adding the rule body instance $\phi(B)$ into the goals. Since $c\#i$ is simplified, we drop $c\#i$ from the goals as it does not exist in the store any more. In case of a propagated goal, step (Propagate), $c\#i$ remains in the goal set as well in the store and thus can possibly fire further rules instances. For both (Simplify) and (Propagate) derivation step, say $\sigma \overset{H_P \backslash H_S}{\rightarrowtail_{\mathcal{G}}} \sigma'$, we record as side-effect the numbered constraints in the store that were propagated (H_P) or simplified (H_S) during the derivation step. We will elaborate on the purpose of side-effects when we introduce the concurrent part of the semantics.

In step (Drop), we remove an active constraint from the set of goals, if the constraint failed to trigger any CHR rule.

Rule (Solve) moves an equation goal e into the store and *wakes up* (reactivates) any numbered constraint in the store which can possibly trigger further CHR rules due to the presence of e. Here is a simple example to show why reactivation is necessary.

$$r1 \ @ \ A(x), B(x) \Longleftrightarrow C(x)$$

		$\langle \{a = 2\} \mid \{A(a)\#1, B(2)\#2\} \rangle$
(Solve)	$\overset{\{A(2)\#1\} \backslash \{\}}{\rightarrowtail_{\mathcal{G}}}$	$\langle \{A(2)\#1\} \mid \{A(2)\#1, B(2)\#2, a = 2\} \rangle$
(Simp $r1$)	$\overset{\{\} \backslash \{A(2)\#1, B(2)\#2\}}{\rightarrowtail_{\mathcal{G}}}$	$\langle \{C(2)\} \mid \{a = 2\} \rangle$
...		

For clarity, we normalize all constraints in the store once an equation is added. Prior to addition of $a = 2$, $A(a)\#1, B(2)\#2$ cannot fire rule $r1$. After adding $a = 2$ however, we can normalize $A(a)\#1$ to $A(2)\#2$, which can now fire $r1$ with $B(2)\#2$. To guarantee exhaustive rule firings, we reactivate $A(2)\#2$ by adding it back to the set of goals. $WakeUp(e, Sn)$ represents a conservative approximation of the to be reactivated constraints [9]. Note we treat reactivated constraints as propagated

$$\textbf{(Lift)} \qquad \frac{\langle G \mid Sn\rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle G' \mid Sn'\rangle}{\langle G \mid Sn\rangle \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \langle G' \mid Sn'\rangle}$$

$$\textbf{(Goal Concurrency)} \qquad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} \langle G_1' \mid H_{S2} \cup S\rangle \\ \langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} \langle G_2' \mid H_{S1} \cup S\rangle \\ \delta_1 = H_{P1}\backslash H_{S1} \quad \delta_2 = H_{P2}\backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2}\backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{c}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \quad \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

$$\textbf{(Closure)} \qquad \frac{\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma'}{\sigma \rightarrowtail^*_{||\mathcal{G}} \sigma'} \qquad \frac{\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma' \quad \sigma' \rightarrowtail^*_{||\mathcal{G}} \sigma''}{\sigma \rightarrowtail^*_{||\mathcal{G}} \sigma''}$$

Figure 3.4: Goal-Based CHR Semantics (Concurrent Part $\overset{\delta}{\rightarrowtail}_{||\mathcal{G}}$)

constraints in the side-effects.

Figure 3.4 presents the concurrent part of the goal-based operational semantics. In the (Lift) step, we turn a sequential goal-based derivation into a concurrent derivation. Note that side-effects are retained. Step (Goal Concurrency) joins together two concurrent derivations operating on a shared store, if their rewriting side-effects δ_1 and δ_2 are non-overlapping as defined below.

Definition 4 (Non-overlapping Rewriting Side-Effects) *Two rewriting side-effects $\delta_1 = H_{P1}\backslash H_{S1}$ and $\delta_2 = H_{P2}\backslash H_{S2}$ are said to be non-overlapping, if and only if $H_{S1} \cap (H_{P2} \cup H_{S2}) = \{\}$ and $H_{S2} \cap (H_{P1} \cup H_{S1}) = \{\}$*

Concurrent derivations with non-overlapping side-effects essentially simplify distinct constraints in the store, as well as propagate constraints which are not simplified by one another. The (Goal Concurrency) step express non-overlapping side-effects by structurally enforcing that simplified constraints H_{S1} and H_{S2} match distinct parts of the store, while propagated constraints H_{P1} and H_{P2} are found in

the shared part of the store S not modified by both concurrent derivations. In the resulting concurrent derivation, the side-effects δ_1 and δ_2 are composed by the union of the propagate and simplify components respectively, forming δ.

The (Closure) step, defines transitive application of the concurrent goal-based derivation. Because side-effect labels are only necessary for the (Goal Concurrency) step, we drop the side-effects in transitive derivations.

An immediate consequence is that we can execute k derivations concurrently by stacking them together as long as all side-effects are mutually non-overlapping. The following lemma summarizes this observation.

Lemma 1 (k-Concurrency) *For any finite k of mutually non-overlapping concurrent derivations,*

$$\frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \overset{H_{P1}\backslash H_{S1}}{\rightarrowtail_{||\mathcal{G}}} \langle G_1' \mid \{\} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \\ ... \\ \langle G_i \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \overset{H_{Pi}\backslash H_{Si}}{\rightarrowtail_{||\mathcal{G}}} \langle G_i' \mid H_{S1} \cup ... \cup \{\} \cup ... \cup H_{Sk} \cup S\rangle \\ ... \\ \langle G_k \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \overset{H_{Pk}\backslash H_{Sk}}{\rightarrowtail_{||\mathcal{G}}} \langle G_k' \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup \{\} \cup S\rangle \\ H_{P1} \subseteq S ... H_{Pi} \subseteq S ... H_{Pk} \subseteq S \\ \delta = H_{P1} \cup ... \cup H_{Pi} \cup ... \cup H_{Pk} \backslash H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk}\end{array}}{\begin{array}{l}\langle G_1 \uplus ... \uplus G_i \uplus ... \uplus G_k \uplus G \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \\ \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \quad \langle G_1' \uplus ... \uplus G_i' \uplus ... \uplus G_k' \uplus G \mid S\rangle\end{array}}$$

we can decompose this into $k-1$ applications of the (pair-wise) (Goal Concurrency) derivation step.

Compared to the abstract CHR semantics, the $\| \mathcal{G}$ semantics provides us with a more systematic parallel execution scheme for executing CHR programs. In Section 3.4, we discuss the important insights and observations we gained from this

formulation of parallel goal-based execution, while in Section 3.5, we will detail our correspondence results showing that any concurrent goal-based derivation of the $\| \mathcal{G}$ semantics can be reproduced in the abstract semantics ($\mathcal{A}$ semantics).

3.4 Discussions

Most of the issues we encounter are related to the problem of ensuring exhaustive rule firings in a concurrent execution environment. These are subtle but important characteristics of our concurrent goal-based semantics which ultimately contributes to it's correctness and correspondence to the abstract CHR semantics. As we shall see in Chapter 4, these issues have also greatly influence the design decisions we made in our concrete implementation of $\| \mathcal{G}$. For brevity, we omit side-effects in derivation steps in the following examples as they don't matter.

3.4.1 Goal Storage Schemes and Concurrency

Recall that in the (Activate) rule of Figure 3.3 we specify that a goal constraint c is immediately stored in the shared constraint store. It may seem that this decision is made rather precariously and may even impose an non-optimal condition that active goals are immediately made visible in the shared constraint store. Yet we wish to highlight that immediate goal storage (as dictated by (Activate)) is crucial for exhaustiveness of rule firing.

Let's consider a concrete example. Suppose we would only store goals after execution (rule head matching). That is, we do not add the goals into the store during (Activate) step, but only during the (Drop) step, as illustrated by the (Activate') and (Drop') rules we shall consider instead:

$$\text{(Activate')} \quad \frac{i \text{ is a fresh identifier}}{\langle \{c\} \uplus G \mid Sn \rangle \rightarrowtail_{\mathcal{G}} \langle \{c\#i\} \uplus G \mid Sn \rangle}$$

$$\text{(Drop')} \quad \frac{\text{(Simplify) and (Propagate) does not apply on } c\#i \text{ in } Sn}{\langle \{c\#i\} \uplus G \mid Sn \rangle \rightarrowtail_{\mathcal{G}} \langle G \mid \{c\#i\} \cup Sn \rangle}$$

Then, for the CHR program

$$r1 \ @ \ A(x), B(y) \Longleftrightarrow C(x, y)$$

we obtain the following derivation

$$\langle \{A(1), B(2)\} \mid \{\} \rangle$$

$$\frac{\begin{array}{c}\text{(Activate')} \quad \langle \{A(1)\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{A(1)\#1\} \mid \{\} \rangle \\ || \\ \text{(Activate')} \quad \langle \{B(2)\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{B(2)\#2\} \mid \{\} \rangle\end{array}}{\langle \{A(1), B(2)\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{A(1)\#1, B(2)\#2\} \mid \{\} \rangle}$$

$$\frac{\begin{array}{c}\text{(Drop')} \quad \langle \{A(1)\#1\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{A(1)\#1\} \rangle \\ || \\ \text{(Drop')} \quad \langle \{B(2)\#2\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{B(2)\#2\} \rangle\end{array}}{\langle \{A(1)\#1, B(2)\#2\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{A(1)\#1, B(2)\#2\} \rangle}$$

Initially both goals $A(1)$ and $B(2)$ are concurrently activated. Since (Activate') does not store goals immediately, both active goals are not visible to each other in

the store. Hence, we wrongfully apply the (Drop') step for both goals. However, there is clearly a complete rule head match $A(1)\#1, B(2)\#2$.

Note that for the actual $\| \mathcal{G}$ semantics (Figure 3.3), we will most certainly take the (Simplify) and (Drop) derivations concurrently after activating both goals $A(1)$ and $B(2)$ [3], thus the rule head match $A(1)\#1, B(2)\#2$ will be triggered. While this necessarily introduces contention between active goal constraints as a side-effect (eg. we now have both goals $A(1)$ and $B(2)$ attempting to fire rule head match $A(1)\#1, B(2)\#2$), it sufficiently guarantees exhaustive rule firing.

3.4.2 Derivations under 'Split' Constraint Store

Notice that our definition of the (Goal Concurrency) rule (Figure 3.4) dictates that concurrent derivations must be done under the context of the entire store. In other words, we cannot split the store and derive concurrent derivations under small subsets of the store. For convenience, we present the (Goal Concurrency) rule again as follows:

$$\textbf{(Goal Concurrency)} \quad \frac{\begin{array}{c}(\text{D1}) \quad \langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} \langle G_1' \mid H_{S2} \cup S\rangle \\ (\text{D2}) \quad \langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} \langle G_2' \mid H_{S1} \cup S\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{l}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \quad \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

Note that we have labeled the premise derivations with (D1) and (D2). By *not splitting* the constraint store, we imply that concurrent derivations (D1) and (D2) must involve the entire store context (i.e. $H_{S1} \cup H_{S2} \cup S$) even though each derivation

[3] (Simplify) and (Drop) derivations by be taken by the goals $A(1)\#1$ and $B(2)\#2$ respectively, or vice versa. But both computation paths are confluent (leads to the same results). Also note that the side-effects (δ) of the $\| \mathcal{G}$ semantics prevents both goals from concurrently firing (Simplify) steps.

seemingly do not involve all constraints in the store. For instance, (D1) does not modify constraints in H_{S2}. This is as oppose to the concurrency rule of the CHR abstract semantics (Figure 2.2) in which derivations in the premise are evaluated under a smaller context of the store.

We will now show that the non-splitting nature of the (Goal-Concurrency) rule is necessary for the exhaustiveness of rule firing in the $\| \mathcal{G}$ semantics. Suppose we allow for concurrent executions on split stores. Specifically, let's assume the following replacement of the (Goal Concurrency) rule:

$$\textbf{(Goal Concurrency')} \quad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup S\rangle \rightarrowtail^{\delta_1}_{||\mathcal{G}} \langle G_1' \mid S\rangle \\ \langle G_2 \mid H_{S2} \cup S\rangle \rightarrowtail^{\delta_2}_{||\mathcal{G}} \langle G_2' \mid S\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{l}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \rightarrowtail^{\delta}_{||\mathcal{G}} \quad \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

Note that the (Goal Concurrency') rule attempts to mimic the concurrency rule of the CHR abstract semantics, by allowing derivations in the premise to be taken under a smaller context of the store. With this rule instead (as oppose to (Goal Concurrency)), we can construct the following derivation:

$$r1 \ @ \ A, B \iff C \qquad r2 \ @ \ D, E \iff F$$

$$\frac{\begin{array}{c}\text{(Drop)} \ \ \langle \{A\#3\} \mid \{A\#3, E\#2\}\rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{A\#3, E\#2\}\rangle \\ \text{(Drop)} \ \ \langle \{D\#4\} \mid \{B\#1, D\#4\}\rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{B\#1, D\#4\}\rangle\end{array}}{\langle \{A\#3, D\#4\} \mid \{A\#3, B\#1, D\#4, E\#2\}\rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{A\#3, B\#1, D\#4, E\#2\}\rangle}$$

For brevity, we omit the earlier derivations which activates and drops $B\#1$ and $E\#2$ as goals. The above derivation illustrates the concurrent execution of the

subsequent goals, $A\#3$ and $D\#4$. Both goals are dropped, since under their local store context, no match to *r1* or *r2* can be completed. However, if we consider the entire store $\{A\#3, E\#2, B\#1, D\#4\}$, it's clearly that goal $A\#3$ can execute rule *r1* and goal $D\#4$ can execute rule *r2*. This definitively shows that considering the $\| \mathcal{G}$ semantics with the (Goal Concurrency') rule, we do not exhaustively apply CHR rules, hence cannot derive a correspondence with the CHR abstract semantics.

We wish to highlight that the reason why (Goal Concurrency') is incorrect, lies in the fact that the $\| \mathcal{G}$ semantics is not *monotonic* in the way that the CHR abstract semantics is monotonic, hence concurrency rule illustrated in Figure 2.2 does not apply for $\| \mathcal{G}$ semantics. We will further discuss what this loss of *classic monotonicity* means to our semantics in Section 3.4.4.

3.4.3 Single-Step Derivations in Concurrent Derivations

The issues discussed in Section 3.4.1 and 3.4.2 illustrates the fundamental causes of non-exhaustiveness of the $\| \mathcal{G}$ semantics if it is defined otherwise. In this section we shall discuss the similar effects that are observed, if we allow multiple step derivations in concurrent derivation steps. For this discussion, notice that the (Goal Concurrency) rule in Figure 3.4 restricts concurrent derivations in it's premise to strictly only single-step derivations. Let us assume the following (Goal Concurrency") rule which allows otherwise:

$$\textbf{(Goal Concurrency")}\quad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \rightarrowtail^{*}_{||\mathcal{G}}{}^{\delta_1} \langle G_1' \mid H_{S2} \cup S\rangle \\ \langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \rightarrowtail^{*}_{||\mathcal{G}}{}^{\delta_2} \langle G_2' \mid H_{S1} \cup S\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{l}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \rightarrowtail^{*}_{||\mathcal{G}}{}^{\delta} \quad \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

While this modification may seem subtle and unassuming, it's implications are not. Note that by allowing multiple derivation steps, we potentially allow derivations to be taken under a smaller context of the constraint store (exactly what we wish to avoid in Section 3.4.2). To illustrate this, we consider an example:

$$r1 \ @ \ A, B \Longleftrightarrow C$$

$$\frac{\begin{array}{ll}(P1) & \langle \{A\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{A\#2\} \mid \{A\#2\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{A\#2\} \rangle \\ (P2) & \langle \{B\} \mid \{\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{B\#3\} \mid \{B\#3\} \rangle \rightarrowtail_{||\mathcal{G}} \langle \{\} \mid \{B\#3\} \rangle \end{array}}{\langle \{A, B\} \mid \{\} \rangle \rightarrowtail^*_{||\mathcal{G}} \langle \{\} \mid \{A\#2, B\#3\} \rangle}$$

The sequence of derivation steps (P1) first activates A which is then dropped. Similarly, (P2) activates B which is then dropped as well which then leads to the stuck state $\langle \{\} \mid \{A\#2, B\#3\} \rangle$. We clearly missed to fire rule *r1*. This shows that single-step concurrent execution are essential to guarantee that newly added constraints are visible to all concurrent active goals, hence we have exhaustive rule firings in the goal-based semantics.

3.4.4 CHR Monotonicity and Shared Store Goal-based Execution

It is clear that the issues discussed in Sections 3.4.1, 3.4.2 and 3.4.3 are some what related. All of which relates to the exhaustiveness of rule firing and the each measure taken contributes to maintaining a *global* or *shared* view of the constraint store for all executing CHR goal constraints. In essence, the underlying reason for the need for a shared store is that the goal-based semantics is not monotonic in its store argument, thus derivations can only be consistent if taken under the full context of the constraint store.

Let's consider an example, which illustrates the non-monotonicity of the CHR

goal-based semantics $\mathcal{G}$, with respect to the constraint store.

$$r1 \ @ \ A, B \Longleftrightarrow C$$

Step	**Transition Type**		**Constraint Store**
			$\langle\{A\} \mid \{\}\rangle$
$D1$	(Activate)	$\rightarrowtail_{\mathcal{G}}$	$\langle\{A\#1\} \mid \{A\#1\}\rangle$
$D2$	(Drop)	$\rightarrowtail_{\mathcal{G}}$	$\langle\{\} \mid \{A\#1\}\rangle$

We consider a simple derivation of the rule $r1$. In Step $D1$ we activate the goal A, thus adding $A\#1$ to the initially empty store. Since we cannot build a complete match of the rule $r1$, we drop the goal $A\#1$ in Step $D2$. If the goal-based semantics is monotonic with respect to the constraint store, we should be able to execute this derivation sequence under a larger context of the constraint store. Yet the following clearly shows that this is not possible:

Step	**Transition Type**		**Constraint Store**
			$\langle\{A\} \mid \{B\#2\}\rangle$
$D1'$	(Activate)	$\rightarrowtail_{\mathcal{G}}$	$\langle\{A\#1\} \mid \{A\#1, B\#2\}\rangle$
$D2'$	(Drop)	$\not\rightarrowtail_{\mathcal{G}}$	$\langle\{\} \mid \{A\#1, B\#2\}\rangle$

Note that we have extended the store with $B\#2$. Step $D1'$ is still valid but Step $D2'$ is clearly invalid, since the (Simplify) rule is applicable on the goal $A\#1$, violating the conditions of dropping the goal $A\#1$.

This shows that the goal-based semantics is non-monotonic with respect to the constraint store, specifically the (Drop) rule cannot always be taken under a larger context of the store. As a result, we must take the appropriate measures discussed in Sections 3.4.1, 3.4.2 and 3.4.3, which collectively imposes a *shared* store restriction on our $\| \mathcal{G}$ semantics. While this formal complexity is a small price to pay, it's benefits are worthwhile as it provides a formal concise description of the behaviour of parallel CHR goal execution over a shared constraint store.

While the goal-based semantics $\mathcal{G}$ is non-monotonic with respect to the constraint store, it is monotonic with respect to the goals. This monotonic property of the goals offers a great degree of flexibility for concurrent execution of CHR goals. We highlight this in the next section.

3.4.5 Lazy Matching and Asynchronous Goal Execution

When executing goals, we lazily compute only matches that contain the specific goal and immediately apply such matches without concerning any further matches. For instance consider the following CHR program and goal-based derivation:

$$r0 \ @ \ A(x), B(y) \iff D(x, y)$$

$$\begin{aligned} & \langle \{A(1)\#4\} \uplus \{A(2), A(3)\} \mid \{B(2)\#1, B(3)\#2, B(4)\#3, A(1)\#4\} \rangle \\ \rightarrowtail_{\mathcal{G}} \quad & \langle \{D(1,2)\} \uplus \{A(2), A(3)\} \mid \{B(3)\#2, B(4)\#3\} \rangle \end{aligned}$$

We have applied the rule instance $A(1)\#4, B(2)\#1$ independently of the existence of the other goals (i.e. $\{A(2), A(3)\}$). In the literature, such a matching scheme is known as a *lazy* matching scheme, and often implemented by variants of the LEAPS algorithm [8].

Lazy matching in the goal-based semantics is possible only because the goal-based semantics is monotonic with respect to the set of goals. The following illustrates

this monotonicity property of goals:

$$\frac{\begin{array}{rl} & \langle \{A(1)\#4\} \mid \{B(2)\#1, B(3)\#2, B(4)\#3, A(1)\#4\}\rangle \\ \rightarrowtail_{\mathcal{G}} & \langle \{D(1,2)\}\mid \{B(3)\#2, B(4)\#3\}\rangle \end{array}}{\begin{array}{rl} & \langle \{A(1)\#4\} \uplus \{A(2), A(3)\} \mid \{B(2)\#1, B(3)\#2, B(4)\#3, A(1)\#4\}\rangle \\ \rightarrowtail_{\mathcal{G}} & \langle \{D(1,2)\} \uplus \{A(2), A(3)\}\mid \{B(3)\#2, B(4)\#3\}\rangle \end{array}}$$

$$\frac{\langle G \mid Sn\rangle \rightarrowtail_{\mathcal{G}} \langle G' \mid Sn'\rangle}{\langle G \uplus G'' \mid Sn\rangle \rightarrowtail_{\mathcal{G}} \langle G' \uplus G'' \mid Sn'\rangle}$$

The above property essentially states that we can execute goals G without prior knowledge of goals G''. Because of monotonicity, we are guaranteed that future executions of G'' will not invalid the executions on G.

Monotonicity of the goals also allows us to execute goals asynchronously. By 'asynchronously', we mean that goals need not *explicitly* synchronize with one another during their execution. For instance, consider the following:

$$r1 \ @ \ A(x), B(y) \Longleftrightarrow C(x,y)$$

$$\frac{\begin{array}{rl} & \langle \{A(1)\#1\} \mid \{A(1)\#1, B(2)\#2\} \cup \{A(3)\#3, B(4)\#4\}\rangle \\ \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(1,2)\}\mid \{\} \cup \{A(3)\#3, B(4)\#4\}\rangle \\ & \langle \{A(3)\#3\} \mid \{A(1)\#1, B(2)\#2\} \cup \{A(3)\#3, B(4)\#4\}\rangle \\ \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(3,4)\}\mid \{A(1)\#1, B(2)\#2\} \cup \{\}\rangle \end{array} \quad \begin{array}{c} \delta_1 = \{\}\backslash\{A(1)\#1, B(2)\#2\} \quad \delta_2 = \{\}\backslash\{A(3)\#3, B(4)\#4\} \\ \delta = \{\}\backslash\{A(1)\#1, B(2)\#2, A(3)\#3, B(4)\#4\} \end{array}}{\begin{array}{rl} & \langle \{A(1)\#1\} \uplus \{A(3)\#3\} \mid \{A(1)\#1, B(2)\#2\} \cup \{A(3)\#3, B(4)\#4\}\rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(1,2)\} \uplus \{C(3,4)\}\mid \{\} \cup \{\}\rangle \end{array}}$$

$$r1 \ @ \ A(x) \Longleftrightarrow C(x)$$
$$r2 \ @ \ A(x), B(x) \Longleftrightarrow D(x)$$

Transition Types		**Constraint Store**
		$\langle \{A(1), A(2), B(1), B(2)\} \mid \{\} \rangle$
(Activate), (Simplify r1)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{C(1), A(2), B(1), B(2)\} \mid \{\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{A(2), B(1), B(2)\} \mid \{C(1)\#1\} \rangle$
(Activate), (Simplify r1)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{C(2), B(1), B(2)\} \mid \{C(1)\#1\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{B(1), B(2)\} \mid \{C(1)\#1, C(2)\#2\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{B(2)\} \mid \{C(1)\#1, C(2)\#2, B(1)\#3\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{\} \mid \{C(1)\#1, C(2)\#2, B(1)\#3, B(2)\#4\} \rangle$

Figure 3.5: Goal/Rule occurrence ordering example

The above describes the concurrent execution of goals $A(1)\#1$ and $A(3)\#3$. Notice that in the derivations of the premise, we can ignore all goals which are not relevant to the derivation. For instance, execution of $A(1)\#1$ does not need goal $A(3)\#3$ to be visible, hence the goals effectively executes asynchronously. Goals do however, implicitly "synchronize" via the shared store. Namely, concurrent derivations must be chosen such that rewrite side-effects involve distinct parts of the store. In Chapter 4, we will discuss efficient means of imposing such restrictions on the side-effects of CHR goals executions in parallel.

3.4.6 Goal and Rule Occurrence Ordering

In this section, we address two issues, namely goal and rule ordering. We will consider the example in Figure 3.5 to illustrate our points in this Section.

Goal ordering refers to the order in which goals are activated and executed. For instance in Figure 3.5, the derivation sequence in this derivation sequence assumes a stack ordering of the goals. This means that we always activate the left-most goal first, and add new goals to the left of the collection as well. Note that for clarity, we assume sequential goal execution for now (Goals are activated one at the time, hence no concurrent goal execution). *Rule occurrence ordering* refers to the order in

which rule-heads are tried by active goals. For instance, if we pick a top-to-bottom ordering for rule-head execution, given a goal $A(i)\#n$, we will always try to match it with $A(x)$ of $r1$ before $A(x)$ of $r2$. Derivation sequence in Figure 3.5 is essentially an example of such a top-to-bottom rule-head execution ordering, hence we will never trigger $r2$ because we can alway match a goal $A(i)\#n$ to $r1$.

One might have noticed that for our goal-based semantics ($\| \mathcal{G}$ semantics, Section 3.3) goals are specified as an unordered multiset[4] and each order tries rule-heads in an unspecified order. By not over specifying goal and rule-head ordering, our goal-based semantics can describe a wider range of behaviours. For instance, the following derivation sequence is still valid as:

Transition Types		**Constraint Store**
		$\langle \{A(1), A(2), B(1), B(2)\} \mid \{\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{A(1), A(2), B(2)\} \mid \{B(1)\#1\} \rangle$
(Activate), (Simplify r2)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{A(2), B(2), C(1)\} \mid \{\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{A(2), C(1)\} \mid \{B(2)\#3\} \rangle$
(Activate), (Simplify r2)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{C(1), C(2)\} \mid \{\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{C(2)\} \mid \{C(1)\#5\} \rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\|\|\mathcal{G}}$	$\langle \{\} \mid \{C(1)\#5, C(2)\#6\} \rangle$

This derivation shows a sequence which stack ordering is not used nor is top-to-bottom ordering. Note that from the constraint store $\{B(1)\#1, A(1)\#2\}$ the activate goal $A(1)\#2$ is free to choose to fire $r1$ or $r2$. If a strict top-to-bottom ordering is used, we are restricted to fire only $r1$. This degree of freedom is important, because the choice of goal order or rule-head order may be determined by domain specific reasons and should not be restricted by the underlying semantics (Section 4.4.3 and 5.3.3 explores this in an implementational point of view).

The last point we wish to highlight is that the $\| \mathcal{G}$ semantics is essentially sim-

[4]The traditional refined CHR operational semantics restrict this to a stack (fixed execution order).

ilar to the refined CHR operational semantics [9] if we impose *all* of the following limitations on the $\| \mathcal{G}$ semantics:

- Goals are activated in left-to-right order and new goals are added only to the right.
- Rule-heads are tried in top-to-bottom right-to-left ordering.
- Only exactly one goal is active at a time.

The first two restrictions are explicit restrictions of the refined CHR operational semantics, hence their requirement are obvious. The third however is less obvious. The problem is that even with stack goal ordering and top-to-bottom rule-head ordering, executing goals concurrently introduces non-determinism which the refined CHR operational semantics cannot reproduce. For instance consider the following example:

Transition Types		**Constraint Store**
		$\langle \{A(1), B(1), ...\} \mid \{\} \rangle$
(Activate) $\|\|$ (Activate)	$\rightarrowtail_{\|\|\mathcal{G}}$	$\langle \{A(1)\#1, B(1)\#2, ...\} \mid \{A(1)\#1, B(1)\#2\} \rangle$
(Drop) $\|\|$ (Simplify r2)	$\rightarrowtail_{\|\|\mathcal{G}}$	$\langle \{C(1), ...\} \mid \{\} \rangle$
...		

We assume that we have 2 execution threads. In the first step, we execute the two left-most goals, namely $A(1)\#1$ and $B(1)\#2$. In this second step, we have goal $B(1)\#2$ firing the CHR rule $r2$, while $A(1)\#1$ is simply dropped. Note that this is not possible if we disallow activation of multiple goals, since after executing $A(1)$ we must match it with $r1$.

As such, the $\| \mathcal{G}$ semantics is obviously more in-deterministic compared to the refined CHR operational semantics. Section 3.5 presents our proofs that the $\| \mathcal{G}$ semantics corresponds to the abstract CHR semantics ($\mathcal{A}$ semantics).

3.4.7 Dealing with Pure Propagation

The CHR $\mathcal{A}$ semantics and $\| \mathcal{G}$ semantics here does not include simpagation rules with empty simplification heads, i.e. pure propagation rules[5] of the form $H \Longrightarrow t_g \mid B$. Handling pure propagtion rules for the sequential goal-execution semantics is a well-studied problem [9, 33] and standard techniques used there (propagation histories) varies little from how they would be used in concurrent goal execution context. Such histories are necessary to prevent similar rule head instances to be triggered infinitely many times. The inclusion of propagation histories to the $\| \mathcal{G}$ semantics is straight-forward but will add little to contributions formal results. For instance, CHR states can be extended with a propagation history P (i.e. $\langle G \mid Sn \mid P \rangle$). Propagation histories contains information on rules that has fired, typically tuples consisting of a set of constraint identifiers (the constraints involved in the multiset rewriting) and a rule label (rule involved in the multiset rewriting). The following illustrates a simple extension to the $\| \mathcal{G}$ transition rules (Figure 3.3), to handling pure propagation rules:

$$\textbf{(Pure Propagate)} \quad \frac{\begin{array}{c}(r \ @ \ H' \Longrightarrow t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists \phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H') = DropIds(H) \\ \delta = \{c\#j\} \cup H \backslash \{\} \quad h \equiv \{i \mid c\#i \in H\} \cup \{r\} \\ h \notin P\end{array}}{\begin{array}{c}\langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H \cup Sn \mid P \rangle \\ \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \quad \langle \phi(B') \uplus \{c\#j\} \uplus G \mid \{c\#j\} \cup H \cup Sn \mid h \cup P \rangle\end{array}}$$

This simply includes a propagation history P which keeps track of all rule instances that has fired. As specified by the premise of the transition rule, propagation rule instance that has already been fired will not be applicable.

For concurrent goal execution, on top of enforcing the uniqueness of propagation

[5] While this may be upsetting for some advocates of Constraint Handling Rules, we insist that it's exclusion has little impact on the concurrent semantics and is purely for brevity and readability of the formalism.

rule instances, propagation histories has an additional responsibility of providing the guarantee that concurrent goal executions rewrite over unique propagation rule instances. Propagation histories need to be synchronized between concurrent goals, with the additional condition that propagation histories cannot be overlapping. The following variant of the (Goal Concurrency) transition rule, handling pure propagation between concurrent goal execution.

$$\textbf{(Goal Concurrency')}\quad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup H_{S2} \cup S \mid P\rangle \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} \langle G_1' \mid H_{S2} \cup S \mid P_1 \cup P\rangle \\ \langle G_2 \mid H_{S1} \cup H_{S2} \cup S \mid P\rangle \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} \langle G_2' \mid H_{S1} \cup S \mid P_2 \cup P\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2} \\ P_1 \cap P_2 \equiv \{\}\end{array}}{\begin{array}{l}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S \mid P\rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \quad \langle G_1' \uplus G_2' \uplus G \mid S \mid P_1 \cup P_2 \cup P\rangle\end{array}}$$

Essentially, we only allow concurrent goal executions which do not propagate the same propagation rule instances ($P_1 \cap P_2 \equiv \{\}$). This sufficiently guarantees that concurrent goal executions rewrites unique instances of propagation rules.

In Section 4.8.2, we shall briefly discuss the technical challenges that will be faced when implementing shared global histories for concurrent goal execution.

3.5 Correspondence Results

We formally verify that the concurrent goal-based semantics is in exact correspondence to the abstract CHR semantics when it comes to termination and exhaustive rule firings. Detailed proofs are given in the appendix (Chapter A). In the following sections, we provide key lemmas and proof sketches.

3.5.1 Formal Definitions

We first introduce some elementary definitions before stating the formal results.

The first two definitions concern the abstract CHR semantics. A store is final if no further rules are applicable.

Definition 2 (Final Store) *A store S is known as a final store, denoted $Final_{\mathcal{A}}(S)$ if and only if no more CHR rules applies on it (i.e. $\neg\exists S'$ such that $S \rightarrowtail_{\mathcal{A}} S'$).*

A CHR program terminates if all derivations lead to a final store in a finite number of states.

Definition 3 (Terminating CHR Programs) *A CHR program $\mathcal{P}$ is said to be terminating, if and only if for any CHR store S, there exists no infinite derivation paths from S, via the program $\mathcal{P}$.*

Next, we introduce some definitions in terms of the goal-based semantics. In an initial state, all constraints are goals and the store is empty. Final states are states which no longer have any goals. We will prove the exhaustiveness of the goal-based semantics by proving a correspondence between final stores in the abstract semantics and final states of the goal-based semantics

Definition 5 (Initial and Final CHR States) *An initial CHR state is a CHR state of the form $\langle G \mid \{\}\rangle$ where G contains no numbered constraints ($c\#n$), while a final CHR state is of the form $\langle \{\} \mid Sn\rangle$*

A state is reachable if there exists a (sequential) goal-based sequence of derivations to this state. We write $\rightarrowtail^*_{\mathcal{G}}$ to denote the transitive closure of $\rightarrowtail_{\mathcal{G}}$.

Definition 6 (Sequentially Reachable CHR states) *For any CHR program $\mathcal{P}$, a CHR state $\langle G' \mid Sn'\rangle$ is said to be sequentially reachable by $\mathcal{P}$ if and only if there exists some initial CHR state $\langle G \mid \{\}\rangle$ such that $\langle G \mid \{\}\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid Sn'\rangle$.*

3.5.2 Correspondence of Derivations

We build a correspondence between the abstract semantics and the concurrent goal-based semantics. We begin with Theorem 2, which states the correspondence of the (sequential) goal-based semantics.

Theorem 2 (Correspondence of Sequential Derivations) *For any reachable CHR state $\langle G \mid Sn\rangle$, CHR state $\langle G' \mid Sn'\rangle$ and CHR program $\mathcal{P}$,*

$$\begin{array}{lll} \mathit{if} & \langle G \mid Sn\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid Sn'\rangle & \\ \mathit{then} & (NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn')) & \vee \\ & (NoIds(G) \uplus DropIds(Sn)) \rightarrowtail^*_{\mathcal{A}} (NoIds(G') \uplus DropIds(Sn')) & \end{array}$$

where $NoIds = \{c \mid c \in G, c \text{ is a CHR constraint}\} \uplus \{e \mid e \in G, e \text{ is an equation}\}$

The above result guarantees that any sequence of sequential goal-based derivations starting from a reachable CHR state either yields equivalent CHR abstract stores (due to goal-based behaviour not captured by the abstract semantics, namely (Solve) (Activate), (Drop)) or corresponds to a derivation in the abstract semantics (due to rule application). A goal-based semantics state $\langle G \mid Sn\rangle$ is related to an abstract semantics store by removing all numbered constraints in G and union it with constraints in Sn without their identifiers. The theorem and its proof is a generalization of an earlier result given in [9].

We formalize the observation that the goal context can be extended without interfering with previous goal executions.

Lemma 2 (Monotonicity of Goals in Goal-based Semantics) *For any goals G,G' and G'' and CHR store Sn and Sn', If $\langle G \mid Sn\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid Sn'\rangle$ then $\langle G \uplus G'' \mid Sn\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \uplus G'' \mid Sn'\rangle$.*

Next, we state that given any goal-based derivation with side-effects δ, we can safely ignore any constraints (represented by S_2) in the store which is not part of δ.

Lemma 3 (Isolation of Goal-based Derivations)

$$\begin{array}{ll} If & \langle G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \overset{H_P \backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1' \cup S_2 \rangle \\ then & \langle G \mid H_P \cup H_S \cup S_1 \rangle \overset{H_P \backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1' \rangle \end{array}$$

Lemma 3 can be straight-forwardly extended to multiple derivation steps. This is stated in Lemma 4.

Lemma 4 (Isolation of Transitive Goal-based Derivations)

$$\begin{array}{ll} If & \langle G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \rightarrowtail_{\mathcal{G}}^* \langle G' \mid H_P \cup S_1' \cup S_2 \rangle \\ & \textit{with side-effects } \delta = H_P \backslash H_S \\ then & \langle G \mid H_P \cup H_S \cup S_1 \rangle \rightarrowtail_{\mathcal{G}}^* \langle G' \mid H_P \cup S_1' \rangle \end{array}$$

The next states that any concurrent derivation starting from a reachable CHR state can be replicated by a sequence of sequential goal-based derivations. Lemma 5 is the first step to prove the correspondence of concurrent goal-based derivations.

Lemma 5 (Sequential Reachability of Concurrent Derivation Steps) *For any sequentially reachable CHR state σ, CHR state σ' and rewriting side-effects δ if $\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma'$ then σ' is sequentially reachable, $\sigma \rightarrowtail_{\mathcal{G}}^* \sigma'$ with side-effects δ.*

> **Proof:**(Sketch) Via Lemma 1, we can always reduce k mutually non-overlapping concurrent derivations into several applications of the (Goal Concurrency) step. Hence we can prove Lemma 5 by structural induction over the concurrent goal-based derivation steps (Lift) and (Goal Concurrency) where we use Lemmas 2 and 4 to show that concurrent derivations can always be replicated by a sequence of sequential goal-based derivations. □

Theorem 3 (Sequential Reachability of Concurrent Derivations) *For any initial CHR state σ, CHR state σ' and CHR Program $\mathcal{P}$, if $\sigma \rightarrowtail_{||\mathcal{G}}^* \sigma'$ then $\sigma \rightarrowtail_{\mathcal{G}}^* \sigma'$.*

The above be directly proven from Lemma 5 by converting each single step concurrent derivation into a sequence of sequential derivations, and showing their composibility.

From Theorem 2 and 3, we have the following corollary, which states the correspondence between concurrent goal-based CHR derivations and abstract CHR derivations.

Corollary 1 (Correspondence of Concurrent Derivations) *For any reachable CHR state $\langle G \mid Sn\rangle$, CHR state $\langle G' \mid Sn'\rangle$ and CHR program $\mathcal{P}$,*

$$\begin{array}{lll} \textit{if} & \langle G \mid Sn\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G' \mid Sn'\rangle & \\ \textit{then} & (NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn')) & \vee \\ & (NoIds(G) \uplus DropIds(Sn)) \rightarrowtail^*_{\mathcal{A}} (NoIds(G') \uplus DropIds(Sn')) & \end{array}$$

where $NoIds = \{c \mid c \in G, c$ is a CHR constraint$\} \uplus \{e \mid e \in G, e$ is an equation$\}$

3.5.3 Correspondence of Exhaustiveness and Termination

We show that all derivations from a initial state to final states in the concurrent goal-based semantics corresponds to some derivation from a store to a final store in the abstract semantics. We first define rule head instances:

Definition 7 (Rule head instances) *For any CHR state $\sigma = \langle G, Sn\rangle$ and CHR program $\mathcal{P}$, any $(H_P \cup H_S) \subseteq Sn$ is known as a rule head instance of σ, if and only if $\exists(r \ @ \ H'_P \backslash H'_P \iff t_g \mid B) \in \mathcal{P}$,$\exists\phi \ Eqs(Sn) \models \phi \wedge t_g$ and $\phi(H'_P \uplus H'_S) = DropIds(H_P \cup H_S)$.*

Definition 8 (Active rule head instances) *For any CHR state $\sigma = \langle G, Sn\rangle$ and CHR program $\mathcal{P}$, a rule head instance H of σ is said to be* active *if and only if there exists at least one $c\#i \in G$ such that $c\#i \in H$.*

Rule head instances (Definition 7) are basically minimal subsets of the store which matches a rule head. Active rule head instance (Definition 8) additional have at least one of it's numbered constraint $c\#i$ in the goals as well. Therefore, by the definition of the goal-based semantics, active rule head instances will eventually be triggered by either the (Simplify) or (Propagate) derivation steps.

