Efficient Compilation of Guarded Join-Patterns via Parallel Implementation of Constraint Handling Rules

Edmund S. L. Lam and Martin Sulzmann

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{lamsoonl,sulzmann}@comp.nus.edu.sg

Abstract. Parallelism is going mainstream and the demand for programs that scales well with parallelism will become ever more stronger. Hence to develope parallel programs of increasing complexity, comprehensive and efficient high level concurrency abstractions would soon become a necessity in existing mainstream programming languages. Our approach extends from a promising high-level concurrency abstraction known as join-patterns. In this paper, we introduce preliminary ideas of a novel approach to solve an existing open problem of join patterns: efficient implementation of guarded join-patterns, by means of a candidate far from the usual suspects, Constraint Handling Rules (CHR), a concurrent committed-choice constraint logic programming language. We introduce a prototype implementation based on CHR compilations of guarded join-patterns in Haskell, known as Haskell-Join-Rules.

1 Introduction

Parallelism is going mainstream and the demand for parallel programs that scales well with parallelism will become ever more stronger. Existing main-stream programming languages often provide concurrency primitives (locks, transactions, monitors, etc..) allowing a programmer to write programs that execute concurrently (consistently). However effciency and scalability (parallelism) of such programs often still rely entirely on the programmer's knowledge and awareness of the concurrency protocols used, as well as ability in micro-managing the concurrency primitives, a tedious and often error-prone task. Hence to develope parallel programs, comprehensive and efficient high-level concurrency abstractions would soon become a necessity in existing programming languages to battle against the ever increasing complexity and demand for parallel applications.

There are numerous calculi and concurrent programming models to support concurrent programming. A particular fruitful and promising model appears to be the join calculus [5] which provides the basis for the concurrency abstractions found in numerous existing implementations (eg. JoCaml [2], Polyphonic C# [1], Join Java [25]). In join calculus, concurrency is expressed via sets of multi-headed reduction rules known as *join-patterns*. Join-patterns are declarative in nature

and easy to understand, for instance the following shows a typical example of a join-pattern definition in JoCaml.

```
let buffer =
  def get() | put(i) = reply i to get
  in reply get,put to buffer
```

Join-patterns like this provide high-level coordination of concurrent processes, without the need of explicit micro-management of concurrency primitives. This particular join-pattern implements a concurrent unbounded buffer. A call to `buffer` returns two operations `get` and `put`, which can be viewed as shared channels containing queued messages symbolically representing calls to them from concurrent processes. A Synchronous `get()` call adds itself to the `get` channel and blocks until it is matched with a join-pattern. A Asynchronous `put(i)` call on the other hand, adds itself to the `put` channels and proceeds immediately after being called. The buffer join-pattern is triggered when a `get()` call and a `put(i)` call are matched together, resulting in the consumption of the two messages (from the channels) and execution of the body `reply i to get`, which binds the return value of `get` to value `i`.

A particularly useful extension not supported by standard join-pattern implementations is *guarded join-patterns* which introduces guarded constraints to join-pattern headers providing selectivity of the matching of join function calls. Consider the following example of a guarded join-pattern in pseudo syntax:

```
let selectbuffer =
  def getGT(j) | put(i) when (j > i) = reply i to getGT
  in reply getGT,put to selectbuffer
```

This models a *selective buffer* join program. A `getGT(n)` call can only trigger a join-pattern if it is matched with a `put(m)` call such that `n > m`. In general, a guarded join-pattern is triggered by a set of matching calls only if all specified guard constraints under the bindings implied by the matching call variables are satisfied. Adding guard conditions to join-patterns is a non-trivial extension, mainly because an efficient implementation involves a highly optimized combinatorial search for multi-object patterns executed in parallel with one another. As such, this problem still remains an open issue to date.

In this proposal, we introduce preliminary ideas of a novel approach to solve this problem. Central to our approach is a candidate far from the usual suspects: Constraint Handling Rules [7] (CHR), a concurrent rule-based constraint logic programming language. CHRs are essentially multi-headed guarded reduction rules that exhaustively rewrite multi-sets of constraints (atomic formulae) into simpler ones, remarkably similar to the matching problem faced by guarded join-patterns. The idea is to translate join-patterns into CHR rules, there by allowing a state-of-the-art CHR solver compute and trigger guarded join-patterns efficiently. This also gives us a formal model (CHR semantics) to precisely define the behaviour of guarded join patterns. For instance, the selective buffer join-pattern can be compiled in the following CHR:

$$\texttt{selectbuffer} \ @ \ \texttt{GetGt(j,o),Put(i)} \Longleftrightarrow \texttt{j > i} \mid \texttt{o = i}$$

This CHR rule simply state that the heads of the rule (`GetGt(j,o),Put(i)`) can be rewritten by `o = i` (binds `o` to value of `i`), only if `j > i`. Note that we use `o` as a variable with output mode in which the caller of `GetGt(j,o)` waits for an answer. A CHR rule execution of this rule is analogous to the triggering of the selective buffer join-pattern.

While the many years of research into efficient CHR systems have yielded highly optimized implementations [11, 3, 21], all of which are single-threaded and do not exploit parallelism. This means that CHR rule executions are sequential and interleaved. We strongly believe that the key to an efficient implementation of guarded join patterns is a parallel CHR system which exploits existing CHR optimizations, while allowing parallel rule execution.

Our previous works and contributions [13, 17, 14, 16, 15, 12] thus far include establishing a parallel CHR operational semantics and implementation, as well as a prototype guarded join-pattern language extension to the functional programming language Haskell [10]. These are summarized by the following:

- In [13] we introduce an agent oriented refinement of CHR semantics, Monadic Action CHR Semantics, which provides a programmer/agent an external interface to explicitly order CHR derivations that representing actions. This is further explored in [17] where a prototype implementation in the Haskell is highlighted.
- We introduce in [14] the parallel CHR semantics and a simplified implementation (propositional CHR) in Haskell. Preliminary experimental results here uncover the presence of significant performance gains when executed over a multicore system.
- In [16, 15], we refined our work in parallel Constraint Handling Rules which addresses the full CHR semantics, while also providing a description of the compilation of parallel CHR in Haskell.
- Recent works in [12] introduces our prototype system known as *Haskell-Join-Rules* which extends Haskell with guarded join-patterns based on the parallel CHR semantics.

This proposal is organized as followings: Section 2 reviews the related works. Section 3 provides a brief introduction on Join-Calculus, CHRs and Haskell. The technical component of this proposal comprises of Section 4, 5 and 6, which highlights the parallel CHR operational semantics, parallel CHR implementation in Haskell (GHC) and the language design and CHR compilation scheme of guarded join-patterns in our protoype Haskell-Join-Rules, respectively. Section 7 highlights our targeted future works and projected timeline to completion. We conclude in Section 8.