Lemma 6 (Rule instances in reachable states are always active) *For any reachable CHR state $\langle G \mid Sn\rangle$, any rule head instance $H \subseteq Sn$ must be active. i.e. $\exists c\#i \in H$ such that $c\#i \in G$.*

Lemma 6 shows that all rule head instances in reachable states are always active. This means that by applying the semantics steps in any way, we must eventually apply the rule head instances as long as all it's constraints remain in the store.

Theorem 4 states that the exhaustiveness of a concurrent goal-based derivation corresponds to exhaustiveness in the abstract semantics. Meaning that for every *terminating* CHR program and initial CHR state $\langle G, \{\}\rangle$, if the exhaustive application of concurrent goal-based derivations yields a final CHR state $\langle \{\}, Sn\rangle$, this state will correspond to a valid final state with respect to the abstract CHR semantics. Simply put, it guarantees that if the concurrent goal-based derivation terminates, the resultant state corresponds to a final CHR abstract state.

Theorem 4 (Correspondence of Exhaustiveness) *For any initial CHR state $\langle G, \{\}\rangle$, final CHR state $\langle \{\}, Sn\rangle$ and terminating CHR program $\mathcal{P}$,*

$$\begin{aligned}&\textit{if } \langle G \mid \{\}\rangle \rightarrowtail^*_{||\mathcal{G}} \langle \{\} \mid Sn\rangle\\ &\textit{then } G \rightarrowtail^*_{\mathcal{A}} DropIds(Sn) \textit{ and } Final_{\mathcal{A}}(DropIds(Sn))\end{aligned}$$

Proof:(Sketch) We prove this theorem by first using Theorem 3 which guarantees that a concurrent goal-based derivation from an initial state to a final state corresponds to some abstract semantics derivation. We

next show that final states corresponds to final stores in the abstract semantics. This is done by contradiction, showing that assuming otherwise contradicts with Lemma 6. □

To establish a property on the termination of the concurrent goal-based semantics we state Lemma 7 and 8, which respectively establishes that there are CHR states $\langle G \mid Sn \rangle$ which corresponds to some final abstract CHR state cannot be applied with (Simplify) or (Propagate) transitions, and that the cannot exist infinite concurrent derivations consisting of only (Solve), (Activate) and (Drop) transitions.

Lemma 7 (Terminal CHR State) *For any CHR State $\langle G \mid Sn \rangle$ and a terminating CHR program $\mathcal{P}$,*

if $Final_{\mathcal{A}}(NoIds(G) \uplus DropIds(Sn))$
then *there exists no proceeding concurrent derivation $\langle G \mid Sn \rangle \rightarrowtail_{||\mathcal{G}} \langle G' \mid Sn' \rangle$ that involves applications of the (Simplify) or (Propagate) derivation rules.*

Lemma 8 (Finite Administrative CHR Goal-Based Derivations) *For any CHR State $\langle G \mid Sn \rangle$, there cannot exist any infinite concurrent derivations consisting of only administrative derivation rules (Solve), (Activate) and (Drop).*

Theorem 5 states that for every concurrent CHR derivation that corresponds to an abstract CHR derivation which results in a final abstract state S', that concurrent CHR derivation would eventually terminate in a CHR state that corresponds with S'. Simply put, this gives us the guarantee that concurrent goal-based derivations that corresponds to a terminating abstract derivation, is terminating as well.

Theorem 5 (Correspondence of Termination) *For any initial CHR state $\langle G \mid \{\} \rangle$, any CHR state $\langle G' \mid Sn \rangle$ and a terminating CHR program $\mathcal{P}$,*

if $\langle G \mid \{\} \rangle \rightarrowtail^*_{||\mathcal{G}} \langle G' \mid Sn \rangle$ *and* $Final_{\mathcal{A}}(NoIds(G') \uplus DropIds(Sn))$
then $\langle G' \mid Sn \rangle \rightarrowtail^*_{||\mathcal{G}} \langle \{\} \mid Sn'' \rangle$ *and* $DropIds(Sn'') = NoIds(G') \uplus DropIds(Sn)$

Proof:(Sketch) We first use Lemma 7 to show that there can be no applications of (Simplify) or (Propagate) transitions from the state $\langle G' \mid Sn \rangle$, since we assume that $\langle G' \mid Sn \rangle$ corresponds to a final state in the abstract semantics. Next, using Lemma 8, we show that with the only permitted CHR transitions ((Solve), (Activate) and (Drop)) we can only have finite concurrent derivations. Finally, by defining a well-founded order based on the number of goals in a CHR state, we show that successive CHR states across CHR concurrent derivations are monotonically decreasing in this ordering, and with the assumption that $\langle G' \mid Sn \rangle$ corresponds to a final state in the abstract semantics, exhaustive derivations will yield that resultant state with empty goals, $\langle \{\} \mid Sn'' \rangle$. □

3.5.4 Concurrent CHR Optimizations

In the sequential setting, there exist a wealth of optimizations [9, 48, 50] to speed up the execution of CHR. Fortunately, many of these methods are still largely applicable to our concurrent goal-based variant as we discuss in the following. For the remainder, we assume that each goal (thread) tries the CHR rules from top-to-bottom to match the rule execution order assumed in [9, 48, 50].

Basic constraint indexing like lookups via hashtables are still applicable with minor adaptations. For instance, the underlying hashtable implementation must be thread safe. Consider the following example:

$$r0@A(x,y), B(x), C(y) \Longleftrightarrow x > y \mid D(x,y)$$

Suppose we have the active constraint $A(1,2)\#n$. To search for a partner constraint of the form $B(1)\#m$ and $C(2)\#p$, standard CHR compilation techniques would optimize with indexing (hashtables) which allows constant time lookup for these constraints. The use of such indexing techniques is clearly applicable in a concurrent

goal execution setting as long as concurrent access of the indexing data structures are handled properly. For example, we can possibly have a concurrent active constraint $A(1,3)\#q$ which will compete with $A(1,2)\#n$ for a matching partner $B(1)\#m$. As such, hashtable implementations that facilitate such indexing must be able to be accessed and modified concurrently.

Guard optimizations/simplifications aim at simplifying guard constraints by replacing guard conditions with equivalent but simplified forms. Since guards are purely declarative, they are not influenced by concurrently executing goal threads/CHR rules. Hence, all existing guard optimizations carry over to the concurrent setting.

The join order of a CHR rule determines the order in which partner constraints are searched to execute a rule. The standard CHR optimization known as *optimal join-ordering* and *early guard scheduling* [9] aims at executing goals with the most optimal order of partner constraints lookup and guard testing. By optimal, we refer to maximizing the use of constant time index lookup. Considering the same CHR rule ($r0$) above, given the active constraint $B(x)$, an optimal join-ordering is to lookup for $A(x,y)$, schedule guard $x > y$, then lookup for $C(y)$. Since our concurrent semantics does not restrict the order in which partner constraints are matched, optimal join ordering and early guard scheduling are still applicable.

Another set of optimizations tries to minimize the search for partner constraints by skipping definitely failing searches. Consider the following example:

$$r1@A \iff ...$$
$$r2@A, B \iff ...$$

If the active goal A cannot fire rule (r1) then we cannot fire rule (r2) either. Hence, after failing to fire rule (r1) we can drop goal A. Thus, we optimize away some definitely failing search. This statement is immediately true in the sequential setting where no other thread affects the constraint store. The situation is different in a concurrent setting where some other thread may have added in between the missing

constraint A. Then, even after failing to fire (r1) we could fire rule (r2). However, we can argue that the optimization is still valid for this example. We will not violate the important condition to execute CHR rules exhaustively because the newly added constraint A will eventually be executed by a goal thread which then fires rule (r1). Hence, the only concern is here that the optimization leads to indeterminism in the execution order of CHR rules which is anyway unavoidable in a concurrent setting.

Yet there are existing optimizations which are not applicable in the concurrent setting. For example, *continuation optimizations* [9, 48] are not entirely applicable. Consider the following CHR rule:

$$r4@A(x), A(y) \Longleftrightarrow x == y \mid ...$$

Given an active constraint $A(1)\#n$, fail continuation optimization will infer that if we fail to fire the rule with $A(1)\#n$ matching $A(x)$, there is no point trying to match it with $A(y)$ because it will most certainly fail as well, assuming that the store never changes. In a concurrent goal execution setting, we cannot assume that the store never changes (while trying to execute a CHR). For instance, after failing to trigger the rule by matching $A(1)\#n$ with $A(x)$, suppose that a new active goal $A(1)\#m$ is added to the store concurrently. Now when we match $A(1)\#n$ to $A(y)$ we can find match the partner $A(1)\#m$ with $A(x)$, hence breaking the assumptions of the fail continuation optimization.

Late (Delayed) storage optimization [9] aims at delaying the storage of a goal g, until the latest point of it's execution where g is possibly a partner constraint of another active constraint. Consider the following example:

$$r1@P_1 \Longrightarrow Q$$
$$r2@P_2, T_1 \Longleftrightarrow R$$
$$r3@P_3, R_1 \Longleftrightarrow True$$
$$r4@P_4 \Longrightarrow S$$
$$r5@P_5, S_1 \Longleftrightarrow True$$

Note to distinguish the rule heads, we annotate each rule head with a subscript integer (eg. P_x). With late storage analysis techniques described in [9], we can delay storage of an active constraint P until just before the execution of the body of $r4$. This is because the execution of goal S (obtained from firing of $r4$) can possibly trigger $r5$. While this is safe in the sequential goal execution scheme, it is possible that rule matches are missing in the concurrent goal execution setting. Consider the case where we have parallel active goals $P\#n$ and $T\#m$. Since $P\#n$ is only stored only when it's execution has reached $r4$, the match $r2$ can be missed entirely by both active parallel goals $P\#n$ and $T\#m$. Specifically, this happens if goal $T\#m$ is activated only after $P\#n$ has tried matching with P_2 (of $r2$), but completes goal execution (by trying T_1 of $r2$, and failing to match) before goal $P\#n$ is stored. It is not entirely surprising that this delayed storage scheme is not sound in a concurrent execution, since the delayed storage scheme highlighted in Section 3.4.1 is after all, a conservative yet still flawed attempt at sound optimization to the concurrent execution strategy. Hence, we conclude that we cannot safely implement late storage in the concurrent setting.

Chapter 4

Parallel CHR Implementation

4.1 Chapter Overview

In this Chapter, we provide details of implementing a Parallel CHR system. Section 4.2 provides a quick review on existing CHR goal based implementations which describes sequential goal execution. This is followed by Section 4.3, which introduces a straight-forward implementation. We will explain in detail why such an implementation will fail to scale well, by highlighting several issues which must be addressed by a practical parallel CHR implementation (Section 4.4.1, 4.4.2, 4.4.2 and 4.4.3). We will also discuss our approaches to address these issues, in the respective sections.

Next, we highlight our parallel CHR implementation in Haskell(GHC) (Section 4.5) and provide experiment results in Section 4.6. Results here provides empirical prove and support for our observations and hypothesis.

4.2 Implementation of CHR Rewritings, A Quick Review

In the execution of CHR goals, rule head matching is essentially the most technically complex and computationally intensive procedure that is involved. As such, any

practical implementation of goal-based CHR execution must include a highly efficient rule-head matching routine. Recall the (simplify) derivation step of the concurrent goal-based semantics $\| \mathcal{G}$:

$$\textbf{(Simplify)} \quad \frac{\begin{array}{c}(r \ @ \ H'_P \backslash H'_S \iff t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists \phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H'_P) = DropIds(H_P) \\ \phi(H'_S) = \phi(\{c\} \uplus DropIds(H_S)) \quad \delta = H_P \backslash \{c\#j\} \cup H_S \end{array}}{\begin{array}{rl} & \langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup H_S \cup Sn \rangle \\ \overset{\delta}{\rightarrowtail}_{\mathcal{G}} & \langle \phi(B') \uplus G \mid H_S \cup Sn \rangle \end{array}}$$

This, as well as the (propagate) derivation step, models CHR rewritings in a declarative manner. But operationally, it specifies little about how the actual matching as well as searching for constraints is done. For instance, the premise of the derivation step simply states that given the goal $c\#j$, there must exist some constraints H_S and H_P in the constraint store that matches with the rule heads for this derivation step to be possible, but not exactly how such constraints in the store are located or how they are selected. In this section, we will provide more details on this problem which we will refer to as the *CHR goal-based matching problem*.

In this section, we review the highlights of existing CHR rewriting implementations. We will highlight the CHR goal-based matching problem in practice, followed by a simple example on how CHR goal execution is implemented. Note that existing implementations are single-threaded, in that they assume that at most one CHR goal can be executed at once.

4.2.1 CHR Goal-Based Rule Compilation

We highlight a compilation scheme for CHR rules which encodes CHR rules as a list of search tasks that locates a complete rule-head match, and a set of body constraints. This CHR compilation scheme, which we shall refer to as the *CHR Goal-*

Non-linearized CHR Rule:

$$r1 \ @ \ A(1, x) \setminus B(x, y), C(z) \Longleftrightarrow y > z \mid D(x, y, z)$$

Linearized CHR Rule:

$$r1 \ @ \ A(1, x1) \setminus B(x2, y), C(z) \Longleftrightarrow y > z \wedge x1 = x2 \mid D(x1, y, z)$$

Figure 4.1: Linearizing CHR Rules

based Rule Compilation, is comparable with those used in existing CHR systems [27].

For convenience, we assume that rule heads are linear. That is, each variable occurs at most once in a constraint in the rule head. It is straightforward to linearize CHR rules. For instance, Figure 4.1 shows the CHR rule $r1$ in it's non-linearized and linearized form respectively.

Figure 4.2 shows the formal description of CHR goal-based rule compilations. For convenience, we also include the relevant fragment of the CHR syntax, shown earlier in Figure 3.2. The idea is to compile a CHR rule, into a set of CHR goal-based rule compilations, where each uniquely corresponds to a rule head of the CHR rule. Each rule compilation is essentially a tuple that represents the sequence of *match tasks* to be executed when a goal is matched to it's associated rule head, and a set of constraints which represents the rule body. A match task specifies one of the three type of nodes, matching a goal (*Goal*), looking for a specific partner constraint (*Lookup*) or checking a guard condition (*Guard*). Each *Goal* or *Lookup* task is annotated by a *rewrite type* which distinguishes whether it's goal/partner constraint is to be simplified (S) or propagated (P).

We illustrate this compilation scheme by example (A formal treatment to the compilation scheme is detailed elsewhere [9, 48]). Let's consider our running example, rule $r1$ and it's corresponding CHR goal-based rule compilations:

$$r1 \ @ \ A(1, x1) \setminus B(x2, y), C(z) \Longleftrightarrow y > z \wedge x1 = x2 \mid D(x1, y, z)$$

CHR Syntax:

Constants	v	::=	$1 \mid true \mid ...$
Terms	t	::=	$x \mid f\ \overline{t}$
Predicates	p	::=	$Get \mid Put \mid ...$
Equations	e	::=	$t = t$
CHR Constraints	c	::=	$p(\overline{t})$
Constraints	b	::=	$e \mid c$
CHR Guards	t_g	::=	t
CHR Heads	H	::=	$\overline{c}$
CHR Body	B	::=	$\overline{b}$
CHR Rule	R	::=	$r\ @\ H \setminus H \Longleftrightarrow t_g \mid B$

CHR Goal-Based Rule Compilation:

Rewrite Type	rw	::=	$S \mid P$
Match Task	mt	::=	$Goal\ rw\ c \mid Lookup\ rw\ c \mid Guard\ t_g$
Match Task Sequence	mts	::=	$\overline{mt}$
Rule Compilation	occ	::=	(mts, B)

Figure 4.2: CHR Goal-Based Rule Compilation

$$mts1 = [Goal\ P\ A(1, x1), Lookup\ S\ B(x2, y), Lookup\ S\ C(z), Guard\ (y > z \wedge x1 = x2)]$$

$$mts2 = [Goal\ S\ B(x2, y), Lookup\ P\ A(1, x1), Lookup\ S\ C(z), Guard\ (y > z \wedge x1 = x2)]$$

$$mts3 = [Goal\ S\ C(z), Lookup\ P\ A(1, x1), Lookup\ S\ B(x2, y), Guard\ (y > z \wedge x1 = x2)]$$

$$comp = \{(mts1, \{D(x1, y, z)\}), (mts2, \{D(x1, y, z)\}), (mts3, \{D(x1, y, z)\})\}$$

Rule $r1$ is compiled into three match tasks, namely $mts1$, $mts2$ and $mts3$, which corresponds to rule heads $A(1, x1)$, $B(x2, y)$ and $C(z)$. For instance, $mts1$ represents the match tasks for executing goals that matches with the rule head $A(1, x1)$, which involves looking for a partner constraint $B(x1, y)$ and then $C(z)$ and finally checking the guard constraints. This match task generates match trees like the one seen in Figure 4.3. Note that all well-formed match tasks have a leading *Goal* task.

4.2.2 CHR Goal-Based Lazy Matching

CHR goal-based matching is essentially a *lazy* matching problem. As opposed to *eagerly* matching all rule head instances in a given constraint store, for each CHR goal, we wish only to locate and execute rule head instances on demand. In essence, this matching problem involves some form of search routine which starts from a CHR goal and searches for matching constraints in the CHR store. This goal-based lazy matching routine is essentially encoded by the goal-based rule compilations discussed in the previous section. Let's consider the CHR rule r from Figure 4.1. We model the search space of such matching problems via *match trees*. The particular match tree shown in Figure 4.3 represents the search space of the constraint matching problem for rule $r1$, triggered by the execution of the goal $A(1, x)$ (Match tasks $mts1$ from Section 4.2.1). Given the *Goal* node $A(1, x)$, we seek for constraints in the store matching rule heads $B(x, y)$ and $C(z)$, in this particular order[1]. For instance, the root (*Goal*) node $A(1, 2)\#1$ has two child nodes, namely *Lookup* $B(2, 10)\#2$ and $B(2, 8)\#3$, each representing possible matches of $B(x, y)$ under the substitution $\{2/x, 10/y\}$ and $\{2/x, 8/y\}$ respectively. *Simp* and *Prop* tokens simply indicates if the constraint is to be simplified or propagated. *Guard* nodes represents the checking of CHR rule guards. Successful leaf nodes contain the complete rule head match which corresponds to all rule heads along the path from the root to the leaf node. By successful, we mean that the guard constraint is satisfied. Note that a complete specification of the matching problem for CHR rule r would include two other match tree, each of which specifies the matching problem starting from the each of the other two rule heads ($B(x, y)$ and $C(z)$).

The match tree in Figure 4.3 specifies four possible rule head instances (also referred to as *successful matches*). However, it is not possible to fire all of them

[1]Note we can similarly have it in the order $C(z)$ then $B(x, y)$, but the abstract CHR semantics leaves this choice open. This flexibility allows us to use known CHR optimizations like optimal join-ordering [9] which orders the CHR rule-head matching to maximize the opportunities to exploit indexing.

A CHR simpagation rule and Constraint Store:

$$r \ @ \ A(1,x) \setminus B(x,y), C(z) \Leftrightarrow y > z \mid D(x,y,z)$$
$$\{A(1,2)\#1, B(2,10)\#2, B(2,8)\#3, C(5)\#4, C(6)\#5, C(12)\#6\}$$

Match tree:

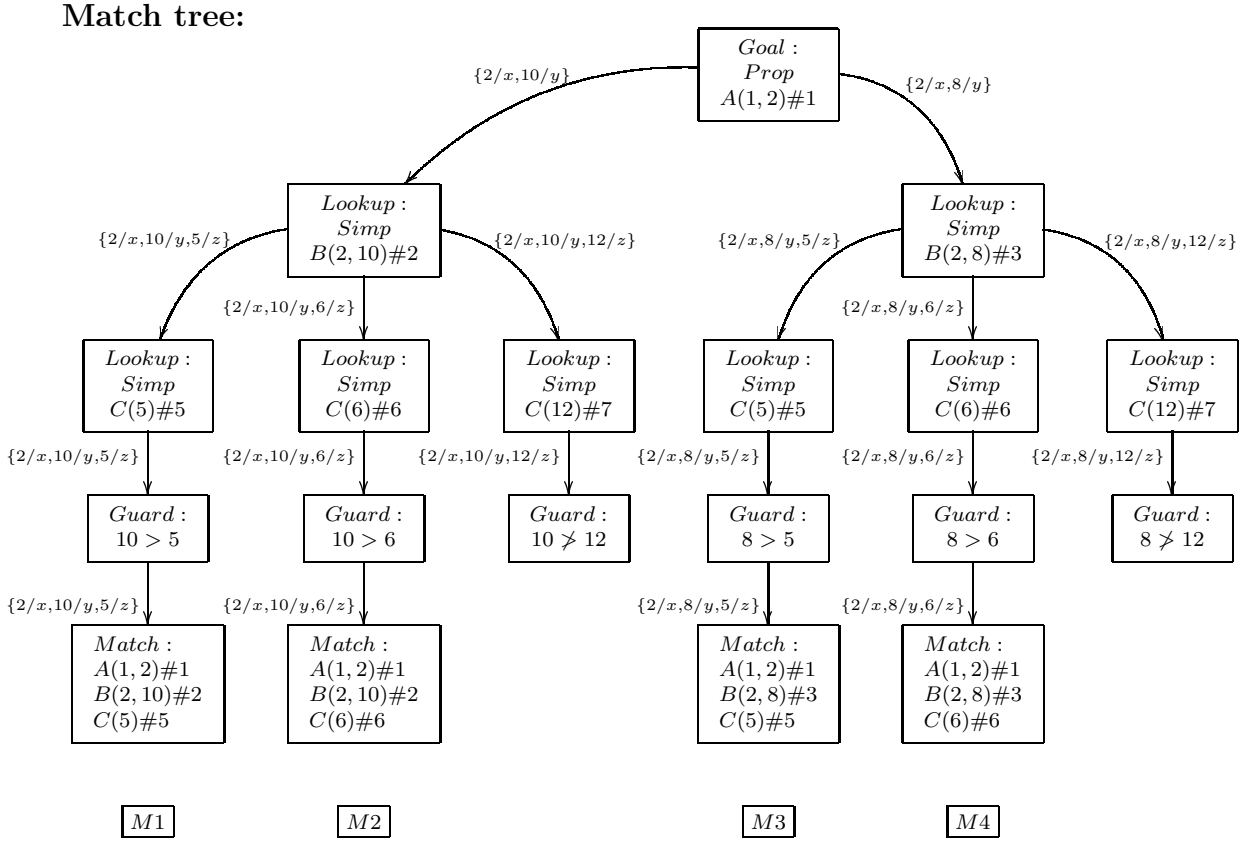

Figure 4.3: Example of CHR rule, derivation and match Tree

---

together. This is because some of the matches are likely to contain overlapping rule heads. Note that for rule $r1$, we propagate $A(1,x)$ but simplify $B(x,y)$ and $C(z)$. If we choose to use match $M1$, match $M2$ becomes invalid because $M1$ and $M2$ share an overlapping constraint $B(2,10)\#2$ which will be simplified. Hence, we can either use match $M1$ or $M2$ but not both. The CHR semantics (eg. $\parallel \mathcal{G}$) of course does not impose any restriction on the choice of which match to use. Similarly, match $M3$ becomes invalid because of the shared simplified constraint $C(5)\#5$. Hence, for each match tree, we can only fire a set of rule head instances which has mutually non-overlapping simplified constraints. For instance, the following illustrates the

```
1  execGoal ⟨G | Sn⟩ A(1,x)#n {
2       ms1 = match Sn B(x,_)
3       for B(x,y)#m in ms1 {
4            ms2 = match Sn C(_)
5            for C(z)#p in ms2 {
6                 if(y > z) {
7                      deleteFromStore Sn [B(x,y)#m,C(z)#p]
8                      addToGoals G [A(1,x)#n,D(x,y,z)]
9                      return true
10                }
11           }
12      }
13      return false
14 }
```

Table 4.1: Example of basic implementation of CHR goal-based rewritings

$\| \mathcal{G}$ derivations that corresponds to the applications of matches $M1$ and $M4$.

$$
\begin{aligned}
& \langle \{A(1,2)\#1\} \mid \{A(1,2)\#1, B(2,10)\#2, B(2,8)\#3, C(5)\#4, C(6)\#5, C(12)\#6\} \rangle \\
\rightarrowtail_r \quad & \langle \{A(1,2)\#1, D(2,10,5)\} \mid \{A(1,2)\#1, B(2,8)\#3, C(6)\#5\} \rangle \\
\rightarrowtail_r \quad & \langle \{A(1,2)\#1, D(2,10,5), D(2,8,6)\} \mid \{A(1,2)\#1, C(12)\#6\} \rangle
\end{aligned}
$$

Similarly, we can apply the alternative set of matches $M2$ and $M3$. In general, we can apply any subsets of matches of a match tree which consist of mutually non-overlapping rule head matches.

Figure 4.1 illustrates pseudo code which implements the execution of goals of the form $A(1,x)\#n$. Description of operations `match`, `deleteFromStore` and `addToGoals` can be found in Section 2.3.2. Line 2 creates an iteration ($ms1$) of constraints in the store Sn that matches the pattern $B(x,\_)$, where the _ symbol represents the 'any' pattern. The 'For' loop of lines $3-13$ tries matching constraints in $ms1$ on the rest of the search procedure. Similar to Line 2, Line 4 creates an iteration of constraints matching $C(\_)$. This is following by the inner 'For' loop of Lines $7-11$ which iterates through constraints in $ms2$. Line 6 checks the rule guard which only executes rewriting (Lines $7-9$) for constraint sets satisfying $y > z$. CHR rewriting is modeled by the following: Line 7 removes the constraints $B(x,y)\#m$ and $C(z)\#p$

```
1  exec_Goal:
2      while ∃ goal
3         select goal G
4         if ∃ r@ P_1,...,P_l \ S_1,...,S_m ⟺ t_g | C_1,...,C_n ∈ 𝒫 and
5            ∃ φ such that
6            St ≡ St_c ⊎ {φ(P_1),...,φ(P_l),φ(S_1),...,φ(S_m)} and
7            ⊨ φ(t_g) and
8            either (G ≡ φ(P_i) for some i ∈ {1,...,l}) or
9                   (G ≡ φ(S_j) for some j ∈ {1,...,m})
10        then let ψ be m.g.u. of all equations in C_1,...,C_n
11             St := St_c ⊎ { φ(P_1),...,φ(P_l), φ∘ψ(C_1),...,φ∘ψ(C_n)}
```

Table 4.2: Goal-based lazy match rewrite algorithm for ground CHR

which matched the simplified heads of the rule. Line 8 adds the rule body $D(x, y, z)$ and the propagated goal constraint $A(1, x)\#n$ [2] into the CHR goals G as new goal(s) to be executed later. Line 9 exits the procedure with success (true). Finally, in Line 13, when no rule head match is found, the goal constraint is dropped and the procedure is exited with failure (false). Note that this procedure essentially traverses the search space specified by match tree in Figure 4.3.

Existing implementations assumes that goal execution routines such as the one found in Figure 4.1 are executed in strictly in isolation[3], hence avoiding the issues and woes of concurrent execution. For the rest of the Chapter, we will detail these issues and highlight our solutions to address them.

Table 4.2 lays out the general structure of a goal-based lazy match rewrite algorithm. We select a goal G which then finds its matching partners. Lines 8 and 9 ensure that the goal must be part of the left-hand side. Our formulation assumes that the CHR rule system is *ground*. That is, equations on right-hand side of rules can immediately be eliminated by applying the m.g.u. This ensures that any derivation starting from a ground constraint store (none of the constraints contains any free variables) can only lead to another ground constraint store. In our experience, the restriction to ground CHR is not onerous because most examples either satisfy

[2]Note that this necessary, as specified by the (Propagate) rule of the $\| \mathcal{G}$ semantics.

[3]In other words, no concurrently running instances, rewriting over the same constraint store

this condition, or it is fairly straightforward to program unification/instantiation on top of CHR (e.g. see our encoding of union-find in the upcoming Section 4.6).

In essence, we wish to extend this CHR execution scheme to execute multiple copies of CHR rewritings (Table 4.2) concurrently, each copy strictly executing a distinct goal but rewriting over the same store St shared among all computation threads.

4.3 A Simple Concurrent Implementation via STM

In Section 2.3.2, we introduced a straight-forward implementation of goal-based concurrent CHR rewritings via traditional locks. Here we shall illustrate another simple implementation via Software Transactional Memory. First, we introduce the basics of software transactional memory in Haskell GHC.

4.3.1 Software Transactional Memory in Haskell GHC

```
newTVar    :: a -> STM (TVar a)     -- Create new transactional variable
readTVar   :: TVar a -> STM a       -- Read a transactional variable
writeTVar  :: TVar a -> a -> STM () -- Write into a transactional variable
atomically :: STM a -> IO a         -- Execute STM operation atomically in IO
forkIO     :: IO () -> IO ThreadId  -- Execute operation in a new thread

incr :: TVar Int -> STM Int          incrN :: TVar Int -> Int -> STM [Int]
incr c = do { v <- readTVar c        incrN c i = mapM (\_ -> incr c) [1..i]
            ; writeTVar c (v+1)
            ; return v }

main = do { c   <- atomically (newTVar 1)
          ; vs1 <- forkIO (incrN c 3)
          ; vs2 <- forkIO (incrN c 3)
          ; ... }
```

Table 4.3: Haskell GHC Software Transaction Memory Library Functions and an example

Software Transactional Memory[4] (STM) is a new and promising concurrency

[4]Please refer to [26] for a detailed introduction to STM in Haskell GHC.

primitive which has gained much interesting in the recent years. STM provides concurrency primitives to program concurrent synchronization in a *composable* manner. This means that operations that modify shared state can be safely composed with other operations that also modify shared states. Sequences of STM operations can be *atomically* composed together into a single *atomic transaction* and the runtime system guarantees that such transactions are executed atomically. By atomically, we mean that their side-effects are visible all at once. Software transactional memory uses *optimistic synchronization*, where shared memory operations are executed as though conflicts never occur and only validated at the end of atomic transactions. If validation fails, the entire operation is re-executed from the start. This approach is known to scale extremely well when conflicting shared memory access occurs on rare occasions. Table 4.3 highlights the basic Haskell STM library functions and an example. `newTVar`, `readTVar` and `writeTVar` creates, reads and writes transactional variables respectively. `atomically` executes a STM operation atomically. Shared memory accesses in this STM are first logged in a transactional log, and the log is only *validated* (checked for global consistency) at the end of the execution. If log is valid, side effects of the operation are made visible *all at once*, otherwise it is *rolled back* (re-executed). `forkIO` simply forks and execute `IO` operations. `incr` increments an integer transaction variable. Thanks to composibility, we implement this simply by composing `readTVar` and `writeTVar` primitive operations. `incrN` demonstrates the power of composibility, showing that we can further compose `incr` dynamically. In `main` we execute two concurrent threads running `incrN`. Since each `incrN` are executed atomically, `vs1` and `vs2` are guaranteed to be contiguous.

4.3.2 Implementing Concurrent CHR Rewritings in STM

Before we proceed to the details of an STM implementation of concurrent CHR rewriting, we introduce the following APIs that provides basic access to the shared store and goals:

$$r1 \ @ \ A(1,x), B(x,y), C(z) \Longleftrightarrow y > z \mid D(x,y,z)$$

```
1  execGoalSTM ⟨G | Sn⟩ A(1,x)#n {
2       atomically {
3            ms1 = matchSTM Sn B(x,_)
4            for B(x,y)#m in ms1 {
5                 ms2 = matchSTM Sn C(_)
6                 for C(z)#p in ms2 {
7                      if(y > z) {
8                           deleteFromStoreSTM Sn [B(x,y)#m,C(z)#p]
9                           addToGoalsSTM G [A(1,x)#n,D(x,y,z)]
10                          return true
11                     }
12                }
13           }
14           return false
15      }
16 }
```

Table 4.4: A straight-forward STM implementation (Example 1)

- `matchSTM` Sn c - Where Sn is the CHR constraint store, and c is a CHR constraint pattern. Returns an iteration of constraints matching c.

- `deleteFromStoreSTM` Sn cs - Where Sn is the CHR constraint store and cs is a list of stored constraints in Sn, deletes all stored constraints in cs from Sn.

- `addToGoalsSTM` G cs - Where G is the goals and cs is a list of CHR constraints, add all CHR constraints in cs into the goals G.

Note that `matchSTM`, `deleteFromStoreSTM` and `addToGoalsSTM` are similar to APIs presented in Section 2.3.2, except they operate on STM shared variables.

Table 4.4 shows a straight-forward approach, using Software Transactional Memory. `execGoalSTM` is the top-level function which implements the execution of goal $A(x)\#n$. The bulk of the code is very much similar to it's single-threaded counterpart (Table 4.1), except for the appearance of the `atomically` construct. This construct allows the atomic execution of it's nested composite STM operation. In this example, we atomically compose the individual STM operations which finds a complete rule match and executes rewriting, starting from a given goal $A(x)$. This

means that effects within the `atomically` block of `execGoalSTM` (Lines $3 - 14$) appears to occur immediately, in an un-interleaving manner.

The atomic composibility of STM operations makes it extremely simple to implement operations like `matchSTM`, `deleteFromStoreSTM` and `addToGoalSTM`. Such implementations can almost be directly lifted from single-threaded implementations[5] without the need of worrying about race-conditions and inconsistent interleaving concurrent executions.

Yet when we consider performance, this implementation will not scale well and is not entirely practical. This is due to a number of reasons, most prominently, false data dependencies, conflicts between multiset rewritings selected by concurrent threads and lack of explicit management of resource limitations. We will discuss these issues in detail in Section 4.4 and highlight the steps we have taken to address these problems.

4.4 Towards Efficient Concurrent Implementations

So far we have highlighted two straight forward implementation of our parallel CHR execution model. Specifically, Section 2.3.2 illustrates a coarse-grained locking approach, while earlier this section we highlighted a coarse-grained software transactional approach. Such simple concurrent implementations of concurrent CHR rewriting do not offer the parallelism and scalability in general. This is because synchronization primitives (locks and transactional memory) here are used in a conservative but overly zealous manner which imposes 'false' data dependencies between concurrently executing goals. These false data dependencies come in the form of false overlapping rule-head matches. For instance, in the coarse-grained locking approach, we lock the entire constraint store when a goal is searching for partner constraints, thus sequentializing all CHR execution. This problem, in the coarse-grained STM

[5]The only major refactoring task is converting the goals and store into shared transactional memory data structures.

$$r2 \ @ \ A(x), B(x), C(x) \Longleftrightarrow D(x)$$

```
1  execGoalSTM ⟨G | Sn⟩ A(x)#n {
2       atomically {
3            ms1 = matchSTM Sn B(x)
4            for B(x)#m in ms1 {
5                 ms2 = matchSTM Sn C(x)
6                 for C(x)#p in ms2 {
7                      deleteFromStoreSTM Sn [B(x)#m,C(x)#p]
8                      addToGoalsSTM G [D(x)]
9                      return true
10                }
11           }
12           return false
13      }
14 }
```

Table 4.5: A straight-forward STM implementation (Example 2)

implementation (details in Section 4.4.1), is much more subtle but yet undeniably present.

Starting from the straight-forward STM implementation of concurrent CHR rewritings highlighted in Section 4.3, we systematically identify various subtle issues (apart for eliminating false data dependencies) which must be specifically addressed in order to 'unlock' parallelism in concurrent execution of CHR rewritings. This section documents these issues and our approach to tackle them, resulting in our optimized implementation of Section 4.5. Results in Section 4.6 supports our discussion here with empirical evidence.

Particularly, we investigate into the problem of **parallel match selection** (Section 4.4.2), **unbounded parallel execution** (Section 4.4.3) and **goal storage policies** (Section 4.4.4) More importantly, we will also highlight the approaches we have taken to address these issues and to mitigate their detrimental effects on parallelism and scalability.

4.4.1 False Overlapping Matches

When implementing concurrent CHR rewritings, it is no doubt that consistency

$$r1 \ @ \ A(x), B(x), C(x) \iff D(x)$$

(A) $\| \mathcal{G}$ semantics concurrent derivation:

$$\begin{array}{lll} & & \langle \{A(1)\#5\} \mid \{C(1)\#1, C(3)\#2, B(1)\#3, B(3)\#4, A(1)\#5, A(3)\#6\} \rangle \\ (D1) & \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(1)\} \mid \{C(3)\#2, B(3)\#4, A(3)\#6\} \rangle \\ & & \quad || \\ & & \langle \{A(3)\#6\} \mid \{C(1)\#1, C(3)\#2, B(1)\#3, B(3)\#4, A(1)\#5, A(3)\#6\} \rangle \\ (D2) & \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(3)\} \mid \{C(1)\#1, B(1)\#3, A(1)\#5\} \rangle \end{array}$$

$$\delta_1 = \{\} \backslash \{\#1, \#3, \#5\} \qquad \delta_2 = \{\} \backslash \{\#2, \#4, \#6\} \qquad \delta = \{\} \backslash \{\#1, \#3, \#5, \#2, \#4, \#6\}$$

$$\begin{array}{ll} & \langle \{A(1)\#5, A(3)\#6\} \mid \{C(1)\#1, C(3)\#2, B(1)\#3, B(3)\#4, A(1)\#5, A(3)\#6\} \rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(1), D(3)\} \mid \{\} \rangle \end{array}$$

(B) Parallel execution based on Table 4.5 implementation:

$$\sigma = \langle \{ \ \underset{\downarrow}{t1} \ A(1)\#5, \ \underset{\downarrow}{t2} \ A(3)\#6 \ \} \mid \{ \ \overset{\text{Search from left-to-right} \longrightarrow}{C(1)\#1 \rightarrow C(3)\#2 \rightarrow B(1)\#3 \rightarrow B(3)\#4 \rightarrow A(1)\#5 \rightarrow A(3)\#6} \ \} \rangle$$

(C) Concurrent execution and accumulated transactional logs:

	execGoalSTM σ $A(1)\#5$	$t1$**'s Transactional Log**	**execGoalSTM** σ $A(3)\#6$	$t2$**'s Transactional Log**
a.	*Start Atomic*	$\{\} \cup$	*Start Atomic*	$\{\} \cup$
b.	`matchSTM st` $B(1)$	$\{\#1, \#2, \#3\} \cup$	`matchSTM st` $B(3)$	$\{\#1, \#2, \#3, \#4\} \cup$
c.	`matchSTM st` $C(1)$	$\{\#1\} \cup$	`matchSTM st` $C(3)$	$\{\#1, \#2\} \cup$
d.	`deleteFromStoreSTM`	$\{\#1, \#3, \#5\}$	`deleteFromStoreSTM`	$\{\#2, \#4, \#6\}$
e.	*End Atomic*	$\{\#1, \#2, \#3, \#5\}$	*End Atomic*	$\{\#1, \#2, \#3, \#4, \#6\}$

Figure 4.4: Example of false overlaps in concurrent matching

of such an implementation is one of our top priority. As such, we must exploit the concurrency synchronization protocols offered by the programming language we have chosen, to model the 'allowed' concurrent behaviours of the $\| \mathcal{G}$ semantics. Particularly, we want to use concurrency primitives like traditional locks or software transactional memory to model the side-effects (δ) of the $\| \mathcal{G}$ semantics derivations ($\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma'$). Let's consider a simple example, as shown in Table 4.5. We consider the following $\| \mathcal{G}$ derivation step for rule $r2$:

$$\begin{array}{ll} & \langle \{A(1)\#5\} \mid \{C(1)\#1, C(3)\#2, B(1)\#3, B(3)\#4, A(1)\#5, A(3)\#6\} \rangle \\ \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(1)\} \mid C(3)\#2, B(3)\#4, A(3)\#6 \rangle \\ & \text{where } \delta_1 = \{\} \backslash \{C(1)\#1, B(1)\#3, A(1)\#5\} \end{array}$$

Recall that we specify each $\| \mathcal{G}$ derivation step with some side-effect δ. Side-effects

$H_P \backslash H_S$ represents the propagated constraints (H_P) and the simplified constraints (H_S) which are involved in the derivation step. In the above instance, no constraints are propagated, while $\{C(1)\#1, B(1)\#3, A(1)\#5\}$ are simplified in the derivation step which fires the rule r. Hence this derivation step can only concurrently execute with other derivation steps that do not have side-effects which include constraints $C(1)\#1$, $B(1)\#3$ or $A(1)\#5$. This restriction guarantees that we only execute non-overlapping rule applications concurrently, and thus guaranteeing that concurrent derivations are consistent.

Figure 4.4 illustrates a detailed example showing how side-effects δ in $\| \mathcal{G}$ derivations are captured in STM transactional logs as shared variable access while executing `execGoalSTM` on the goals $A(1)\#5$ and $A(3)\#6$, concurrently. Part (A) of the figure illustrates the concurrent $\| \mathcal{G}$ derivation of two non-overlapping rule applications of $r2$. It represents our desired concurrent behaviour. Note that derivation $D1$ causes side-effects $\delta_1 = \{\}\backslash\{\#1, \#3, \#5\}$ while $D2$ causes $\delta_2 = \{\}\backslash\{\#2, \#4, \#6\}$. In short, we will we will refer to stored constraints via their unique identifiers (ie. $A(1)\#5$ by $\#5$).

Part (B) states some of the assumption we make of the concurrent execution in practice, namely we assume that we have two concurrent threads of computation $t1$ and $t2$, executing goals $A(1)\#5$ and $A(3)\#6$ respectively, and both threads are attempting to complete a rule head match to CHR rule $r2$, via the goal execution procedure of Table 4.5. For simplicity, we assume that the search procedure (`matchSTM`) searches for matches in left-to-right order, hence the shared constraint store is just a shared linked-list[6]. Finally, each constraint in the store is assumed to be stored in a unique shared transactional memory location. Hence we shall refer to stored constraints (identifiers) also as transactional memory locations.

Part (C) of the figure illustrates the transactional logs accumulated by the con-

[6]It is possible that in certain special cases, with the use of hash indexing and more complex (shared) store data structures [9], such linear searching can be avoided. But this is a reasonable assumption for the general case.

current executions of the two goals. For simplicity, we assume that threads $t1$ and $t2$ executes in discrete locked steps. In step a, both threads begin executing the atomic transactional memory procedure. In step b, both threads executes their respective `matchSTM` procedures to locate a matching $B(x)$ constraint. In a left-to-right searching order, thread $t1$ accesses $\{C(1)\#1, C(3)\#2\}$ before it finds $B(1)\#3$ from the iterator returned by `matchSTM` ($ms1$ in the pseudo codes of Table 4.5), while $t2$ accesses $\{C(1)\#1, C(3)\#2, B(1)\#3\}$ before it finds $B(3)\#4$. In step c, both threads executes the next `matchSTM` procedures to locate the final rule head $C(x)$, during which $t1$ and $t2$ obtains $C(1)\#1$ and $C(3)\#2$ after searching through $\{\#1\}$ and $\{\#1, \#3\}$ respectively. Since both $t1$ and $t2$ have each found a complete match, in step d `deleteFromStore` routines are executed to remove the rule heads $\{A(1)\#5, B(1)\#3, C(1)\#1\}$ and $\{A(3)\#6, B(3)\#4, C(3)\#2\}$. In step e, both threads completes their respective atomic transactions, hence logs are validated. Unfortunately, we will find that the logs are overlapping, will be ruled as potentially inconsistent by the STM protocol, even though the constraints actually used (instantiation of guard variables and deleted from store) for each computation are non-overlapping. This will result to the rolling back of either one of the executed sequence, thus the goals will not execute in parallel.

It is clear that there is some form of disparity between the side-effects (δ) of the $\| \mathcal{G}$ semantics and the transactional logs accumulated by the simple implementation, `execGoalSTM`. For instance, in Figure 4.4, derivation $D1$ execution goal $A(1)\#5$ involves side-effects (simplified constraints) $\{\#1, \#3, \#5\}$ while executing `execGoalSTM` σ $A(1)\#5$ (by $t1$) validates the transactional log $\{\#1, \#2, \#3, \#5\}$. Similarly, derivation $D2$ involves $\{\#2, \#4, \#6\}$ while `execGoalSTM` σ $A(3)\#6$ validates the logs $\{\#1, \#2, \#3, \#4, \#6\}$. Note that in both cases, the transactional logs validated are supersets of the actual side-effects of the respective $\| \mathcal{G}$ derivations. While this means that `execGoalSTM` consistently models $\| \mathcal{G}$ derivations as STM log validation sufficiently includes side-effects δ of $\| \mathcal{G}$ derivations, the logs unfortunately

$$r2 \ @ \ A(x), B(x), C(x) \Longleftrightarrow D(x)$$

```
1  execGoalSTM_ARV ⟨G | Sn⟩ A(x)#n {
2       ms1 = match Sn B(x)
3       for B(x)#m in ms1 {
4            ms2 = match Sn C(x)
5            for C(x)#p in ms2 {
6                 atomically {
7                      if(containsSTM Sn [B(x)#m,C(x)#p]){
8                           deleteFromStoreSTM Sn [B(x)#m,C(x)#p]
9                           addToGoalsSTM G [D(x)]
10                          return true
11                     }
12                }
13           }
14      }
15      return false
16 }
```

Table 4.6: STM implementation with atomic rule-head verification

include more constraints than necessary to guarantee consistently, thus introducing *false CHR rewriting overlaps*. In other words, the STM synchronization protocol for this particular naive implementation, behaves like a conservative but inaccurate approximation of the $\| \mathcal{G}$ derivation side-effects. Such false overlaps are extremely detrimental to scalability. This is because concurrent derivations which can be consistently executed in parallel are falsely ruled as executing conflicting updates to the shared memory and hence are executed sequentially as a result. In Section 4.6, we provide empirical results that show that a coarse grained STM implementation will not scale well in general.

Atomic Rule-Head Verification

Upon observation of the simple approach illustrated in Table 4.4.1, we can see that STM protocols behave like an inaccurate approximation of $\| \mathcal{G}$ side-effects because each 'atomic' run of `execGoalSTM` accumulates a transactional log that contains more than just the constraints (memory location) of the constraints involved in the rule-head instance. These are the constraints read during the traversal of the shared store, and incidentally not all such constraints are semantically in-

volved in the executed CHR rewriting. For instance, in the example of Figure 4.4, derivation $D1$ involves the rewriting of $C(1)\#1, B(1)\#3, A(1)\#5$, but executing of our naive STM implementation accumulates the transactional log containing $C(1)\#1, C(3)\#2, B(1)\#3, A(1)\#5$.

There are several approaches to minimize false CHR rewriting overlaps. For instance, explicitly removing excess constraint(non-rule head) memory locations from the STM transactional logs, via an extended Haskell STM library [52]. Our approach is to introduce what we refer to as *atomic rule-head verification*. The idea is to "push" the store traversal part of the CHR rule execution out of the STM atomic transaction. This effectively excludes constraint read during the matching constraint search routines of CHR goal execution from the accumulated STM transactional log, but includes the rule heads of the CHR rule instance that is executed. Table 4.6 illustrates this improved goal execution procedure, `execgoalSTM_ARV`. Lines $2-5$ shows the new matching constraint search routine. We assume that the function `match` is similar to `matchSTM` except it is called externally from STM transactions. Lines $6-12$ contain the atomic rule-head verification procedure. Essentially, it verifies that all selected rule-heads (matching constraints $B(x)\#m$,$C(x)\#p$) are in the shared store (via `containsSTM` of Line 7) and remove them while adding the body constraints ($D(x)$) into the goals (Line $8-9$).

While pseudo code of `execgoalSTM_ARV` illustrates this improvement as a simple change from `execgoalSTM` of Table 4.5, the introduction of atomic rule-head verification requires more subtle implementation effort in the underlying shared concurrent data structures to guarantee the consistency of parallel CHR rewritings. For instance, we must now ensure that non-atomic `match` are consistently executed in the presence of possible removal of constraints from concurrent executions of `deleteFromStoreSTM` routines. In Section 4.5, we shall illustrate these subtleties via a concrete example in Haskell (GHC).

$$r2 \ @ \ A(x), B(y) \Longleftrightarrow C(x, y)$$

Example of non-overlapping match selection:

$$\begin{array}{lll} & & \langle \{A(1)\#1\} \mid \{A(1)\#1, A(2)\#2, B(3)\#3, B(4)\#4\} \rangle \\ (D1) & \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(1,3)\} \mid \{A(2)\#2, B(4)\#4\} \rangle \\ & & \quad || \\ & & \langle \{A(2)\#2\} \mid \{A(1)\#1, A(2)\#2, B(3)\#3, B(4)\#4\} \rangle \\ (D2) & \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(2,4)\} \mid \{A(1)\#1, B(3)\#3\} \rangle \end{array}$$

$$\delta_1 = \{\}\backslash\{\#1, \#3\} \qquad \delta_2 = \{\}\backslash\{\#2, \#4\} \qquad \delta = \{\}\backslash\{\#1, \#3, \#2, \#4\}$$

$$\begin{array}{ll} & \langle \{A(1)\#1, A(2)\#2\} \mid \{A(1)\#1, A(2)\#2, B(3)\#3, B(4)\#4\} \rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(1,3), C(2,4)\} \mid \{\} \rangle \end{array}$$

Example of overlapping match selection:

$$\begin{array}{lll} & & \langle \{A(1)\#1\} \mid \{A(1)\#1, A(2)\#2, B(3)\#3, B(4)\#4\} \rangle \\ (D1') & \overset{\delta'_1}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(1,3)\} \mid \{A(2)\#2, B(4)\#4\} \rangle \\ & & \quad \not|| \\ & & \langle \{A(2)\#2\} \mid \{A(1)\#1, A(2)\#2, B(3)\#3, B(4)\#4\} \rangle \\ (D2') & \overset{\delta'_2}{\rightarrowtail}_{||\mathcal{G}} & \langle \{C(2,3)\} \mid \{A(1)\#1, B(4)\#4\} \rangle \end{array}$$

$$\delta'_1 = \{\}\backslash\{\#1, \#3\} \qquad \delta'_2 = \{\}\backslash\{\#2, \#3\}$$

Figure 4.5: Non-overlapping and overlapping match selections

4.4.2 Parallel Match Selection

Apart from false overlaps introduced by conservative but inaccurate approximations of the $\| \ \mathcal{G}$ derivation side-effects, another problem that will be faced by an implementation of $\| \ \mathcal{G}$ is the *parallel match selection problem*. To focus on this problem, let's assume that concurrency synchronization of our implementation accurately models $\| \ \mathcal{G}$ side-effects (ie. does not introduce false overlaps). Even in this ideal case, it is possible that concurrently executing CHR goals select over-lapping matches, even though entirely non-overlapping matches exists in a particular CHR state.

Figure 4.5 illustrates this problem with a simple example. We consider concurrent derivations of the CHR rule $r2$ starting from the CHR store $\{A(1)\#1, A(2)\#2$ $, B(3)\#3, B(4)\#4\}$, modeling the parallel execution of goals $A(1)\#1$ and $A(2)\#2$. Derivations $D1$ and $D2$ illustrates an ideal match selection, specifically $D1$ models execution of goal $A(1)\#1$ with match $A(1)\#1, B(3)\#3$ selected, while $D2$ the execu-

tion of goal $A(2)\#2$ with match $A(2)\#2, B(4)\#4$ selected. Since side-effects δ_1 and δ_2 (of $D1$ and $D2$ respectively) are non-overlapping, they can consistently execute in parallel. Derivations $D1'$ and $D2'$ on the other hand, illustrates an unfortunate case where overlapping matches are selected instead, specifically $D1'$ models execution of $A(1)\#1$ with match $A(1)\#1, B(3)\#3$ selected, while $D2'$ the execution of goal $A(2)\#2$ with match $A(2)\#2, B(3)\#3$ selected. Since side-effects δ_1' and δ_2' (of $D1'$ and $D2'$ respectively) are overlapping on $B(3)\#3$, derivations $D1'$ and $D2'$ cannot be consistently executed in parallel. In a simple STM implementation (similar to those illustrated in Table 4.4 and Table 4.5), this would imply that when derivations $D1'$ and $D2'$ execute in parallel, either one will ultimately be doomed to failure and inevitably 'rolled-back' by the STM protocol to prevent inconsistent parallel rewriting. Derivations $D1$ and $D2$, on the other hand, can execute in parallel.

In essence, we require some form of *match selection policy* that enables parallel goals to select non-overlapping constraints. An ideal match selection policy would always pick the parallel execution of $D1$ and $D2$ over $D1'$ and $D2'$. Yet a practical implementation of such a policy should not impose unrealistic overheads or impose additional synchronization routines over our asynchronously running goal execution routines.

Bag Constraint Store/Iterations

While it is clear that on the extreme end, we can impose a *synchronized* parallel computation of a maximal set of non-overlapping rule-head matches[7] in a given shared CHR constraint store, such an approach is not suitable when we are considering an asynchronous model of parallelism. Particularly because we expect parallel CHR goals to execute asynchronously with minimal synchronization between each goal executions.

The approach we have chosen focuses on providing a best-effort match selection policy that can be implemented in a manner which imposes no additional synchro-

[7]See related works in Section 6.3 for synchronized parallel multiset matching algorithms in parallel production rule systems

nization over-heads between parallel goal execution routines. It is based on the observation that overlapping matches are most likely to be selected over non-overlapping ones if all goals search through the shared constraint store in the same arbitrary order. For instance, in the example illustrated in Figure 4.5, derivations $D1'$ and $D2'$ will most certainly be chosen for execution by goals $A(1)\#1$ and $A(2)\#2$ if all goals try constraints in the given left-to-right texture order (specifically, all goals will try $B(3)\#3$ before $B(4)\#4$). This does not exploit the fact that the shared CHR store is a multi-set (as opposed to an ordered list) and a search can be conducted consistently by iterating through constraints in the store in *any* order. Our approach is to use bag data structures [8] to represent the *iterations of the constraint store*, which ensures that goals executed in parallel search through the shared constraint store (multiset) each in a unique order (i.e. observing constraints in a unique order), hence increasing the chances that goals pick unique constraints and locate non-overlapping matches. In the simplest form, we implement the shared bag constraint store with a circular linked-list that has multiple entry heads[9]. Each goal execution thread is issue a unique entry head, in which it begins all it's search routines. Therefore, each goal thread will observe stored constraints in it's unique sequence.

Figure 4.6 illustrates an example of parallel goal execution with an underlying bag store implementation. For simplicity, we assume that the shared constraint store is a single circular linked-list. In this example, $A(1)\#1$ is executed by goal thread T_1, while $A(2)\#2$ by goal thread T_2. As indicated, goal threads T_1 and T_2 are assigned unique entry points to the store and will search for partner constraints in a clock-wise direction. Given this particular instance, T_1 will try the partner constraint $B(3)\#3$, while T_2 will try $B(4)\#4$, and hence trigger non-overlapping rule head instances $A(1)\#1, B(3)\#3$ and $A(2)\#2, B(4)\#4$ without needless synchronization.

It is of course, possible on the contrary, to find an arbitrary contrived example

[8] Unordered collection of elements(constraints in our context) that may have duplicates

[9] Note that more practical implementations will include top-level indexing data structures. Hence iterators of the shared store created during parallel goal executions are bag data structures (multi-headed circular linked-lists) instead.

$$r2 \ @ \ A(x), B(y) \Longleftrightarrow C(x, y)$$

Parallel goal derivation, from Figure 4.5:

$$\begin{array}{rl} & \begin{array}{cccc} T_1 & & T_2 & \\ \downarrow & & \downarrow & \end{array} \\ & \langle\{ \quad A(1)\#1, \quad A(2)\#2 \quad \} \mid \{A(1)\#1, A(2)\#2, B(3)\#3, B(4)\#4\}\rangle \\ \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} & \langle\{ \quad C(1,3), \quad C(2,4) \quad \} \mid \{\}\rangle \end{array}$$

Bag representation of constraint store:

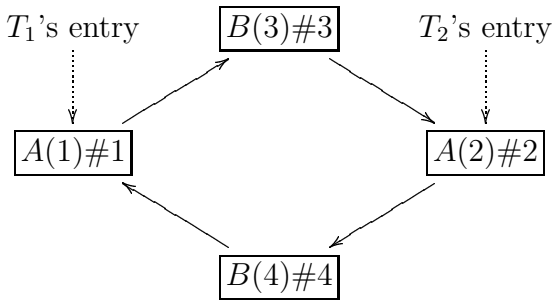

Figure 4.6: Example of a 'bag' store and match selection

---

which shows how a bag constraint store would not enable the selection of non-overlapping matches. Our experimental results in Section 4.6 shows that in general, the bag constraint store help in performance and scalability of asynchronous parallel goal execution.

4.4.3 Unbounded Parallel Execution

We define *unbounded parallel goal execution* as the parallel execution strategy which does not bound the number of goals executing in parallel by some finite integer n, but is only bounded by the number of CHR goals in a particular CHR state. While it may seems most nature to implement a parallel CHR system that aggressively executes in parallel *all* CHR goals in a given CHR state, it is not entirely practical to do so in general, especially when underlying system resources (processor cores, memory, etc..) are limited.

Unbounded execution of goals will almost certainly introduce unnecessary conflicts, as parallel goals are more likely to compete for similar/overlapping rule head

$$r3 \ @ \ A(x), B(y), C(z) \iff D(x, y, z)$$

$$\begin{array}{lll} & & \langle \{A(1)\#1\} \uplus G \mid \{A(1)\#1, B(2)\#2, C(3)\#3\} \uplus S \rangle \\ (D1) & \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(1,2,3)\} \uplus G \mid S \rangle \\ & & \not| \\ & & \langle \{B(2)\#2\} \uplus G \mid \{A(1)\#1, B(2)\#2, C(3)\#3\} \uplus S \rangle \\ (D2) & \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(1,2,3)\} \uplus G \mid S \rangle \\ & & \not| \\ & & \langle \{C(3)\#3\} \uplus G \mid \{A(1)\#1, B(2)\#2, C(3)\#3\} \uplus S \rangle \\ (D2) & \overset{\delta_3}{\rightarrowtail}_{||\mathcal{G}} & \langle \{D(1,2,3)\} \uplus G \mid S \rangle \end{array}$$

$$\delta_1 = \delta_2 = \delta_3 = \{\} \backslash \{\#1, \#2, \#3\}$$

Figure 4.7: Example of contention for rule-head instances

instances. Figure 4.7 illustrates a simple example of this upper bound. Derivations $D1$, $D2$ and $D3$ represent three parallel CHR goal executions ($A(1)\#1$, $B(2)\#2$ and $C(3)\#3$ respectively) attempting to match and rewrite the same rule-head instance. Note that in spite of any arbitrary interleaving of execution of the three goals, it is almost certain that at most one of the goal execution can successfully execute, while the other two cannot and must be dropped via the (Drop) derivation rule in a *subsequent* derivation[10].

In essence, executing too many goals in parallel causes technical problems in practice. Not only will a system's limited memory be contended for by more goal executions, the larger number of concurrent goal executions will likely invoke more synchronization routines between conflicting goal executions. Considering an STM implementation, contentions between two goal threads will result to the 'rolling back' of either one (due to in-validate STM transactions caused by inconsistent side-effects) and thus contributing to wasted computations. In a realistic scenario, we have limited resources (CPU cores, memory, etc...) and by execution goals aggressively, we will possibly swarm the limited system resources with higher number of such wasted computations and this will have detrimental effects on scalability

[10]we can possibly execute other goals in G in parallel as well, but Figure 4.7 illustrates that we can at most executed $m + 1$ goals in parallel (of the $m + 3$ goals available in the example), where m is the number of goals in G

and performance. In such instances, executing any one of the goals of Figure 4.7 ($A(1)\#1$, $B(2)\#2$ or $C(3)\#3$) to trigger the rule instance $A(1)\#1, B(2)\#2, C(3)\#3$ may actually perform better than executing all three goals in parallel.

In general, if all goal constraints are immediately activated and executed in parallel, an n-headed rule-instance will be executed by one successful computation and up to $n-1$ unsuccessful computations. As such, it almost seems more efficient to execute only as much goal as we have physical CPU cores to process them, otherwise we risk introducing more conflicts.

Scheduling and Thread Pools

The solution to this problem is a well known concept in parallel programming, *scheduling*. In our context, CHR goals are the jobs to be scheduled to be executed by a thread pool, consisting of a *bounded* number n of execution threads (where n is normally the number of CPU cores available).

While most modern programming languages already have thread pooling libraries or built-in schedulers[11], which provides an abstraction for programmers to write multi-threaded programs without being aware of the number of physical system cores, implementing parallel goal executions on such thread pooling libraries naively may not be optimal. One particular interesting observation which we identified is that active goals threads executing CHR goals should *not be preempted*(interrupted) by concurrent goal threads competing for system resources. To illustrate this, we consider the example in Figure 4.7, suppose we have three threads executing the goals $A(1)\#1$, $B(2)\#2$ and $C(3)\#3$ respectively but only one physical CPU core to run the three threads of computation. If preemptive scheduling is used, we can possibly waste time context switching between redundant alternatives to the same rule-head match $A(1)\#1, B(2)\#2, C(3)\#3$. On the other hand, a non-preemptive scheduler would focus on a single goal execution and find the rule-head match more efficiently. Specifically, we will fully execute $A(1)\#1$ and move on to the 'administrative' clean

[11] Eg. Haskell (GHC) and Scala both provides light-weight threads and thread pooling

up of $B(2)\#2$ and $C(3)\#3$ ((Drop) rule derivations, because $B(2)\#2$ and $D(3)\#3$ are deleted by execution of $A(1)\#1$).

For our implementation in Haskell GHC, we *throttle* the number CHR goal threads available for goal execution. In other words, we ensure that the number of active CHR goals never exceed the number of CPU cores (processors) available. This means that given a CHR state with m goals waiting to be activated, we execute at most n (where $n < m$) of them at a time by n CHR goal execution threads, each mapped to a physical CPU core. In this case, a goal thread executing a specific active goal will less likely (or even never) be preempted by another goal thread, since we have as many active goal threads as we have physical CPU cores. In Section 4.6, we will examine this further and provide experiment results to support this.

Goal Ordering

Thread pooling and scheduling of CHR goals introduces a tightly associated issue which must occasionally be addressed. Our experiment results in Section 4.6 also show that the performance of parallel goal-execution in certain domains, can be highly sensitive to the *order* in which goals are scheduled, as well as sensitive to the *type* of goals scheduled in parallel. For example, recall the merge sort example introduced earlier in Figure 2.3.

$$\mathit{merge1} \ @ \ \mathit{Leq}(x,a) \backslash \mathit{Leq}(x,b) \iff a < b \mid \mathit{Leq}(a,b)$$
$$\mathit{merge2} \ @ \ \mathit{Merge}(n,a), \mathit{Merge}(n,b) \iff a < b \mid \mathit{Leq}(a,b), \mathit{Merge}(n+1,a)$$

In the merge sort problem, the order in which Leq and $Merge$ constraints are scheduled greatly affects performance. We wish to always exhaustively schedule and execute *merge1* rule instances, before introducing new Leq constraints via *merge2*. This is because it is more efficient to execute the binary tests $a < b$ immediately, rather than introducing more Leq constraints first[12]. Hence, it seems optimal to stack Leq goal constraints and queue $Merge$ goal constraints, so Leq goals are

[12]Introducing more Leq constraints will make the lookup procedures for Leq constraints more computationally intensive, hence more inefficient.

exhaustively executed, while $Merge$ goals are executed only when there are no Leq goals.

Another orthogonal observation is that scheduling Leq constraints in parallel would less likely introduce conflicts when compared with scheduling $Merge$ constraints in parallel, since in *merge1* we only simplify one constraint ($Leq(x, b)$) as oppose to two ($Merge(n, a)$ and $Merge(n, b)$) in *merge2*. Stacking Leq and queuing $Merge$ goal constraints will also make it more likely that Leq goals executed in parallel, since parallel goal threads will choose Leq constraints over $Merge$ constraints.

In general, the optimal selection and execution ordering is a domain specific problem[13]. In our implementation, we provide program annotations that allows the user to specify whether each type of goals are to be stacked or queued for execution. In other words, the user is allowed to specify a boolean value tagged to each rule-body constraint, indicating whether the new goal constraint is to be stacked or queued in the CHR goals. The default value is 'stacked' (with respect to the refined CHR operational semantics). We like to note that inferring optimal goal ordering annotations is an extremely interesting topic of discussion, but we will leave that for future works.

4.4.4 Goal Storage Policies

In the sequential goal execution context, delaying goal storage to the latest point of execution is most certainly an optimization (specifically, late storage optimization from Section 3.5.4). Delaying goal storage reduces the size of a constraint store to the minimal, optimizing in both space and time efficiency.

In general, the storage of constraints is a burden to performance, as excessive storage increases memory usage and time complexity of constraint lookup procedure. Yet in the context of concurrent goal execution, eager storage (storing early) of goals

[13] As the saying goes, "Different strokes for different folks." Different scheduling strategies work better or worst for different domains (CHR programs). Some problems even exhibit little performance change when different scheduling strategies are used.

may not entirely be bad for performance. There are cases where eager storage has an advantage over late storage. We explain this point via the following contrived example, which consist of 2 rules:

$$r2 \ @ \ A(x), B(x) \Longleftrightarrow ...$$
$$r3 \ @ \ C(0) \Longleftrightarrow D(1), A(1), ..., D(n), A(n)$$

Suppose the initial store is $\{B(1), ..., B(n), C(0)\}$, we assume there are $n+1$ threads, n threads executing goals $B(1)$,...,$B(n)$ and one thread executing $C(0)$. To improve performance, we should be allowed to specify that A constraints are stored eagerly after firing of rule *r3*, thus allowing threads 1,...,n to fire rule *r2* in parallel. There is no need to store D constraints immediately, hence we can retain the original storage scheme for D constraints. Similar to goal orderings, the choice between whether to eagerly or lazily store goals is domain specific and in worst case, should be left to be decided by the programmer (via annotations of body constraints).

4.5 Parallel CHR System in Haskell GHC

In this section, we dive down into the details of implementing a concrete parallel CHR system, known as ParallelCHR , that implements the $\| \mathcal{G}$ semantics in a scalable manner. Our choice of programming language is Haskell, a lazy functional programming language. In particular, we use the Glasgow Haskell Compiler [23] because of its good support for shared memory, multi-core architectures. Haskell also provides high-level abstraction facilities (polymorphic types higher-order functions etc) and its clean separation between pure and impure computations invaluable in the development of our system. In principle, our system can of course be re-implemented in other main-stream languages such as C and Java. Our implementation in Haskell GHC is available for download at $http://code.google.com/p/parallel-chr/$.

4.5.1 Implementation Overview

We take a high-level look at finding matches in parallel and atomic rule execution. In our implementation, a thread pool consisting of several light-weight Haskell GHC threads are used to execute CHR goals in a shared collection of goals. Goal execution threads like these executes CHR rewriting asynchronously by searching the shared store for matching partner constraints (to complete rule head instances), deleting the simplified constraints of the rule head instance and finally adding body constraints into the collection of goals. The challenge we face in this parallel execution problem is that the partners found by asynchronous threads running in parallel be overlapping (share similar simplified heads). As defined in the $\| \mathcal{G}$ semantics (Definition 4), parallel goal execution must rewrite over non-overlapping rule-heads. Here, we briefly introduce two approaches which uses different concurrency primitives to implement this non-overlapping parallel rule-head matching routine.

Fine-grained Lock-based parallel matching: This approach is a standard refinement of the coarse-grained locking approach (highlighted in Table 2.1). Rather than guarding the shared store with a single global lock, we restrict the access of each constraint in the shared store with a unique dedicated lock. The parallel matching task at hand now includes incrementally acquire locks of partner constraints. However, we must be careful to avoid deadlocks. For example, suppose that thread 1 and 2 seek partners A and B to fire any of the rules $A, B, C \iff rhs_1$ and $A, B, D \iff rhs_2$. We assume that C is thread 1's goal constraint and D is the goal constraint of thread 2. Suppose that thread 1 first encounters A and locks this constraint. By chance, thread 2 finds B and imposes his lock on B. But then none of the two threads can proceed because thread 1 waits for thread 2 to release the lock imposed on B and thread 2 waits for the release of the locked constraint A.

The scenario illustrated above is a classic (deadlock) problem when programming with locks. The recently popular becoming concept of Software Transactional Memory (STM) is meant to avoid such issues. Instead of using locks directly, the

programmer declares that certain program regions are executed atomically. The idea is that atomic program regions are executed optimistically. That is, any read/write operations performed by the program are recorded locally and will only be made visible at the end of the program. Before making the changes visible, the underlying STM protocol will check for read/write conflicts with other atomically executed program regions. If there are conflicts, the STM protocol will then (usually randomly) commit one of the atomic regions and rollback the other conflicting regions. Committing means that the programs updates become globally visible. Rollback means that we restart the program. The upshot is that the optimistic form of program execution by STM avoids the typical form of deadlocks caused by locks. In our setting, we can protect critical regions via STM as follows.

STM-based parallel matching means that we perform the search for partner constraints and their removal from the store atomically. For the above example, where both threads attempt to remove constraints A and B as well as their specific goal constraints we find that only one of the threads will commit whereas the other has to rollback, i.e. restart the search for partners.

The downside of STM is that unnecessary rollbacks can happen due to the conservative conflict resolution strategy. Here is an example to explain this point. Suppose that thread 1 seeks partner A and thread 2 seeks partner B. There is clearly no conflict. However, during the search for A, thread 1 reads B as well. This can happen in case we perform a linear search and no constraint indexing is possible or the hash-table has many conflicts. Suppose that thread 2 commits first and removes B from the store. The problem is that thread 1 is forced to rollback because there is a read/write conflict. The read constraint B is not present anymore. STM does not know that this constraint is irrelevant for thread 1 and therefore conservatively forces thread 1 to rollback.

We have experimented with a pure STM-based implementation of atomic search for partners and rule execution. The implementation is simple but unnecessary roll-

backs happen frequently which in our experience results in some severe performance penalties, for reasons stated in Section 4.4.1. We provide concrete evidence in the upcoming Section 4.6. In our current implementation, we use a **hybrid STM-based** scheme which uses both Software Transactional Memory and traditional shared memory access techniques. The search for matching partner constraints is performed "outside" STM (to avoid unnecessary rollbacks), this means that accessing constraint memory locations at this stage does not invoke STM concurrency synchronization protocols. Once a set of constraints forming a complete match is found, we perform an atomic STM procedure which atomically checks that all the constraints are still available, and logically deletes the simplified constraints[14]. This essentially implements *atomic rule-head verification* (as described in Section 4.4.1) which guarantees the atomic deletion (logical) of rule-head instances. Logically deleted constraints will eventually be physically delinked from the constraint store, either immediately after the atomic rule-head verification step or by an amortized delete procedure, both of which can be implemented with relative ease with traditional concurrency primitives (e.g. compare-and-swap, locks, etc...).

4.5.2 Data Representation and Sub-routines

We briefly discuss our data representation of the constraint handling rules language in Haskell, illustrated by Figure 4.8. **Abstract Data Type** shows the Haskell data type representation of CHR language elements, like constraints, substitution, store etc. **Rule Occurrence Data Types** represent the goal-based compilation of CHR rules, detailed in Section 4.2.1. Essentially, a CHR Program is a list of CHR rule compilations. A rule compilation `Comp` is a tuple, which consist of a list of match tasks (`MatchTask`) and a list of constraints (`Cons`). Note that we will represent sets with lists. Note that for presentation, we shall focus entirely on CHR matching and

[14]By logically delete, we mean that the constraint is not physically removed from the data structure, but simply marked as deleted

Abstract Data Types

Integer Value:	`Int`	Boolean Value:	`Bool`
List of a's:	`[a]`	Substitution:	`Subst`
CHR Constraint:	`Cons`	Rule Guard:	`Guard`
CHR Store:	`Store`	CHR Goals:	`Goals`

Rule Occurrence Data Types

Head Type:	`data Head = Simp | Prop`
Match Task:	`data MatchTask = LpHead Head Cons | SchdGrd Guard`
Rule Compilation:	`type Comp = ([MatchTask],[Cons])`
CHR Program:	`type Prog = [Comp]`

Figure 4.8: Interfaces of CHR data types

---

rewriting of CHR constraints. Hence we will not include builtin constraints in our CHR language here. We defer a treatment of builtin constraints till Section 4.8.1.

The following provide brief descriptions of the basic CHR Solver Sub-routines. These sub-routines represents basic interfaces to the underlying shared store and goal data structures, as well as substitution framework.

- `isAlive :: Cons -> Bool`

  Given CHR constraint `c`, returns true if and only if `c` is still stored.

- `match :: Subst -> Cons -> Cons -> IO (Maybe Subst)`

  Given a substitution and two CHR constraints `c` and `c'`, returns resultant substitution of matching `c` with `c'`, if they match. Otherwise return nothing.

- `consApply :: Subst -> [Cons] -> [Cons]`

  Given a substitution and a list of CHR constraints, apply the substitution on each constraint of the list and return the results.

- `grdApply :: Subst -> Guard -> Bool`

  Given a substitution and a guard condition, apply the substitution on the guard and return true iff guard condition is satisfiable.

- `emptySub :: Subst`

  Returns the empty substitution.

- `addToStore :: Store -> Cons -> IO Cons`

  Given a CHR store `st` and a CHR constraint `c`, add `c` into `st`. Returns the stored constraint c containing additional book-keeping information (store back-pointers, etc..).

- `getCandidates :: Store -> Cons -> IO [Cons]`

  Given a CHR Store `st` and a CHR constraint `c`, return a list of constraints from the `st` that matches `c`.

- `getGoal :: Goals -> IO (Maybe Cons)`

  Given CHR goals, returns the next goal if one exists, otherwise returns nothing.

- `addGoals :: Goals -> [Cons] -> IO ()`

  Given CHR goals `gs` and a list of CHR constraints `cs`, add `cs` into `gs`.

- `notRepeat :: [(Head,Cons)] -> Cons -> Bool`

  Given a list of matching heads, and a constraint **c** returns true if **c** is not already found in the list of heads.

- `isStored :: Store -> Cons -> STM Bool`

  Given a CHR store `st` and a constraint `c`, returns true if and only if `c` is stored in `st`.

- `logicalDeleteFromStore :: Store -> Cons -> STM ()`

  Given a CHR store and a constraint in the store, logically mark the specified constraint as deleted from the store.

- `delinkFromStore :: Store -> Cons -> IO ()`

  Given a CHR store and a constraint in the store, physically delink the constraint from the store.

```
1  goalBasedThread :: Goals -> Store -> Prog -> IO ()
2  goalBasedThread gs st prog =
3     rewriteLoop
4     where
5       rewriteLoop = do
6         { mb <- getGoal gs
7         ; case mb of
8            Just g -> do { a <- addToStore st g
9                         ; executeGoal a prog
10                        ; rewriteLoop }
11           Nothing -> return () }
12      executeGoal a (occ:occs) = do
13        { matchGoal gs st a occ
14        ; if isAlive a then executeGoal a occs
15                       else return () }
16      executeGoal _ [] = return ()
```

Table 4.7: Top-level CHR Goal Execution Routine

- `atomically :: STM a -> IO a`

  Given a `STM` operation, execute it atomically in the `IO` monad.

Next, Section 4.5.3 and 4.5.4 will introduce the main high-level goal execution routine which uses these sub-routines.

4.5.3 Implementing Parallel CHR Goal Execution

We introduce our parallel CHR implementation from a top-down approach, starting from the function `goalBasedThread`, as shown in Table 4.7. The parallel CHR solver comprises of multiple copies of this function, executed asynchronously in parallel by multiple threads of computation. Each execution essentially implements the execution of a CHR goal. For now, we focus on execution of CHR goals only, and defer a treatment for builtin constraints till Section 4.8.1.

This function is given the references to the shared goals `gs` and store `st`, and the CHR program `prog`. Goals are exhaustively executed via the `rewritingLoop` procedure, which terminates only when the goals are empty (line 11). As specified by the (Activate) rule of the $\| \mathcal{G}$ semantics (Figure 3.3 of Section 3.3), goals are added to the store only when they are executed (line 8). Procedure `executeGoal`

```
1  matchGoal :: Goals -> Store -> Cons -> Occ -> IO ()
2  matchGoal goals store g (mtasks,body) = do
3   { let (LpHead hd c):rest = mtasks
4   ; mb <- match emptySub c g
5   ; case mb of
6      Just subst -> do { execMatch [(hd,g)] subst rest ; return () }
7      Nothing    -> return () }
8   where
9    execMatch hds subst ((SchdGrd guard):mts) =
10    if grdApply subst guard then execMatch hds subst mts
11                            else return False
12   execMatch hds subst ((LpHead hd c):mts) =
13    let execMatchCandidates (nc:ncs) =
14         if (notRepeat hds nc) && (isAlive nc)
15         then do { mb <- match subst c nc
16                 ; case mb of
17                    Just subst' -> do
18                        { succ <- execMatch ((h,nc):hds) subst' mts
19                        ; if not succ then execMatchCandidates ncs
20                                      else return False }
21                    Nothing     -> execMatchCandidates ncs }
22         else execMatchCandidates ncs
23        execMatchCandidates [ ] = return False
24    in do { cans <- getCandidates store c
25          ; execMatchCandidate cans }
26   execMatch hds subst [ ] = do
27    { succ <- atomically (verifyRuleHeads store hds)
28    ; if succ then do { let simpHds = filter (\(h,_) -> h == Simp) hds
29                      ; mapM (\(_,g) -> delinkFromStore store g) simpHds)
30                      ; addGoals goals (consApply subst body)
31                      ; let (h,_) = first hds
32                      ; return (h == Simp) }
33              else return False }
```

Table 4.8: Implementation of Goal Matching

attempts to match the active goal `g` with each of the occurrence compilations (via the `matchGoal` operation at line 13, whose definition is deferred till later). Procedure `executeGoal` stops when the goal is no longer alive (line 15) or all occurrence have been tried (line 16), both of which are cases which lead to the application of the (Drop) rule of the $\| \mathcal{G}$ semantics.

Procedure `matchGoal` in Table 4.8 implements the main parallel matching algorithm which searches for matching constraints of the active goal constraint. This search is specified by the match tasks of CHR goal-based rule compilations, de-

scribed earlier in Section 4.2.1. We assume that the first match task is the lookup of the active goal pattern (line 3)[15]. In line 4 the active goal is then matched with the head pattern (from the lookup task)[16]. If the active goal successfully matches the head pattern (line 6) we call `execMatch`. If matching fails, we abort the procedure (line 7).

Procedure `execMatch` essentially implements the search traversal through CHR match trees (Section 4.2.2). It checks for the remaining match tasks, which can be either looking up a partner constraint, or checking a guard condition. It controls the branching of the search by returning $True$ if the search is to terminate at the current branch, or $False$ if the search is to proceed. Lines $9-11$ implements the scheduling of a guard constraint `grd`. We proceed on with the rest of the match tasks if the guard evaluates to true. Lines $12-25$ on the other hand, implements the lookup of a partner constraint, as specified by the matchtask `LpHead hd c`. We first collect all possible candidates `cans` matching `c` from the store (line 24)[17]. Then, we call `exec_match_candidate` (line 25) which tries to find a complete match for the entire rule head by iterating over the set of candidates. Note that we only iterate through as many candidate as required (lines $19-20$) for exhaustiveness of the goal execution (details in Section 4.5.7).

In case we find a complete match (line 26), we fire the rule. Note that this step can happen in parallel with multiple goal executions, hence to guarantee consistency, we must atomically verify and commit this match via `verifyRuleHeads` (line 27). This procedure checks that all heads are still alive and logically marks all the simplified heads as deleted. All these operation are done in one atomic transactional step. That is, if any of the intermediate steps fails the entire transaction fails with no visible side effect (We defer details on how atomic rule-head verification is implemented with

[15]This is because CHR rules have at least one head, hence this constraint lookup task must exist.

[16]Note that with known pre-compilation analysis, this matching of active goal and head pattern can be avoided. Such optimizations are covered in [9] and will not be discussed here.

[17]Note that this procedure can be implemented 'lazily', or with iterators representing a collection of all candidate matching constraints and hence only retrieved on demand.

```
1  verifyRuleHeads :: Store -> [(Head,Cons)] -> STM Bool
2  verifyRuleHeads store hds = do
3   { bs <- mapM (\(_,g) -> isStored store g) hds
4   ; if and bs
5     then do { let simpHds = filter (\(h,_) -> h == Simp) hds
6             ; mapM (\(_,g) -> logicalDeleteFromStore store g) simpHds
7             ; return True }
8     else return False }
```

Table 4.9: Implementation of Atomic Rule-Head Verification

STM in Haskell till Section 4.5.4). Follow a successful run of `verifyRuleHeads` (line $28 - 32$), we will physically delink all the simplified constraints (line 29) and add the body constraints of the rule instance into the goals (line 30). If the executed goal is a simplified head, we end the search by return $True$ (since the goal currently executed will be deleted from the store), otherwise we proceed to the next candidate (line 32). In a failed run of `verifyRuleHeads` (line 33), we return $False$ to indicate that the goal execution should try another partner constraint.

Note that the delinking of simplified constraints (line 29) are done in a seemingly unsafe ("unatomic") sequence of `IO` operations. Yet it is safe to do so, thanks to the fact that the constraints to be delinked at line 29 are the same constraints marked as deleted by `verifyRuleHeads` in line 27. Hence we have the guarantee that no two concurrent goal executions will attempt to delink the same constraints.