2 Related Works

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. Naming the key players, we start off with JoCaml [2] the join-pattern language extension of OCaml. JoCaml introduces basic join-patterns, including join-patterns with multiple synchronous calls, hence allowing rendervous style synchronization (eg. 2 way synchronizations). Polyphonic $C\#$ [1] is a language extension of $C\#$ with join-patterns. Polyphonic $C\#$ disallows join-patterns with multiple synchronous calls to avoid explicit spawning of threads in join-pattern bodies. By restricting join-patterns to strictly contain at most one synchronous call, ambiguilties concerning which thread should execute the join-pattern body are avoided (since there are at most one synchronous thread blocking on each join pattern). A java implementation of join-pattern is also available in [24] which is similar to Polyphonic $C\#$. A Hardware implementation of Join Java is explored in [**?**], for higher efficiency in triggering join-patterns with hardware support. Yet these implementations are only based on standard join-patterns without guards. It is difficult to introduce guards into these existing implementations as they are based on state machine compilation schemes highlighted in [4]. Such compilation schemes enforce sequential triggering of join-patterns and it is not directly obvious how they can support the parallel combinatorial search for multi-object patterns to execute guarded join-patterns.

In [1], the authors briefly mentioned a prototype extension of Polyphonic $C\#$ with guarded join-patterns. As admitted by the authors, the 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. [20] introduces join-patterns to $C\#$, while [23] 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 [23], 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 [**?**,**?**]. [**?**] 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 [23], but also does not address efficient compilation of guarded join-patterns in general. [**?**] 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. The authors however, do not provide a formal model in which their implementation is based on.

Apart from CHR, there are other candidates (constraint logic programming languages, rule bases systems, databases) which are possibly applicable as solutions to the problem of efficient guarded join-pattern compilation. For instance, the *Rete algorithm* [**?**] used by many production rule systems is an efficient approach for computing multi-object pattern matchings by incrementally building matches from partial matches. However, such incremental matching algorithms are not entirely suitable in the context of join-patterns due to a fundamental problem: Such approaches incrementally and exhaustively accumulates partial matchings to build complete matchings. In join-patterns messages are consumed (deleted) immediately when they match with a join-pattern, making all matchings involve these deleted constraints obselete and hence constituting wasted computations. Solutions from the database context (in building joins between tables) also suffers from the same redundancy. The CHR refined operational semantics [3] on the other hand defines a goal directed execution strategy for processing multi-set constraint pattern matching only by demand, since most of the CHR rule rewritings involve deleting matched constraints. This makes CHR multi-set constraint matching algorithms highly appropriate for triggering guarded join-patterns.

Logic programming languages like prolog [**?**] or CLP(R) [**?**] provides the base matching facilities in which such multi-object pattern matching algorithms can be implemented, but it's backward chaining backtracking semantics lacks the close resemblance of the CHR forward chaining commit choice semantics with the join-pattern semantics.

3 Background

To keep this proposal self-contained, this section provides a brief introduction on Join-Calculus, CHRs and Haskell. For a more thorough introduction of the respective topics, please refer to [6], [8] and [19]. Readers who are familar with any of the related fields are welcomed to skip the respective background information. Before we begin, we define some notions that will be used through the paper: $\uplus$ denotes multi-set union and $\wedge$ to denote conjunction among predicates. Sequences are denoted by $[c \mid C]$ where c is the head item and C the tail of the sequence, or simply just $\bar{c}$. Concatanation of two sequences S_1 and S_2 is denoted by $S_1 + +S_2$. Depending on the context we treat substitutions θ, ϕ as conjunctions of equations. We write $\equiv$ to denote syntactic equivalence and $\supset$ to denote Boolean implication. Lastly, empty sets and sequences are denoted by $\emptyset$ and ϵ.

3.1 Join-Calculus: JoCaml

We review the syntax and semantics of Join-Calculus. Join-Calculus systems traditionally use a model of concurrency based on the chemical abstract machine. We focus on the core Join-Calculus language in JoCaml syntax, since it's host language is very much similar to Haskell. Much of the materials in this section is adapted from [4]. The core Join-Calculus language is summarized by

the following:

$$\begin{array}{lll} P ::= c(\bar{x}) & D ::= J = P & J ::= c(\bar{x}) \\ \quad \mid \texttt{let}\ D\ \texttt{in}\ P & \quad \mid D\ \texttt{and}\ D & \quad \mid J \mid J \\ \quad \mid P \mid P & & \end{array}$$

A process P can be a message $c(\bar{x})$ with name c and arguments $\bar{x}$, a `let` definition of D in P, or a parallel composition of two other processes. A definition D is either a join definition, which is a set of message patterns J paired with a guarded process P, or the conjunction of two definitions. Variables in J are only allowed to appear once.

The Join-Calculus semantics is defined by the reflexive chemical abstract machine (RCHAM) [6], which specifies transformations over a chemical soup $\mathcal{D} \models \mathcal{P}$ where $\mathcal{D}$ and $\mathcal{P}$ are multi-sets over active definitions D and running processes P. These transformations are divided into two types of rules, structural ($\rightleftharpoons$) which are reversible and describes syntactical rearrangements of terms, and reduction ($\rightarrow$) which describes the actual triggering of join definitions. The following summarizes the two kinds of rules.

RCHAM Structural Rules: $(\mathcal{D} \models \mathcal{P}) \rightleftharpoons (\mathcal{D}' \models \mathcal{P}')$

$$(\quad \models P_1 \mid P_2) \rightleftharpoons (\quad \models P_1, P_2) \quad \texttt{(S-Par)}$$

$$(D_1\ \texttt{and}\ D_2 \models\ ) \rightleftharpoons (D_1, D_2 \models\ ) \quad \texttt{(S-And)}$$

$$\frac{D' \equiv \mathit{rename}(D)}{(\quad \models \texttt{let}\ D\ \texttt{in}\ P) \rightleftharpoons (D' \models P)}\ \texttt{(S-Def)}$$

RCHAM Reduction Rules: $(\mathcal{D} \models \mathcal{P}) \rightarrow (\mathcal{D}' \models \mathcal{P}')$

$$\frac{\exists\theta \quad J' \equiv \theta(J)}{(J = P \models J') \rightarrow (J = P \models \theta(P))}$$

The $\mathit{rename}(D)$ procedure in the `S-Def` rule performs α-renaming of D with fresh variables to enforce lexical scoping.

On top of the core calculus, Join programming languages introduces the notion of synchronous messages. Synchronous messages function the same way as asynchronous ones, except that they expect return results. Hence a process calling a synchronous message will block until a time when a join definition is triggered and a result is written to it. The JoCaml System implement this as a continuation passing style model, simply by appending a continuation variable to all synchronous messages, in which the process calling it will wait on. A join pattern call `c()` in JoCaml is synchronous if there exists a `reply` construct within the join-pattern body that replies a value `v` to it (ie. `reply v to c()`), otherwise it is asynchronous.

$$\boxed{\langle G, S, B\rangle_n \rightarrowtail_{w_t} \langle G', S', B'\rangle_{n'}}$$

1. **Solve:**

$$\frac{b \texttt{ is a Built-in Constraint}}{\langle \{b\} \uplus G, S, B\rangle_n \rightarrowtail \langle G, S, b \wedge B\rangle_n}$$

2. **Introduce:**

$$\frac{c \texttt{ is a CHR Constraint}}{\langle \{c\} \uplus G, S, B\rangle_n \rightarrowtail \langle G, \{c\#n\} \uplus S, B\rangle_{(n+1)}}$$