4.5.4 Implementing Atomic Rule-Head Verification

We detail the atomic rule-head verification (highlighted in Section 4.4.1) implementation via Software Transactional Memory in Haskell (GHC).

Table 4.9 illustrates the implementation of atomic rule-head verification with STM in Haskell GHC. The STM operation `verifyRuleHeads` works as follows: Given the shared store and a set of matching constraints (presumably the complete rule-head instance), we check that all the constraints are still in the store and not deleted by any other parallel goal execution routines (line 3). If so (line $5 - 7$) we delete (from the store) all the constraints that are matched as simplified heads

and return true. Otherwise (line 8) we return false. Note that since this STM operation is guaranteed to execute atomically, a successful run (resulting to the return of `True`) indicates that we were able to independently observe the presence of all constraints involved in the store and delete the simplified constraint. Most importantly, constraints involved in this STM validation process corresponds directly to the constraints that form the rule head instance, thus we will not introduce any false overlaps (Section 4.4.1).

4.5.5 Logical Deletes and Physical Delink

Recall that we have chosen to use logical deletes during atomic rule-head verification, while the physical delinking of constraints from the store data structure is only executed in subsequent non-atomic steps (Sections 4.5.3 and 4.5.4). This approach is beneficial in two ways. Firstly, we can implement atomic rule-head verification with **smaller STM transactions**. This is because multiset logical deletes can be straight-forwardly implemented as the toggling of boolean flags stored in STM transactional variables. Hence, logically deleting n constraints is essentially just writing into n boolean variables. Logical deletes are much cheaper operations, compared to implementing physical removal of constraints (from the store) which involves delinking of nodes from a list data structure (implemented on STM). As such, our atomic rule-head verification can be implemented with smaller STM transactions which most certainly incur less conflicts from STM roll backs.

Besides reducing the number of STM roll backs, we can now implement other list operations (list traversal, delinking of list nodes) via **lighter weight concurrency primitives**. In our works on comparing Haskell concurrency primitives [54], we have demonstrated with empirical evidence that a concurrent list data structure implemented via traditional compare-and-swap operations is much more efficient than one implemented via STM. Yet, STM provides the most elegant solution to

atomic multiset operations[18]. Our multiset logical delete via STM and physical delink via compare-and-swap implementation essentially adopts the best of both worlds (or rather, concurrency primitives) and provides the alternative with least concurrency synchronization overheads.

4.5.6 Back Jumping in Atomic Rule-Head Verification

Consider the following example:

$$A(x), B(x,y), C(y,z), D(z) \Longleftrightarrow E(x,y,z)$$

Suppose while executing the goal $A(1)\#n$ we have found the rule-head instance $[A(1)\#n, B(1,2)\#m, C(2,3)\#p, D(3)\#q]$ in that specific sequence and now attempts to rule atomic rule-head verification on the four constraints. Further suppose that the verification procedure failed because the constraint $B(1,2)\#m)$ has already been deleted by some other executing thread. Our implementation of atomic rule-head verification in Table 4.9 will return $False$ suggesting that one of the constraint has been deleted, but without more information other than the boolean flag, our goal execution procedure in Table 4.8 has to explore other alternate branches of the match tree, iterating through possible alternative candidates of $D(z)$, $C(y,z)$, before reaching $B(x,y)$ lookup node, where the verification had failed.

To avoid such pointless traversals of the match tree, we can implement a well known optimization technique for backtracking search algorithms, known as **back-jumping**. By keeping track of exactly which constraint has failed the atomic rule-head verification, we can precisely backtrack our search to the "highest" point of the match-tree which is possibly still valid and resume the search from that point.

Table 4.10 illustrates the atomic rule-head verification function `verifyRuleHeads` with backjumping indicator. `verifyRuleHeadsBackJump` is similar to that of in Ta-

[18] as oppose to traditional locks or compare-and-swap synchronization variables, which are prone to errors and other overheads incurred by complex synchronization acrobatics.

```
1  verifyRuleHeadsBackJump :: Store -> [(Head,Cons)] -> STM Int
2  verifyRuleHeadsBackJump store hds = do
3   { bs <- mapM (\(_,g) -> isStored store g) hds
4   ; if and bs
5     then do { let simpHds = filter (\(h,_) -> h == Simp) hds
6             ; mapM (\(_,g) -> logicalDeleteFromStore store g) simpHds
7             ; return 0 }
8     else let Just j_index = elemIndex False bs
9          in return (j_index + 1) }
```

Table 4.10: Atomic Rule-Head Verification with Backjumping Indicator

ble 4.9, but instead returns an integer. If verification is successful, 0 is returned (line 7). Otherwise, it returns the 1-index of the left-most constraint which has failed the verification[19]

Table 4.11 illustrates the modified `matchGoalBackJump` operation that utilizes the backjumping indexes provided by `verifyRuleHeadsBackJump`. Note that the most important change is in lines $19 - 20$, where new candidates are tried only if the jump index i returned by the previous branch (line 18) is equal to 1. Otherwise, we simply return the index decrement by one (line 20). Successful run of the `verifyRuleHeadsBackJump` indicated by index $i == 0$ (line 28) results to the same delinking of simplified constraint (line 29) and adding of body constraints into the goals (line 30). If goal is a simplified constraint (line 32), we return the number of the rule heads (effective "backjumping out" of the goal execution), otherwise we procedure on with the search through the match tree.

4.5.7 Implementation and $\| \mathcal{G}$ Semantics

In this section, we informally discuss the correspondence of our parallel CHR system in Haskell GHC, and the $\| \mathcal{G}$ semantics. Our implementation implements the CHR $\| \mathcal{G}$ semantics in that given the shared goals gs (initially containing goals cs), shared store st (initially empty) and CHR program compilation $prog$ (of CHR program $\mathcal{P}$), when multiple concurrent execution of the `goalBasedThread gs st prog` goal

[19]Since new rule-heads are appended to the end of our rule-heads hds, left-most constraint which has failed represents the "highest" point of the match tree which has failed the verification.

```
1  matchGoalBackJump :: Goals -> Store -> Cons -> Occ -> IO ()
2  matchGoalBackJump goals store g (mtasks,body) = do
3   { let (LpHead hd c):rest = mtasks
4   ; mb <- match emptySub c g
5   ; case mb of
6      Just subst -> do { execMatch [(hd,g)] subst rest ; return () }
7      Nothing    -> return () }
8   where
9    execMatch hds subst ((SchdGrd guard):mts) =
10    if grdApply subst guard then execMatch hds subst mts
11                            else return 1
12   execMatch hds subst ((LpHead hd c):mts) =
13    let execMatchCandidates (nc:ncs) =
14         if (notRepeat hds nc) && (isAlive nc)
15         then do { mb <- match subst c nc
16                 ; case mb of
17                    Just subst' -> do
18                        { i <- execMatch ((h,nc):hds) subst' mts
19                        ; if i == 1 then execMatchCandidates ncs
20                                    else return (i-1) }
21                    Nothing     -> execMatchCandidates ncs }
22         else execMatchCandidates ncs
23        execMatchCandidates [ ] = return 1
24    in do { cans <- getCandidates store c
25          ; execMatchCandidate cans }
26   execMatch hds subst [ ] = do
27    { i <- atomically (verifyRuleHeadsBackJump store hds)
28    ; if i==0 then do { let simpHds = filter (\(h,_) -> h == Simp) hds
29                      ; mapM (\(_,g) -> delinkFromStore store g) simpHds)
30                      ; addGoals goals (consApply subst body)
31                      ; let (h,_) = first hds
32                      ; return (if h == Simp then (length hds) + 1 else 1) }
33             else return i }
```

Table 4.11: Goal Matching with Back-Jumping

execution routine terminates[20], shared goals gs will be empty and shared store st will contain constraints cs' such that for the CHR program $\mathcal{P}$, $\langle cs \mid \{\}\rangle \rightarrowtail_{\|\mathcal{G}} \langle\{\} \mid cs'\rangle$.

To summarize, each $\| \mathcal{G}$ transition rule (Figure 3.3) corresponds to our implementation in the following manner:

- (Solve): Addressed in Section 4.8.1. An equation constraint e is not physically stored in the constraint st as suggested in the $\| \mathcal{G}$ semantics, but imposes it's side-effects on the builtin theory when e is executed by the routine `solve`

[20]This also includes the termination of all goal reactivation threads (Section 4.8.1)

(Table 4.12). The set of constraints $WakeUp(e, st)$ awaken (reactivated) by the (Solve) transition is replicated in our implementation by the execution of `reactivateWhenGround` by reactivation threads spawned specifically for this purpose (add the affected constraint back into the goals).

- (Activate): This transition immediately corresponds to the execution of `addToStore`, line 8 of `goalBasedThread` in Table 4.7.

- (Simplify): This transition corresponds to an execution of `executeGoal a prog` (line 9, Table 4.7) which results to the deletion of active goal constraint `a` (line 29, Table 4.8). This means that the active constraint `a` is one of the simplified constraint of the successful rule-head match. The CHR rewriting (removal of simplified constraints and adding of body to the goals) is eventually completed by the execution of lines $28-32$ of Table 4.8. Atomic rule head verification (line 27, Table 4.8) guarantees that concurrent goal execution selects mutually exclusive simplified constraints, hence rewrites non-overlapping rule instances. This is exactly specified by the merging of side-effects δ in the $\| \mathcal{G}$ semantics (Figure 3.4).

- (Propagate): Very much similar to the above (Simplify), except execution of `executeGoal a prog` results in a successful rule-head match where `a` matches a propagated constraint.

- (Drop): This transition models the removal of a goal constraint after it has exhaustively searched the store for matching partner constraints and has not been simplified. It corresponds to the end of an execution of `executeGoal a prog` which does not end with the simplification of the active constraint `a`. Essentially, all complete execution of `executeGoal a prog` corresponds to a either a sequence (empty allowed) of (Propagate) transitions followed by a (Drop) transition, or a sequence (empty allowed) or (Propagate) transitions followed by a (Simplify) transition.

Our parallel CHR implementation faithfully implements the concurrent CHR goal-based semantics, in that every execution on a termination CHR program corresponds to a valid $\| \mathcal{G}$ concurrent derivation. But because of practical limitations of hardware, it is likely that our implementation cannot replicate all possible executions modeled by the semantics. For instance consider the following derivation:

$$r \ @ \ A(x), B(x) \Longleftrightarrow C(x)$$

$$\begin{aligned} & \langle A(1)\#n_1, .., A(j)\#n_j \mid A(1)\#n_1, B(1)\#m_1, .., A(j)\#n_j, B(j)\#m_j \rangle \\ \rightarrowtail_{||\mathcal{G}} \quad & \langle C(1), .., C(j) \mid \{\} \rangle \end{aligned}$$

This derivation specifies the concurrent derivation of j pairs of A and B constraints. Suppose that j is a significantly huge number, say $10,000$, we will require ten thousand physical CPU cores to execute all ten thousand goals in parallel and faithfully implement what is specified by this derivation.

In essence, what this means is that our parallel CHR implementation is a sound and faithfully implementation of the concurrent CHR goal-based semantics, but it is nonetheless inevitably incomplete as it is likely not able to achieve all theoretical concurrent derivations in practice, due to the bounds of hardware limitations.

4.6 Experimental Results

In this section, we present the experiments we have conducted on our parallel CHR system and the empirical results we have collected. We focus on eight distinct CHR programs, which represents a diverse spread of CHR rules with varying characteristics. Note that these programs were chosen because they represent the most common examples of CHR used in general-purpose programming throughout the literature (Eg. Gcd, mergesort, prime, fibonacci and unionfind, found in [9, 20, 19]), while also representing excellent examples of parallel programming in practice (Eg. blockworld, dining philosophers). The following highlights each of these CHR programs,

and the experiment parameters we have used:

- **Merge Sort**:

$$merge1 \ @ \ Leq(x,a) \setminus Leq(x,b) \iff a < b \mid Leq(a,b)$$
$$merge2 \ @ \ Merge(n,a), Merge(n,b) \iff a < b \mid Leq(a,b), Merge(n+1,a)$$

  CHR implementation of Merge sort. CHR goal threads essentially compare different pairs of integers in parallel. We optimize with a specific goal ordering scheme (stack Leq goals and queue $Merge$ goals) which minimizes the number of comparisons between Leq constraints and the number of conflicts between goal executions (see Section 4.4.3 for details). For our experiment, we run merge sort on a collection of 1024 integers.

- **Gcd**:

$$gcd1 \ @ \ Gcd(n) \setminus Gcd(m) \iff m \geq n \&\& n > 0 \mid Gcd(m-n)$$
$$gcd2 \ @ \ Gcd(0) \iff True$$

  CHR implementation of greatest common divisor Euclid's algorithm. We optimize by queuing Gcd goals. For our experiments, we find the greatest common divisor of 1000 integers. Finding the Gcds of distinct pairs of integers can be executed in parallel.

- **Parallelized Union Find**:

$$union \ @ \ Union(a,b), Fresh(x) \iff Fresh(x+2), Find(a,x), Find(b,x+1), Link(x,x+1)$$
$$findNode \ @ \ Edge(a,b) \setminus Find(a,x) \iff Find(b,x)$$
$$findRoot \ @ \ Root(a) \setminus Find(a,x) \iff Found(a,x)$$
$$found \ @ \ Edge(a,b) \setminus Found(a,x) \iff Found(b,x)$$
$$linkeq \ @ \ Link(x,y), Found(a,x), Found(a,y) \iff True$$
$$link \ @ \ Link(x,y), Found(a,x), Found(b,y), Root(a), Root(b) \iff Edge(b,a), Root(a)$$

Adapted from [20], Union find is basically a data structure which maintains the union relationship among disjoint sets. Sets are represented by trees ($Edge(x, y)$) in which root notes ($Root(x)$) are the representatives of the sets. The union operation between two sets of a and b ($Union(a, b)$) is executed by finding the representatives x and y of the sets a and b ($Find(a, x)$ and $Find(b, y)$), and then linking them together ($Link(x, y)$). The *union* rule initiates the union operation. The constraint $Fresh(x)$ introduces "fresh variables" since our current prototype only supports ground CHR rules/stores. Rule *findNode* traverses edges until we reach the root in rule *foundRoot*. Rule *found* re-executes a find if the tree structure has changed. This is necessary since union find operations can be executed in parallel. Rule *linkeq* removes redundant link operations and rule *link* performs the actual linking of two distinct trees. In experiments, we test an instance of parallelized union find, where 300 union operations are issued in parallel to unite 301 disjoint sets (binary trees) of depth 5.

- **Blockworld**:

$$grab \ @ \ Grab(r, x), Empty(r), Clear(x), On(x, y) \iff Hold(r, x), Clear(y)$$
$$puton \ @ \ PutOn(r, y), Hold(r, x), Clear(y) \iff Empty(r), Clear(x), On(x, y)$$

A simple simulation of robot arms re-arranging stacks of blocks. $Grab(r, x)$ specifies that robot r grabs block x, only if r is empty and block x is clear on top and on y ($On(x, y)$). The result is that robot r will be holding block x ($Hold(r, x)$) and block x is no longer on block y, thus y is clear. $PutOn(r, y)$ specifies that robot r places a block on block y, if r is holding some block x and y is clear. In our experiments, we simulate 8 agents each moving a unique stack of 1000 blocks. Robots can be executed in parallel as long as their actions do not interfere.

- **Dining Philosophers**:

$$grabforks \ @ \ Think(c, 0, x, y), Fork(x), Fork(y) \iff Eat(c, 20, x, y)$$
$$thinking \ @ \ Think(c, n, x, y) \iff n > 0 \mid Think(c, n - 1, x, y)$$
$$putforks1 \ @ \ Eat(0, 0, x, y) \iff Fork(x), Fork(y)$$
$$putforks2 \ @ \ Eat(c, 0, x, y) \iff Fork(x), Fork(y), Think(c - 1, 20, x, y)$$
$$eating \ @ \ Eat(c, n, x, y) \iff Eat(c, n - 1, x, y)$$

The classic dining philosopher problem, simulating a group of philosophers thinking and eating on a round table, and sharing a fork with each of her neighbors. In our implementation, Forks are represented by the constraints $Fork(x)$ where x is a unique fork identifier. A thinking and eating philosopher is represented by the constraints $Think(c, n, x, y)$ and $Eat(c, n, x, y)$ where x and y are the fork identifiers, c represents the number of eat/think cycles left and n a counter that simulates the delay of thinking/eating process. Rules *thinking* and *easting* delay thinking and eating. If there any think/eat cycles left, we return both forks and issue a new thinking process. See rule *putforks2*. Otherwise, we only return both forks. See rule *putforks1*. In our experiments, we simulated the dining philosopher problem with 150 philosophers, each eating and thinking for 50 cycles with a delay of 20 steps.

- **Prime**:

$$prime1 \ @ \ Candidate(1) \iff True$$
$$prime2 \ @ \ Candidate(x) \iff x > 1 \mid Prime(x), Candidate(x - 1)$$
$$prime3 \ @ \ Prime(y) \setminus Prime(x) \iff x \ mod \ y == 0 \mid True$$

A CHR program that computes the first n prime numbers. In our experiments, we find the first 1500 prime numbers. Parallelism comes in the form of parallel comparison of distinct pairs of candidate numbers.

- **Fibonacci**:

$$\begin{aligned}
\mathit{fibo1} \ @ \ \mathit{FindFibo}(0) &\iff \mathit{Fibo}(1) \\
\mathit{fibo2} \ @ \ \mathit{FindFibo}(1) &\iff \mathit{Fibo}(1) \\
\mathit{fibo3} \ @ \ \mathit{FindFibo}(x) &\iff \mathit{FindFibo}(x-1), \mathit{FindFibo}(x-2) \\
\mathit{fibo4} \ @ \ \mathit{Fibo}(x), \mathit{Fibo}(y) &\iff \mathit{Fibo}(x+y)
\end{aligned}$$

A CHR program that computes the value of the n^{th} Fibonacci number. We find the 25^{th} Fibonacci number. Parallelism is present when evaluating different parts of the Fibonacci tree.

- **Turing Machine:**

$$\begin{aligned}
\mathit{delta_left} \quad & @ \ \mathit{Delta}(qs, ts, qs', ts', \mathit{LEFT}) \setminus \mathit{CurrState}(i, qs), \mathit{TapePos}(i, ts) \\
& \iff \mathit{CurrState}(i-1, qs'), \mathit{TapePos}(i, ts') \\
\mathit{delta_right} \quad & @ \ \mathit{Delta}(qs, ts, qs', ts', \mathit{RIGHT}) \setminus \mathit{CurrState}(i, qs), \mathit{TapePos}(i, ts) \\
& \iff \mathit{CurrState}(i+1, qs'), \mathit{TapePos}(i, ts')
\end{aligned}$$

A simple formulation of the classic Turing machine in CHR (Originally found in [51]). In our implementation, *delta_left* and *delta_right* define the state transitions of the Turing machine. The constraint $\mathit{Delta}(qs, ts, qs', ts', dir)$ specifies the state transition mapping $(qs, ts) \mapsto (qs', ts', dir)$ where qs, qs' are state symbol and ts, ts' are tape symbols and dir is the direction which the tape is moved. $\mathit{CurrState}(i, qs)$ states that the current state of the machine is qs at tape position i. $\mathit{TapePos}(i, ts)$ states that tape position i has the symbol ts. In our experiments, we tested a Turing machine instance which determines if a tape (string of 0's and 1's) of length 200 is of the form $\{0^n 1^n \mid n > 1\}$. The Turing machine simulator is inherently single thread (rules cannot fire in parallel), as it involves state transitions of a single state machine. This serves to investigate the effects of parallel rewriting applied to a single threaded problem.

Our experiments are conducted to find empirical evidence that our parallel CHR implementation is scalable and practical. By scalable and practical, we mean that performance scales (improves) with the number of active processor threads added to execute the parallel CHR multiset rewritings invoked by the parallel CHR solver. To do so in a systematic way, we investigate into the effects of optimizations targeted at improving parallel goal execution and show that each are crucial in 'unlocking' true parallelism in the implementation of the concurrent goal-based CHR semantics. We observe the performance of the eight CHR programs with each optimization versus a default alternative. To summarized, we focus on the following concurrency specific optimizations:

- **Throttled/Bounded Thread Pools** (Section 4.4.3): Aimed to reduce number of conflicting parallel executions and to prevent limited system resources from being swarmed by redundant concurrent goal executions. The alternative to this is to rely entirely on GHC's thread pooling system, hence we spawn a lightweight GHC thread to execute each new active goal.

- **Atomic Rule-Head Verification** (Section 4.4.1): Aimed to reduce the number of false-overlaps during parallel goal executions. The alternative to this is a simple STM implementation that does not use atomic rule-head verification (This implementation executes each goal as a single STM operation, see Section 4.3).

- **Bag Constraint Store and Store Iterators** (Section 4.4.2): Aimed to reduce number of overlapping matches selected by parallel goals, by making each goal thread observe stored constraints in a unique order. The alternative to this are basic list constraint stores and list store iterators.

- **Domain Specific Goal Ordering** (Section 4.4.3): Aimed to optimally schedule goals for execution. As detailed in Section 4.4.3, goal-ordering in our implementation come in the form of user-annotations tagged to each CHR rule-body

constraint, indicating whether the newly added goal is to be *stacked* or *queued* in the CHR goals. Goal ordering is specifically customized for each CHR program and only crucial for some examples, specifically *Gcd* and *Mergesort*. The alternative to this is the basic stack ordering of goals, which is the traditional ordering used by most CHR implementations.

On top of the concurrency optimizations mentioned here, our implementation also includes existing CHR optimizations which are still applicable to the concurrent context. Specifically, our implementation includes constraint indexing (hashing), optimal join ordering and early guard scheduling.

Our experiments are conducted on an Intel Core i7-920 processor[21] with 6 GB of memory running 64-bit Windows XP and Haskell GHC 6.10.1. For each experiment, we measure the relative performance of executing with 1, 2, 4, 8 and unbounded goal thread(s) against a *base* non-concurrent implementation in Haskell. Final results shown are the medians of 20 runs of the same experiment. This non-concurrent implementation serves as a benchmark for our concurrent implementation and is free from the overheads of concurrent execution (e.g. invoking STM runtime synchronization, atomic rule-head verification, etc..).

4.6.1 Results with Optimal Configuration

Figure 4.9 illustrates the experimental results conducted with our parallel CHR system in optimal configuration. In other words, **atomic rule-head verification**, **Bag constraint store and iterators**, **throttled goal thread pool** and **domain specific goal ordering**[22] concurrent optimizations are enabled. Measurements are based on the percentage time against execution time of the basic non-concurrent implementation (we will refer to this execution time as the base execution time).

[21] An Intel Core i7-920 processor is essentially a quad core processor, but is equipped with Hyper-threading technology that effectively allows it to run 8 concurrent threads of computation.

[22] Where applicable. Namely, Merge sort and Gcd

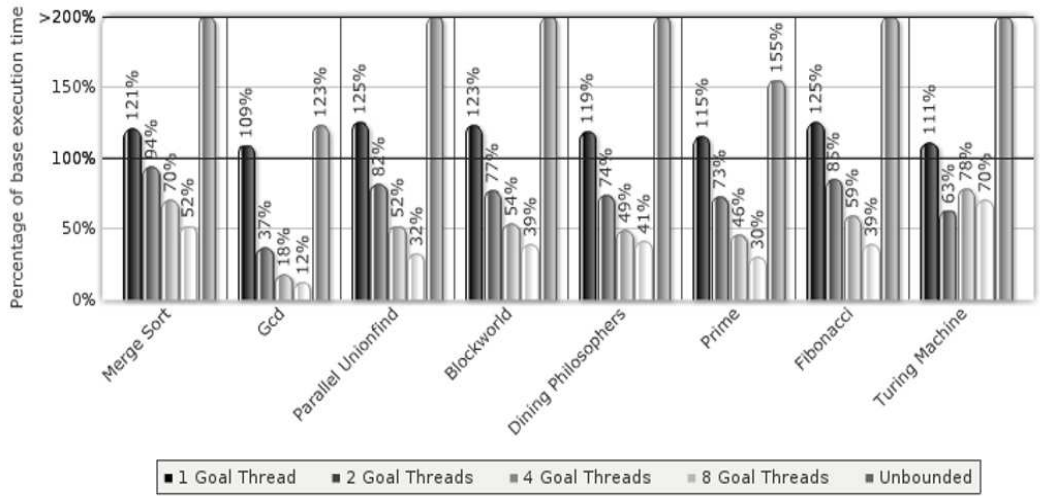

Figure 4.9: Experimental results, with optimal configuration (on 8 threaded Intel processor)

There are several important observations. Firstly, executing our parallel implementation with 1 goal thread is inferior (at all examples) compared to the non-concurrent implementation for obvious reasons (overheads of concurrent execution are introduced, with no benefits of concurrent goal execution being exploited). Executions with 2, 4 and 8 goal threads scale well against base execution time in general, with exception of the Turing Machine example. This is expected as the Turing machine example is inherently single-threaded. Interestingly, we still obtain improvements from parallel execution of administrative procedures (for example dropping of goals, due to failed matching). Relative drop in performance (between 2 and 4/8 goal threads) indicates a upper bound of parallelism of such "administrative" procedures.

One interesting result that our experiment uncovered is the presence of super-linear speed-up for certain examples, like Gcd. The reasons for this is often very subtle and domain specific. Figure 4.10 illustrates why we get super-linear speed-up for the Gcd example. For presentation purpose, we annotate each constraint with a unique identifier and each derivation with the rule name parameterized by the constraints that fired it and the number of times it fired. For instance $g2(x, y) \times t$

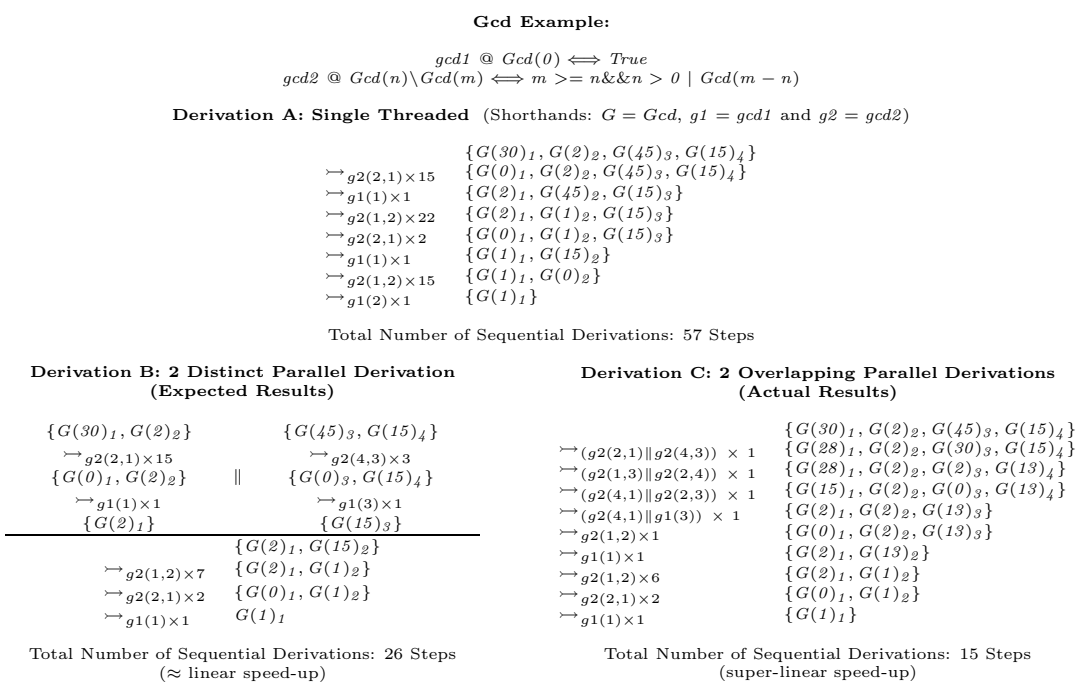

Figure 4.10: Why Super-Linear Speed-up in Gcd

denotes that rule *g2* fired on constraints x and y for t number of times. We examine derivations of the Gcd example from the initial store $\{Gcd(30), Gcd(2), Gcd(45), Gcd(15)\}$ Derivation A shows the single threaded case where we get a total of 57 derivation steps to reach the final store. Derivation B shows the parallel derivation of 2 threads which yield the expected results (linear speed-up of 26 sequential derivation steps). This assumes an unlikely scenario where derivations between 2 pairs of Gcd constraints do not overlap (i.e. interfere with each other). Derivation C shows the actual result which yields super-linear speed-up. Derivations overlap, that is, there can be rule firings across parallel derivations. [23] This allows Gcd constraints of higher values to be matched together, cutting down tediously long derivations initiated by Gcd constraints of lower values (which is typical in the single threaded case). By queuing *Gcd* goals (domain specific goal ordering), we encourage derivations similar to Derivation C to be chosen over other possibilities, since goals are processed in a

[23] Of course, this behavior is also possible in a sequential execution scheme where we interleave the execution of goal constraints, thus, effectively simulating the parallel execution scheme.

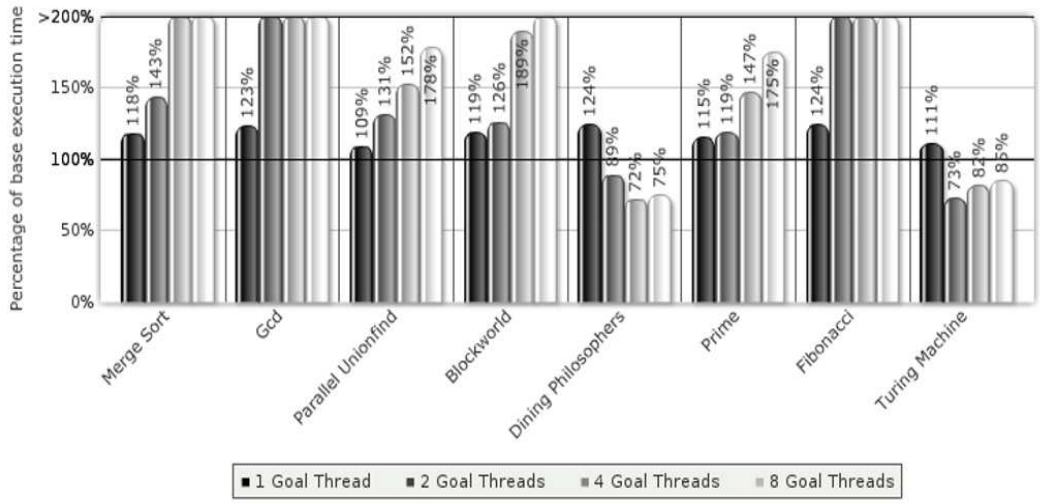

Figure 4.11: Experimental results, with atomic rule-head verification disabled

breadth first manner (See results in Section 4.6.4 for confirmation of this point).

The final important insight lies in the right most data set of each CHR example. "Unbounded" refers to the performance of the parallel CHR system when we do not bound the number of CHR goal threads. This means that we spawn as many Haskell GHC lightweight threads as there are goal constraints, hence representing the abandonment of the **bounded goal thread pool** concurrency optimization. Results here show definitively that unbounded thread pooling (Section 4.4.3) is harmful to parallel CHR goal execution, with all CHR examples in this configuration performing sub-optimally.

4.6.2 Disabling Atomic Rule Head Verification

Figure 4.11 illustrates the experiment results conducted with **atomic rule-head verification** disabled. The alternative implementation we use here is similar to the simple implementation described in Section 4.3 and has the potential to introduce many false-overlaps (illustrated in Section 4.4.1) in concurrent goal execution. In general, results here shows that multi-threaded goal execution performs worse than a single threaded execution or even the basic non-concurrent implementation. This

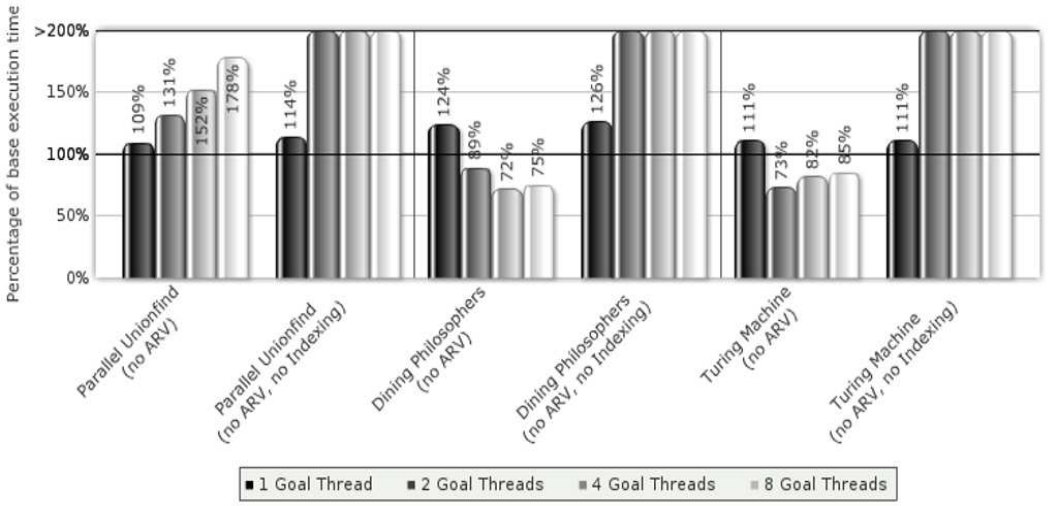

Figure 4.12: Experimental results with and without constraint indexing (atomic rule-head verification disabled)

essentially highlights the importance of minimizing false overlaps in concurrent goal execution, via atomic rule-head verification, or other fine-grained micro management of lower level concurrency primitives.

Dining Philosophers and Turing Machine examples demonstrate slight speed ups over base execution time, showing that there are domains which are more tolerant to the absence of fine-grained synchronization (introduced by atomic rule-head verification). It is not surprising, since Dining philosophers and Turing machine are examples in which CHR rule head matching heavily relies on constraint indexing. For instance, looking at the dining philosopher's problem, the active goal $Think(c, 0, a, b)\#n$ can seek for partners $Fork(a)\#m$ and $Fork(b)\#p$ via specifying indexed lookups for arguments a and b in the $Fork$ constraint store (as oppose to a linear iteration of all $Fork$ constraint, until a and b are found). This reduces the number of shared memory reads and thus reducing number of false overlaps, even without the presence of streamlined synchronization introduced by atomic rule-head verification.

To further support this argument, we investigate further by repeating the experiments, this time with constraint indexing also disabled. Figure 4.12 shows the

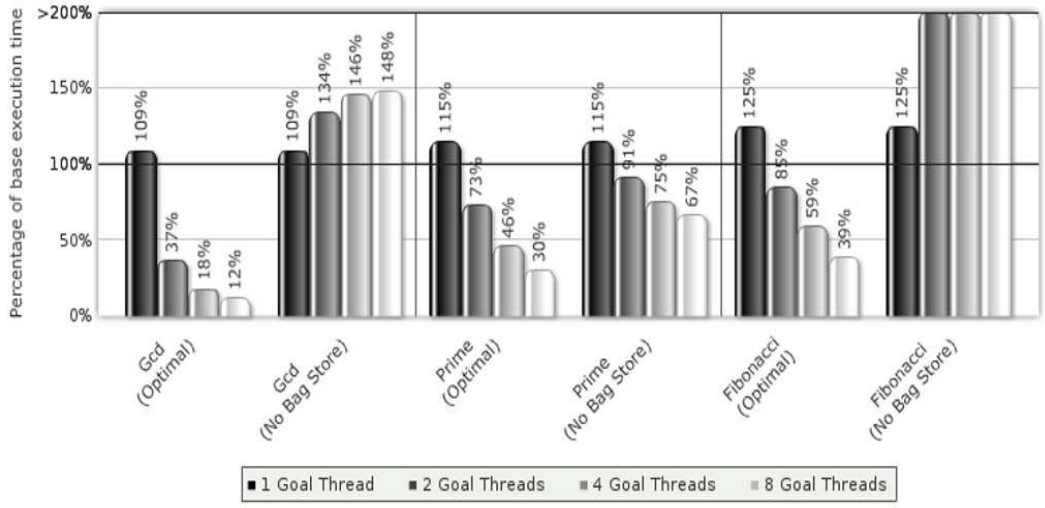

Figure 4.13: Experimental results with and without bag constraint store

highlights of this follow up experiment (we omit the examples which present insignificant or no difference from results in Figure 4.11). In this experiment, we see that Unionfind, Dining philosophers and Turing machine demonstrate terrible performance when constraint indexing is disabled. Since these examples heavily rely on constraint indexing, using linear lookups instead forces goal executions to iterate through many more shared memory locations in the constraint store, thus increasing number of failed concurrent execution due to false overlaps. This explains why results (no ARV and no Indexing) worsens with more goal threads executing in parallel.

4.6.3 Disabling Bag Constraint Store

Figure 4.13 illustrates experiment results conducted without the use of **bag constraint stores and iterators** versus our optimal results from Section 4.6.1. Here, we highlight only the Gcd, Prime and Fibonacci examples as all others show little significant changes in scalability and performance. Without bag constraint store and iterators, Gcd and Fibonacci performs worse when more goal threads are executed in parallel. Even though Prime demonstrate better performance with more goal

threads, it's scalability with more threads is still less impressive than our optimal results.

These results are not entirely surprising, since Gcd, Fibonacci and Prime are indeed CHR problems where CHR goals likely share overlapping sets of potential candidates for partner constraints. For instance, consider the Fibonacci rule *fibo4*:

$$\textit{fibo4} \ @ \ Fibo(x), Fibo(y) \Longleftrightarrow Fibo(x+y)$$

An active goal $Fibo(x1)\#n$ is free to match with any $Fibo$ constraint (variable y of rule head $Fibo(y)$ is unbounded), as such if all parallel goals iterate through potential candidate $Fibo$ constraints in the same order, they will frequently select overlapping constraints. Hence, more computation time is wasted for synchronizing between parallel goal threads (STM roll back and continue search for another available partner). As the experiment results in this section show, such unnecessary synchronization procedures are avoided in our optimal configuration by the use of bag constraint stores and iterators. The Gcd example CHR rule *gcd1* and the Prime example rule *prime3* also shares this similarly with the Fibonacci example rule *fibo4*.

4.6.4 Disabling Domain Specific Goal Ordering

Figure 4.14 illustrates our experiment results without the domain specific goal ordering optimization. In our examples, only Mergesort and Gcd examples specifies a goal ordering[24], hence disabling goal ordering will only affect these two examples.

The results show that without goal ordering, Mergesort without goal ordering does not scale with increasing goal threads. This supports our arguments in Section 4.4.3, confirming that the optimal goal ordering reduces number of conflicting concurrent goal executions and number of *Leq* comparisons. Gcd without goal ordering still performs decently, but without the super linear speed up it experiences with

[24]For Mergesort, *Leq* goals are stacked, $Merge$ goals are queued. For Gcd, goals are queued.

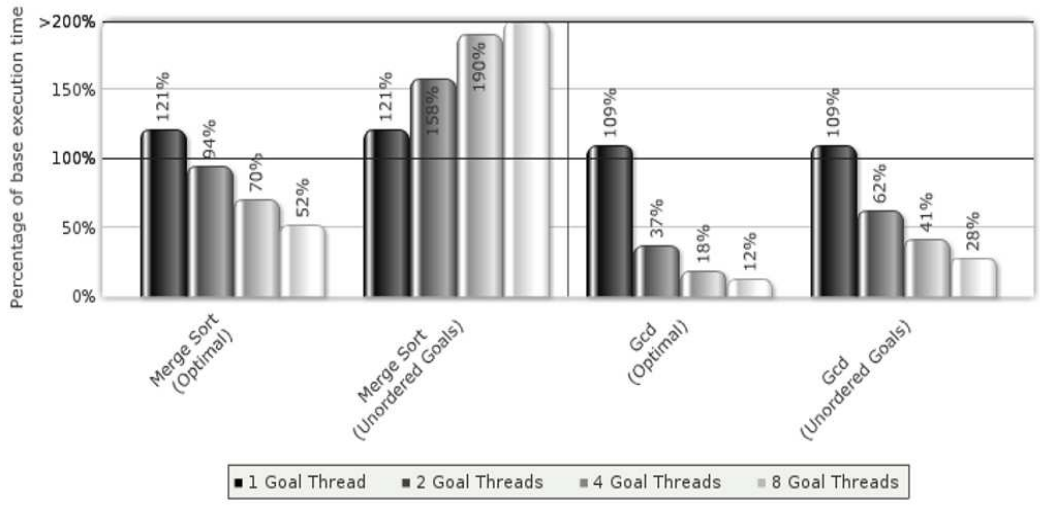

Figure 4.14: Experimental results, with domain specific goal ordering disabled

goal ordering.

4.7 External Benchmarks

We have benchmarked our prototype parallel CHR implementation against a state-of-the-art CHR implementation, SWI Prolog CHR [55] as well as against hand-coded programs in Haskell and it's concurrency primitives. Our parallel CHR implementation when running on one core is 8 to 9 times slower than both SWI Prolog CHR and hand-coded Haskell programs. Running with 8 cores, our implementation is still 2 times slower.

Even though these results seem unmotivating to our course, we like to point out that these relative poor performance is not due to the overheads of a parallel implementation, but more likely other factors. For instance, our prototype implementation still lacks many CHR optimizations orthogonal to parallelism, many of which are available in SWI Prolog CHR. Further more, to avoid the hassles of maintaining a full-fletched CHR compiler, our parallel CHR implementation is implemented as a library based domain specific language (DSL), this has one disadvantage: execution time of programs also includes compile time elements (for instance, generating

optimal match ordering for each CHR rule instance, etc..). Yet, the over-heads of parallelism (concurrency synchoronization overheads) only accounts for $10 - 20\%$ (see Figure 4.9) additional execution time.

While these results do show that our prototype implementation is still far from industrial strength, our main empirical results (Section 4.6.1) have shown our implementation's scalability of execution with number of processor cores, which is aligned to current technological improvement trends of multicore architectures, something not present in existing implementations. Using CHR as a declarative high-level concurrency abstraction allows the programmer to implicitly write scalable parallel applications, without the hassle of hand-coding complex concurrency synchronization routines. Yet this will not be without performance over-heads, hence one of our future works will be to reduce such over-heads to a more reasonable level where parallel multiset rewrite in CHR can provide programmers with high-level concurrency abstractions in the same way modern general purpose programming languages provide programmers high-level programming constructs that are more intuitive than turing machine operations.

4.8 Extensions

In this section, we consider two extensions to our current implementation.

4.8.1 Dealing with Ungrounded Constraints: Reactivation with STM

Our implementation of goal execution illustrated in Table 4.7 and 4.8 does not account for *equation constraints* and *goal reactivation*. Recall that in the $\| \mathcal{G}$ semantics

of Figure 3.3 (Section 3.3) we have the (Solve) transition rule:

$$\textbf{(Solve)} \quad \frac{W = WakeUp(e, Sn)}{\langle \{e\} \uplus G \mid Sn\rangle \overset{W\setminus\{\}}{\underset{\mathcal{G}}{\rightarrowtail}} \langle W \uplus G \mid \{e\} \cup Sn\rangle}$$

$$\begin{array}{lrcl} \text{where} & Eqs(S) & = & \{e \mid e \in S, e \text{ is an equation}\} \\ & WakeUp(e, Sn) & = & \{c\#i \mid c\#i \in Sn \wedge \phi \text{ m.g.u. of } Eqs(Sn) \wedge \\ & & & \theta \text{ m.g.u. of } Eqs(Sn \cup \{e\}) \wedge \phi(c) \neq \theta(c)\} \end{array}$$

Simply put, when an equation constraint e (in general, a builtin constraint) is introduced to the store, we have to reactivate (return to the goals, for future execution) certain constraints represented by the set $WakeUp(e, Sn)$. This is because by introducing equation e, we possibly ground certain variables held by ungrounded constraints in store Sn. Thus, such constraints can possibly trigger new rule-instances and must be re-executed to maintain exhaustiveness of CHR rule execution. For example, consider the following CHR rule:

$$r1@A(1) \Longleftrightarrow B(1)$$
$$r2@B(x) \Longleftrightarrow x = 1$$

Suppose we have two goals $A(a)\#m, B(a)\#n$ for some term variable a. Suppose we execute $A(a)\#m$, but it cannot trigger rule $r1$ because it does not match $A(1)$, hence we drop it. But after executing $B(a)\#n$, we have $a = 1$, which grounds $A(a)\#m$ to $A(1)\#m$. Hence we need to reactivate all constraints containing the term variable a.

The type of builtin constraints available depends on builtin theory which the CHR language is built on top of (eg. linear inequality, herbrand, etc..). For our implementation, we assume a simple builtin store consisting of only equations of the form $x = v$, where x is a term variable and v a value[25].

[25]Note we do not attempt to deal with full-blown parallel unification and hence we do not include equations of the form $x1 = x2$

Term Variable:	`TVar (Maybe String)`
Equation:	`EqCons (TVar (Maybe String)) String`

Figure 4.15: Term Variables via STM Transactional Variables

```
1 solve :: EqCons -> STM Bool
2 solve (EqCons x v) = do
3   { mb <- readTVar x
4   ; case mb of
5       Just v' -> return (v==v')
6       Nothing -> do { writeTVar x (Just v)
7                     ; return True }
8   }
```

Table 4.12: Implementation of Builtin Equations

We represent term variables with STM transactional variables. Figure 4.15 shows our representation of term variables x and equations $x = v$ in Haskell. Term variables are essentially represented by STM transactional memory variables (`TVar`) storing values of type `Maybe String`. An unassigned variable simply contains a `Nothing`, while a variable grounded to the value v contains a `Just v`. An equation is represented by the data type `EqCons x v` where x is a term variable, and v a value. For simplicity, we assume that values are strings. Table 4.12 illustrates the implementation of a simple equation solver. An equation `EqCons x v` is solved by reading the value of x (line 3) checking if it is grounded (lines 5) or if it is not (lines $6-7$). For grounded x, we simply return $True$ if it's value is equal to v (line 5), otherwise we return $False$ indicating a builtin error. For a non-ground x, we write the value v into term variable x and return $True$. The `solve` routine is executed on each equation in the body of a rule instance successfully matched during CHR goal matching and execution (line $28-32$ of `matchGoal` from Table 4.8).

Implementing term variables with STM transactional variables offers us an elegant and unique way of handling constraint reactivation. Essentially, for each goal constraint g with at least one non-ground term variable, we retrieve the set of all it's non-ground term variables xs right before executing g. After executing g, if g

```
1 reactivateWhenGround :: [TVar (Maybe String)] -> Cons -> Goals -> IO ()
2 reactivateWhenGround xs g gs = do
3   { atomically (do { vs <- mapM readTVar xs
4                    ; let ground_vs = filter (\ v -> Nothing == v) vs
5                    ; if (length ground_vs) > 0 then return ()
6                                                else retry })
7   ; addGoals gs [g] }
```

Table 4.13: Goal reactivation thread routine

is not deleted from the store, we spawn a light-weight thread which has the sole purpose of "sleeping" until g's set of ungrounded variables is no longer xs, but a subset of it[26]. We stress the importance of "capturing" ungrounded variables xs strictly *before* goal execution of g, otherwise we may possibly miss rule instances. This is because xs represents a conservative snap-shot of the term variables of g at the time of goal execution, and thus the goal g should be reactivated if xs changes.

Figure 4.13 illustrates the reactivation routine to be executed by goal reactivation threads. We assume each thread runs the routine on a unique ungrounded goal constraint g, and is given the ungrounded term variables of g right before goal execution (xs) and the pointer to the shared goals gs. Lines $3-6$ essentially implements the sleeping procedure, which completes if any term variable in xs has been grounded (line 5), otherwise the entire STM operation is *retried* (line 6). This effectively blocks the computation until a term variable in xs is grounded, during which, the goal g will be added to gs (line 7). Note that `retry` is a Haskell GHC STM library function which is implemented not by busy-wait polling, but will schedule re-execution of the STM operation only when variables it has read before are modified.

Our current implementation does not support ungrounded constraint, even though the solution documented here can be straight-forwardly integrated. This is because in our experience, CHR programs that utilizes ungrounded constraints can be rewritten to a form which only uses grounded constraints (see parallel union find in Section 4.6 for an example). While this restriction does not hinder the expressiveness

[26]Note Haskell uses light-weight threads, which are highly suitable for such non-computational intensive synchronization tasks (unlike, CHR goal execution). If light-weight threads are not available, we can always use thread pooling techniques similar to that illustrated in Section 4.4.3

of programming in Constraint Handling Rules, it is of course not without other practical implications. For instance, without unground constraints, we must specify variable assignments explicitly by means of user-defined constraints (since such features are not built-in in the CHR implementation) making programming in CHR a slightly more tedious experience. This will also most certainly introduce additional rule-heads to the user-defined CHR rules, thus incurring more rule-head matching overheads.

4.8.2 Dealing with Pure Propagation: Concurrent Dictionaries

Section 3.4.7 considered the semantic refinements necessary to deal with pure propagation. Here we discuss the implementation efforts necessary. Recall that we need to guarantee that concurrent goals rewrite unique propagation rule instances. This means that the propagation history must be a shared data structure whose changes must be globally visible to all concurrent threads. The most suitable data structure for this application is a concurrent dictionary. Propagation rule instances (simply represented by a list of constraint identifiers and rule identifiers) are kept as keys of the dictionary, which provides us with the testing interfaces to prone whether a rule instance has been previous fired by some execution thread.

While the data structure required is quite obvious, what is more subtle and interesting is the nature of the history check that needs to be conducted during rule execution. Because the act of commit to a rule instance must be atomic (Atomic rule head verification, see Section 4.5.4), checking and extending the shared propagation history must be done as part of this atomic procedure. In the case of our implementation, this means that we must implement the shared history as a STM data structure.

Table 4.14 illustrates a variant of atomic rule head verification that integrates

```
1  historyContains :: History -> [(Head,Cons)] -> STM Bool
2
3  addToHistory :: History -> [(Head,Cons)] -> STM ()
4
5  verifyRuleHeadsProp :: Store -> History -> [(Head,Cons)] -> STM Bool
6  verifyRuleHeadsProp store hist hds = do
7   { bs <- mapM (\(_,g) -> isStored store g) hds
8   ; if and bs
9     then do { let simpHds = filter (\(h,_) -> h == Simp) hds
10            ; if length simpHds > 0
11              then mapM (\(_,g) -> logicalDeleteFromStore store g) simpHds
12              else do { hasFired <- historyContains hist hds
13                      ; if hasFired
14                        then return False
15                        else do { addToHistory hist hds
16                                ; return True }
17                      }
18            ; return True }
19     else return False }
```

Table 4.14: Atomic rule head verification with propagation history

propagation history handling[27]. We assume that the abstract datatype `History` is the shared propagation history, with two interfaces: `historyContains` *hist hds* returns true if and only if the rule head instance *hds* has been fired, while `addToHistory` *hist hds* adds rule head instance *hds* to the history *hist*. We check if the simplified heads are empty (line 10). If not we proceed with the standard logical delete of simplification heads, otherwise we check if the current rule instance has already been fired (line 12). We will only add the rule instance to the history if and only if the rule instance has not been fired (lines $15 - 16$).

Note that having the a single global shared history built on transactional memory for all propagation rules of a CHR program is most likely to have significant impact on performance. This is because we essentially have to synchronize all triggering of propagation rules on that single shared data structure. A simple optimization is to have one propagation history for each propagation rule of the program. As such, we will only interleave instances of the same propagation rule. This means

[27]Note that for simplicity, we extend from the basic `verifyRuleHeads` rather than `verifyRuleHeadsBackJump` from Table 4.10. Both extensions are orthogonal and can be integrated together.

that concurrent execution of non-overlapping propagation rule-instances of distinct propagation rules can execute in parallel.

Chapter 5

Join-Patterns with Guards and Propagation

5.1 Chapter Overview

In this Chapter, we introduce a non-trivial application of parallel CHR rewriting, namely formalizing and implementing a goal-based execution model for Join-Patterns with guards and propagation, based on our earlier introduced $\| \mathcal{G}$ semantics. Section 5.2 provides a quick review on Join-Calculus and Join-Patterns, and highlights the similarities with CHR rewritings. This is followed by Section 5.3, which introduces guard and propagation extensions to Join-Patterns. Section 5.4 formally introduce the Join-Pattern goal-based semantics. Finally, Section 5.5 concludes with highlights of our implementation and experimental results on a multi-core system.

5.2 Join-Calculus and Constraint Handling Rules

Constraint Handling Rules is a concurrent committed-choice constraint logic programming language to describe rewritings among multi-sets of constraints. Join-

Calculus [18], on the other hand, is a process calculus designed to provide expressive concurrency abstractions in the form of multi-headed reaction rules (known as Join-Patterns). Rule triggering depends on the simultaneous consumption of messages [1] matching each of the rule heads. It is clear that Join-Calculus semantics share a lot in common with the CHR multiset rewriting, yet surprisingly, CHR and Join-Calculus have been studied so far in complete isolation. We believe that a comparison between both calculi is long overdue and should enable a fruitful exchange of ideas and results.

In this Section, we provide a quick review of the Join-Calculus language (Section 5.2.1) as well as the highlights of the standard Join-Pattern compilation schemes used by existing Join-Pattern implementation. Following this, we introduce a simple CHR goal-based compilation scheme adapted for the triggering of Join-Patterns (Section 5.2.3).

5.2.1 Join-Calculus, A Quick Review

There are numerous calculi and concurrent programming models to support concurrent programming. A particular fruitful and promising model appears to be the join calculus [17] which provides the basis for the concurrency abstractions found in numerous existing implementations (eg. JoCaml [7], Polyphonic C# [5], Join Java [59]). In join calculus, concurrency is expressed via sets of multi-headed reduction rules known as *Join-Patterns*. As demonstrated in Section 2.3.3, Join-Patterns are declarative in nature and easy to understand, providing high-level coordination of concurrent processes without the need of explicit micro-management of concurrency primitives.

Figure 5.1 shows the essential core Join-Calculus language. Processes (or events) are typically modeled as unique names p each with a fixed number of term arguments. A collection of concurrently running processes (denoted M) is represented by

[1] Messages can be received from multiple shared channels, from various concurrent computational entities (program threads, remote procedure calls, etc..)

Primitives:

Process(Event) Name	p	Variable	x
Constant Value	v	List of a's	$\overline{a}$

Join-Calculus Expressions:

Term	t	::=	$x \mid v$
Process	P	::=	$p(\bar{t})$
Concurrent Processes	M	::=	$P \mid M\ ,\ M$
Join-Pattern	J	::=	$P \mid J \mid J$
Join-Body	B	::=	$P \mid B \mid B$
Reaction Rule	D	::=	$J \rhd B$

Figure 5.1: Join-Calculus Core Language

processes composed together with a binary operator ",". This collection is treated as an unordered set of processes. For instance, the following illustrates a collection of concurrent processes, representing the state of the printer spooler, denoted $\mathcal{S}$:

$$\mathcal{S} \quad = \quad \mathit{Ready}(P1), \mathit{Ready}(p2), \mathit{Job}(J1), \mathit{Job}(J2), \mathit{Job}(J3)$$

A printer p which is available for printing will call the process $\mathit{Ready}(p)$, while a print job j is submitted to the spooler via calling the process $\mathit{Job}(j)$. To stay true to the notation used in this thesis, we shall use standard Haskell[2] notation to represent variables and constants: Uppercase references for constant names and lowercase references for variables/function names. Hence the above illustrates a state consisting of two available printers and three outstanding print jobs. A print job j is to be matched with any available printer p, during which printing can be initiated by sending j to p ($\mathit{Send}(p, j)$). This behavior is captured by the reaction rule $\mathcal{D}$, defined as follows:

$$\mathcal{D} \quad = \quad \mathit{Ready}(p) \mid \mathit{Job}(j) \rhd \mathit{Send}(p, j)$$

A reaction rule ($J \rhd B$) has two parts. We refer to the left-hand side J as the Join-Pattern and to the right-hand side B as the Join-Body (in our simplified setting

[2]Yes, we love Haskell that much.

rule processes). The Join-Pattern J specifies that processes matching Join-Patterns J can be consumed and replaced by rule processes B. Note that we will sometimes refer to the reaction rules as Join-Patterns as well if there is no ambiguity doing so.

The Chemical Abstract Machine (CHAM) [6] provides the semantic foundations for the Calculus. A set of reaction rules can be applied to a collection of concurrent processes. This is defined by two forms of transition steps, namely structural steps $(R \vdash M) \rightleftharpoons (R \vdash M')$ and reduction steps $(R \vdash M) \longrightarrow (R \vdash M')$ where R is the set of reaction rules and M, M' are collections of concurrent processes. This exploits the analogy that concurrent processes are a "chemical soup" of atoms and molecules, while reaction rules define chemical reactions in this chemical soup. Structural steps heat/cool atoms to and from molecules (switching to-and-from ',' and '|'), while reduction steps apply reaction rules to the matching molecules. The following shows a possible sequence of structural/reduction steps which results from applying the printer spooler rule $\mathcal{D}$ on the spooler state $\mathcal{S}$:

$$\mathcal{D} = \mathit{Ready}(p) \mid \mathit{Job}(j) \rhd \mathit{Send}(p, j)$$

$$\begin{aligned}
& (\{\mathcal{D}\} \vdash \mathit{Ready}(P1), \mathit{Ready}(P2), \mathit{Job}(J1), \mathit{Job}(J2), \mathit{Job}(J3)) \\
\rightleftharpoons \quad & (\{\mathcal{D}\} \vdash \mathit{Ready}(P2), \mathit{Job}(J2), \mathit{Job}(J3), \mathit{Ready}(P1) \mid \mathit{Job}(J1)) \\
\longrightarrow \quad & (\{\mathcal{D}\} \vdash \mathit{Ready}(P2), \mathit{Job}(J2), \mathit{Job}(J3), \mathit{Send}(P1, J1)) \\
\rightleftharpoons \quad & (\{\mathcal{D}\} \vdash \mathit{Job}(J3), \mathit{Send}(P1, J1), \mathit{Ready}(P2) \mid \mathit{Job}(J2)) \\
\longrightarrow \quad & (\{\mathcal{D}\} \vdash \mathit{Job}(J3), \mathit{Send}(P1, J1), \mathit{Send}(P2, J2))
\end{aligned}$$

When concurrent processes J' matches a reaction rule $J \rhd B$ (ie. $J' = \theta(J)$ for some substitution θ) causing the rule to be applied, we say that J' has *triggered* the Join-Patterns J. Note the inherent non-determinism in matching processes with Join-Patterns: any pair of $\mathit{Ready}(p)$ and $\mathit{Job}(j)$ can be arbitrarily chosen by a structural step and matched with the Join-Patterns.

There is an obvious similarity between the CHR semantics and Join-Calculus semantics. Transitions of the chemical abstract machine essentially describes rewrit-

ings among multisets of processes, the same way the abstract CHR semantics (Chapter 3) describe rewritings among multisets of constraints via CHR simplification rules. Let's consider the same rewritings of our print spooler example (earlier in the CHAM rewritings), this time in abstract CHR derivations:

$$r \ @ \ Ready(p), Job(j) \Longleftrightarrow Send(p, j)$$

$$\begin{aligned} & \{Ready(P1), Ready(P2), Job(J1), Job(J2), Job(J3)\} \\ \rightarrowtail_{\mathcal{A}} \quad & \{Send(P1, J1), Ready(P2), Job(J2), Job(J3)\} \\ \rightarrowtail_{\mathcal{A}} \quad & \{Send(P1, J1), Send(P2, J2), Job(J3)\} \end{aligned}$$

In the following Sections, we will introduce a concrete Join-Pattern language extension and review the standard compilation scheme of Join-Patterns and show that a similar compilation scheme can be derived from the CHR goal-based semantics.

5.2.2 Programming with Join-Patterns

We consider a simple Join-Pattern extension to Haskell GHC. For now, we will defer details of the actual underlying implementation, but focus on introducing the language. This will provide more concrete examples on how Join-Patterns are used in practical programming.

For clarity, we repeat our example from Chapter 2. Table 5.1 illustrates a simple shared communication buffer implemented our Join-Pattern extension[3]. Join-Patterns are introduced via the `event` keyword. we introduce two events to consume (`Get`) and produce (`Put`) buffer elements. Via the Join-Pattern `Get(x) & Put(y)` we look for matching consumer/producer events. If present, the matching events are removed and the join body is executed. This expression `x := y` simply assigns variable `x` the value of `y`, modeling the retrieval of a buffered item. In general, the

[3]We present our language will slightly sugared syntax for simplicity. In our actual implementation, we implementation our Join-Pattern extension as a combinator library extension

```
event Put(Async Int)
event Get(Sync Int)

Get(x) & Put(y) = x := y

t1 = do { Put(3)              t2 = do { Put(5)
        ; Put(4)                       ; x2 <- newSync
        ; x1 <- newSync                ; Get(x2)
        ; Get(x1)                      ; v2 <- readSync x2
        ; v1 <- readSync x1            ; print v2 }
        ; print v1 }

main = do { forkIO t1
          ; forkIO t2
          ; -- sleep and wait for t1 and t2
          ; ... }
```

Table 5.1: Get-Put Communication Buffer in Join-Patterns

join body is simply a call back function executed when the matching events specified by the Join-Pattern are present.

Events are essentially called like function calls. For instance, in Table 2.2 operation `t1` and `t2` make calls to `Get` and `Put`. Arguments of events can either be asynchronous (ground input values), synchronous (output variables). Synchronous arguments, generated via the `newSync` primitive, serve to transmit buffer elements. We can access the transmitted values via primitive `readSync` which blocks until the variable is bound to a value. Synchronous variables are written into via `:=`. We assume that `print` is a primitive function that prints it's argument on the shell terminal.

Suppose we execute the two threads executing `t1` and `t2` respectively. Events are non-blocking, they will be recorded in the store and we proceed until we hit a blocking operation. Hence, both threads potentially block once we reach their first `readSync` statement. `main` represents the top level function that runs the operations `t1` and `t2` concurrently. It basically calls `forkIO` tx which forks off a light-weight Haskell thread that executes the given operation. We summarize the important

library functions of this language extension,

- `forkIO` *op* - Given `IO` operation *op*, forks off a light-weight Haskell thread that executes *op*. This is a Haskell GHC library function.
- `newSync` - Returns a new synchronization variable, which is empty (i.e. unassigned).
- `readSync` x - Attempts to retrieve the value assigned to x. If x is unassigned, this operation blocks until x is assigned a value by another thread.
- x `:=` y - Assigns x the value of y. This operation assumes that y is assigned a value.

5.2.3 Join-Pattern Compilation and Execution Schemes

We start off by reviewing the standard Join-Pattern compilation and execution schemes, used by existing Join-Pattern implementations. Following this, we introduce our CHR goal-based compilation scheme for Join-Pattern execution by example.

Standard Join-Pattern Compilation Scheme

Existing implementations compile Join Patterns into state machines that maintain the matching states of the Join-Patterns [13]. This compilation involves constructing n message channels (which are typically queues) and a finite state machine (automaton) for each set of Join-Patterns, such as the one in Figure 5.2. These message queues, together with the finite state machine keeps track of the *matching status* of the Join-Patterns. Each message channel is assigned to one of the process name (P, Q or R), represents the collection of calls to this process by concurrent computation threads. Hence, a call to a process is analogous to the arrival of a new message in the corresponding message channel.

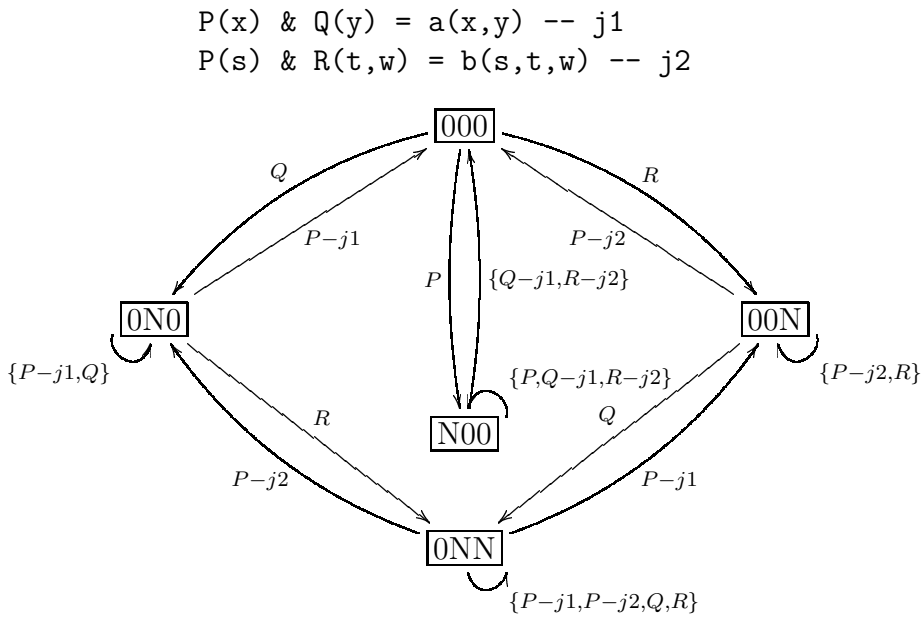

Figure 5.2: A Matching Status Automaton with two Join-Patterns

---

Figure 5.2 shows an example of a Join-Calculus expression consisting of two reaction rules, as well as it's corresponding matching status automaton that is constructed. We label the first reaction rule as $j1$ and the other as $j2$. $j1$ has the Join-Pattern $P(x)\&Q(y)$ and the join body $a(x, y)$, while $j2$ has $P(s)\&R(t, w)$ and $b(s, t, w)$. We assume that function calls a and b reduces to sequences of primitive operations that may consist of other Join-Pattern process calls (eg. P, Q or R).

States of the finite state machine are labeled by a sequence of n bits, one assigned to each message queue stating whether it is empty (0) or non-empty (N). Let's assume that the order is P, Q then R. This automaton is updated every time a new process is call (ie, a new message has arrived) or when a Join-Pattern is successfully matched.

Each edge labeled with a transition label, which is either of the form $m - j$ stating that arrival of message m has triggered Join-Pattern j (and hence consumed m together with other messages involved in the join), or just m which states that arrival of message m has triggered nothing and therefore is just queued. If there are more than one alternative transitions between two states of the automaton, we will

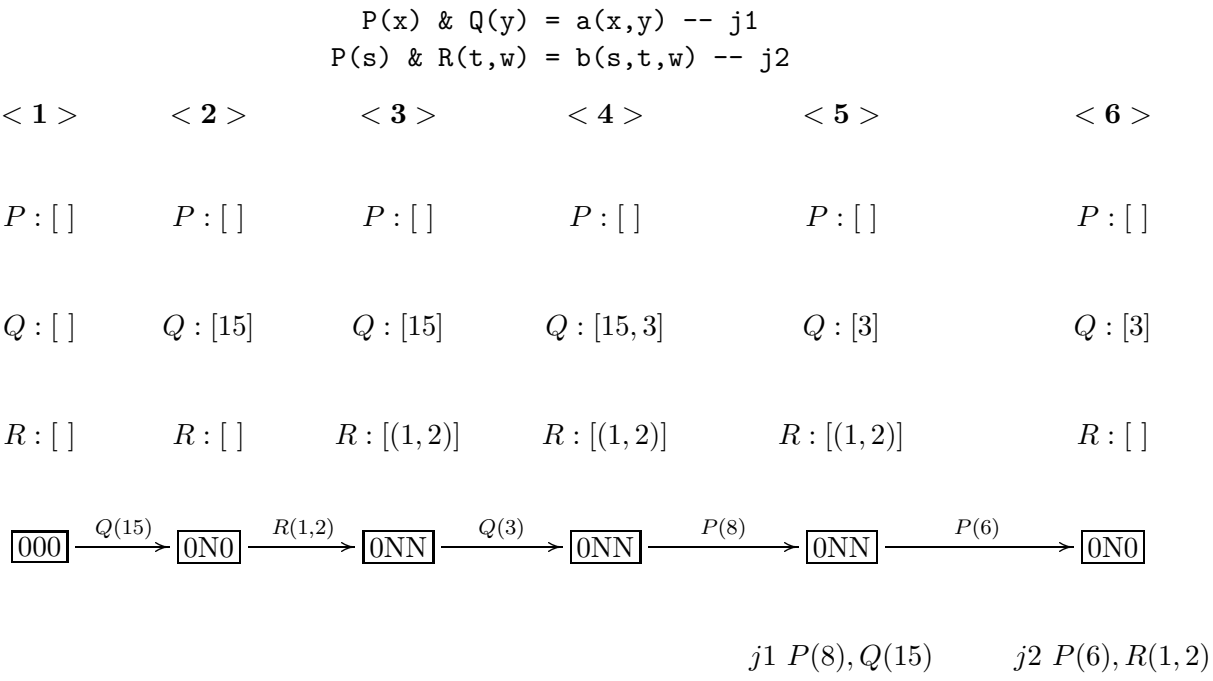

Figure 5.3: Example of Join-Pattern Triggering with Finite State Machine

represent them as a single edge with the set of alternative transitions. Since the Join-Patterns considered do not have guard conditions, messages channels are normally implemented with shared queue data structures, and messages are consumed in a first-in-first-out manner.

Figure 5.3 illustrates an example of the triggering of Join-Patterns from Figure 5.2. We illustrate a scenario when join processes (messages) are received in the sequence $Q(15), R(1,2), Q(3), P(8), P(6)$[4]. Iterations $< 1 >$ to $< 6 >$ represents the arrival of the respective join processes. For each iteration $< x >$, we show the queue and state of the finite state machine generated for the Join-Patterns. Initially, the queues are all empty and we are at state 000 ($< 1 >$). Iterations $< 2 >$ and $< 3 >$ represents the arrival of messages $Q(15)$ and $R(1,2)$ respectively, transiting to the states $0N0$ then $0NN$, as specified by the finite state machine. In $< 4 >$, a $Q(15)$ arrives, but we remain in the state $0NN$ even though the queue has two Q

[4]In theory, join processes can be called concurrently, but for the purpose of this example, we consider this specific sequence.

messages. Finally in $< 5 >$, we see a $P(8)$ message and we trigger the Join-Pattern $j1$ (as specified by state machine transition $P - j1$) and remove the messages $P(8)$ and $Q(15)$. This is followed by $< 6 >$ another message $P(6)$ and hence the triggering of the Join-Pattern $j2$ via messages $P(6)$ and $R(1, 2)$.

Note that in iteration $< 5 >$, we could have selected $P-j2$ instead of $P-j1$, thus triggering Join-Pattern $j2$ on $P(8)$ and $R(1, 2)$ instead. Both alternatives are valid, but in actually implementations, a Join-Pattern compiler will arbitrarily choose one.

CHR Goal-Based Compilation Scheme for Join-Patterns

We present an alternative compilation scheme for Join-Patterns, which essentially compiles Join-Pattern into CHR rules. There are of course, fundamental differences between the execution of CHR rules and that of Join-Pattern reaction rules (asynchronous rule body execution), but here we focus on how we can compile the *triggering* of Join-Patterns with CHR rewritings.

Rather than compiling Join-Pattern reaction rules into message queues and a finite state machine, we compile Join-Patterns into a set of CHR rules and a CHR shared store. We consider the example in Figure 5.2. Join Patterns of $j1$ and $j2$ can basically be represented by the following CHR simplification rules:

$$r1 \ @ \ P(x), Q(y) \Longleftrightarrow ExecA(x, y)$$
$$r2 \ @ \ P(s), R(t, w) \Longleftrightarrow ExecB(s, t, w)$$

where $r1$ corresponds to $j1$ and $r2$ to $j2$. Constraints $ExecA(x, y)$ and $ExecB(s, t, w)$ symbolically represent the execution of processes $A(x, y)$ and $B(s, t, w)$[5]. Derivations of these CHR rules essentially represents the triggering of the reaction rules $j1$ and $j2$. In this approach, execution of CHR goals essentially maps to the execution of Join-Pattern processes. For instance, considering the example in Figure 5.3, where join processes arrive in the sequence $Q(15), R(1, 2), Q(3), P(8), P(6)$, we have the

[5]For now, we shall treat our CHR rewritings as symbolic representations of Join-Pattern triggering, while later in Section 5.3 and 5.4 we will formally define a goal-based Join-Pattern semantics for a Join-Pattern extension in the language Haskell.

following CHR derivations:

Iteration	**CHR Transitions**		**Message Store**
$< 1 >$			$\langle \{Q(15), R(1,2), Q(3), P(8), P(6)\} \mid \{\} \rangle$
$< 2 >$	(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle \{R(1,2), Q(3), P(8), P(6)\} \mid \{Q(15)\#1\} \rangle$
$< 3 >$	(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle \{Q(3), P(8), P(6)\} \mid \{Q(15)\#1, R(1,2)\#2\} \rangle$
$< 4 >$	(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle \{P(8), P(6)\} \mid \{Q(15)\#1, R(1,2)\#2, Q(3)\#3\} \rangle$
$< 5 >$	(Activate), (Simplify r1)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle \{P(6), ExecA(8,15)\} \mid \{R(1,2)\#2, Q(3)\#3\} \rangle$
$< 6 >$	(Activate), (Simplify r2)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle \{ExecA(8,15), ExecB(6,1,2)\} \mid \{Q(3)\#3\} \rangle$

For this example, we illustrate sequential goal execution in left-to-right sequence, but note that CHR goals (join processes) are concurrent and can execute in any arbitrary ordering. In this compilation, iterations $< x >$ are represented by a series of CHR goal-based semantics transitions. The CHR constraint store represents the set of messages that awaits to be matched to Join-Patterns. As such, we will refer to it as the *message store* from now. For instance, iteration $< 2 >$ is represented by the activation of goal $Q(15)$, followed by the dropping of the same goal (since no CHR rules can be fired). In the standard Join-Pattern compilation, this is equivalent to the transition $000 \xrightarrow{Q} 0N0$ in the finite state machine and the adding of 5 to the Q message queue. The triggering of a Join-Pattern ($< 5 >$ and $< 6 >$) is represented by the activation of a goal, followed by the firing of a CHR rule instance, specifically $P(8), Q(15) \Longleftrightarrow ExecA(8,15)$ in iteration $< 5 >$ and $P(6), R(1,2) \Longleftrightarrow ExecB(6,1,2)$ in iteration $< 6 >$. When compared to the standard Join-Pattern compilation, these correspond in the following manner:

CHR Goal-Based Compilation	**Standard Join-Pattern Compilation**
$\langle\{P(8), P(6)\} \mid \{Q(15)\#1, R(1,2)\#2, Q(3)\#3\}\rangle$	State transition $0NN \overset{P-j1}{\rightarrow} 0NN$
$\rightarrowtail^*_{\mathcal{G}}$ $\langle\{P(6), ExecA(8,15)\} \mid \{R(1,2)\#2, Q(3)\#3\}\rangle$	Remove 15 from Q queue
	Remove 8 from P queue
$\langle\{P(6), ExecA(8,15)\} \mid \{R(1,2)\#2, Q(3)\#3\}\rangle$	State transition $0NN \overset{P-j2}{\rightarrow} 0N0$
$\rightarrowtail^*_{\mathcal{G}}$ $\langle\{ExecA(8,15), ExecB(6,1,2)\} \mid \{Q(3)\#3\}\rangle$	Remove $(1,2)$ from R queue
	Remove 6 from P queue

We wish to point out that triggering Join-Patterns with CHR rewritings would obviously seem a little "over-kill", incurring unnecessary overheads (compared to the highly efficient state-machine compilation highlighted earlier). In the next section (Section 5.2.4), we will demonstrate the advantages and new possibilities ushered in by this alternative compilation.

5.2.4 $\| \mathcal{G}$ Semantics and Join-Patterns

Our CHR goal-based compilation scheme for Join-Patterns offers more than just an alternative compilation scheme for Join-Patterns, but also provides us with a parallel execution model for Join-Pattern triggering. Standard Join-Pattern compilation into a finite state machine has one fundamental limitation: it is essentially a single lock shared resource. Each join process call (eg. P, Q or R) must essentially attempt to acquire exclusive access to the finite state machine representing the Join-Pattern and invoke the state transition before releasing access rights for the next process. This basically means that all processes involved in the same set of Join-Patterns will ultimately synchronize on the same shared resource (the finite state machine).

Our concurrent CHR goal-based semantics ($\| \mathcal{G}$ semantics from Chapter 3) essentially provides the necessary formalism to execute such triggering in parallel, hence

multiple processes attempting to trigger Join-Patterns can compute their matching state in parallel.

Yet we must be realistic and evaluate the benefits of parallelized Join-Pattern triggering. Recall that in the standard Join-Pattern compilation scheme, we compile Join-Patterns into n message queues, where n is the number of unique Join-Pattern processes. State-of-the-art Join-Pattern implementations [13, 5] employ highly efficient bit masking techniques to implement Join-Pattern triggering mechanisms. In this technique, the runtime system simply keeps track of n bits each associated to a process type, which is 0 if the queue of that process is empty, and 1 otherwise. The task of triggering a Join-Pattern is simply to apply a bit-wise boolean operation. For instance, considering the example in Figure 5.2, Join-Pattern of $j1$ will be represented by the bits 001 (A 0 bit indicates that it's associated queue must be not empty, hence P and Q must not be empty to trigger $j1$), while $j2$ by 010. Hence, given a state 110 (P and Q queues are not empty), to test if we can trigger $j1$ we simply do a disjunction between $j1$'s bits and the current state's, i.e. $110 \vee 001 \equiv 111 \equiv True$. Approaches in [59] even attempt introduce hardware support in the execution of Join-Pattern triggers via hardware logical gates, hence making it a highly efficient operation. As such the triggering of Join-Pattern in standard compilation, is an inherently sequential but highly efficient task. This raises questions on the effectiveness of parallelizing an already highly efficient task[6]

While this probably means that little parallelism and performance benefits can be gained from compiling Join-Patterns into CHR rules, we will show in the following section when this approach will be beneficial.

[6]Simply put, by Amdahl's Law [2], "Overall system speed is governed by the slowest component". Hence, parallelizing Join-Pattern triggering which probably accounts for a minute fraction of runtime is as good as blasting microorganisms on the moon (if any exists) with nuclear weapons.

```
P(x) & Q(y) when (x > y || x == y) = a(x,y) -- j3
        P(s) & R(s,w) = a(s,s,w) -- j4
```

Figure 5.4: Example of Join-Patterns With Guards

5.3 Join-Patterns with Guards and Propagation

Let's consider an extension to the Join-Pattern language, known as *Join-Patterns with guards*. Now, Join-Patterns are not only a join between a multiset of processes, each Join-Pattern may optionally specify a boolean expression which determines if the arguments of a given set of join processes are acceptable to trigger the reaction rule.

Figure 5.4 illustrates an extension of our example from Figure 5.2, using Join-Patterns with guards to specify more complex synchronization patterns. Each Join-Pattern now has an optional `when` clause which specifies a guard expression. $j3$ is similar to $j1$ (from Figure 5.2) except it will only react to processes $P(x), Q(y)$ such that `x > y || x = y`, where || is logical disjunction. $j4$ is similar to $j2$ except it only reacts to processes $P(s_1)$ and $R(s_2, w)$, such that $s_1 = s_2$[7]. Considering the arrival of join process calls (messages) in the order $Q(15), R(1,2), Q(3), P(8), P(6)$, we notice that $P(8)$ and $Q(15)$ no longer can trigger any Join-Pattern, as the guard condition for $j3$ cannot be satisfied by these two process ($8 \not> 15$), hence we must search for another suitable Q process (if any). In this case, we have $P(6), Q(3)$ that can trigger $j3$. Similarly, $P(6), R(1,2)$ cannot trigger $j4$ ($6 \neq 1$) nor can we trigger $j3$ with $P(6), Q(15)$ ($6 \not> 15$), thus $P(6)$ will not trigger any Join-Patterns in this example.

[7]Note in Figure 5.4, we illustrate $j4$ with a non-linear pattern (variable s appears twice in the Join-Pattern). This is equivalent to `P(s) & R(t,w) when (s == t) = B(s,t,w)`

5.3.1 Parallel Matching and The Goal-Based Semantics

Implementing the triggering of Join-Patterns with Guards require a search procedure to iterate through all available messages and find a matching set of messages that triggers a Join-Pattern. This is oppose to triggering standard Join-Patterns, where no search is required (we only test if message queues are empty). Join-Patterns with guards can be straight-forwardly implemented by compiling Join-Patterns into CHR rules. The CHR goal-based execution model provides the exact search procedure required to locate matching sets of join processes which satisfy a given guard condition. For instance, compiling the Join-Patterns with guards $j3$ and $j4$ in Figure 5.4, we have the following CHR rules:

$$r3 \ @ \ P(x), Q(y) \Longleftrightarrow x > y \ || \ x == y \mid ExecA(x, y)$$
$$r4 \ @ \ P(s), R(s, w) \Longleftrightarrow ExecB(s, s, w)$$

CHR Transitions		**Message Store**
		$\langle\{Q(15), R(1,2), Q(3), P(8), P(6)\} \mid \{\}\rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle\{R(1,2), Q(3), P(8), P(6)\} \mid \{Q(15)\#1\}\rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle\{Q(3), P(8), P(6)\} \mid \{Q(15)\#1, R(1,2)\#2\}\rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle\{P(8), P(6)\} \mid \{Q(15)\#1, R(1,2)\#2, Q(3)\#3\}\rangle$
(Activate), (Simplify r1)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle\{P(6), ExecA(8,3)\} \mid \{Q(15)\#1, R(1,2)\#2\}\rangle$
(Activate), (Drop)	$\rightarrowtail^*_{\mathcal{G}}$	$\langle\{ExecA(8,3)\} \mid \{Q(15)\#1, R(1,2)\#2, P(6)\#4\}\rangle$

Guards of the Join-Patterns are compiled directly into the guards of the CHR rule. CHR derivations (as shown above) also models the exact behaviour which we expect from the Join-Patterns. When each process is call (activated, in CHR lingo), it searches the message store for matching partners to trigger either $r3$ or $r4$. If no matches can be found, we conclude the execution of the process with not Join-Pattern triggered (Drop), otherwise we execute the rewriting specified by the triggering of the Join-Pattern (Simplify).