3. **Apply:**

$$\frac{\begin{array}{c}\exists (r \ @ \ H_1' \setminus H_2' \Leftrightarrow g \mid C) \wedge \exists \theta \texttt{ such that} \\ cons(H_1) \equiv \theta(H_1') \wedge cons(H_2) \equiv \theta(H_2') \wedge \mathcal{CT} \models B \supset \exists_r(\theta \wedge g)\end{array}}{\langle G, H_1 \uplus H_2 \uplus S, B\rangle_n \rightarrowtail \langle \theta(C) \uplus G, H_1 \uplus S, B\rangle_n}$$

Fig. 1. Theoretical Semantics of CHR w_t

3.2 Constraint Handling Rules (CHR)

We review the syntax and theoretical semantics of CHR. CHRs essentially describe the multi-set rewriting of constraints, which are either built-in constraints or CHR constraints. Built-in constraints are defined by a built-in theory $\mathcal{CT}$ (eg. Herbrand, or integer equality) while CHR constraints are defined by the user. These rewritings are specified by CHR rules r of the general form:

$$r \ @ \ H_1 \backslash H_2 \Longleftrightarrow g \mid C$$

where rule *heads* H_1 and H_2 consist of CHR constraints, the *guard* g is a built-in constraint and the rule *body* C consists of CHR and built-in constraints.

A CHR program is a set of CHR rules. To distinguish between different copies of equivalent CHR constraints in a multiset, we identify a CHR constraint c with a uniquely assigned identifier n, denoted by $c\#n$. Such are known as numbered CHR constraints. We denote the CHR state as the tuple $\langle G, S, B\rangle_n$ abreviated by σ, where G is a multiset of CHR and built-in constraints (Goals), S is a multiset of numbered CHR constraints (CHR store), B is a logic conjunction of built-in constraints (built-in store) and n an integer. The goals represents the constraints yet to be processed, while the stores are constraints which have been and awaits to be matched. The integer n simply provides a source of new identifiers for numbering CHR constraints. We define two auxiliary functions, $chr(C)$ and $builtin(C)$ which returns all CHR and built-in constraints of C respectively. $cons(c\#n)$ and $id(c\#n)$ returns the CHR constraint and identifier respectively. The CHR theoretical semantics (denoted w_t) is formally presented in 1.

The theoretical semantics w_t is driven by three kinds of derivation rules. The (Solve) rule adds a new built-in constraint from the goals to the built-in store, while (Introduce) rule labels and adds a new CHR constraint to the CHR store. The (Apply) rule essentially states the application of a CHR rule r, given that

Gcd Rules:

$$\texttt{gcd1@ gcd(0)} \Leftrightarrow \texttt{true}$$
$$\texttt{gcd2@ gcd(n)\gcd(m)} \Leftrightarrow m >= n \;\&\&\; n > 0 \mid \texttt{gcd(m-n)}$$

Example Derivation:

$$\begin{array}{ll} & \langle \{Gcd(2), Gcd(2)\}, \emptyset \rangle_1 \\ \rightarrowtail_{intro} & \langle \{Gcd(2)\}, \{Gcd(2)\#1\} \rangle_2 \\ \rightarrowtail_{intro} & \langle \emptyset, \{Gcd(2)\#1, Gcd(2)\#2\} \rangle_3 \\ \rightarrowtail_{apply} & \langle \{Gcd(0)\}, \{Gcd(2)\#2\} \rangle_3 \\ \rightarrowtail_{intro} & \langle \emptyset, \{Gcd(2)\#2\} \rangle \end{array}$$

Fig. 2. Greatest Common Divisor CHR Program

we can find constraints in the store $H_1 \uplus H_2$, matching rule heads H_1' and H_2'. By matching, we mean that there exists a substitution θ which applied to H_1' and H_2', makes them syntactically equivalent to H_1 and H_2. Matching substitution θ and rule guard condition g must of course be derivable by the current built-in store B under the built-in theory $\mathcal{CT}$. The result of the rule application is the removal of H_2 from the current CHR store, and the introduction of constraints in rule body C into their respective stores. The exhaustive application of CHR derivations is denoted by $\sigma \rightarrowtail^*_{w_t} \sigma'$ where either no rules are applicable in σ' or built-in store in σ' is contridicting. Note that at times, we omit the built-in store if it is not crucial for a particular presentation, and also similarily for the constraint numbering. Figure 2 shows an example of a CHR program and CHR exhaustive derivation. For clarity, we annotate each derivation explicitly with the kind of rule derivation taken.

3.3 Haskell

In this section we provide a brief review on Haskell [19]. Haskell is a strongly-typed lazy functional programming language with powerful concurrency abstractions and imperative-style programming features. It's strongly-typed system also makes it easy to identify the meaning of a function by just examining its type signature. On top of this, Haskell provides features like higher ordered functions and polymorphic functions. For instance, the following is the type signature of a polymorphic library function provided by most Haskell implementations:

```
map :: (a -> b) -> [a] -> [b]
```

The `map` function takes in a function of type `a -> b` and a list (sequence) of `a`'s as arguments, and maps the function on each element of the list to produce a list of `b`'s. This function can be re-used in countless numbers of ways, for instance running the following:

```
map (\a -> a+1) [1,2,3,4]
```

simply maps an increment function on the list `[1,2,3,4]` resulting to `[2,3,4,5]`. These features allow us to implement highly re-usable library codes. Haskell also

allow the programmer to define overloaded functions by means of it's type class mechanism, for instance the following type class declares the class of equality functions for some type parameter `a`, accompanied by a primitive instance for booleans and recursive instance for lists:

```
class Eq a where
    (==) :: a -> a -> Bool

instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False

instance Eq a => Eq [a] where
  [] == [] = True
  (a:as) == (b:bs) = (a == b) && (as == bs)
  _ == _ = False
```

Lazy evaluation is the technique of delaying computations until the time where the result of the computation is required. This allow us to program in a purely declarative manner, without the worry of any runtime performance penalties. For instance consider,

```
integers :: Int -> [Int]
integers i = i:(integers (i+1))

head :: [a] -> a
head (a:_) = a

zero = head (integers 0)
```

The function `integers` declaratively defines the list of all non-negative integers, seemingly in a non-terminating way. `head` simply returns the first element of a list, ignoring everything else. With lazy evaluation, applying `integers 0` to the `head` function results terminally in the singleton list containing a `'0'` (other integers more than zero are not required, thus not evaluated).

The Haskell codes we have shown so far contains only pure functions, in that they do not have side effects, like reading/writing values from/to mutable memory locations. Haskell provides *monads* as a mechanism to allow us to define operations with side-effects. For instance, `IO` (input/output) is an instance of monads defined in standard Haskell libraries. A function of type `IO a` performs some input/output action (which may contain side-effects) and produces a value of type `a`. Note that we shall refer to such monadic functions as *operations* or *actions*. A simple example of `IO` operation is `putStr` of the type signature:

```
putStr :: String -> IO ()
```

This operation takes a string (sequence of characters) and prints it on the a standard output terminal. The return type `()` denoted *unit* type, is similar to the *void* return type of C or Java. Haskell provides the `do` syntax that composes together monadic functions of the same monad type. For example,

```
put2StrAndInc :: String -> String -> Int -> IO Int
put2StrAndInc s1 s2 i = do { putStr s1
                           ; putStr s2
                           ; return (i+1) }
```

```
data TVar -- Transactional variables

data STM a -- Software transactional memory monad.

-- Basic TVar interfaces
newTVar :: STM (TVar a)              readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

-- STM monad interfaces
atomically :: STM a -> IO a          forkIO :: IO () -> IO ThreadId
```

Fig. 3. GHC Software Transactional Memory Interfaces

This operation prints two given strings on screen, increments and return the given integer in the given sequential order. The `return` function of type `a -> IO a` has no side-effects and simply lifts the value of `a` into the monadic type `IO a`. Note that `do` and `return` are overloaded and used by monads of other types.