With CHR style matching and search for matching processes as a crucial compo-

nent of Join-Pattern execution, the need for parallel matching now becomes apparent and perhaps highly critical. The cost of triggering of Join-Pattern with guards is much higher compared to triggering standard Join-Patterns, hence we cannot afford to have such concurrent procedures be sequentially executed one after another. Our works on the concurrent CHR goal-based semantics (Chapter 3) and parallel CHR implementation (Chapter 4) essentially highlights the necessary ingredients to implement a scalable parallel implementation of Join-Patterns with guards.

CHR goal-based lazy matching (Section 3.4.5) is a highly suitable model for computing the triggering of Join-Pattern with guards. This is because each process (CHR goal) essentially will strictly compute only it's *own* rule-head matches[8] asynchronously (without directly synchronizing with other processes), and proceeds immediately. In essence, this is the ideal execution strategy for executing Join-Pattern processes.

The standard Join-Patterns compilation scheme is heavily tailored made for triggering standard Join-Patterns (simple message queues and bit-wise testing operations). While this makes it highly optimized for triggering standard Join-Patterns, it cannot natively handle the compilation of Join-Patterns with guards, unless serious modifications are implemented. Existing work in [49, 36, 5] addresses implementation of Join-Patterns with guards to a limited capacity, but do not address parallel matching and optimized compilations in general.

```
1  event Item(Async Key,Async Data)
2  event Set(Async Key,Async Data)
3  event Get(Async Key,Sync Data)
4
5  Item(k1,x) & Set(k2,y)
6         when k1 == k2 = Item(k,y)
7  Item(k1,x) & Get(k2,y)
8         when k1 == k2 = do { y:= x
9                            ; Item(k1,x) }
```

Table 5.2: Concurrent Dictionary in Join-Patterns with Guards

[8]By this, we mean it only searches for the rule-head matches that it is part of.

Let's consider another example. Table 5.2 illustrates an implementation of a concurrent dictionary in Join-Patterns with Guards. `Item(k,x)` represents an item `x` mapped by key `k`. Via the first Join-Pattern, `Set(k,y)` essentially replaces an item `x` mapped to key `k` (`Item(k,x)`) with `y` (`Item(k,y)`). `Get(k,y)` simply retrieves the value mapped to key `k` in the dictionary, or blocks until one is available. Note that the guards conditions `k1 == k2` critically provides the join of the keys between the processes.

It may be tempting to try to implement Join-Patterns with guards, by "pushing" the guard expression into the body of the Join-Pattern. For instance, the following attempts to do this:

```
Item(k1,x) & Set(k2,y) = if k1 == k2
                         then Item(k,y)
                         else do { Item(k1,x)
                                 ; Set(k2,y) }
Item(k1,x) & Get(k2,y) = if k1 == k2
                         then do { y:= x
                                 ; Item(k1,x) }
                         else do { Item(k1,x)
                                 ; Get(k2,y) }
```

Specifically, we attempt to implement the concurrent dictionary via standard Join-Patterns. The guard condition `k1 == k2` is pushed into the respective join bodies as if-statements. Note that if the guard conditions fail, we simply "return" the join processes as though we have never triggered the Join-Pattern. There are several problems to this approach. Firstly, we possibly test, re-execute and re-add the same processes over and over again. Since processes are likely to be added into queues, there is no way we can observe if each `Get` process have attempted to match all `Item` processes. This execution model is essentially a busy-wait model of concurrency and is highly inefficient because it simply waste away CPU computation

time on meaningless queuing and dequeuing of processes. On the other hand, the concurrent CHR semantics provides the systematic lazy goal-based search for such multi-headed patterns which avoids such busy-wait cycles and can be executed in parallel.

5.3.2 Join-Patterns with Propagation

Let's take a second look at Table 5.2, the second Join-Pattern removes the item, reads its content, and then "puts" the item back into the dictionary by call the same process `Item(k1,x)`. This can be inefficient simply because we have removed `Item(k1,x)` when we trigger the Join-Pattern and later in the body re-inserted it into the message store. Notice that if we haven't removed the `Item(k1,x)` when we triggered the Join-Pattern, we would have avoid the overheads of the delete and insert operations from and into the message store. This motivates the introduction of CHR style propagation (Section 2.2.4) into Join-Patterns as well.

```
Item(k1,x) & Set(k2,y) when k1 == k2 = Item(k,y)
Item(k1,x) \ Get(k2,y) when k1 == k2 = y:= x
```

Table 5.3: Concurrent Dictionary in Join-Patterns with Guards and Propagation

Table 5.3 reformulates the concurrent dictionary in Join-Patterns with Guards and Propagation. Borrowing from CHR syntax, processes before the \will be propagated (i.e. not removed from the message store) while processes after will be simplified (i.e. removed from the message store) as usual.

Propagation also promotes better parallelism behaviour. For instance, consider two threads running `Get(k,y1)` and `Get(k,y2)` in parallel attempting to match `Item(k,x)`. Since `Item(k,x)` is propagated on the second Join-Pattern, we can practically trigger the two instances of the Join-Pattern in parallel (`Item(k,x) Get(k,v1)` and `Item(k,x) \Get(k,v2)`).

5.3.3 More Programming Examples

We present several more examples which exploits Join-Patterns with guards and propagation.

Join Patterns as Bounded Atomic Transactions We demonstrate how with Join-Patterns, we can model bounded atomic transactions. Transactions are bounded because the number of elements in a Join-Pattern is fixed at compile-time.

A typical task in concurrent programming is to guarantee the atomic execution of certain programs parts. Let's consider our concurrent dictionary example from Table 5.3 again. Suppose that we want to guarantee an atomic swap between two dictionary mappings `k1` and `k2`.

```
1  -- repeated definitions
2  Item(k,x) & Set(k,y) = Item(k,y)
3  Item(k,x)  Get(k,y) = y:= x
4
5  -- failed atomic transfer attempt
6  swap k1 k2 = do { y1 <- newSync
7                  ; Get(k1,y1)
8                  ; v1 <- readSync y1
9                  ; y2 <- newSync
10                 ; Get(k2,y2)
11                 ; v2 <- readSync y2
12                 ; Set(k1,v2)
13                 ; Set(k2,v1) }
14
15 -- atomic transfer via join patterns
16 event Swap(Async Int, Async Int, Async Int)
17 Item(k1,v1) & Item(k2,v2) & Swap(k1,k2) = do
18     { Item(k1,v2)
19     ; Item(k2,v1) }
```

Table 5.4: Atomic swap in concurrent dictionary

Table 5.4 shows a naive implementation which does not guarantee that the swap happens atomically. The problem is that in between the join process calls in the body of `swap`, the values retrieved from keys could have been updated by other concur-

rently running processes before either `Set(k1,v2)` or `Set(k2,v1)` can be executed. The problem is that join bodies are not executed atomically (as though a single instantaneous operation), hence the correctness of the `swap` cannot be guaranteed.

Table 5.4 also illustrates the solution in Join-Patterns with guards: we declare a new `Swap` join process and a new Join-Pattern that pairs a `Swap(k1,k2)` with two matching `Item` join processes. The underlying CHR rewriting semantics guarantees that join processes `Swap(k1,k2)`, `Item(k1,v1)` and `Item(k2,v2)` are consumed atomically, so even if the join body is not executed atomically, both key mappings `k1` and `k2` are exclusively acquired by the swap join process.

```
1  event Think(Async Philo,Async Fork,Async Fork,Sync Bool)
2  event Fork(Async Fork)
3
4  Think(p,l,r,o) & Fork(l) & Fork(r) = (o := True)
5
6  philosopher p l r = do
7      { o <- newSync
8      ; Think(p,l,r,o)
9      ; threadDelay 100
10     ; v <- readSync o
11     ; Fork(l)
12     ; Fork(r)
13     ; philosopher p l r }
```

Table 5.5: Dining Philosophers

Similarly, we encode dining philosophers. See Table 5.5. Synchronous argument `o` has a single purpose, in line 10, it is read simply to block the philosopher until she/he has acquired her/his allocated forks. To avoid deadlocks, each philosopher must atomically grab one fork to its left and one fork to its right. This property is guaranteed by the Join-Pattern.

N-Way Synchronization Another typical programming pattern is n-way synchronization. Several parties wait for each others arrival and then exchanges some data. We can models this via n-headed Join-Patterns which contain a mix of asyn-

chronous and synchronous arguments. For concreteness, let's consider the *Gossiping girls* example, where this feature is highly useful.

A number of girls initially know one distinct secret each. Each girl has access to a phone which can be used to call another girl to share their secrets. Each time two girls talk to each other they always exchange all secrets with each other (thus after the phone call they both know all secrets they knew together before the phone call). The girls can communicate only in pairs (no conference calls) but it is possible that different pairs of girls talk concurrently.

```
1  -- two-girl calls
2   event GirlCall(Async GirlId,Async Secret,Sync Secret)
3
4   GirlCall(g1,s1,o1) & GirlCall(g2,s2,o2)
5     when notSubsets s1 s2 = do { let s = union s1 s2
6                                ; o1 := s
7                                ; o2 := s }
8
9   girl id curSecret = do
10    { s <- newSync
11    ; GirlCall(id,curSecret,s)
12    ; newSecret <- readSync s
13    ; girl id newSecret }
14
15 -- multi-girl calls
16  GirlCall(g1,s1,o1) \ GirlCall(g2,s2,o2)
17    when sALL_SECRETS == s1 && notSubset s1 s2 = (o2 := s1)
18
19  GirlCall(g1,s1,o1) & GirlCall(g2,s2,o2)
20    when not (sALL_SECRETS == s1) &&
21         not (sALL_SECRETS == s2) &&
22         (notSubset s1 s2) = do { let s = union s1 s2
23                                ; o1 := s
24                                ; o2 := s }
```

Table 5.6: Gossiping Girls

The top part of Table 5.6 gives a solution where only two girls can call each other. For brevity, we omit the (obvious) definitions of some primitive functions (For instance, boolean check `notSubset` and set operation `union`). The guard `notSubset`

`s1 s2` holds iff `s1` or `s2` is not a subset of the other. Thus, each call leads to the exchange of a new secret which guarantees that we eventually reach a state where all girls know all secrets.

Let's make the example more interesting by allowing girls who know all secrets (checked via the guard `sALL_SECRETS == s1`) to have simultaneous calls with girls who don't know all secrets yet. This 'multi-girl call' variation makes use of propagation. Girls who know all secrets are propagated and thus can be shared among other girls who don't know all secrets yet. The second Join-Pattern is copied from above but we impose the stronger condition that both parties don't know all secrets yet.

Top to Bottom Join-Pattern Execution Order Our implementation always tries Join-Patterns in top to bottom order as specified in the program text. Figure

```
1  event Get(Sync (Maybe Int))
2  event Put(Async Int)
3
4  Get(x) & Put(y) = (x := (Just y)) -- j1
5  Get(x) = (x := Nothing)           -- j2
```

Table 5.7: Concurrent Optional Get

5.7 illustrates an example which is only possible if we match `Get` join process calls to the Join-Patterns in top-to-bottom order. Basically, Join-Pattern $j1$ will be matched if a `Put(y)` exists, during which we return *just* the value stored in the `Put`. We will only match a `Get` join process call to $j2$ if no `Put(y)` calls are in the message store, during which we return `Nothing`. Note that if we allow join process calls to match Join-Patterns in any arbitrary order, we might have `Get` calls returning `Nothing` even if there exist `Put` messages[9].

We consider another example. The programmer can also exploit this execution strategy to make a stack concurrent. Commonly, a stack is a strictly sequential data

[9] As a note, our $\| \mathcal{G}$ semantics (Chapter 3) allows such arbitrary ordered rule matching, but our implementation picks strictly top-to-bottom ordering

structure. Each `Pop` or `Push` accesses the top of the stack, so in case of concurrent accesses we expect that they will be serialized. However, we argue that concurrent pop and push operations can immediately consume each other without having to go via the stack.

```
1  event Pop(Sync Int)
2  event Push(Async Int)
3  event Stack(Async [Int])
4
5  Push(x) & Stack(xs) = Stack(x:xs)
6  Pop(y) & Stack(x:xs) = do { y:= x
7                            ; Stack(xs) }
8
9  Push(x) & Pop(y) = (y:= x)
```

Table 5.8: Concurrent Stack

Table 5.8 shows an encoding of this idea in our system. The first two Join-Patterns are the standard pop and push operations. In cases where contention is high, we can include the third Join-Pattern that applies if a push can be matched against a pop. Thus, we can execute pops and pushes concurrently. Execute the Join-Patterns in top-down order ensures that the actual stack operations are always tried before our concurrent `Push(x) & Pop(y)` Join-Pattern is attempted. Our experiments in Section 5.5 show that the concurrent stack version scales better with the number of processor cores.

More on propagation in Join Patterns We consider another example which demonstrate the usefulness of propagation in Join-Patterns. While propagated join patterns promote better concurrency behaviour, because they can be shared among concurrently executing threads, propagation can also be used to model exhaustive processing of events.

Suppose we have multiple requests from clients which submit (asynchronous) their requests via join processes (events). We process these requests via some background jobs, that is, asynchronous events. Table 5.9 gives a possible encoding using

```
1  event Apply()
2  event Request(Async Client_Id)
3
4  -- variant 1
5  Apply()  Request(cid) = do { ...
6                             -- eg send client data
7                             -- to the server }
8
9  -- variant 2
10 Apply() & Request(cid) = do { ...
11                            ; Apply() }
12
13 Apply() = return ()
```

Table 5.9: Iteration via Propagation

join patterns with propagation (variant 1). The propagated event `Apply()` in the first join pattern guarantees that we exhaustively process requests. We could of course avoid propagation (see variant 2). However, the propagation variant is more efficient than first simplifying (removing) the call followed by invocation of the same call again.

We can even invoke multiple `Apply()` calls to concurrently process requests. The atomic rewriting semantics of join patterns guarantees that each request can only be processed once.

To terminate processing in case there are no more requests (instead of suspending), we can exploit the top to bottom execution order of join patterns in our implementation. Under top to bottom execution, the second Join-Pattern will be matched and triggered once we run out of requests.

5.4 A Goal-Based Execution Model for Join-Patterns

5.4.1 Overview of Goal-Based Execution

We consider the issue of how to execute Join-Patterns with guards and propagation efficiently. The idea is to treat join processes calls as goals to be executed the same

way as CHR goals are executed in the CHR $\| \mathcal{G}$ semantics (Chapter 3), only we will be matching active join processes to Join-Patterns and looking for partner processes in the message store. The high-level level structure of the execution algorithm is as follows:

For each active join processes call j, we match j to each Join-Pattern, `h1 & ... & hp \hm & ... & hn when g = body`. After which, for each Join-Pattern head `hi` that j matches with, we perform the following steps:

1. Search (in the message store) for matching copies of the remaining Join-Pattern heads, i.e. `h1,...,hp,hm,...,hn` − `hi`.

2. If the guard `g` holds under the matching substitution:

   (a) Atomically check all `h1,...,hp,hm,...,hn` are in the message store, and delete from the message store the simplified heads (`hm`,...,`hn`).

   (b) Execute Join-Pattern body `body` under the matching substitution. If active process j is not deleted, proceed with the next step, otherwise we are done.

3. Otherwise, we try the next match Join-Pattern head match `hi`, or try to match j with heads of another Join-Pattern. If all are tried, we are done.

This is essentially the same as the execution strategy we have detailed in Section 3.2. Our guards are assumed to be side-effect free. Hence, the guard check can be performed outside the atomically statement.

5.4.2 Goal Execution Example

We illustrate in detail goal-based execution of Join-Patterns via an example. In particular, we focus on the interplay between join process call (goal) and program execution. For brevity, we assume that the search for matching elements of a Join-

```
1  Item(k,x) & Set(k,y) = Item(k,y) -- j1
2  Item(k,x) \ Get(k,y) = y:= x     -- j2
3
4  main = do Item(k,1)
5            forkIO (do Set(k,2)
6                       x <- newSync
7                       Get(k,x)
8                       v1 <- readSync x)
9            forkIO (do Set(k,3)
10                      y <- newSync
11                      Get(k,y)
12                      v2 <- readSync y)
```

Table 5.10: Goal Execution Example

Pattern happens instantaneously[10]. Figure 5.5 shows the individual execution steps of the concurrent dictionary, shown again in Table 5.10. We will refer to *goal threads* as the computation threads that execute join process goal matching, while *program threads* as the computation threads that execute actual program codes, which includes the Join-Pattern bodies. Program and goal threads run concurrently. We consider a specific interleaving execution. On the left we show the evolution of the message store. The solid arrows refer to program threads whereas the dotted arrows refer to goal threads. Goals are written in italics to distinguish them from program text.

Execution of `main` proceeds as follows:

1. Execution of `Item(k,1)` activates the goal $Item(k, 1)$. This goal thread immediately terminates with no effect because there are no partners yet to fire a Join-Pattern.

2. Next, we spawn off two program threads. Execution of their first statement yields two goal threads $Set(k, 2)$ and $Set(k, 3)$. Goal $Set(k, 2)$ in combination with partner `Item(k,1)` triggers the Join-Pattern $j1$. The program thread resulting from execution of the body is labeled with the events which triggered

[10]In practice, multiset matching is a computationally intensive operation and would obviously not be instantaneous.

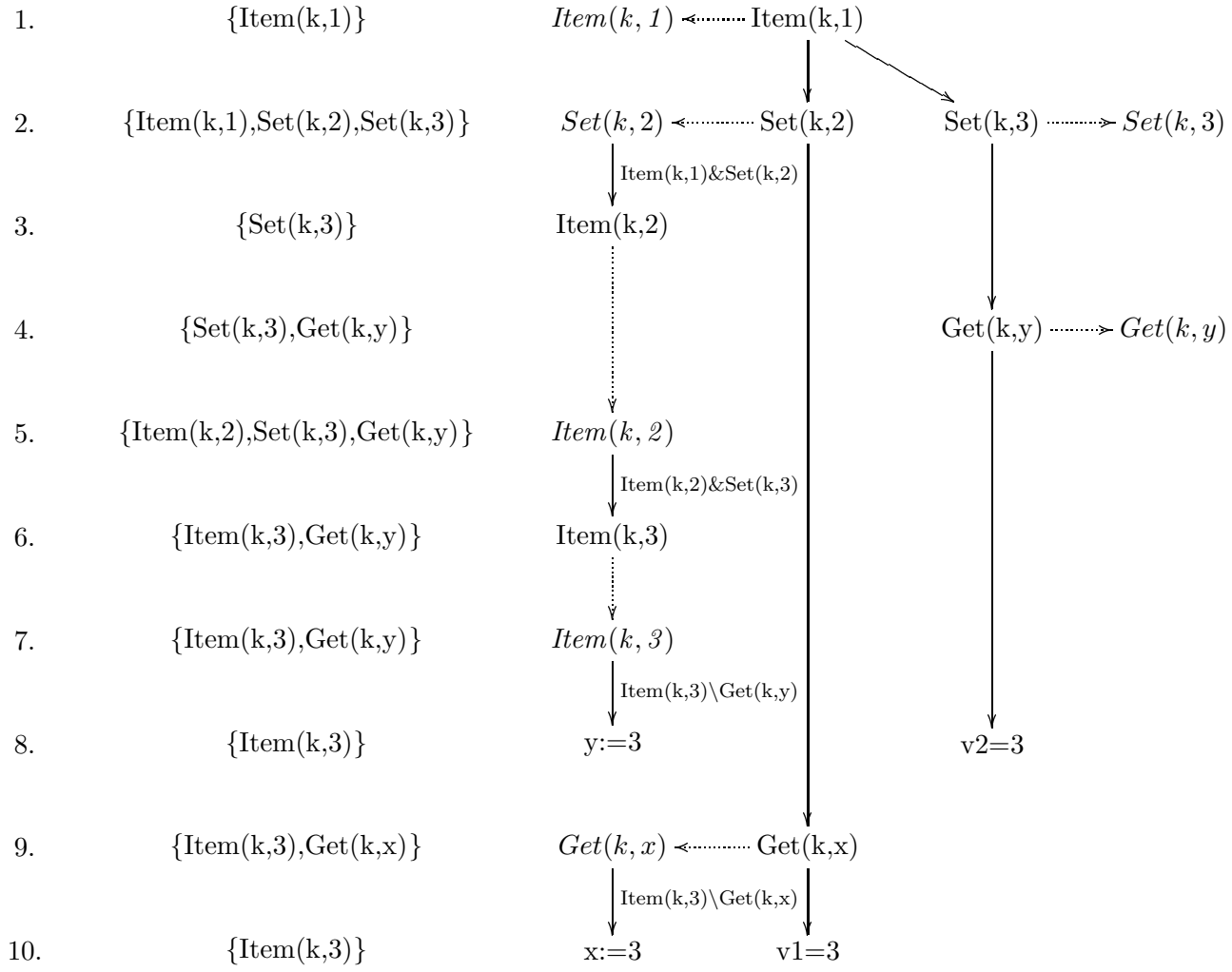

Figure 5.5: Goal and Program Execution Steps

---

the join pattern. Here, `Item(k,1)&Set(k,3)`. Goal $Set(k, 3)$ terminates with no effect.

3. We show an intermediate execution step. Goal $Set(k, 2)$ has been removed from the store because of the second step. Execution of `Item(k,2)` yields a new goal thread $Item(k, 2)$. This thread will only become active in step 5.

4. The second program thread advances. We omit `newSync` for brevity. The goal thread $Get(k, y)$ immediately terminates because of missing partners.

5. The goal thread $Item(k, 2)$ that resulted from the execution of the Join-Pattern body in step 2 becomes active. The partner `Set(k,3)` is selected which leads to the firing of Join-Pattern $j1$.

m	::=	$proc(v)$	Event
p, s	::=	$proc(x)$	Pattern
e	::=	$() \mid proc(v) \mid proc(x)$	Expressions
	$\mid$	$e; e \mid fork(e)$	
ϕ	::=	$[x_1 \mapsto v_1, ..., x_k \mapsto v_k]$	Substitution
g	::=	$True \mid False \mid x > y \mid ...$	Boolean guard
jp	::=	$p_1 \& ... \& p_k \backslash s_1 \& ... \& s_l$	Join pattern
jd	::=	$jp\ when\ g = e$	Join definition
$\mathcal{P}$	::=	$\{e_1, ..., e_n\}$	Program threads
$\mathcal{G}$	::=	$\{m_1, ..., m_k\}$	Goal threads
$\mathcal{S}$	::=	$\{m_1, ..., m_k\}$	Store

Figure 5.6: Syntax and Notations

6. Execution of the right-hand side yields the goal thread $Item(k, 3)$.

7. This goal then unblocks the second program thread by triggering Join-Pattern $j2$. After firing, the goal thread $Item(k, 2)$ is still active because it belongs to a propagated join pattern part. However, no further partner are available. Hence, the goal terminates with no effect.

8. Execution of the right-hand side `y:=3` unblocks `v2 <- readSync y`.

9. The first program thread reaches `Get(k,x)`. The goal thread $Get(k, x)$ fires the second join pattern.

10. We show the effect of the unblocked `v1<-readSync x` after execution of `x:=3`.

5.4.3 Join-Pattern Goal-Based Semantics

Figure 5.6 shows the syntax and notations we shall be using. In Figure 5.7, we formalize the goal-based execution scheme for a join pattern language extended with guards and propagation. Pure propagation is not supported (i.e. all Join-Patterns must have an least one simplified head). For brevity, we ignore synchronous arguments and only consider a simple expression language with events, sequencing and forking of new program threads. We assume that () terminates an expression.

$$\text{(Simp)}\quad \frac{\begin{array}{c} p_1\&...\&p_k \backslash s_1\&...\&s_l \text{ when } g = e \quad m \in \mathcal{G} \quad \mathcal{G}' = \mathcal{G} - \{m\} \\ \phi(p_1), ..., \phi(p_k), \phi(s_1), ..., \phi(s_l) \in S \quad \phi(g) \text{ is true and } \phi(s_i) = m \\ \text{for some } i \in \{1, ..., l\} \text{ and some substitution } \phi \\ \mathcal{P}' = \mathcal{P} \uplus \{\phi(e)\} \quad \mathcal{S}' = \mathcal{S} - \{\phi(s_1), ..., \phi(s_l)\} \end{array}}{(\mathcal{P}, \mathcal{G}, \mathcal{S}) \rightarrow (\mathcal{P}', \mathcal{G}', \mathcal{S}')}$$

$$\text{(Prop)}\quad \frac{\begin{array}{c} p_1\&...\&p_k \backslash s_1\&...\&s_l \text{ when } g = e \quad m \in \mathcal{G} \\ \phi(p_1), ..., \phi(p_k), \phi(s_1), ..., \phi(s_l) \in S \quad \phi(g) \text{ is true and } \phi(p_i) = m \\ \text{for some } i \in \{1, ..., k\} \text{ and some substitution } \phi \\ \mathcal{P}' = \mathcal{P} \uplus \{\phi(e)\} \quad \mathcal{S}' = \mathcal{S} - \{\phi(s_1), ..., \phi(s_l)\} \end{array}}{(\mathcal{P}, \mathcal{G}, \mathcal{S}) \rightarrow (\mathcal{P}', \mathcal{G}, \mathcal{S}')}$$

$$\text{(Drop)}\quad \frac{\mathcal{G} = \mathcal{G}' \uplus \{m\}}{(\mathcal{P}, \mathcal{G}, \mathcal{S}) \rightarrow (\mathcal{P}, \mathcal{G}', \mathcal{S})}$$

$$\text{(Evt)}\quad \frac{\mathcal{P} = \mathcal{P}' \uplus \{proc(v); e\} \quad \mathcal{G}' = \mathcal{G} \uplus \{proc(v)\} \quad \mathcal{S}' = \mathcal{S} \uplus \{proc(v)\}}{(\mathcal{P}, \mathcal{G}, \mathcal{S}) \rightarrow (\mathcal{P}' \uplus \{e\}, \mathcal{G}', \mathcal{S}')}$$

$$\text{(Fork)}\quad \frac{\mathcal{P} = \mathcal{P}' \uplus \{fork(e_1); e_2\} \quad \mathcal{P}'' = \mathcal{P}' \uplus \{e_1, e_2\}}{(\mathcal{P}, \mathcal{G}, \mathcal{S}) \rightarrow (\mathcal{P}'', \mathcal{G}, \mathcal{S})}$$

$$\text{(Unit)}\quad \frac{\mathcal{P} = \mathcal{P}' \uplus \{()\}}{(\mathcal{P}, \mathcal{G}, \mathcal{S}) \rightarrow (\mathcal{P}', \mathcal{G}, \mathcal{S})}$$

Figure 5.7: Goal-Based Operational Semantics

We describe the meaning of programs in terms of a small-step semantics among configurations $(\mathcal{P}, \mathcal{G}, \mathcal{S})$ where $\mathcal{P}$ denotes the set of program threads, $\mathcal{G}$ denotes the set of goal threads and $\mathcal{S}$ denotes the set of stored events. Our sets are multisets and we write $\in$ to denote member-ship test, $\uplus$ to denote multiset union and $-$ to denote multiset difference.

Evaluation of an expression e starts in the initial configuration $(\{e\}, \{\}, \{\})$. Reduction rules simulate an interleaved execution of program and goal threads. They are applied in top to bottom order.

Rules (Simp) and (Prop) cover application of a join definition based on a given goal. For both cases, simplified events in a join pattern are removed from the store

and the join body becomes a new program thread. The difference is that in rule (Simp) the goal is simplified, i.e. the goal is removed from $\mathcal{G}$. Rule (Prop) propagates the goal, i.e. the goal store remains unchanged. Propagated goals can lead to further join definition execution. Hence, they are not removed.

In case goal m could not trigger any join pattern, we drop the goal (thread). See rule (Drop) which only applies if rules (Prop) or (Simp) are not applicable. Exhaustive firings of join patterns when dropping goals is still guaranteed. If a complete match with the join pattern exists, the (previously) missing partners will act as goals and trigger the join pattern.

The (implementation) advantage of dropping goals is that we won't waste (system) resources. For example, consider the stack example from an earlier section. We assume that a large number of concurrent `Pop` operations, and no other operation such as `Push`, tries to access the stack. Hence, the following join definition is only relevant.

```
Pop(y) & Stack(x:xs) = do { y:= x
                          ; stack(xs) }
```

Each `Pop` operation becomes initially a goal (active event) and each goal runs in its own thread and tries to fire the join pattern `Pop(y) & Stack(x:xs)`. But only one `Pop` goal at a time can access the stack. Hence, many goals will fail to fire the above join pattern. Instead of wastefully retrying (leading possibly to another failure), we simply drop each failed `Pop` goal. Once the right-hand side of the above join definition is executed the `Stack` call becomes active (i.e. acts as a goal) and then can select its partner among the 'failed' `Pop`s in the store.

Rules (Evt), (Fork) and (Unit) describe single-step execution of a random program thread. In each single step, we perform one of the following. An (asynchronous) event is stored and a new goal is generated (Evt). A new program thread is forked (Fork). A program thread is terminated (Unit).

5.4.4 Implementation Issues

We examine some issues of our Join-Patterns with guards and propagation implementation.

Asynchronous and Synchronous Join Process Arguments Join Processes with only asynchronous arguments are non-blocking. After execution of the event (join process), evaluation continues as normal. Technically, calls with synchronous arguments are also non-blocking. However, we can impose a blocking mechanism by waiting until the synchronous argument is bound.

For example, consider a variant of the earlier example from Table 5.1. The commented out parts will be considered later.

```
event put(Async Int)
event get(Sync Int)


put(x) & get(y) = y:= x
                  -- y:= 2  -- L1


prog = do { put(3)
          ; y <- newSync
          ; get(y)
          ; v <- readSync y
          -- v2 <- readSync y -- L2
          ; return v }
```

We implement synchronized variables via Haskell's STM variables. We basically use them as a one-place buffer which can either be full or empty. Function `readSync` creates an empty buffer which can be filled via the statement `:=` (assignment). Function `readSync` blocks until the buffer is full.

Our current (library-based) implementation, does not impose any restriction on the use of `:=` and `readSync`. This possible leads to 'bogus' code. Suppose we uncomment location `L2`. The read at location `L2` will be blocked forever unless we also uncomment location `L1`. The fact that we use a synchronization variable twice is a questionable feature. We could reject such 'bogus' uses by imposing some (type) conditions. For instance, we could guarantee the 'one-time' reading and writing from/to a synchronization variable via a linear type system. Enforcing such conditions is left for future work.

Out of Order Join Pattern Execution Join patterns are tried in top to bottom order as specified by the program text. But it is possible that join patterns are executed out of order in case of contention among concurrent threads. Consider a variant of an earlier example, the concurrent dictionary.

```
Item(k,x) & Set(k,y) = Item(k,y)    -- (Set)
Set(k,y) = Item(k,y)                -- (Default)
```

The last (default) pattern is only meant to be applied in case there is no item of key `k` in the store yet. We start in the initial configuration

$$(\{\}\ ,\ \{\ \texttt{Set(k,1)},\texttt{Set(k,2)}\ \},\ \{\ \texttt{Item(k,3)},\texttt{Set(k,1)},\texttt{Set(k,2)}\ \})$$

The program thread is empty and there are two goal threads `Set(k,1)` and `Set(k,2)`. Here is a possible reduction sequence where we underline each goal.

$$\begin{array}{ll} & (\{\},\{\underline{\texttt{Set(k,1)}},\texttt{Set(k,2)}\},\{\texttt{Item(k,3)},\texttt{Set(k,1)},\texttt{Set(k,2)}\}) \\ \rightarrow_{Simp} & (\{\texttt{Item(k,1)}\},\{\underline{\texttt{Set(k,2)}}\},\{\texttt{Set(k,2)}\}) \\ \rightarrow_{Simp} & (\{\texttt{Item(k,1)},\texttt{Item(k,2)}\},\{\},\{\}) \\ \rightarrow_{2*Evt} & (\{\},\{\texttt{Item(k,1)},\texttt{Item(k,2)}\},\{\texttt{Item(k,1)},\texttt{Item(k,2)}\}) \end{array}$$

Goal `Set(k,1)` executes rule (Simp) on the first join pattern. The program resulting thread `Item(k,1)` will be activated as a goal only later. Therefore, goal `Set(k,2)`

executes the last join pattern which yields another program thread `Item(k,2)`. Finally, both program threads yield new goal threads.

This result is somewhat unexpected. The programmer most likely expected the the following reduction.

$$
\begin{array}{ll}
 & (\{\}, \{\texttt{Set(k,1)}, \texttt{Set(k,2)}\}, \{\texttt{Item(k,3)}, \texttt{Set(k,1)}, \underline{\texttt{Set(k,2)}}\}) \\
\rightarrow_{Simp} & (\{\texttt{Item(k,2)}\}, \{\texttt{Set(k,1)}\}, \{\texttt{Set(k,1)}\}) \\
\rightarrow_{Call} & (\{\}, \{\texttt{Set(k,1)}, \texttt{Item(k,2)}\}, \{\underline{\texttt{Set(k,1)}}, \texttt{Item(k,2)}\}) \\
\rightarrow_{Simp} & (\{\texttt{Item(k,1)}\}, \{\texttt{Item(k,2)}\}, \{\}) \\
\rightarrow_{Call} & (\{\}, \{\texttt{Item(k,2)}, \texttt{Item(k,1)}\}, \{\texttt{Item(k,1)}\})
\end{array}
$$

Execution of goal `Set(k,2)` is immediately followed by the activation of the resulting program thread which then yields goal `Item(k,2)` (which is also stored). Then, goal `Set(k,1)` fires the first join pattern. We omit the symmetric case where goal `Set(k,1)` executes before `Set(k,2)`.

We believe the out of order execution of join patterns in case of contention is acceptable. To guarantee a strict top to bottom order execution of join pattern (i.e. ruling out the first reduction sequence) some significant implementation (synchronization) effort is required. For example, would need to arbitrarily delay goal execution, or execute the left-hand side and right-hand side of a join pattern definition atomically. Both choices are not acceptable in our opinion.

5.5 Experiment Results: Join-Patterns with Guards

We conducted experiments of our parallel join implementation with a range of examples which uses Join-Pattern with guards and/or propagation. These set of examples represent a collection of common parallel programming problems widely found in the literature of parallel programming, each of which involves various forms of synchronization between parallel operations that as a whole are governed certain

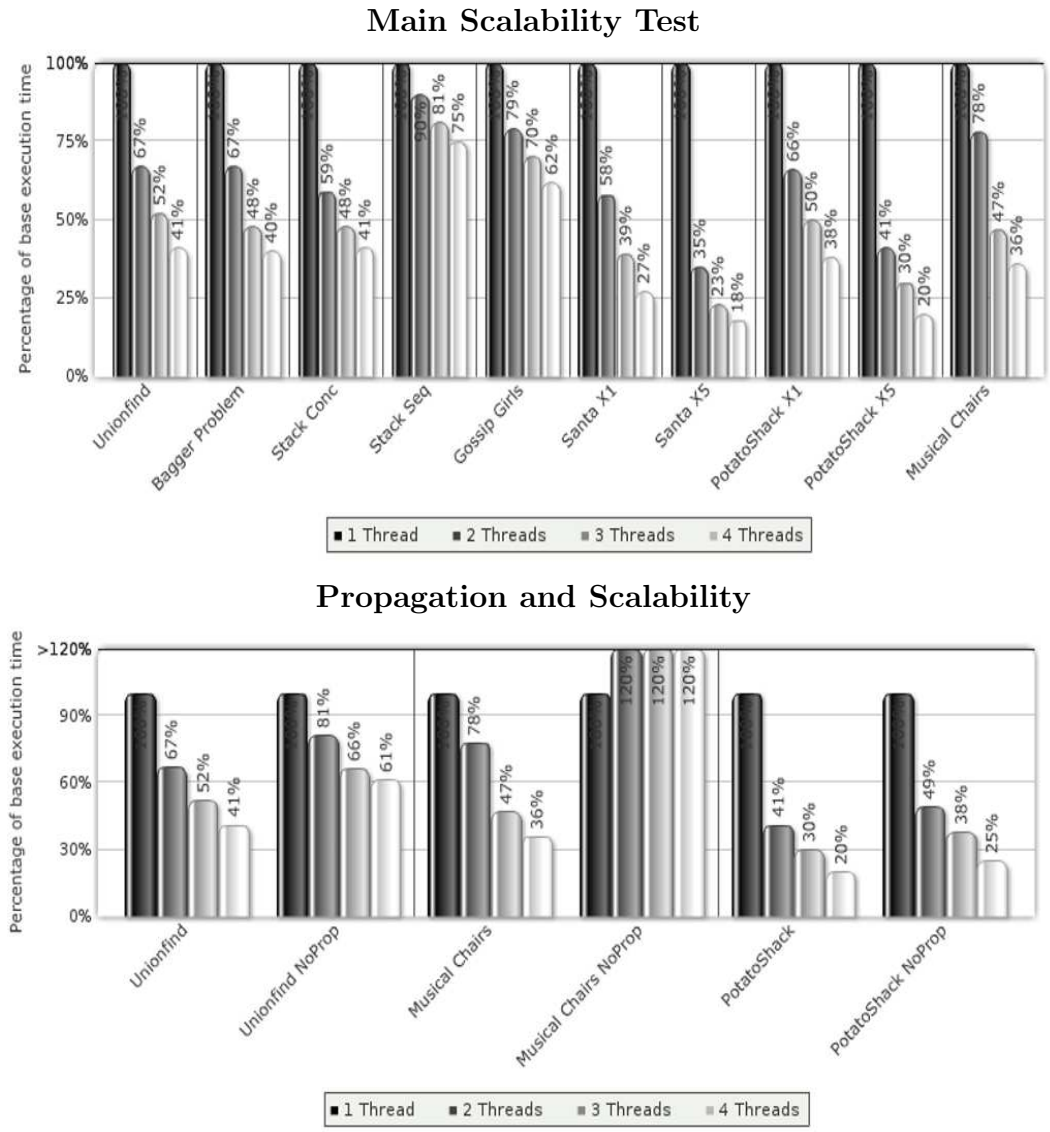

Figure 5.8: Experiment Results

---

rules specified in the form of Join-Patterns. Experiments are ran on a quad-core Intel Xeon 1.86 GHz with 1GB memory, we were using Glasgow Haskell Compiler (GHC) 6.10.1 . Results shown are the relative performance of running 2-4 cores against running on a single core, and are averaged over several test runs. We briefly describe each join program in this benchmark, while details and implementations can be found at http://code.haskell.org/parallel-join

- **UnionFind** Adopted from [20]. Parallelize union find implementations a concurrent data structure which maintains union relationship among disjoint sets. In experiments, we test an instance where 8 parallel union operations attempts to unite 9 disjoint sets of size 200.

- **BaggerProblem** The bagger problem simulates a packing problem where n bags are packed with objects of three sizes and larger objects cannot be stacked on smaller ones. In experiments, we test an instance where 1000 items of various sizes are packed into 40 bags.

- **StackConc/StackSeq** Two Implementation of a stack with our join patterns. StackConc is shown in Table 5.8, while StackSeq is a variant with the last join pattern removed In experiments, we test an instance of 500 parallel push and pop operations.

- **GossipGirls** Shown in Table 5.6, the gossiping girls problem simulates concurrent processes (girls) communicating and exchanging information, until all girls have the full set of information. In experiments, we test an instance where 50 girls start with mutable disjoint sets of secrets to tell.

- **SantaX**n Adopted from [57], the Santa Claus problem is an exercise of concurrency, where Santa must synchronize with either 3 of 10 elves to discuss toy designs, or all 9 reindeers to deliver toys, with reindeers having higher priority. In experiments, we test an instance where Santa must make 80 deliveries or toy discussions (SantaX1). We also investigated a variant where we have 5 Santa Claus' (SantaX5).