Haskell (specifically, Glasgow Haskell Compiler [10] (GHC)) provides a new and promising concurrency primitive, software transactional memory [18] (STM) which provides concurrent and composable shared memory access inspired by transactional management protocols of the database systems. Figure 3 shows the interfaces of STM. A transactional variable `TVar` is only accessible in the `STM` monad via the interfaces `newTVar`, `readTVar` and `writeTVar` which creates, read and write into a transactional variable respectively. STM operations can be composed the same way any monadic operation in haskell is composed. For instance, the following increments an integer transactional variable by read it's value and writing the successor back into it:

```
incTVar :: TVar Int -> STM ()
incTVar tv = do { x <- readTVar tv
                ; writeTVar tv (x+1) }
```

STM operations can be safely executed in parallel by concurrently running IO threads with the `atomically` operation. This `IO` operation takes a `STM` operation and executes it *atomically* as a `IO` computation. By atomically, we mean that it's side effects happens as though they were one atomic action. Consider the following program:

```
manyIncs :: TVar Int -> Int -> IO ()
manyIncs _ 0 = return ()
manyIncs x n = do { forkIO (atomically (incTVar x))
                  ; manyIncs x (n-1) }
```

Given a transactional variable `x` that initially contains `0`, STM protocols ensure that a call to `manyIncs x m` will always result to `x` containing the value `m` when all `IO` threads created terminate.

$$\frac{\langle \{\texttt{Gcd(2)},\texttt{Gcd(4)}\}, \emptyset\rangle \rightarrowtail^* \langle \emptyset, \{\texttt{Gcd(2)}\}\rangle \;||\; \langle \{\texttt{Gcd(5)},\texttt{Gcd(10)}\}, \emptyset\rangle \rightarrowtail^* \langle \emptyset, \{\texttt{Gcd(5)}\}\rangle}{\langle \{\texttt{Gcd(2)},\texttt{Gcd(4)},\texttt{Gcd(5)},\texttt{Gcd(10)}\}, \emptyset\rangle}$$
$$\begin{aligned} &\rightarrowtail \quad ... \\ &\rightarrowtail \quad \langle \emptyset, \{\texttt{Gcd(2)},\texttt{Gcd(5)}\}\rangle \\ &\rightarrowtail^* \langle \emptyset, \{\texttt{Gcd(1)}\}\rangle \end{aligned}$$

Fig. 4. Concurrency in CHR

4 Parallel CHR Operational Semantics

This section highlights our works in [14, 16]. The CHR semantics is concurrent in that rule executions can occur concurrently as long as they do not match on over lapping constraints. This means that in theory, we can naively compose CHR derivations that do not overlap. For instance, Figure 4 shows two separate CHR derivations of the Gcd program, one finding gcd of 2 and 4, the other of 5 and 10, resulting to 2 and 5 respectively. In theory we can compose the two derivations together, yielding $\{\texttt{Gcd(2)},\texttt{Gcd(5)}\}$ which will evaluate to $\{\texttt{Gcd(1)}\}$. Such composition and non-interference between derivations strongly suggests the possibility in parallel executions of CHR derivations by multiple threads, hence allowing CHR performance to scale with the number of active processors. This section highlights our parallel CHR operational semantics that aims to provide a formal and operational description of parallel CHR derivations by multiple solver threads.

The most straight forward way to implement a parallel CHR system is probably to spawn an active thread for each goal constraint in the program. Each thread is tasked to find partners for it's goal constraint that completes a rule head match. This unfortunately is not a good approach for two main reasons:

- Each goal attempting to match with a n headed rule is likely to be competing with $n-1$ other goals and ultimately only one will ever succeed, while the rest must abort.
- Total number of goals is unbounded and changes during runtime. Spawning threads depending on the number of goals with no respect for system resources (eg. number of processors) would not scale well in general.

Our approach is to define a CHR operational semantics that specifies parallel CHR rule derivation with a bounded number of active solver threads. This is done by explicitly modelling the number of solver threads active as a multiset known as the *thread pool*, denoted by TP.

Figure 5 shows the interleaved CHR operational semantics w_i which specifies the CHR derivations of a single solver thread. Note that w_i is an adaptation of the refined CHR operational semantics w_r defined in [3]. A CHR state in w_i is the tuple $\langle (t, n, g), G, S, B\rangle$. G, S and B are the goals CHR and Built-in Store as before, except that they are to be shared by multiple solver threads. Threads are formally represented by a tuple (t, n, g) where t is an integer uniquely assigned

– **Schedule:**

$$\langle (t, n, \emptyset), g \uplus G, S, B\rangle \rightarrowtail_{w_i} \langle (t, n, \{g\}), G, S, B\rangle$$

– **Solve:**

$$\frac{\begin{array}{c} b \texttt{ is a built-in constraint} \\ W \equiv \{c \mid c \in S \wedge \mathcal{CT} \not\models ground(c, B) \wedge \exists (r @ H_1' \backslash H_2' \Leftrightarrow g \mid C) \\ \texttt{such that } \exists \theta\ \theta(c) \in cons(H_1' \uplus H_2') \wedge \mathcal{CT} \not\models (B \supset \exists_B \theta \wedge g) \wedge \\ \mathcal{CT} \models (b \wedge B \supset \exists_B \theta \wedge g)\} \end{array}}{\langle (t, n, \{b\}), G, S, B\rangle \rightarrowtail_{w_i} \langle (t, n, \emptyset), W \uplus G, S, b \wedge B\rangle}$$

– **Introduce:**

$$\frac{c \texttt{ is a CHR constraint}}{\langle (t, n, \{c\}), G, S, B\rangle \rightarrowtail_{w_i} \langle (t, n+1, \{c\#(t, n)\}), G, \{c\#(t, n)\} \uplus S, B\rangle}$$

– **Simplify:**

$$\frac{\begin{array}{c} \exists (r\ @\ H_1' \setminus H_2', c', H_3' \Leftrightarrow g \mid C) \wedge \exists \theta \texttt{ such that} \\ cons(H_1) \equiv \theta(H_1') \wedge cons(H_2) \equiv \theta(H_2') \wedge cons(H_3) \equiv \theta(H_3') \\ \wedge c \equiv \theta(c') \wedge \mathcal{CT} \models B \supset (\theta \wedge g) \end{array}}{\begin{array}{c} \langle (t, n, \{c\#(t, m')\}), G, \{c\#(t, m')\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B\rangle \\ \rightarrowtail_{w_i} \\ \langle (t, n, \emptyset), \theta(C) \uplus G, H_1 \uplus S, B\rangle \end{array}}$$

– **Propagate:**

$$\frac{\begin{array}{c} \exists (r\ @\ H_1', c', H_2' \setminus H_3' \Leftrightarrow g \mid C) \wedge \exists \theta \texttt{ such that} \\ cons(H_1) \equiv \theta(H_1') \wedge cons(H_2) \equiv \theta(H_2') \wedge cons(H_3) \equiv \theta(H_3') \\ \wedge c \equiv \theta(c') \wedge \mathcal{CT} \models B \supset (\theta \wedge g) \end{array}}{\begin{array}{c} \langle (t, n, \{c\#(t, m')\}), G, \{c\#(t, m')\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B\rangle \\ \rightarrowtail_{w_i} \\ \langle (t, n, \{c\#(t, m')\}), \theta(C) \uplus G, \{c\#(t, m')\} \uplus H_1 \uplus H_2 \uplus S, B\rangle \end{array}}$$

– **Drop**

$$\frac{\begin{array}{c} \neg \exists (r\ @\ H_1' \setminus H_2' \Leftrightarrow C) \texttt{ such that} \{c\#(t, m')\} \uplus H_1 \uplus H_2 \subseteq S \wedge \\ \exists \theta\ (\{c\} \uplus cons(H_1) \uplus cons(H_2)) \equiv \theta(H_1' \uplus H_2') \wedge \mathcal{CT} \models B \supset (\theta \wedge g) \end{array}}{\langle (t, n, \{c\#(t, m')\}), G, S, B\rangle \rightarrowtail_{w_i} \langle (t, n, \emptyset), G, S, B\rangle}$$

Fig. 5. Interleaving CHR Operational Semantics w_i

to each thread, n is an integer counter. This means that stored constraints are now uniquely identified by a tuple (t, n) . g is a set of goal constraints which is either a singleton set or empty set. Integers t and n provide each thread a local name supply for numbering constraint without synchronizing with other threads, while g represents the current goal the solver thread is working on, analogous to the active constraints in w_r.

The w_i operational semantics introduces the (Schedule) rule, in which a fresh goal is fetched from the shared goals. The (Solve) rule adds a Built-in constraint b into the Built-in store. Note that since this is an operational semantics, we are explicit about reprocessing stored CHR constraints that may trigger rules due to the introduction of b. This reactivation of constraints is modelled by a wake up policy W similar to what is done in w_r. The (Introduce) rule 'broadcasts' the current goal CHR constraint c to store by tagging it with the thread's local name supplies. Note that once this happens, the solver thread no longer has execlusive ownership of c as other threads are free to match it with other partner constraints

and rewrite it. (Simplify) and (Propagate) are derived from the (Apply) rule of w_t, only difference being that we can more explicit about the active goal matching a simplify or propagate head of a rule. In the latter case, the goal is kept even after trigger of the rule, since it is propagated and not deleted during the rewriting step. Lastly, the (Drop) rule is taken when the current goal fails to trigger any CHR rules. Note that w_i differs from the refined operational semantics w_r [3] in another major way: the order in which rules are tried is not specified, hence treated as a implementational design choice. Our decision to keep this abstract will be discussed later in this section.

– **Parallel Rewriting:**

$$\frac{\begin{array}{c}\langle (t,n,E), G_1 \uplus G_2 \uplus G, S_1 \uplus S_2 \uplus S, B\rangle \rightarrowtail_{w_i} \langle (t,n',E'), G_1' \uplus G_2 \uplus G, S_1' \uplus S_2 \uplus S, B_1 \wedge B\rangle \\ \langle TP, G_1 \uplus G_2 \uplus G, S_1 \uplus S_2 \uplus S, B\rangle \rightarrowtail_{w_{||}} \langle TP', G_1 \uplus G_2' \uplus G, S_1 \uplus S_2' \uplus S, B_2 \wedge B\rangle \\ \mathcal{CT} \models B_1 \wedge B_2 \wedge B\end{array}}{\begin{array}{c}\langle \{(t,n,E)\} \uplus TP, G_1 \uplus G_2 \uplus G, S_1 \uplus S_2 \uplus S, B\rangle \\ \rightarrowtail_{w_{||}} \\ \langle \{(t,n',E')\} \uplus TP', G_1' \uplus G_2' \uplus G, S_1' \uplus S_2' \uplus S, B_1 \wedge B_2 \wedge B\rangle\end{array}}$$

– **Interleaved Rewriting:**

$$\frac{\langle (t,n,E), G, S, B\rangle \rightarrowtail_{w_i} \langle (t,n',E'), G', S', B'\rangle}{\langle \{(t,n,E) \uplus TP, G, S, B\}\rangle \rightarrowtail_{w_{||}} \langle \{(t,n',E')\} \uplus TP, G', S', B'\rangle}$$

Fig. 6. Parallel CHR Operational Semantics $w_{||}$

Figure 6 shows the parallel CHR operational semantics $w_{||}$ which essentially is the parallel composite of the w_i operational semantics. A state here is the tuple $\langle TP, G, S, B\rangle$, where G, S and B are the shared goals, CHR and Built-in store and TP is the thread pool. The (Parallel Rewriting) rule specifies the parallel derivations of CHR rule executions over non-overlapping portions of the shared CHR store and different goals. Note that we reply heavily on the monotonicity of CHRs (adding new CHR constraints will not suppress any rule execution) and the monotonicity of the built-in theory (non-contradicting entailment of the built-in store never suppress any rule execution). These are properties satisfied by the CHR semantics that allows parallelism. (Interleaved Rewriting) rule specifies the case where parallelism is not exploited and a single solver thread takes a rule execution step.

Figure 7 shows an example of a $w_|$ | derivation of the Gcd program (Figure 2) in finding for the greatest common divisor of 4, 2 and 5. In this example, we consider 2 solver threads labelled 1 and 2. We omit the built-in store and label each derivation step with the transition rules taken.

4.1 Parallelism and Indeterminism

The $w_i/w_{||}$ semantics is in-deterministic in four main ways:

– Goals are scheduled for execution by solver threads in a unspecified order. (In *Schedule* transition rule of w_i)