- **PotatoShackX**n A simulation of a fast-food restaurant serving fries or baked potatoes. The problem consist of concurrent processes, running either customer, cook or kitchen helper routines which must communicate and synchronize with each other. In experiments, we test an instance where 24 customers

are served by 1 cook and 1 kitchen helper (PotatoShackX1). We also investigated a variant where we have 5 cooks and 5 helpers (PotatoShackX5).

- **MusicalChairs** A simulation of the game of musical chairs. The game starts with $n + 1$ players and n chairs and continues until only one player is left. In experiments, we test an instance where n is 30.

Figure 5.8 show our main experimental results. **Main Scalability Tests** illustrates the relative speed up in performances, and the scalability of each program up to 4 processors. As shown, the test programs experience consistent speed up in performance as we increase number of processors. In some cases (SantaX5, PotatoShackX5), we see significant super-linear speed ups. Experiments SantaX1 and PotatoShackX1 show that super-linear speed ups are largely attributed to running 'output' processes in parallel (processes that produces the actual outputs measured, eg. Santa Claus and the cooks). In these experiments (SantaX1 and PotatoShackX1) we only have one such 'output' process, thus we see a significant drop in such super-linear speed up behavior.

Discussed in Section 5.4.4, StackConc shows high scalability as we allow pairing of parallel push and pop operations. For StackSeq, since we disallow this (remove the last join pattern of Table 5.8), all push and pop operations must synchronize on a single 'Stack' event, hence we do not get much speed up.

We also investigated on the empirical repercussions of not using propagation where possible. In **Propagation & Scalability**, we see the programs Union Find, Musical Chairs and Potato Shack along with 3 respective variants which do not use propagation (No-Prop). These three examples are chosen because they heavily relied on propagated patterns. As seen in the results, the no propagation variants scale worst in general, and for the case of Musical Chairs, propagation is critical for scalability.

To summarize, our experiments show that programs implemented in our join pattern implementation scale relatively well and propagation is a useful feature for parallel programming.

Chapter 6

Related Works

6.1 Existing CHR Operational Semantics and Optimizations

While the abstract CHR semantics [19] formally defines the behaviour of the CHR language, existing implementations are derived from refined operational semantics [9, 48, 33] which provide more precise formalism of CHR rule execution strategies. The refined CHR operational semantics [9] describes the compilation and execution of CHR programs in terms of the execution of CHR goal constraints which trigger rule instances. Our concurrent CHR operational semantics (Section 3.3) essentially generalizes from this approach, extending it's semantics to allow concurrent execution of CHR goals. [33] introduces Constraint Handling Rules with user-definable rule priorities (CHR^{rp}) and presents a CHR goal-based operational semantics which provides user specified control over the ordering of CHR goal execution. In [58], the authors explored an extension of the CHR semantics with negated rule-heads. This allows the user to specify in CHR rule-heads negated constraints, which triggers rule firing in the absence of the specified constraints (from the store). Recent works in [12] revisits the formal definitions CHR state equivalence and provides a more simplified formulation of the CHR operational semantics.

Optimized compilation and analysis of CHR programs have been well studied over the years. [9, 27, 48] highlights a range of standard optimizations and analysis techniques for CHR goal-based compilations, from constraint indexing, late storage to optimal join-ordering. In Section 3.5.4, we have identified the optimizations which are still applicable in the parallel execution context.

Our work presented here complements our earlier works on the parallel CHR execution on a shared memory multi-core architecture [34, 53]. Prior to our previous work, there has been no research into the implementation of parallel execution of CHR rewritings. Studies in [21] specifically investigated into parallelizing the Union-Find problem in CHR via confluence analysis, while we provide empirical evidence here (Section 4.6) that proves that the parallelized formulation of Union find in CHR scales with multicore execution.

6.2 From Sequential Execution to Concurrent Execution

Works in [44, 43] introduces an extension to Haskell GHC in the form of a primitive `unreadTVar` that allows the programmer to explicitly remove memory locations from STM transactional logs. Similar to *atomic rule-head verification* in our context, the motivation of providing this new primitive is to allow programmers to "trim" STM transactions and to remove false data dependencies in a general way. As such, this proposed extension of Haskell GHC can provide the basis for an alternative implementation of our parallel CHR system. Experiment results in [44], includes scalability analysis of our concurrent CHR implementation and provides third party confirmation of the scalability of our approach.

Our works in [54] compares the performance of Haskell GHC's various concurrency primitives (`MVar`, `STM` and compare-and-swap `IORef`). It has provided the empirical evidence that helped us conclude that using a mix between compare-and-

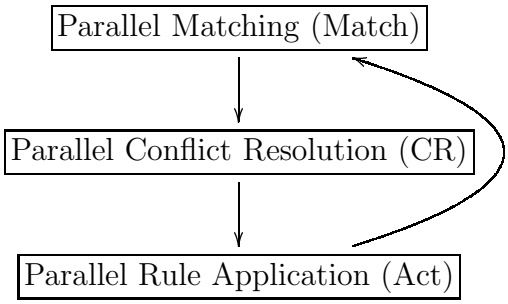

Figure 6.1: Parallel Production Rule Execution Cycles

swap `IORef` for list traversal and physical delinking, while `STM` for multi-set logical deletes provides a highly competitive implementation of CHR multi-set rewriting.

6.3 Parallel Production Rule Systems

Parallel execution models of forward chaining production rule based languages (eg. OPS5 [16]) have been widely studied in the context of production rule systems. A production rule system is defined by a set of multi-headed production rules (analogous to CHR rules) and a set of assertions (analogous to the CHR store). Production rule systems are richer than the CHR language, consisting of user definable execution strategies and negated rule heads. This makes parallelizing production rule execution extremely difficult, because rule application is not monotonic (rules may not be applied in a larger context). As such, many previous works in parallel production rule systems focuses on efficient means of maintaining correctness of parallel rule execution (eg. data dependency analysis [28], sequential to parallel program transformation [22]), with respect to such user specified execution strategies. These works can be classified under two approaches, namely synchronous and asynchronous parallel production systems.

For synchronous parallel production systems (eg. UMPOPS [24]), multiple processors/threads run in parallel. They are synchronized by execution cycles of the production systems. Figure 6.1 illustrates the production cycle of a typical produc-

tion rule system, consisting of three execution phases. In the (Match) phase, all rule matches are computed. Conflict resolution (CR) involves filtering out matches that do not conform to the user specified rule execution strategy, while (Act) applies the rule matches that remains (known as the eligible set) after the (CR) phase. By synchronizing parallel rule execution in production cycles, a larger class of user specified execution strategies can be supported since execution is staged.

Matching in synchronous production rule systems often use some variant of the RETE network [15]. RETE is a incremental matching algorithm where matching is done eagerly (data driven) in that each newly added assertion (constraint in CHR context) triggers computation of all it's possible matches to rule heads. Figure 6.2 illustrates a RETE network (acyclic graph), described in CHR context. Root node is the entrance where new constraints are added. Intermediate nodes with single output edges are known as alpha nodes. Intermediate nodes with two output edges are beta nodes, representing joins between alpha nodes. Each alpha node is associated with a set of constraint matching it's pattern, while a beta node is associated with a set of partial/complete matches. Parallel implementation of RETE [37] allows distinct parts of the network to be computed in parallel.

The most distinct characteristic of RETE is that partial matches are computed and stored. This and the eager nature of RETE matching is suitable for production rule systems as assertions (constraints) are propagated (not deleted) by default. Hence computing all matches rarely result to redundancy. Traditional CHR systems do not advocate this eager matching scheme because doing so results to many redundancies, due to overlapping simplified matching heads. Eager matching algorithms is also proved in [8] to have a larger asymtoptic worst-case space complexity than lazy matching algorithms.

In [40], the matching algorithm TREAT is proposed. TREAT is similar to RETE, except it does not store partial matches. TREAT performs better than RETE if the overhead of maintaining and storing partial matches outweighs that of re-computing

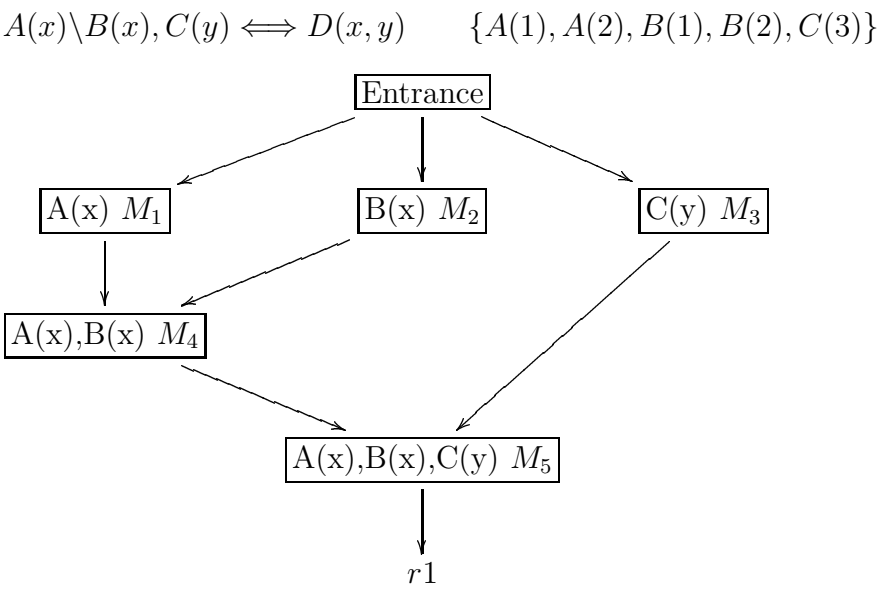

Figure 6.2: Example of a RETE network, in CHR context

---

partial matches.

Asynchronous parallel production rule systems (eg. Swarm [22], CREL [41]) introduces parallel rule execution via asynchronously running processor/threads. In such systems, rules can fire asynchronously (not synchronized by production cycles), hence enforcing execution strategies is more difficult and limited. Similar to implementations of goal based CHR semantics rule matching is such systems often use a variant of the LEAPS [8] lazy matching algorithm.

Staging executions in synchronous parallel production rule systems allows for flexibility in imposing execution strategies, but at a cost. In [42], synchronous execution of UMPOPS production rule system is shown to be less efficient than asynchronous execution. Hence it is clear that synchronous systems will only be necessary if we wish to impose some form of execution strategies on top of the abstract CHR semantics (eg. rule-priority, refined operational semantics). We are

interested in concurrent CHR semantics on the abstract CHR semantics. It's non-determinism and monotonicity property provides us with the flexibility to avoid executing threads in strict staging cycles. Thus our approach is very similar to asynchronous parallel production rule systems.

6.4 Join Pattern Guard Extensions

Join-calculus have been widely studied in various context. There are a number of existing implementations of join-pattern language extensions based on mainstream programming languages. In [5], the authors briefly mentioned a prototype extension of Polyphonic *C#* with guarded join-patterns. This approach does not scale well as it uses a simple and naively way of triggering guarded join-patterns: sequential and exhaustive combinatorial search for all possible join-pattern matchings. Library extensions that introduces join-patterns have also been studied. [46] introduces join-patterns to *C#*, while [49] to Haskell, both by means of library extensions. Library extensions are extremely versatile and convinent for prototyping (no external compiler needed), but are normally not as efficient as highly optimized language extensions. Mentioned in [49], this join-pattern library extension to Haskell supports a limited class of guarded join-patterns: localized guard constraints on a single join-pattern head (message). Unfortunately, this is highly restrictive and does not allow selectivity of combinations between join pattern heads (messages).

Other extensions of join-patterns have been explored in [35, 36, 36, 25]. [35, 36] introduces an extension of join-patterns (in JoCaml) with ML style pattern matching, via a source (join-patterns with pattern matching) to source (basic join-pattern) compilation scheme. Even though not mentioned in the paper, we believe that the compilation scheme described, supports a limited class of guarded join-patterns similar to [49], but also does not address efficient compilation of guarded join-patterns in general. [25] extends the language Scala with join-patterns via

extensible pattern matching facilities of Scala, while also attempting to integrate Erlang style actor programming into join-patterns.

Chapter 7

Conclusion And Future Works

7.1 Conclusion

Constraint Handling Rules (CHR) is a concurrent committed choice rule based programming language. It's semantics essentially involves multi-set rewriting over a multi-set of constraints. This computational model is highly concurrent as theoretically rewriting steps over non-overlapping multi-sets of constraints can execute concurrently.

In this thesis, we study the parallel execution of Constraint Handling Rules (CHR). Our work here can be classified into three main areas:

Concurrent Goal-based Semantics We introduced a concurrent goal based semantics ($\| \mathcal{G}$ semantics) for Constraint Handling Rules (CHR) (Chapter 3). This concurrent semantics describes concurrent execution of CHR goals and the execution of CHR rewritings in parallel. Specifically, we make the following contributions:

- We formally define the concurrent CHR goal-based semantics (Section 3.3).
- We identify the main issues which makes the formalism of the $\| \mathcal{G}$ semantics non-trivial (Section 3.4).

- We provide a formal correspondence results between the $\| \mathcal{G}$ semantics and the abstract CHR semantics (Section 3.5, A.1 and A.2)

Parallel Implementation of CHR Rewriting The $\| \mathcal{G}$ semantics provides the semantic foundation of our parallel CHR system (Chapter 4). Empirical results show that this implementation executes CHR multiset rewritings in parallel and scales with the number of CPU cores executing the parallel solver. Specifically, our contributions are as follows:

- We identify the main challenges (Section 4.4) to implement a parallel CHR rewrite system, which is practical and can exploit multicore architecture.
- We implement a parallel CHR system in Haskell GHC, based on our $\| \mathcal{G}$ semantics (Section 4.5).
- We provide in-depth experiment results (Section 4.6) to show that our parallel CHR system scales and that our listed optimizations for parallelism are crucial for scalability.

Join-Pattern with Guards and Propagation We introduce a unique execution model for Join-Patterns with Guards, based on our parallel CHR execution model. This provides a possible solution to the known problem of efficient execution of Join-Patterns with Guards. This represents a non-trivial application of our concurrent CHR goal-based semantics. Specifically, our contributions here are as follows:

- We introduce and motivate Join-Patterns with guards and propagation by means of an array of examples. (Section 5.3)
- We formally specify our concurrent goal-based execution model for Join-Pattern with Guards and propagation (Section 5.4)
- We implemented a prototype system in Haskell GHC and provide basic experiment results to illustrate the scalability of this approach (Section 5.5).

7.2 Future Works

There are several directions of future works which we wish to pursue. Similarly, we will list these works in the following three areas of study:

Concurrent Goal-based Semantics In [32], more explicit execution control strategies are explored for sequential goal-based CHR execution. For instance, Constraint Handling Rules with user definable priorities were studied and the authors provided strong motivation for that extension in the CHR language. Our works in parallel goal execution (Section 4.4.3) also encounters the need for such execution control strategies, thus it will be beneficial to investigate how this work can be adapted for the parallel CHR goal execution context.

In our works here, we make no attempt to define the correspondence with the refined CHR operational semantics [9], which describes sequential CHR goal-based execution. An interesting future work is to explore the possibility (and feasibility) of defining a parallel CHR execution model that has a correspondence with the refined CHR operational semantics. The main challenge of this work will be to identify the restrictions we need on concurrent goal-based execution and identify the conditions where concurrent goals are free to execute asynchronously.

Parallel Implementation of CHR Rewriting Works in [44] introduces a new primitive operation in the Haskell STM library, allowing the programmer to explicitly "trim" transactional logs of STM transactions. This can provide the basis of an alternative implementation to our current parallel CHR system, which uses atomic rule-head verification (Section 4.5.4), with the latter highly likely to be a simpler implementation. Empirical results will be another focus of this future works, to support scalability analysis of this approach.

While results in Section 4.7 do show that our prototype implementation is still far from industrial strength, using CHR as a declarative high-level concurrency ab-

straction allows the programmer to implicitly write scalable parallel applications, without the hassle of hand-coding complex concurrency synchronization routines. Yet this will not be without performance over-heads, hence one of our future works will be to reduce such over-heads to a more reasonable level where parallel multiset rewrite in CHR can provide programmers with high-level concurrency abstractions and still function competitively against handcoded parallel programs.

Works in [39] investigates into the theoretical scalability of the preflow-push algorithm implemented in CHR, while [20] does the same for the union find algorithm. An interesting course of future work will be to correlate such theoretical works with practical findings of our parallel CHR implementation. Such studies could possibly yield interesting insights on the behaviour of parallel programs implemented in CHR and aid us in improving the performance or usability of future implementations.

Our studies here are so far confined to the domain of symmetric shared memory processor (SMP) frameworks. A natural extension to our work will be to explore parallel CHR rewritings in the area of distributed programming. This will involve exploring new related issues of parallelism, which is not visible (or inconsequential) in shared memory architectures. For instance, when dealing with shared memory, we can practically assume that all execution threads have access to all constraints in the constraint store (a global view). In a distributed framework, CHR goal execution threads can be distributed through out a network consisting of multiple computation nodes and need not be homogeneous[1]. Similarly, we may have to account for the possibility of having the constraint store distributed between different computation nodes as well. The introduces a new problem which is to identify an optimal goal/store constraint distribution that maximize data proximity (relevant constraints are kept local to goal execution threads that match them frequently) and minimize data migration (stored constraints are transfered between nodes less frequently). Future works in this direction will likely to be related to distributed

[1]Meaning that goal execution threads in different nodes may be tasked to executed different matching routines. For instance, match distinct rule occurrences

rule-base systems [45] and distributed constraint solving [3].

Join-Patterns with Guards and Propagation Our work to introduce CHR parallel multiset rewritings into Join-Patterns is still relatively preliminary. In future, we intend to provide formal correspondence results between our concurrent goal-based Join-Pattern execution model, and Join-Calculus. Our current implementation of Join-Patterns with Guards and Propagation is still a prototype and more implementation work (e.g.. introducing CHR optimizations, refining the language design) is still required.

One interesting direction of exploration is to investigate if a composite compilation scheme can be derived. In other words, we compile Join-Patterns with guards and propagation into CHR goal-based occurrences, while standard Join-Patterns are compiled with the standard compilation techniques. Issues like termination or exhaustiveness of join process execution must be re-addressed in such a composite compilation scheme.

Bibliography

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[3] Farhad Arbab and Eric Monfroy. Coordination of heterogeneous distributed cooperative constraint solving. *SIGAPP Appl. Comput. Rev.*, 6(2):4–17, 1998.

[4] Maryam Bavarian and Verónica Dahl. Constraint based methods for biological sequence analysis. *J. UCS*, 12(11):1500–1520, 2006.

[5] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[6] G. Berry and G. Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM.

[7] Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *ASAMA '99: Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, page 22, Washington, DC, USA, 1999. IEEE Computer Society.

[8] B.J. Lofaso D.P. Miranker, D. Brant and D. Gadbois. On the performance of lazy matching in production systems. In *In proceedings of International Conference on Artificial Intelligence AAAI*, pages 685–692, 1990.

[9] G. J. Duck. *Compilation of Constraint Handling Rules.* PhD thesis, The University of Melbourne, 2005.

[10] Gregory J. Duck, Simon L. Peyton Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies. In *ESOP*, pages 49–63, 2004.

[11] Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *ICLP*, pages 90–104, 2004.

[12] H. Betz F. Raiser and T. Frühwirth. Equivalence of chr states revisited. *CHR '09: Proc. 6th Workshop on Constraint Handling Rules, Pasadena, California*, pages 33–48, 2009.

[13] F. Le Fessant and L. Maranget. Compiling join-patterns. In *HLCL '98: High-Level Concurrent Languages, volume 16(3) of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, Sept. 1998.*, 1998.

[14] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electr. Notes Theor. Comput. Sci.*, 16(3), 1998.

[15] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[16] Charles Forgy and John P. McDermott. Ops, a domain-independent production system language. In *IJCAI*, pages 933–939, 1977.

[17] C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332. Springer-Verlag, 2002.

[18] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM.

[19] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.

[20] T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence analysis. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 113–127. Springer-Verlag, 2005.

[21] Thom Frhwirth. Parallelizing union-find in constraint handling rules using confluence. In *Logic Programming: 21st International Conference, ICLP 2005, volume 3668 of Lecture Notes in Computer Science*, pages 113–127. Springer, 2005.

[22] Rose F. Gamble. Transforming rule-based programs: from the sequential to the parallel. In *IEA/AIE '90: Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 854–863, New York, NY, USA, 1990. ACM.

[23] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[24] Anoop Gupta, Charles Forgy, Dirk Kalp, Allen Newell, and Milind Tambe. Parallel ops5 on the encore multimax. In *ICPP (1)*, pages 71–280, 1988.

[25] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *COORDINATION*, pages 135–152, 2008.

[26] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[27] C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.

[28] T. Ishida. Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):11–17, 1991.

[29] G. Stewart Von Itzstein and Mark Jasiunas. On implementing high level concurrency in java. In *Asia-Pacific Computer Systems Architecture Conference*, volume 2823 of *LNCS*, pages 151–165. Springer-Verlag, 2003.

[30] K.u.leuven jchr system home page. http://dtai.cs.kuleuven.be/CHR/JCHR/.

[31] L. De Koninck, P.J. Stuckey, and G.J. Duck. Optimizing compilation of CHR with rule priorities. In *Proc. of FLOPS'08*, volume 4989 of *LNCS*, pages 32–47. Springer-Verlag, 2008.

[32] Leslie De Koninck. Execution control for chr. In *ICLP*, pages 479–483, 2009.

[33] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for chr. In *PPDP*, pages 25–36, 2007.

[34] E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory. In *Proc. of ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, pages 19–24, 2007.

[35] Qin Ma and Luc Maranget. Compiling pattern matching in join-patterns. In *CONCUR*, pages 417–431, 2004.

[36] Qin Ma and Luc Maranget. Algebraic pattern matching in join calculus. *Logical Methods in Computer Science*, 4(1), 2008.

[37] Milind Mahajan and V. K. Prasanna Kumar. Efficient parallel implementation of rete pattern matching. *Comput. Syst. Sci. Eng.*, 5(3):187–192, 1990.

[38] Louis Mandel and Luc Maranget. Programming in JoCaml (tool demonstration). In *Proc. of ESOP'08*, volume 4960 of *LNCS*, pages 108–111. Springer-Verlag, 2008.

[39] Marc Meister. Concurrency of the preflow-push algorithm in Constraint Handling Rules. In *CSCLP'07: Proc. 12th Intl. Workshop on Constraint Solving and Constraint Logic Programming*, pages 160–169, 2007.

[40] Daniel P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[41] Daniel P. Miranker, Chin Kuo, and James C. Browne. Parallelizing transformations for a concurrent rule execution language. Technical report, University of Texas at Austin, Austin, TX, USA, 1989.

[42] Daniel E. Neiman. Control issues in parallel rule-firing production systems. In *in Proceedings of National Conference on Artificial Intelligence*, pages 310–316, 1991.

[43] C. Perfumo, N. Sonmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 67–78, New York, NY, USA, 2008. ACM.

[44] C. Perfumo, N. Sonmez, O. S. Unsal, A. Cristal, M. Valero, and T. Harris. Dissecting transactional executions in Haskell. In *The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2007.

[45] William Perrizo, Joseph Rajkumar, and Prabhu Ram. Hydro: a heterogeneous distributed database system. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 32–39, New York, NY, USA, 1991. ACM.

[46] C. Russo. The Joins concurrency library. In *Proc. of PADL'07*, volume 4354 of *LNCS*, pages 260–274. Springer-Verlag, 2007.

[47] H. Meuss S. Abdennadher, T. Fruhwirth. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4, 1999.

[48] T. Schrijvers. Analyses, optimizations and extensions of Constraint Handling Rules: Ph.D. summary. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 435–436. Springer-Verlag, 2005.

[49] S. Singh. Higher-order combinators for join patterns using stm, 2006. Proc. of TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.

[50] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and continuation optimization for occurrence representations of chr. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 83–97. Springer-Verlag, 2005.

[51] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of constraint handling rules. *ACM Trans. Program. Lang. Syst.*, 31:8:1–8:42, February 2009.

[52] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero. Unreadtvar: Extending haskell software transactional memory for performance. In *in Eighth Symposium on Trends in Functional Programming (TFP 2007)*, 2007.

[53] M. Sulzmann and E. S. L. Lam. Parallel execution of multi-set constraint rewrite rules. In *Proc. of PPDP'08*, pages 20–31. ACM Press, 2008.

[54] M. Sulzmann, E. S. L. Lam, and S. Marlow. Comparing the performance of concurrent linked-list implementations in haskell. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 37–46, New York, NY, USA, 2008. ACM.

[55] Swi prolog chr home page. http://www.swi-prolog.org/man/chr.html.

[56] Michael Thielscher. Reasoning about actions with chrs and finite domain constraints. In *ICLP*, pages 70–84, 2002.

[57] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[58] Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. pages 125–140.

[59] G. Stewart von Itzstein and M. Jasiunas. On implementing high level concurrency in java. In *Advances in Computer Systems Architecture, Aizu Japan*. Springer Verlag, 2003.

Appendix A

Proofs

In this section, we provide the proofs of the Lemmas and Theorems discussed in this paper. Because many of our proofs rely on inductive steps on the derivations, we define k-step derivations to facilitate the proof mechanisms. Figure A.1 shows k-step derivations of the sequential goal-based derivations $\overset{\delta}{\rightarrowtail}_{\mathcal{G}}$ and the concurrent goal-based derivations $\overset{\delta}{\rightarrowtail}_{||\mathcal{G}}$.

A.1 Proof of Correspondence of Derivations

Theorem 2 (Correspondence of Sequential Derivations) For any reachable CHR state $\langle G \mid Sn\rangle$, CHR state $\langle G' \mid Sn'\rangle$ and CHR Program $\mathcal{P}$,

$$\begin{array}{ll} \text{if} & \langle G \mid Sn\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid Sn'\rangle \\ \text{then} & (NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn')) \quad \vee \\ & (NoIds(G) \uplus DropIds(Sn)) \rightarrowtail^*_{\mathcal{A}} (NoIds(G') \uplus DropIds(Sn')) \end{array}$$

where $NoIds = \{c \mid c \in G, c \text{ is a CHR constraint}\} \uplus \{e \mid e \in G, e \text{ is an equation}\}$

Proof: We prove that for all finite n and reachable states $\langle G \mid Sn\rangle, \langle G' \mid Sn'\rangle$, $\langle G \mid Sn\rangle \rightarrowtail^n_{\mathcal{G}} \langle G' \mid Sn'\rangle$ either yields equivalent abstract stores or corresponds to some abstract semantics derivation. We prove by induction on the derivation steps n. Showing that goal-based derivation of any finite n steps satisfying one of the following conditions:

- **(C1)** $(NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn'))$
- **(C2)** $(NoIds(G) \uplus DropIds(Sn)) \rightarrowtail^*_{\mathcal{A}} (NoIds(G') \uplus DropIds(Sn'))$

We have the following axioms, by definition of the functions $NoIds$ and $DropIds$, for any goals G or store Sn:

- **(a1)** For any equation e, $NoIds(\{e\} \uplus G) = \{e\} \uplus NoIds(G)$
- **(a2)** For any equation e, $DropIds(\{e\} \cup Sn) = \{e\} \uplus DropIds(Sn)$
- **(a3)** For any numbered constraint $c\#i$, $NoIds(\{c\#i\} \uplus G) = NoIds(G)$

Sequential Goal-based Semantics k-closure

$$\text{(k-Step)} \qquad \sigma \rightarrowtail^0_{\mathcal{G}} \sigma \qquad\qquad \frac{\sigma \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \sigma' \quad \sigma' \rightarrowtail^k_{\mathcal{G}} \sigma''}{\sigma \rightarrowtail^{k+1}_{\mathcal{G}} \sigma''}$$

Concurrent Goal-based Semantics k-closure

$$\text{(k-Step)} \qquad \sigma \rightarrowtail^0_{||\mathcal{G}} \sigma \qquad\qquad \frac{\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma' \quad \sigma' \rightarrowtail^k_{||\mathcal{G}} \sigma''}{\sigma \rightarrowtail^{k+1}_{||\mathcal{G}} \sigma''}$$

Figure A.1: k-closure derivation steps

- **(a4)** For any numbered constraint $c\#i$, $DropIds(\{c\#i\} \cup Sn) = \{c\} \uplus DropIds(Sn)$
- **(a5)** For any CHR constraint c, $NoIds(\{c\} \uplus G) = \{c\} \uplus NoIds(G)$
- **(a6)** For any store Sn', $DropIds(Sn \cup Sn') = DropIds(Sn) \uplus DropIds(Sn')$

$(a1)$ and $(a2)$ are so because $NoIds$ and $DropIds$ have no effect on equations. $(a3)$ is true because $NoIds$ is defined to drop numbered constraints. $(a4)$ is true because $DropIds$ is defined to remove identifier components of numbered constraints. We have $(a5)$ because $NoIds$ has no effect on CHR constraints. By definition of $DropIds$, $(a6)$ is true.

Base case: We consider $\langle G \mid Sn\rangle \rightarrowtail^0_{\mathcal{G}} \langle G' \mid Sn'\rangle$. By definition of $\rightarrowtail^0_{\mathcal{G}}$, we have $G = G'$ and $Sn = Sn'$. Hence $(NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn'))$ and we are done.

Inductive case: We assume that the theorem is true for some finite $k > 0$, hence $\langle G \mid Sn\rangle \rightarrowtail^k_{\mathcal{G}} \langle G' \mid Sn'\rangle$ have some correspondence with the abstract semantics.

We now prove that by extending these k derivations with another step, we preserve correspondence, namely $\langle G \mid Sn\rangle \rightarrowtail^k_{\mathcal{G}} \langle G' \mid Sn'\rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle G'' \mid Sn''\rangle$ has a correspondence with the abstract semantics. We prove this by considering all possible form of derivation step, step $k+1$ can take:

- (Solve) $k+1$ step is of the form $\langle \{e\} \uplus G''' \mid Sn'\rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle W \uplus G''' \mid \{e\} \cup Sn'\rangle$ such that for some G''' and W

  $$G' = \{e\} \uplus G''', G'' = W \uplus G''' \text{ and } Sn'' = \{e\} \cup Sn' \quad (\mathbf{a_{solve}})$$

  where e is an equation, $W = WakeUp(e, Sn)$ contains only goals of the form $c\#i$. This is because (Solve) only wakes up stored

numbered constraints. Hence,

$$\begin{array}{lll} NoIds(G'') \uplus DropIds(Sn'') & & \\ \quad = \; NoIds(W \uplus G''') \uplus DropIds(\{e\} \cup Sn') & (\mathbf{a_{solve}}) \\ \quad = \; NoIds(G''') \uplus DropIds(\{e\} \cup Sn') & (\mathbf{a3}) \\ \quad = \; NoIds(G''') \uplus \{e\} \uplus DropIds(Sn') & (\mathbf{a2}) \\ \quad = \; NoIds(\{e\} \uplus G''') \uplus DropIds(Sn') & (\mathbf{a1}) \\ \quad = \; NoIds(G') \uplus DropIds(Sn') & (\mathbf{a_{solve}}) \end{array}$$

Hence we can conclude that evaluated store of derivation step $k+1$ is equivalent to abstract store of evaluated store of step k, therefore satisfying condition (**C1**).

- (Activate) $k+1$ step is of the form $\langle \{c\} \uplus G''' \mid Sn' \rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle \{c\#i\} \uplus G''' \mid \{c\#i\} \cup Sn' \rangle$ such that for some G'''

  $$G' = \{c\} \uplus G''', G'' = \{c\#i\} \uplus G''' \text{ and } Sn'' = \{c\#i\} \cup Sn' \quad (\mathbf{a_{act}})$$

  Hence,

  $$\begin{array}{lll} NoIds(G'') \uplus DropIds(Sn'') & & \\ \quad = \; NoIds(\{c\#i\} \uplus G''') \uplus DropIds(\{c\#i\} \cup Sn') & (\mathbf{a_{act}}) \\ \quad = \; NoIds(G''') \uplus DropIds(\{c\#i\} \cup Sn') & (\mathbf{a3}) \\ \quad = \; NoIds(G''') \uplus \{c\} \uplus DropIds(Sn') & (\mathbf{a4}) \\ \quad = \; NoIds(\{c\} \uplus G''') \uplus DropIds(Sn') & (\mathbf{a5}) \\ \quad = \; NoIds(G') \uplus DropIds(Sn') & (\mathbf{a_{act}}) \end{array}$$

  Hence we can conclude that evaluated store of derivation step $k+1$ is equivalent to abstract store of evaluated store of step k, therefore satisfying condition (**C1**).

- (Simplify) $k+1$ step is of the form $\langle \{c\#i\} \uplus G''' \mid H_P \cup \{c\#i\} \cup H_S \cup Sn''' \rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle B \uplus G''' \mid H_P \cup Sn''' \rangle$ for some H_P,H_S and B such that for some G''' and Sn'''

  $$\begin{array}{l} Sn' = H_P \cup \{c\#i\} \cup H_S \cup Sn''', Sn'' = H_P \cup Sn''', \\ G' = \{c\#i\} \uplus G''' \text{ and } G'' = B \uplus G''' \qquad (\mathbf{a1_{simp}}) \end{array}$$

  and there exists a CHR rule $r \; @ \; H'_P \backslash H'_S \iff t_g \mid B'$ such that exists ϕ where

  $$\begin{array}{l} DropIds(\{c\#i\} \cup H_S) = \phi(H'_S) \quad DropIds(H_P) = \phi(H'_P) \\ Eq(Sn''') \models \phi \wedge t_g \quad B = \phi(B') \qquad (\mathbf{a2_{simp}}) \end{array}$$

  Hence,

$$\begin{array}{lll} NoId(G') \uplus DropIds(Sn') & & \\ \quad = NoIds(\{c\#i\} \uplus G''') \uplus DropIds(H_P \cup \{c\#i\} \cup H_S \cup Sn''') & (\mathbf{a1_{simp}}) \\ \quad = NoIds(G''') \uplus DropIds(H_P \cup \{c\#i\} \cup H_S \cup Sn''') & (\mathbf{a_3}) \\ \quad = NoIds(G''') \uplus DropIds(H_P) \uplus DropIds(\{c\#i\} \cup H_S) \uplus DropIds(Sn''') & (\mathbf{a_6}) \\ \quad = NoIds(G''') \uplus \phi(H'_P) \uplus \phi(H'_S) \uplus DropIds(Sn''') & (\mathbf{a2_{simp}}) \end{array}$$

By definition of the abstract semantics and $a2_{simp}$, we know that we have the rule application $\phi(H'_P) \cup \phi(H'_S) \rightarrowtail_{\mathcal{A}} \phi(B')$ Therefore, by monotonicity of CHR rewriting (Theorem 1)

$$\begin{array}{lll} NoId(G') \uplus DropIds(Sn') & & \\ \quad = & NoIds(G''') \uplus \phi(H'_P) \uplus \phi(H'_S) \uplus DropIds(Sn''') & \\ \rightarrowtail_{\mathcal{A}} & NoIds(G''') \uplus \phi(B') \uplus DropIds(Sn''') & (\textbf{Theorem 1}) \\ \quad = & NoIds(\phi(B') \uplus G''') \uplus DropIds(Sn''') & (\textbf{a1}), (\textbf{a3}) \\ \quad = & NoIds(G'') \uplus DropIds(Sn'') & (\textbf{a1}_{\textbf{simp}}) \end{array}$$

Hence we have $NoId(G) \uplus DropIds(Sn) \rightarrowtail^*_{\mathcal{A}} NoId(G') \uplus DropIds(Sn') \rightarrowtail_{\mathcal{A}} NoIds(G'') \uplus DropIds(Sn'')$, such that the $k+1$ goal-based derivation step satisfy condition (**C2**).

- (Propagate) $k+1$ step is of the form $\langle \{c\#i\} \uplus G''' \mid H_P \cup \{c\#i\} \cup H_S \cup Sn''' \rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle B \uplus \{c\#i\} \uplus G''' \mid H_P \cup \{c\#i\} \cup Sn''' \rangle$ for some H_P,H_S and B such that for some G''' and Sn'''

$$\begin{array}{l} Sn' = H_P \cup \{c\#i\} \cup H_S \cup Sn''', Sn'' = H_P \cup \{c\#i\} \cup Sn''', \\ G' = \{c\#i\} \uplus G''' \text{ and } G'' = B \uplus \{c\#i\} \uplus G''' \qquad (\textbf{a1}_{\textbf{prop}}) \end{array}$$

and there exists a CHR rule $r \ @ \ H'_P \backslash H'_S \iff t_g \mid B'$ such that exists ϕ where

$$\begin{array}{l} DropIds(H_S) = \phi(H'_S) \quad DropIds(\{c\#i\} \cup H_P) = \phi(H'_P) \\ Eq(Sn''') \models \phi \wedge t_g \quad B = \phi(B') \qquad (\textbf{a2}_{\textbf{prop}}) \end{array}$$

Hence,

$$\begin{array}{lll} NoId(G') \uplus DropIds(Sn') & & \\ = NoIds(\{c\#i\} \uplus G''') \uplus DropIds(H_P \cup \{c\#i\} \cup H_S \cup Sn''') & & (\textbf{a1}_{\textbf{prop}}) \\ = NoIds(G''') \uplus DropIds(H_P \cup \{c\#i\} \cup H_S \cup Sn''') & & (\textbf{a}_{\textbf{3}}) \\ = NoIds(G''') \uplus DropIds(\{c\#i\} \cup H_P) \uplus DropIds(H_S) \uplus DropIds(Sn''') & & (\textbf{a}_{\textbf{6}}) \\ = NoIds(G''') \uplus \phi(H'_P) \uplus \phi(H'_S) \uplus DropIds(Sn''') & & (\textbf{a2}_{\textbf{prop}}) \end{array}$$

By definition of the abstract semantics and $a2_{simp}$, we know that we have the rule application $\phi(H'_P) \cup \phi(H'_S) \rightarrowtail_{\mathcal{A}} \phi(B')$ Therefore, by monotonicity of CHR rewriting (Theorem 1)

$$\begin{array}{lll} NoId(G') \uplus DropIds(Sn') & & \\ \quad = & NoIds(G''') \uplus \phi(H'_P) \uplus \phi(H'_S) \uplus DropIds(Sn''') & \\ \rightarrowtail_{\mathcal{A}} & NoIds(G''') \uplus \phi(B') \uplus DropIds(Sn''') & (\textbf{Theorem 1}) \\ \quad = & NoIds(\phi(B') \uplus G''') \uplus DropIds(Sn''') & (\textbf{a1}), (\textbf{a5}) \\ \quad = & NoIds(\phi(B') \uplus \{c\#i\} \uplus G''') \uplus DropIds(Sn''') & (\textbf{a3}) \\ \quad = & NoIds(G'') \uplus DropIds(Sn'') & (\textbf{a1}_{\textbf{prop}}) \end{array}$$

Hence we have $NoId(G) \uplus DropIds(Sn) \rightarrowtail^*_{\mathcal{A}} NoId(G') \uplus DropIds(Sn') \rightarrowtail_{\mathcal{A}} NoIds(G'') \uplus DropIds(Sn'')$, such that the $k+1$ goal-based derivation step satisfy condition (**C2**).

- (Drop) $k+1$ step is of the form $\langle \{c\#i\} \uplus G'' \mid Sn' \rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle \{G'' \mid Sn' \rangle$

such that for some G'''

$$G'' = \{c\#i\} \uplus G' \text{ and } Sn' = Sn'' \quad (\mathbf{a_{drop}})$$

Hence,

$$\begin{array}{rcll} NoIds(G'') \uplus DropIds(Sn'') & = & NoIds(\{c\#i\} \uplus G') \uplus DropIds(Sn') & (\mathbf{a_{drop}}) \\ & = & NoIds(G') \uplus DropIds(Sn') & (\mathbf{a3}) \end{array}$$

Hence we can conclude that evaluated store of derivation step $k+1$ is equivalent to abstract store of evaluated store of step k, therefore satisfying condition (**C1**).