$$
\begin{array}{ll}
 & \langle\{(1,1,\emptyset),(2,1,\emptyset)\},\{Gcd(4),Gcd(2),Gcd(5)\},\emptyset\rangle \\
\rightarrowtail_{Sche||Sche} & \langle\{(1,1,\{Gcd(4)\}),(2,1,\{Gcd(2)\})\},\{Gcd(5)\},\emptyset\rangle \\
\rightarrowtail_{Intro||Intro} & \langle\{(1,2,\{Gcd(4)\#(1,1)\}),(2,2,\{Gcd(2)\#(2,1)\})\}, \\
 & \{Gcd(5)\},\{Gcd(4)\#(1,1),Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Simp||Drop} & \langle\{(1,2,\emptyset),(2,2,\emptyset)\},\{Gcd(2),Gcd(5)\},\{Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Sche||Sche} & \langle\{(1,2,Gcd(2)),(2,2,Gcd(5))\},\emptyset,\{Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Intro||Intro} & \langle\{(1,3,Gcd(2)\#(1,3)),(2,3,Gcd(5)\#(2,3))\}, \\
 & \emptyset,\{Gcd(2)\#(2,1),Gcd(2)\#(1,3),Gcd(5)\#(2,3)\}\rangle \\
\rightarrowtail_{Simp||Simp} & \langle\{(1,3,\emptyset),(2,3,\emptyset)\},\{Gcd(0),Gcd(3)\},\{Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Sche||Sche} & \langle\{(1,3,\{Gcd(0)\}),(2,3,\{Gcd(3)\})\},\emptyset,\{Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Intro||Intro} & \langle\{(1,4,\{Gcd(0)\#(1,3)\}),(2,4,\{Gcd(3)\#(2,3)\})\}, \\
 & \emptyset,\{Gcd(2)\#(2,1),Gcd(0)\#(1,3),Gcd(3)\#(2,3)\}\rangle \\
\rightarrowtail_{Simp||Simp} & \langle\{(1,4,\emptyset),(2,4,\emptyset)\},\{Gcd(1)\},\{Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Sche} & \langle\{(1,4,Gcd(1)),(2,4,\emptyset)\},\emptyset,\{Gcd(2)\#(2,1)\}\rangle \\
\rightarrowtail_{Intro} & \langle\{(1,5,Gcd(1)\#(1,4)),(2,4,\emptyset)\},\emptyset,\{Gcd(2)\#(2,1),Gcd(1)\#(1,4)\}\rangle \\
\rightarrowtail_{Prop} & \langle\{(1,5,Gcd(1)\#(1,4)),(2,4,\emptyset)\},\{Gcd(1)\},\{Gcd(1)\#(1,4)\}\rangle \\
\rightarrowtail_{Drop||Sche} & \langle\{(1,5,\emptyset),(2,4,Gcd(1))\},\emptyset,\{Gcd(1)\#(1,4)\}\rangle \\
\rightarrowtail_{Intro} & \langle\{(1,5,\emptyset),(2,5,Gcd(1)\#(2,4))\},\emptyset,\{Gcd(1)\#(1,4),Gcd(1)\#(2,4)\}\rangle \\
\rightarrowtail_{Simp} & \langle\{(1,5,\emptyset),(2,5,\emptyset\},\{Gcd(0)\},\{Gcd(1)\#(1,4)\}\rangle \\
\rightarrowtail_{Sche} & \langle\{(1,5,Gcd(0)),(2,5,\emptyset\},\emptyset,\{Gcd(1)\#(1,4)\}\rangle \\
\rightarrowtail_{Intro} & \langle\{(1,6,Gcd(0)\#(1,5)),(2,5,\emptyset\},\emptyset,\{Gcd(1)\#(1,4),Gcd(0)\#(1,5)\}\rangle \\
\rightarrowtail_{Simp} & \langle\{(1,6,\emptyset),(2,5,\emptyset\},\emptyset,\{Gcd(1)\#(1,4)\}\rangle
\end{array}
$$

Fig. 7. Example CHR $w_{||}$ Derivations

- Rule head matching is executed in an unspecified order. ie. Order of which rule head is matched first and order of which matching partner constraint is tried are unspecified. (In *Simplify* and *Propagate* rules of w_i)
- Order in which CHR rules are tried is unspecified. (In *Simplify* and *Propagate* rules of w_i)
- Interleaved and Parallel rewriting transition rules of $w_{||}$ are overlapping.

Goals are processed in a unordered manner, allowing each solver thread to avoid interleaving on a single shared ordered goal store, which ultimately results in the use of some non-random access shared data structure (eg. queue, stack, etc..). It is important that solver threads as much as possible try to match non-overlapping partner constraints to reduce competition between conflicting rule executions. Consider the following derivations:

Parallel Derivation with distinct partner constraints: $Gcd(4)\#l_3$ and $Gcd(5)\#l_4$

$$
\begin{array}{ll}
 & \langle\{(t,n,Gcd(2)\#l_1),(t',n',Gcd(3)\#l_2)\},\emptyset,\{Gcd(4)\#l_3,Gcd(5)\#l_4,Gcd(2)\#l_1,Gcd(3)\#l_2\}\rangle \\
\rightarrowtail_{Prop||Prop} & \langle\{(t,n,Gcd(2)\#l_1),(t',n',Gcd(3)\#l_2)\},\{Gcd(4-2),Gcd(5-3)\},\{Gcd(2)\#l_1,Gcd(3)\#l_2\}\rangle
\end{array}
$$

Interleaved Derivation with due to overlapping partner constraints: $Gcd(4)\#l_3$

$$
\begin{array}{ll}
 & \langle\{(t,n,Gcd(2)\#l_1),(t',n',Gcd(3)\#l_2)\},\emptyset,\{Gcd(4)\#l_3,Gcd(5)\#l_4,Gcd(2)\#l_1,Gcd(3)\#l_2\}\rangle \\
\rightarrowtail_{Prop} & \langle\{(t,n,Gcd(2)\#l_1),(t',n',Gcd(3)\#l_2)\},\{Gcd(4-2)\},\{Gcd(5)\#l_4,Gcd(2)\#l_1,Gcd(3)\#l_2\}\rangle \\
\rightarrowtail_{Prop} & \langle\{(t,n,Gcd(2)\#l_1),(t',n',Gcd(3)\#l_2)\},\{Gcd(4-2),Gcd(5-3)\},\{Gcd(2)\#l_1,Gcd(3)\#l_2\}\rangle
\end{array}
$$

The top most derivation illustrates the scenario where solver threads t and t' grab distinct partner constraints, hence operate on entirely non-overlapping rule executions ($\{Gcd(2)\#l_1, Gcd(4)\#l_3\}$ and $\{Gcd(3)\#l_2, Gcd(5)\#l_4\}$). The latter shows the interleaving derivations when both t and t' attempt to grab $Gcd(4)\#l_3$, causing t' to fail and retry on another match in the second derivation.

This is the reason why we do not dictate the order in which matching partner constraints are tested (eg. ordering partners by value of their constraint identifiers). Ideally, each solver thread such have a unique view (ordering of elements) of the shared stores to avoid unneccessary interleaving. The order in which rules are tried is left as a design decision still yet to be explored. For instance, it may be possible that certain ordering of rules are more favourable for parallelism, hence this degree of freedom is necessary for such optimizations. This is unlike the w_r operational semantics which enforces a texture ordering of rules for more deterministic and controlled behaviour.

The last source of indeterminism is the result of the in-deterministic nature of parallelism. Ideally, the (Parallel Rewriting) transition rule should be taken aggressively with the largest set of parallel rule executions taken in parallel locked steps. (Interleaved Rewriting) transition rule should only be taken when parallel executions are not possible due to presence of only conflicting rule executions from a particular CHR state. It is however impossible to efficiently compute a maximum set of non-overlapping rule head matchings, hence unreasonable to specify such restrictions in the operational semantics. In practical implementations, we must rely on heuristics and specialized shared data-structures to ensure that parallel rewritings are executed as much as efficiently possible.

No doubt by under-specifying the operational semantics $w_{||}$ we allow more flexibility in implementation and opportunities for optimizations to exploit more parallelism, it's in-determinism compared to the refined semantics w_r, makes it not suitable for many existing programs written with the w_r semantics. Many such CHR programs rely on specific execution orderings (eg. ordering of goals, ordering or CHR rules) to guarantee the correctness of the program. While this is in general a bad programming habit to rely on implementation design specific characteristics of a language, such determinism is at times desirable even in the parallel setting. In Section 7, we will elaborate more on possible future works towards a more deterministic parallel CHR operational semantics.