Considering all forms of $k+1$ derivation steps, (Solve), (Activate) and (Drop) satisfies condition $bf(C1)$, while (Simplify) and (Propagate) satisfy condition (**C2**). Hence we can conclude that Theorem 2 holds. □

Lemma 1 (k-Concurrency) For any finite k of mutually non-overlapping concurrent derivations,

$$\frac{\begin{array}{c} \langle G_1 \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \overset{H_{P1}\backslash H_{S1}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_1 \mid \{\} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \\ ... \\ \langle G_i \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \overset{H_{Pi}\backslash H_{Si}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_i \mid H_{S1} \cup ... \cup \{\} \cup ... \cup H_{Sk} \cup S\rangle \\ ... \\ \langle G_k \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \overset{H_{Pk}\backslash H_{Sk}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_k \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup \{\} \cup S\rangle \\ H_{P1} \subseteq S ... H_{Pi} \subseteq S ... H_{Pk} \subseteq S \\ \delta = H_{P1} \cup ... \cup H_{Pi} \cup ... \cup H_{Pk} \backslash H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \end{array}}{\begin{array}{l} \langle G_1 \uplus ... \uplus G_i \uplus ... \uplus G_k \uplus G \mid H_{S1} \cup ... \cup H_{Si} \cup ... \cup H_{Sk} \cup S\rangle \\ \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \quad \langle G'_1 \uplus ... \uplus G'_i \uplus ... \uplus G'_k \uplus G \mid S\rangle \end{array}}$$

we can decompose this into $k-1$ applications of the (pair-wise) (Goal Concurrency) derivation step.

Proof: We prove the soundness of k-concurrency by showing that k mutually non-overlapping concurrent derivation can be decomposed into $k-1$ applications of the (Goal Concurrency) step. We prove by induction on the number of concurrent derivations k.

Base case: $k = 2$. 2-concurrency immediately corresponds to the (Goal Concurrency) rule, hence it is true by definition.

Inductive case: We assume that for $j > 2$ and $j < k$, we can decompose j mutually non-overlapping concurrent derivations. into $j-1$ applications of the (Goal Concurrency) step. We now consider $j+1$ mutually non-overlapping concurrent derivations. Because all derivations are non-overlapping, we can compose any two derivations amongst these $j+1$ into a single concurrent step via the (Goal Concurrency) rule. We

pick any two concurrent derivations, say the j^{th} and $(j+1)^{th}$ (Note that by symmetry, this choice is arbitrary):

$$\langle G_j \mid H_{S1} \cup ... \cup H_{Sj} \cup H_{Sj+1} \cup S\rangle \overset{H_{Pj}\setminus H_{Sj}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_j \mid H_{S1} \cup ... \cup \{\} \cup H_{Sj+1} \cup S\rangle$$

$$\langle G_{j+1} \mid H_{S1} \cup ... \cup H_{Sj} \cup H_{Sj+1} \cup S\rangle \overset{H_{Pj+1}\setminus H_{Sj+1}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_{j+1} \mid H_{S1} \cup ... \cup H_{Sj} \cup \{\} \cup S\rangle$$

$$H_{Pj} \subseteq S \quad H_{Pj+1} \subseteq S$$

By applying the above two non-overlapping derivations with an instance of the (Goal Concurrency) rule, we have:

$$\begin{array}{c}\langle G_{j'} \mid H_{S1} \cup ... \cup H_{Sj'} \cup S\rangle \overset{H_{Pj'}\setminus H_{Sj'}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_{j'} \mid H_{S1} \cup ... \cup \{\} \cup S\rangle \\ \text{where} \quad G_{j'} = G_j \uplus G_{j+1} \quad G'_{j'} = G'_j \uplus G'_{j+1} \\ H_{Sj'} = H_{Sj} \cup H_{Sj+1} \quad H_{Pj'} = H_{Pj} \cup H_{Pj+1}\end{array}$$

Hence we have reduced $j+1$ non-overlapping concurrent derivations into j non-overlapping concurrent derivations by combining via the (Goal Concurrency) derivation step.

$$\frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup ... \cup H_{Sj'} \cup S\rangle \overset{H_{P1}\setminus H_{S1}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_1 \mid \{\} \cup ... \cup H_{Sj'} \cup S\rangle \\ ... \\ \langle G_{j'} \mid H_{S1} \cup ... \cup H_{Sj'} \cup S\rangle \overset{H_{Pj'}\setminus H_{Sj'}}{\rightarrowtail_{||\mathcal{G}}} \langle G'_{j'} \mid H_{S1} \cup ... \cup \{\} \cup S\rangle \\ H_{P1} \subseteq S ... H_{Pj'} \subseteq S \\ \delta = H_{P1} \cup ... \cup H_{Pj'} \setminus H_{S1} \cup ... \cup H_{Sj'}\end{array}}{\begin{array}{l}\langle G_1 \uplus ... \uplus G_{j'} \uplus G \mid H_{S1} \cup ... \cup H_{Sj'} \cup S\rangle \\ \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \quad \langle G'_1 \uplus ... \uplus G'_{j'} \uplus G \mid S\rangle\end{array}}$$

Hence, by our original assumption, the above is decomposable into $j-1$ applications of the (Goal Concurrency) step. This implies that $j+1$ concurrent derivations are decomposable into j (Goal Concurrency) step. □

Lemma 2 (Monotonicity of Goals in Goal-based Semantics) For any goals G,G' and G'' and CHR store Sn and Sn', if $\langle G \mid Sn\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid Sn'\rangle$ then $\langle G \uplus G'' \mid Sn\rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \uplus G'' \mid Sn'\rangle$

Proof: We need to prove that for any finite k, if $\langle G \mid Sn\rangle \rightarrowtail^k_{\mathcal{G}} \langle G' \mid Sn'\rangle$ we can always extend the goals with any G'' such that $\langle G \uplus G'' \mid Sn\rangle \rightarrowtail^k_{\mathcal{G}} \langle G' \uplus G'' \mid Sn'\rangle$.

We prove this by induction on the number of derivation steps k, showing that for any finite $i \leq k$, goals are monotonic.

Base case: We consider $\langle G \mid Sn\rangle \rightarrowtail^0_{\mathcal{G}} \langle G' \mid Sn'\rangle$. By definition of $\rightarrowtail^0_{\mathcal{G}}$, we have $G = G'$ and $Sn = Sn'$. Hence we immediately have $\langle G \uplus G'' \mid Sn\rangle \rightarrowtail^0_{\mathcal{G}} \langle G' \uplus G'' \mid Sn'\rangle$

Inductive case: We assume that the lemma is true for some finite $i > 0$, hence $\langle G \mid Sn\rangle \rightarrowtail^{i}_{\mathcal{G}} \langle G' \mid Sn'\rangle$ is monotonic with respect to the goals.

We now prove that by extending these i derivations with another step, we still preserve monotonicity of the goals. Namely, if $\langle G \mid Sn\rangle \rightarrowtail^{i}_{\mathcal{G}} \langle \{g\} \uplus G_i \mid Sn_i\rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle G_{i+1} \mid Sn_{i+1}\rangle$ then $\langle G \uplus G'' \mid Sn\rangle \rightarrowtail^{i}_{\mathcal{G}} \langle G_i \uplus G'' \mid Sn_i\rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle G_{i+1} \uplus G'' \mid Sn_{i+1}\rangle$ We prove this by considering all possible form of derivation step, step $i + 1^{th}$ can take:

- (Solve) Consider $i + 1^{th}$ derivation step of the form $\langle \{e\} \uplus G_i \mid Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle W \uplus G \mid \{e\} \cup Sn_i\rangle$ for some equation e and $W = WakeUp(e, Sn_i)$.

  By definition, the (Solve) step only make reference to e and Sn_i, hence we can extend G_i with any G'' without affecting the derivation step, ie.

  $$\langle \{e\} \uplus G_i \uplus G'' \mid Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle W \uplus G_i \uplus G'' \mid \{e\} \cup Sn_i\rangle$$

  Hence, given our assumption that the first i derivations are monotonic with respect to the goals, extending with a $i+1^{th}$ (Solve) step preserves monotonicity of the goals.

- (Activate) Consider $i + 1^{th}$ derivation step of the form $\langle \{c\} \uplus G_i \mid Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle \{c\#j\} \uplus G_i \mid \{c\#j\} \cup Sn_i\rangle$ for some CHR constraint c, goals G_i and store Sn_i.

  By definition, the (Activate) step only make reference to goal c, hence we can extend G_i with any G'' without affecting the derivation step, ie.

  $$\langle \{c\} \uplus G_i \uplus G'' \mid Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle \{c\#j\} \uplus G_i \uplus G'' \mid \{c\#j\} \cup Sn_i\rangle$$

  Hence, given our assumption that the first i derivations are monotonic with respect to the goals, extending with a $i + 1^{th}$ (Activate) step preserves monotonicity of the goals.

- (Simplify) Consider $i+1^{th}$ derivation step of the form $\langle \{c\#j\} \uplus G_i \mid \{c\#j\} \uplus H_S \cup Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle B \uplus G_i \mid Sn_i\rangle$ for some CHR constraints H_S and body constraints B.

  By definition, the (Simplify) step only make reference to goal $c\#j$, and H_S of the store, hence we can extend G_i with any G'' without affecting the derivation step, ie.

  $$\langle \{c\#j\} \uplus G_i \uplus G'' \mid \{c\#j\} \cup H_S \cup Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle B \uplus G_i \uplus G'' \mid Sn_i\rangle$$

  Hence, given our assumption that the first i derivations are monotonic with respect to the goals, extending with a $i + 1^{th}$ (Simplify) step preserves monotonicity of the goals.

- (Propagate) Consider $i+1^{th}$ derivation step of the form $\langle \{c\#j\} \uplus G_i \mid$

$H_S \cup Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle B \uplus \{c\#j\} \uplus G_i \mid Sn_i\rangle$ for some CHR constraints H_S and body constraints B.

By definition, the (Propagate) step only make reference to goal $c\#j$, and H_S of the store, hence we can extend G_i with any G'' without affecting the derivation step, ie.

$$\langle \{c\#j\} \uplus G_i \uplus G'' \mid H_S \cup Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle B \uplus \{c\#j\} \uplus G_i \uplus G'' \mid Sn_i\rangle$$

Hence, given our assumption that the first i derivations are monotonic with respect to the goals, extending with a $i+1^{th}$ (Propagate) step preserves monotonicity of the goals.

- (Drop) Consider $i+1^{th}$ derivation step of the form $\langle \{c\#j\} \uplus G_i \mid Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle G_i \mid Sn_i\rangle$ for some numbered constraint $c\#j$.

  By definition, the (Drop) step only make reference to goal $c\#j$, while it's premise depend on Sn_i, hence we can extend goals G_i with any G'' without affecting the derivation step, ie.

  $$\langle \{c\#j\} \uplus G_i \uplus G'' \mid Sn_i\rangle \rightarrowtail_{\mathcal{G}} \langle G_i \uplus G'' \mid Sn_i\rangle$$

  Hence, given our assumption that the first i derivations are monotonic with respect to the goals, extending with a $i+1^{th}$ (Drop) step preserves monotonicity of the goals.

Hence, with our assumption of monotonicity of goals for i steps, the goals are still monotonic for $i+1$ steps regardless of the form of the $i+1^{th}$ derivation step. □

Lemma 3 (Isolation of Goal-based Derivations) If $\langle G \mid H_P \cup H_S \cup S_1 \cup S_2\rangle \overset{H_P \backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1' \cup S_2\rangle$ then $\langle G \mid H_P \cup H_S \cup S_1\rangle \overset{H_P \backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1'\rangle$

Proof: We need to show that for any goal-based derivation, we can omit any constraint of the store which is not a side-effect of the derivation. To prove this, we consider all possible forms of goal-based derivations:

- (Solve) Consider derivation of the form

  $$\langle \{e\} \uplus G \mid W \cup \{\} \cup S_1 \cup S_2\rangle \overset{W \backslash \{\}}{\rightarrowtail_{\mathcal{G}}} \langle W \uplus G \mid W \cup \{\} \cup \{e\} \cup S_1 \cup S_2\rangle$$

  Since wake up side-effect is captured in W, we can drop S_2 without affecting the derivation. Hence we also have:

  $$\langle \{e\} \uplus G \mid W \cup \{\} \cup S_1\rangle \overset{W \backslash \{\}}{\rightarrowtail_{\mathcal{G}}} \langle W \uplus G \mid W \cup \{\} \cup \{e\} \cup S_1\rangle$$

- (Activate) Consider derivation of the form

  $$\langle \{c\} \uplus G \mid \{\} \cup \{\} \cup S_1 \cup S_2\rangle \overset{\{\} \backslash \{\}}{\rightarrowtail_{\mathcal{G}}} \langle \{c\#i\} \uplus G \mid \{\} \cup \{\} \cup \{c\#i\} \cup S_1 \cup S_2\rangle$$

Since (Activate) simply introduces a new constraint $c\#i$ into the store, we can drop S_2 without affecting the derivation. Hence we also have:

$$\langle \{c\} \uplus G \mid \{\} \cup \{\} \cup S_1 \rangle \overset{\{\}\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} \langle \{c\#i\} \uplus G \mid \{\} \cup \{\} \cup \{c\#i\} \cup S_1 \rangle$$

- (Simplify) Consider derivation of the form

$$\langle \{c\#i\} \uplus G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \overset{H_P\backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle B \uplus G \mid H_P \cup S_1 \cup S_2 \rangle$$

Since S_2 is not part of the side-effects of this derivation, we can drop S_2 without affecting the derivation. Hence we also have:

$$\langle \{c\#i\} \uplus G \mid H_P \cup H_S \cup S_1 \rangle \overset{H_P\backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle B \uplus G \mid H_P \cup S_1 \rangle$$

- (Propagate) Consider derivation of the form

$$\langle \{c\#i\} \uplus G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \overset{H_P\backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle B \uplus \{c\#i\} \uplus G \mid H_P \cup S_1 \cup S_2 \rangle$$

Since S_2 is not part of the side-effects of this derivation, we can drop S_2 without affecting the derivation. Hence we also have:

$$\langle \{c\#i\} \uplus G \mid H_P \cup H_S \cup S_1 \rangle \overset{H_P\backslash H_S}{\rightarrowtail_{\mathcal{G}}} \langle B \uplus \{c\#i\} \uplus G \mid H_P \cup S_1 \rangle$$

- (Drop) Consider derivation of the form

$$\langle \{c\#i\} \uplus G \mid \{\} \cup \{\} \cup S_1 \cup S_2 \rangle \overset{\{\}\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} \langle G \mid \{\} \cup \{\} \cup S_1 \cup S_2 \rangle$$

(Drop) simply removes the goal $c\#i$ when no instances of (Simplify) or (Propagate) can apply on it. Note that it's premise references to the entire store, so removing S_2 may seems unsafe. But since removing constraints from the store will not cause $c\#i$ to be applicable to any instances of (Simplify) or (Propagate), hence we also have:

$$\langle \{c\} \uplus G \mid \{\} \cup \{\} \cup S_1 \rangle \overset{\{\}\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} \langle G \mid \{\} \cup \{\} \cup S_1 \rangle$$

□

Lemma 4 (Isolation of Transitive Goal-based Derivations) If $\langle G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G' \mid H_P \cup S_1' \cup S_2 \rangle$ with side-effects $\delta = H_P \backslash H_S$, then $\langle G \mid H_P \cup H_S \cup S_1 \rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G' \mid H_P \cup S_1' \rangle$

Proof: We need to prove that for all k, $\langle G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \overset{k}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1' \cup S_2 \rangle$ with side-effects $\delta = H_P \backslash H_S$ we can always safely omit

affected potions of the store from the derivation. We prove by induction on $i \leq k$.

Base case: $i = 1$. Consider, $\langle G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \rightarrowtail^1_{\mathcal{G}} \langle G' \mid H_P \cup S'_1 \cup S_2 \rangle$. This corresponds to the premise in Lemma 3, hence we can safely omit S_2 from the derivation.

Inductive case: $i > 1$. we assume that for any $\langle G \mid H_{Pi} \cup H_{Si} \cup S_{1i} \cup S_{2i} \rangle \rightarrowtail^i_{\mathcal{G}} \langle G' \mid H_{Pi} \cup S'_{1i} \cup S_{2i} \rangle$ with side-effects $\delta_i = H_{Pi} \backslash H_{Si}$, we can safely omit S_{2i} from the derivation. Let's consider a $j = i + 1$ derivation step from here, which contains side-effects $\delta_j = H_{Pj} \backslash H_{Sj}$ non-overlapping with δ_i. Hence H_{Pj} and H_{Sj} must be in S_{2i} (ie. $S_{2i} = H_{Pj} \cup H_{Sj} \cup S_{1j} \cup S_{2j}$).

$$
\begin{array}{ll}
 & \langle G \mid H_{Pi} \cup H_{Si} \cup S_{1i} \cup H_{Pj} \cup H_{Sj} \cup S_{1j} \cup S_{2j} \rangle \\
\rightarrowtail^i_{\mathcal{G}} & \langle G' \mid H_P \cup S'_{1i} \cup H_{Pj} \cup H_{Sj} \cup S_{1j} \cup S_{2j} \rangle \\
\rightarrowtail^{\delta_j}_{\mathcal{G}} & \langle G'' \mid H_P \cup S'_{1i} \cup H_{Pj} \cup S'_{1j} \cup S_{2j} \rangle
\end{array}
$$

Hence consider the following substitutions:

$$
\begin{array}{ll}
H_P = H_{Pi} \cup H_{Pj} & H_S = H_{Si} \cup H_{Sj} \\
S_1 = S_{1i} \cup S_{1j} & S'_1 = S'_{1i} \cup S'_{1j} \\
\delta = H_P \backslash H_S &
\end{array}
$$

we have $\langle G \mid H_P \cup H_S \cup S_1 \cup S_{2j} \rangle \rightarrowtail^{i+1}_{\mathcal{G}} \langle G \mid H_P \cup S'_1 \cup S_{2j} \rangle$ with side-effects δ such that no constraints in S_{2j} is in δ. Hence we can safely omit S_{2j} from the derivation and we have isolation for $i+1$ derivations as well. □

Lemma 5 (Sequential Reachability of Concurrent Derivation Steps) For any sequentially reachable CHR state σ, CHR state σ' and rewriting side-effects δ if $\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma'$ then σ' is sequentially reachable, $\sigma \rightarrowtail^*_{\mathcal{G}} \sigma'$ with side-effects δ.

Proof: From the k-concurrency Lemma (Lemma 1) we showed that any finite k mutually non-overlapping concurrent goal-based derivations can be replicated by nested application of the (Goal Concurrency) step. Hence, to prove sequential reachability of concurrent derivations, we only need to consider the derivation steps (Lift) and (Goal Concurrency) which sufficiently covers the concurrent behaviour of any k concurrent derivations.

We prove by structural induction of the concurrent goal-based semantics derivation steps (Lift) and (Goal Concurrency).

- (Lift) is the base case. Application of (Lift) simply lifts a goal-based derivation $\sigma \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \sigma'$ into a concurrent goal-based derivation $\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma'$. Thus states σ' derived from the (Lift) step is immediately sequentially reachable since $\sigma \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \sigma'$ implies $\sigma \rightarrowtail^*_{\mathcal{G}} \sigma'$.

- (Goal Concurrency)

$$\frac{\begin{array}{c}\textbf{(D1)}\quad \langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_1}{\rightarrowtail}_{||\mathcal{G}} \langle G_1' \mid \{\} \cup H_{S2} \cup S\rangle \\ \textbf{(D2)}\quad \langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_2}{\rightarrowtail}_{||\mathcal{G}} \langle G_2' \mid H_{S1} \cup \{\} \cup S\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{rl} & \langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \textbf{(C)}\quad \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} & \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

we assume that **(D1)** and **(D2)** are sequentially reachable. This means that we have the following:

$$\langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G_1' \mid \{\} \cup H_{S2} \cup S\rangle \text{ with side-effects } \delta_1 = H_{P1} \backslash H_{S1} \text{ such that } H_{P1} \subseteq S \quad (\mathbf{a_{D1}})$$

$$\langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G_2' \mid H_{S1} \cup \{\} \cup S\rangle \text{ with side-effects } \delta_2 = H_{P2} \backslash H_{S2} \text{ such that } H_{P2} \subseteq S \quad (\mathbf{a_{D2}})$$

Since both derivations are by definition non-overlapping in side-effects, we can show that **(C)** is sequentially reachable, using monotonicity of goals (Lemma 2) and isolation of derivations (Lemma 3):

$$\begin{array}{rll} & \langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle & \\ \rightarrowtail_{\mathcal{G}}^{*} & \langle G_1' \uplus G_2 \uplus G \mid H_{S2} \cup S\rangle & (\mathbf{Lemma2}, \mathbf{a_{D1}}) \\ \rightarrowtail_{\mathcal{G}}^{*} & \langle G_1' \uplus G_2' \uplus G \mid S\rangle & (\mathbf{Lemma2}, \mathbf{Lemma4}, \mathbf{a_{D2}})\end{array}$$

Hence, the above sequential goal-based derivation shows that (Goal Concurrency) derivation step is sequentially reachable with side-effect δ.

□

Theorem 3 (Sequential Reachability of Concurrent Derivations) For any initial CHR state σ, CHR state σ' and CHR Program $\mathcal{P}$, if $\sigma \rightarrowtail_{||\mathcal{G}}^{*} \sigma'$ then $\sigma \rightarrowtail_{\mathcal{G}}^{*} \sigma'$.

Proof: We prove that for all finite k number of concurrent derivation steps $\sigma \rightarrowtail_{||\mathcal{G}}^{k} \sigma'$, we can find a corresponding sequential derivation sequence $\sigma \rightarrowtail_{\mathcal{G}}^{*} \sigma'$.

Base case: $k = 1$. We consider $\sigma \rightarrowtail_{||\mathcal{G}}^{1} \sigma'$. From Lemma 5, we can conclude that we have $\sigma \rightarrowtail_{\mathcal{G}}^{*} \sigma'$ as well.

Inductive case: $k > 1$. We consider $\sigma \rightarrowtail_{||\mathcal{G}}^{k} \sigma'$, assuming that it is sequentially reachable, hence we also have $\sigma \rightarrowtail_{\mathcal{G}}^{*} \sigma'$. We consider extending this derivation with the $k + 1^{th}$ step $\sigma' \rightarrowtail_{||\mathcal{G}} \sigma''$. By Lemma 5, we can conclude that the $k + 1^{th}$ concurrent derivation is sequential reachable, hence $\sigma' \rightarrowtail_{\mathcal{G}}^{*} \sigma''$. Hence we have $\sigma \rightarrowtail_{\mathcal{G}}^{*} \sigma' \rightarrowtail_{\mathcal{G}}^{*} \sigma''$ showing that $\sigma \rightarrowtail_{||\mathcal{G}}^{k+1} \sigma''$ is sequentially reachable. □

A.2 Proof of Correspondence of Termination and Exhaustiveness

Lemma 6 (Rule instances in reachable states are always active) For any reachable CHR state $\langle G \mid Sn\rangle$, any rule head instance $H \subseteq Sn$ must be active. ie. $\exists c\#i \in H$ such that $c\#i \in G$.

Proof: We will prove this for the sequential goal-based semantics. Since Theorem 3 states all concurrent derivation is sequentially reachable, this Lemma immediately applies to the concurrent goal-based semantics as well.

We prove that for all finite k derivations from any initial CHR state $\langle G \mid \{\}\rangle$, ie. $\langle G \mid \{\}\rangle \rightarrowtail^k_{\mathcal{G}} \langle G' \mid Sn'\rangle$, all rule head instances $H \subseteq Sn'$ has at least one $c\#i \in H$ such that $c\#i \in G$. We prove by induction on $i < k$ that states reachable by i derivations from an initial stage have the above property.

Base case: $i = 0$. Hence $\langle G \mid \{\}\rangle \rightarrowtail^0_{\mathcal{G}} \langle G' \mid Sn'\rangle$. By definition, $G = G'$ and $Sn' = \{\}$. Since Sn' is empty, the base case immediately satisfies the Lemma.

Inductive case: $i > 0$. We assume that for any $\langle G \mid \{\}\rangle \rightarrowtail^i_{\mathcal{G}} \langle G' \mid Sn'\rangle$, all rule head instances $H \subseteq Sn'$ is active, hence have at least one $c\#i \in H$ such that $c\#i \in G'$. We extend this derivation with an $i+1^{th}$ step, hence $\langle G \mid \{\}\rangle \rightarrowtail^i_{\mathcal{G}} \langle G' \mid Sn'\rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle G'' \mid Sn''\rangle$. We now prove that all rule head instances in Sn'' are active. We consider all possible forms of this $i+1^{th}$ derivation step. We omit side-effects.

- (Solve) $i+1$ derivation step is of the form $\langle \{e\} \uplus G''' \mid Sn'\rangle \rightarrowtail_{\mathcal{G}} \langle W \uplus G''' \mid \{e\} \cup Sn'\rangle$ for some goals G''' and $W = WakeUp(e, Sn')$. Our assumption provides that all rule head instances in Sn' are active. Introducing e into the store will possibly introduce new rule head instances. This is because for some CHR rule $(r \ @ \ H_P \backslash H_S \iff t_g \mid B) \in \mathcal{P}$ since we may have a new ϕ such that $Eqs(\{e\} \cup Sn') \models \phi \wedge t_g$ and $\phi(H_P \cup H_S) \in Sn'$. This means that there is at least one $c\#i$ in $\phi(H_P \cup H_S)$ which is further grounded by e. Thankfully, by definition of $W = WakeUp(e, Sn')$, we have $c\#i \in W$. Hence new rule head instances will become active because of introduction of W to the goals.
- (Activate) $i+1$ derivation step is of the form $\langle \{c\} \uplus G''' \mid Sn'\rangle \rightarrowtail_{\mathcal{G}} \langle \{c\#i\} \uplus G''' \mid \{c\#i\} \cup Sn'\rangle$. Our assumption provides that all rule head instances in Sn' are active. By adding $c\#i$ to the store, we can possibly introduce new rule head instances $\{c\#i\} \cup H$ such that $H \in Sn'$. Since $c\#i$ is also retained as a goal, such new rule head instances are active as well.
- (Simplify) $i+1$ derivation step is of the form $\langle \{c\#i\} \uplus G''' \mid \{c\#i\} \cup H_S \cup Sn'\rangle \rightarrowtail_{\mathcal{G}} \langle B \uplus G''' \mid Sn'\rangle$. Our assumption provides that all rule

head instances in Sn' are active. $c\#i$ has applied a rule instance, removing $c\#i$ and some H_S from the store. Since $c\#i$ is no longer in the store, we can safely remove $c\#i$ from the goals. Removing H_S from the store will only (possibly) remove other rule head instance from the store. Hence rule head instances in Sn' still remain active.

- (Propagate) $i+1$ derivation step is of the form $\langle\{c\#i\} \uplus G''' \mid \{c\#i\} \cup H_S \cup Sn'\rangle \rightarrowtail_{\mathcal{G}} \langle B \uplus \{c\#i\} \uplus G''' \mid \{c\#i\} \cup Sn'\rangle$. Our assumption provides that all rule head instances in Sn' are active. $c\#i$ has applied a rule instance, removing some H_S from the store. Since $c\#i$ is still in the store, we cannot safely remove $c\#i$ from the goals, thus it is retained. Removing H_S from the store will only (possibly) remove other rule head instance from the store. Hence rule head instances in Sn', including those that contains $c\#i$, still remain active.
- (Drop) $i+1$ derivation step is of the form $\langle\{c\#i\} \uplus G''' \mid Sn'\rangle \rightarrowtail_{\mathcal{G}} \langle G''' \mid Sn'\rangle$. Our assumption provides that all rule head instances in Sn' are active. Premise of the (Drop) step demands that no (Simplify) and (Propagate) steps apply on $c\#i$. This means that $c\#i$ is not part of any rule head instances in Sn'. Hence we can safely remove $c\#i$ from the goals without risking to deactivate any rule instances.

Hence (Solve) and (Activate) guarantees that new rule head instances become active, (Drop) safely removes a goal without deactivating any rule head instances and (Simplify) and (Propagate) only removes constraint from the store. In all cases, existing rule head instances remain active while new rule head instances become active, thus we have proved the lemma. □

Theorem 4 (Correspondence of Exhaustiveness) For any initial CHR state $\langle G, \{\}\rangle$, final CHR state $\langle\{\}, Sn\rangle$ and terminating CHR program $\mathcal{P}$,

$$\begin{array}{l}\text{if } \langle G \mid \{\}\rangle \rightarrowtail^*_{||\mathcal{G}} \langle\{\} \mid Sn\rangle \\ \text{then } G \rightarrowtail^*_{\mathcal{A}} DropIds(Sn) \text{ and } Final_{\mathcal{A}}(DropIds(Sn))\end{array}$$

Proof: We prove that for any concurrent derivation $\langle G \mid \{\}\rangle \rightarrowtail^*_{||\mathcal{G}} \langle\{\} \mid Sn\rangle$, we have a corresponding abstract derivation $G \rightarrowtail^*_{\mathcal{A}} DropIds(Sn)$. Theorem 3 states that we can replicate the above concurrent derivation, with a sequential derivation. Hence we have $\langle G \mid \{\}\rangle \rightarrowtail^*_{\mathcal{G}} \langle\{\} \mid Sn\rangle$. By instantiating Theorem 2, we immediately have $G \rightarrowtail^*_{\mathcal{A}} DropIds(Sn)$ from this sequential goal-based derivation.

Next we show that $DropIds(Sn)$ is a final store ($Final_{\mathcal{A}}(DropIds(Sn))$) with respect to some CHR program $\mathcal{P}$. We prove by contradiction: Suppose $DropIds(Sn)$ is not a final store, hence $\langle\{\} \mid Sn\rangle$ has at least one rule head instance H of $\mathcal{P}$ in Sn which is not active, since the goals

are empty. However, this contradicts with Lemma 6, which states that all reachable states have only active rule instances. Since $\langle \{\} \mid Sn \rangle$ is sequentially reachable, it must be the case that Sn has no rule head instances of $\mathcal{P}$. Therefore $DropIds(Sn)$ must be a final store. □

Lemma 7 (Terminal CHR State) For any CHR State $\langle G \mid Sn \rangle$ and a terminating CHR program $\mathcal{P}$,

if $Final_{\mathcal{A}}(NoIds(G) \uplus DropIds(Sn))$
then there exists no proceeding concurrent derivation $\langle G \mid Sn \rangle \rightarrowtail_{||\mathcal{G}} \langle G' \mid Sn' \rangle$ that involves applications of the (Simplify) or (Propagate) derivation rules.

Proof: We prove by contradiction: Suppose that we have some proceeding concurrent derivation $\langle G \mid Sn \rangle \rightarrowtail_{||\mathcal{G}} \langle G' \mid Sn' \rangle$ which involves an application of at least one (Simplify) or (Propagate) derivation. By Theorem 3, we have $\langle G \mid Sn \rangle \overset{\delta}{\rightarrowtail}_{\mathcal{G}} \langle G' \mid Sn' \rangle$. Specifically, there must exist some CHR derivation $\langle G'' \mid Sn'' \rangle \overset{\delta'}{\rightarrowtail}_{\mathcal{G}} \langle G''' \mid Sn''' \rangle$ which is a (Simplify) or (Propagate) transition such that

$$\langle G \mid Sn \rangle \rightarrowtail^*_{\mathcal{G}} \langle G'' \mid Sn'' \rangle \overset{\delta'}{\rightarrowtail}_{\mathcal{G}} \langle G''' \mid Sn''' \rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid Sn' \rangle$$

Yet by Theorem 2, there is a corresponding abstract derivation,

$$(NoIds(G) \uplus DropIds(Sn)) \rightarrowtail^*_{\mathcal{A}} (NoIds(G''') \uplus DropIds(Sn'''))$$

which involves the application of a (Simplify) or (Propagate) rule. This contradicts with the assumption that $NoIds(G) \uplus DropIds(Sn)$ is a final state (ie. we have $\neg Final_{\mathcal{A}}(NoIds(G) \uplus DropIds(Sn))$). Hence we cannot have any proceeding concurrent derivations $\langle G \mid Sn \rangle \rightarrowtail_{||\mathcal{G}} \langle G' \mid Sn' \rangle$ which involves an application of at least one (Simplify) or (Propagate) derivation. □

Lemma 8 (Finite Administrative CHR Goal-Based Derivations) For any CHR State $\langle G \mid Sn \rangle$, there cannot exist any infinite concurrent derivations consisting of only administrative derivation rules (Solve), (Activate) and (Drop).

Proof: We prove by first constructing a well-founded total order of CHR states across concurrent goal-based derivations consisting only of administrative transitions (ie. (Solve), (Activate) and (Drop) transitions), and then showing that this ordering monotonically decreases across successive CHR states of well-formed derivations until a minimal value is reached. We define goal ranks over CHR states $\langle G \mid Sn \rangle$, $GoalRank$ as follows:

$GoalRank(\langle G \mid Sn \rangle) = (m, n, p)$
where m is the number of equations in G
n is the number of CHR constraints in G
p is the number of numbered CHR constraints in G

Essentially, goal ranks keep track of the number of each type of goal constraints in a CHR state. As such, the minimal value (bottom, $\perp$) is $(0,0,0)$ We define a total well-founded order over goal ranking tuples (m,n,p) as follows:

$$(m1,n1,p1) \succ (m2,n2,p2)$$
$$\text{if and only if}$$
$$(m1 > m2) \vee (m1 = m2 \wedge n1 > n2) \vee (m1 = m2 \wedge n1 = n2 \wedge p1 > p2)$$

Given a goal-based derivation of any length k, $\delta \rightarrowtail^k_{\mathcal{G}} \delta'$ that consists of only the administrative transitions, we prove that $GoalRank(\delta) \succ GoalRank(\delta')$, hence the derivation is finite and terminating as it will eventually (and ultimately) reach the bottom value $(0,0,0)$. We prove by structural induction over the (Solve), (Activate), (Drop) transitions, assuming that derivations of length $i < k$ have the above property, ie. for $i < k$

$$\text{if } \delta \rightarrowtail^i_{\mathcal{G}} \delta' \text{ then } GoalRank(\delta) \succ GoalRank(\delta')$$

we now need to prove inductively that if $\delta \rightarrowtail^i_{\mathcal{G}} \delta' \rightarrowtail_{\mathcal{G}} \delta''$ then we must have $GoalRank(\delta) \succ GoalRank(\delta'')$. We consider all possible administrative transitions for $\delta' \rightarrowtail_{\mathcal{G}} \delta''$, where $GoalRank(\delta') = (m,n,p)$ and $GoalRank(\delta'') = (m',n',p')$:

- (Solve): $i+1$ derivation step is of the form $\langle \{e\} \uplus G''' \mid Sn' \rangle \rightarrowtail_{\mathcal{G}} \langle W \uplus G''' \mid \{e\} \cup Sn' \rangle$ for some goals G''' and $W = WakeUp(e, Sn')$. Since equation e is removed from the goals, hence we have $m' = m - 1$. By definition of $Wakeup$, W is a finite set of numbered CHR constraints, hence $p' = p + len(W)$. No CHR constraints are affected, hence $n' = n$. As such we have $GoalRank(\delta'') = (m-1, n, p + len(W))$. Since $(m,n,p) \succ (m-1,n,p+len(W))$, therefore we have $GoalRank(\delta') \succ GoalRank(\delta'')$.
- (Activate): (Activate) $i+1$ derivation step is of the form $\langle \{c\} \uplus G''' \mid Sn' \rangle \rightarrowtail_{\mathcal{G}} \langle \{c\#i\} \uplus G''' \mid \{c\#i\} \cup Sn' \rangle$. Since a CHR constraint c is traded for a numbered CHR constraint $c\#i$, we have $n' = n - 1$ and $p' = p + 1$. No equations are affected, hence $m' = m$. As such we have $GoalRank(\delta'') = (m, n-1, p+1)$ Since $(m,n,p) \succ (m,n-1,p+1)$, therefore we have $GoalRank(\delta') \succ GoalRank(\delta'')$.
- (Drop) $i+1$ derivation step is of the form $\langle \{c\#i\} \uplus G''' \mid Sn' \rangle \rightarrowtail_{\mathcal{G}} \langle G''' \mid Sn' \rangle$. Since numbered CHR constraint $c\#i$ is removed, hence we have $p' = p - 1$ No equations or CHR constraints are affected, hence $m = m'$ and $n' = n$. As such we have, $GoalRank(\delta'') = (m,n,p-1)$. Since $(m,n,p) \succ (m,n,p-1)$, therefore we have $GoalRank(\delta') \succ GoalRank(\delta'')$.

We have shown in all structural cases that $GoalRank(\delta') \succ GoalRank(\delta'')$. Combining with our assumption, we have $GoalRank(\delta) \succ GoalRank(\delta') \succ$

$GoalRank(\delta'')$. This means that CHR states are monotonically decreasing in goal ranks. Since $\succ$ is a well-founded total order with a minimal value $(0,0,0)$, we have proven that all goal-based derivations $\delta \rightarrowtail^*_{\mathcal{G}} \delta'$ consisting of only (Solve), (Activate) and (Drop) administrative transitions are finite. **(P1)**

Suppose that we have a concurrent derivation $\delta \rightarrowtail^*_{||\mathcal{G}} \delta'$ consisting of only administrative transitions that is infinitely long. By Theorem 3, all concurrent derivations $\delta \rightarrowtail^*_{||\mathcal{G}} \delta'$ have at least one corresponding sequential goal-based derivation $\delta \rightarrowtail^*_{\mathcal{G}} \delta'$. This would mean that sequential goal-based derivation $\delta \rightarrowtail^*_{\mathcal{G}} \delta'$ could be infinitely long as well. Yet, that would contradict with **(P1)**. Therefore it must be the case that all concurrent derivation $\delta \rightarrowtail^*_{||\mathcal{G}} \delta'$ consisting of only administrative transitions are finite.

□

Theorem 5 (Correspondence of Termination) For any initial CHR state $\langle G \mid \{\}\rangle$, any CHR state $\langle G' \mid Sn\rangle$ and a terminating CHR program $\mathcal{P}$,

if $\langle G \mid \{\}\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G' \mid Sn\rangle$ and $Final_{\mathcal{A}}(NoIds(G') \uplus DropIds(Sn))$
then $\langle G' \mid Sn\rangle \rightarrowtail^*_{||\mathcal{G}} \langle \{\} \mid Sn''\rangle$ and $DropIds(Sn'') = NoIds(G') \uplus DropIds(Sn)$

Proof: We first show that from $\langle G' \mid Sn\rangle$, there must be a finite sequence of concurrent derivations that leads to the terminal CHR State $\langle \{\} \mid Sn''\rangle$. Lemma 7 states that given $Final_{\mathcal{A}}(NoIds(G') \uplus DropIds(Sn))$ any valid concurrent derivation $\langle G' \mid Sn\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G'' \mid Sn''\rangle - (D)$ must not involve any applications of (Simplify) or (Propagate) transition rules. Hence (D) must only consist of administrative transitions (Solve), (Activate) and (Drop). From Lemma 8, we have that $\langle G' \mid Sn\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G'' \mid Sn''\rangle$ must be finite and terminating.

We now show that this terminal state $\langle G'' \mid Sn''\rangle$ is such that $G'' = \{\}$ and that $\langle \{\} \mid Sn\rangle$ corresponds to the final CHR abstract state $NoIds(G') \uplus DropIds(Sn)$. In other words, $GoalRanks(\langle G'' \mid Sn''\rangle) = (0,0,0)$[1] For any CHR state $\langle G' \mid Sn\rangle$ such that $GoalRanks(\langle G' \mid Sn\rangle) = (m,n,p)$, we can apply m number of (Solve) transitions $\langle G' \mid Sn\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G'_2 \mid Sn_2\rangle$ where $GoalRanks(\langle G'_2 \mid Sn_2\rangle) = (0,m',p)$. From here, we can apply m' number of (Activate) transitions $\langle G'_2 \mid Sn_2\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G'_3 \mid Sn_3\rangle$ where $GoalRanks(\langle G'_2 \mid Sn_2\rangle) = (0,0,p')$. Since we have $Final_{\mathcal{A}}(NoIds(G') \uplus DropIds(Sn))$, no (Simplify) or (Propagate) transition can apply for $\langle G' \mid Sn\rangle$ or any successor states, hence we can exhaustively apply (Drop) transitions $\langle G'_2 \mid Sn_2\rangle \rightarrowtail^*_{||\mathcal{G}} \langle \{\} \mid Sn''\rangle$ and naturely $GoalRanks(\langle \{\} \mid Sn''\rangle) = (0,0,0)$.

By Corollary 1, $\langle G \mid \{\}\rangle \rightarrowtail^*_{||\mathcal{G}} \langle G' \mid Sn\rangle \rightarrowtail^*_{||\mathcal{G}} \langle \{\} \mid Sn''\rangle$ means we have $NoIds(G) \rightarrowtail^*_{\mathcal{A}} NoIds(G') \uplus DropIds(Sn) \rightarrowtail^*_{\mathcal{A}} DropIds(Sn'')$.

[1] See proof in Lemma 8 for detailed description and definition of $GoalRanks$.

Since $Final_{\mathcal{A}}(NoIds(G') \uplus DropIds(Sn))$, no abstract semantics transition can apply from $NoIds(G') \uplus DropIds(Sn)$, hence we must have $DropIds(Sn'') = NoIds(G') \uplus DropIds(Sn)$. □