5 Implementing Parallel CHR in Haskell

This section highlights our works in [15], describing some of the important aspects of our parallel CHR implementation in Haskell. Implementation of a parallel CHR system is a non-trivial task that involves a great deal of engineering feats, most of which focuses not only at faithful implementation of the CHR semantics, but efficiency and scalability of the system. Due to space constraints, for this presentation we shall focus entire on the implementation of the parallel CHR rule matching algorithm, arguably the most important aspect of any CHR system.

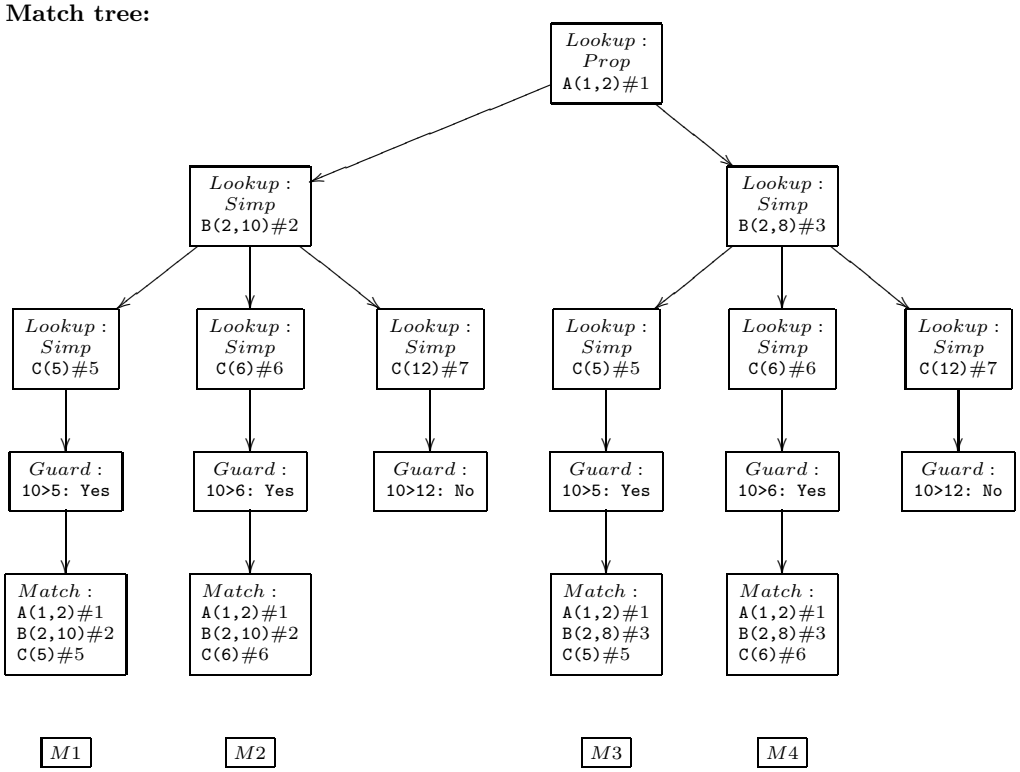
A CHR simpagation rule:

$$\texttt{r} \ @ \ \texttt{A(1,x)} \setminus \texttt{B(x,y)}, \texttt{C(z)} \Leftrightarrow \texttt{y>z} \mid \texttt{D(x,y,z)}$$

Constraint store:

$$\{\texttt{A(1,2)}\#1, \texttt{B(2,10)}\#2, \texttt{B(2,8)}\#3, \texttt{C(5)}\#4, \texttt{C(6)}\#5, \texttt{C(12)}\#6\}$$

Match tree:



Execution of rule r using matches $M1$ and $M4$:

$$\begin{aligned} & \{\texttt{A(1,2)}\#1, \texttt{B(2,10)}\#2, \texttt{B(2,8)}\#3, \texttt{C(5)}\#4, \texttt{C(6)}\#5, \texttt{C(12)}\#6\} \\ \rightarrowtail_r^* \ & \{\texttt{A(1,2)}\#1, \texttt{C(12)}\#6, \texttt{D(2,10,5)}, \texttt{D(2,8,6)}\} \end{aligned}$$

Fig. 8. Example of CHR rule, derivation and match Tree

We illustrate the problem of parallel multi-headed rule matching via an example. Figure 8 defines a CHR rule which fires if we find a matching copy of `A(1,x)`, `B(x,y)` and `C(z)` in the constraint store such that the guard `y>z` is satisfied. The search space for constraints which match rule heads forms a tree to which we refer to as the match tree. Figure 8 gives the match tree for rule `r` where we seek for matchings of rule heads `A(1,x)`, `B(x,y)` and `C(z)` (in this order). Successful leaf nodes contain the complete rule head match which corre-

```
class CHRConstraint c

data HeadType = Simp | Prop
data RuleHead c = RuleHead { htype::HeadType , hcons::c }
data SearchTask c = Lookup HeadType (c -> Bool) | Guard ([c] -> Bool)

data CHRStore c
getMatches :: CHRConstraint c => CHRStore c -> (c -> Bool) -> IO [c]
```

Fig. 9. Basic Haskell Interface

sponds to all rule heads along the path from the root to the leaf node. Successful means that the guard constraint is satisfied.

There are four successful matches. However, it is not possible to fire all of them together. This is because we only propagate `A(1,x)` but simplify `B(x,y)` and `C(z)` by the left-hand side `D(x,y,z)`. Hence, if we choose to use match $M1$ match $M2$ becomes invalid. This is the case because $M1$ and $M2$ share the constraint `B(2,10)`$\#2$ which will be simplified. Hence, we can either use match $M1$ or $M2$ but not both. Similarly, match $M3$ becomes invalid because of the shared, simplified constraint `C(5)`$\#5$. Hence, we conclude that by choosing match $M1$ first, the only remaining valid match is $M4$. Figure 8 gives the execution of rule `r` using matches $M1$ and $M4$.

Parallelism adds another complexity to the rule matching problem: Multiple solver threads are now allowed to execute the matching algorithm over the shared stores in parallel. This means that solver threads must not only explore the match tree correctly, they must do so without blocking each other unnecessarily. In the next two subsections, we describe how the search space (match tree) is built lazily, followed by the parallel rule head matching algorithm.

5.1 Building the Match Tree

We introduce some basic data structures to describe the search for matching constraints. 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 example, the earlier rule `r` is linearized as follows,

$$\texttt{r} \ @ \ \texttt{A(1,x1)} \setminus \texttt{B(x2,y)}, \texttt{C(z)} \Leftrightarrow \texttt{y>z} \wedge \texttt{x1==x2} \mid \texttt{D(x,y,z)}$$

Rule heads and guards are compiled into a sequence of search tasks. A search task either consists of looking up a matching head (which can either be propagated or simplified) , or testing a guard. Here is a possible search task for the above (linearized) rule `r` represented in Haskell.

```
data MyCons = A (Int,Int) | B (Int,Int) | C Int | D (Int,Int,Int)

s1 = [Lookup Prop patA, Lookup Simp patB,
      Guard g1, Lookup Simp patC, Guard g2]
```

```
where
  -- A(1,x1)
  patA (A (1,_)) = True
  patA _ = False
  -- B(x2,y)
  patB (B (_,_)) = True
  patB _ = False
  -- x1 == x2
  g1 [A (_,x1),B (x2,_)] = x1 == x2
  --C(z)
  patC (C _) = True
  patC _ = False
  -- y > z
  g2 [_,B (_,y),C z] = y > z
  -- g2 _ = error "impossible"
```

We assume the data type definitions from Figure 9. User (CHR) constraints are declared as instances of the type class `CHRConstraint`. For each lookup we define a function which checks whether the current constraint is a succesful match. The idea is to collect all matching constraints by mapping this function over the entire constraint store. For the guard check we define a function which takes in as argument the succesful matches so far and then performs the specific test. Note that we assume that all guard functions are well-formed, hence their definitions always match the matching rule heads so far. In essence, we use functions to avoid the explicit treatment of substitutions. How to actually compute search tasks from CHR rules is described elsewhere [3, 22]. The CHR store contains a collection of user constraints and is a shared data structure built on top of transactional memory and accessable by multiple computation threads. For simplicity, we abstract away its underlying implementation.

Building a match tree for given search task is now straightforward. Figure 10 contains the details. Let us first take a look at the `MTree` data type. Each `MNode` consists of a rule head and a list of sub-trees. Successful complete matches are stored in `MLeaf` as a list of rule heads. An unsuccessful branch is represented by an `MNode` with an empty list of subtrees. This may happen in case the search for matching constraints fails or the test of a guard condition yields false.

The `buildMTree` operation builds a match tree given the constraint store and a list of search tasks. We leave the constraint store abstract and assume a function `getMatches` to find all matching constraints in the store for a given lookup task. We assume that the constraint store is an "imperative" data structure living in the heap. This is reflected by the monadic type of `getMatches`. See Figure 9. Building the entire tree is an expensive and not always necessary task (at least for simplified matches). For this, we use lazy evaluation (in the form of the `mapIOLazily` operation), which lazily evaluates all sub-trees of a node. In Haskell, monadic operations are generally strict. Via the `unsafeInterleaveIO` library operation we can delay the execution of the `IO` argument until the value

```
data MTree c = MNode (RuleHead c) [MTree c] | MLeaf [RuleHead c]

buildMTree :: CHRConstraint c => CHRStore c -> [SearchTask c] -> IO [MTree c]
buildMTree str tasks = build tasks
  where
  build :: CHRConstraint c => [RuleHead c] -> [SearchTask c] -> IO [MTree c]
  build matches ((Lookup ht filter):st) = do
    ms <- getMatches str filter
    mapIOLazily (\m -> do let rhead = RuleHead ht m
                          mts <- build (matches ++ [rhead]) st
                          return (MNode rhead mts))
                ms
  build matches ((ScheduleGuard guard):st) =
    if guard (map hcons matches) then build matches st
                                 else return []
  build matches [] = return (MLeaf matches)

mapIOLazily :: (a -> IO b) -> [a] -> IO [b]
mapIOLazily f (a:as) = do b <- f a
                          bs <- unsafeInterleaveIO (mapIOLazily f as)
                          return (b:bs)
mapIOLazily _ [] = return []
```

Fig. 10. MTree DataType and Building an MTree

is required. Thus, how much of the match tree is actually evaluated depends on the specific search strategies.

5.2 Parallel Multi-headed CHR Rule Matching

This section highlights the parallel multi-headed CHR rule matching algorithm. Note that we essentially use lazy evaluation to a emulate backtracking search, the same way we use laziness to build match trees. Figure 11 shows the `lazySearch` operation which executes lazy search on a given match tree.

One might notice a strange characteristic of this algorithm: It is of the `IO` monad type yet it accesses the CHR store, which is in transactional memory. This means that the exploration of the search space is essentially an unsafe operation, but the purpose for this is easy to explain: Naively, we can write a concurrent matching algorithm simply by composing primitive matching operations into one big STM operation (This is analogous to protecting the shared store with one critical section.). This approach unfortunately never scales as the STM protocols will almost certainly force these concurrent but huge transactional operations to interleave. Our solution is to conduct the matching operation in an unsafe manner, only to atomically revelidate complete matches just before commiting.

The search begins at the root of the match tree. For nodes with propagated rule heads, we explore all its sub-trees and return all matches found (denoted `{l-1}` in the code). The `joinLists` function simply concatenates and unfolds a list of list of matches into a single top-level list. For nodes with simplified

```
lazySearch :: CHRConstraint c => CHRStore c -> MTree c -> IO [[RuleHead c]]
lazySearch str mt = do
  lzSearch mt
  where
    lzSearch :: CHRConstraint c => MTree c -> IO [[RuleHead c]]
    lzSearch (MNode rh mts) | (htype rh) == Prop = do
      b <- atomically (areAlive str [rh]) -- {1-3}
      if b then do mss <- mapM (\mt -> forkIO (lzSearch mt)) mts
                   return (joinLists mss) -- {1-1}
           else return []
    lzSearch (MNode rh mts) | (htype rh) == Simp = do
      b <- atomically (areAlive str [rh]) -- {1-3}
      if b then do mss <- mapLazilyIO (\mt -> forkIO (lzSearch mt)) mts
                   return (headOnly (joinLists mss)) -- {1-2}
           else return []
    lzSearch (MLeaf rhds) = do
      b <- atomically (revalidateAndCommit str rhds) -- {1-4}
      if b then return [rhds]
           else return []

headOnly :: [a] -> [a]     joinLists :: [[a]] -> [a]
headOnly (a:_) = [a]       joinLists (a:as) = a ++ (joinLists as)
headOnly [] = []           joinLists [] = []
```

Fig. 11. Lazy Search

```
areAlive :: CHRConstraint c => CHRStore c -> [RuleHead c] -> STM Bool
deleteSimpHeads :: CHRConstraint c => CHRStore c -> [RuleHead c] -> STM Bool

revalidateAndCommit :: CHRConstraint c => CHRStore c -> [RuleHead c] -> STM Bool
revalidateAndCommit str rhds = do
  b <- areAlive str rhds
  if b then do deleteSimpHeads str rhds
               return True
       else return False
```

Fig. 12. More CHR Store Interface, Revalidate and Commit

rule heads, we only return the first match found ({1-2}). This is done by using the `mapLazilyIO` function (Figure 10) to expand subtrees of the node and `headOnly` function to return only the head of the list of matches. Note we do a liveness test ({1-3}) each rule heads. This prunes away hopeless branches where the current rule head has already been deleted. The base case (`MLeaf`) returns the complete match only if the match can be successfully validated (all heads are alive in store) and committed (all simplified heads are deleted) via the `revalidateAndCommit` ({1-4}) operation shown in Figure 12. This is the heart of the algorithm that ensures correctness of parallel rule executions. Each

```
data CHRHandler c          type LVar a = TVar (Maybe a)

newLVar :: IO (LVar a)

class CHRConstraint c => CHRSolver c where
    newSolver :: Int -> [c] -> IO (CHRHandler c)

tell :: CHRSolver c => CHRHandler c -> c -> IO ()
ask :: CHRSolver c => CHRHandler c -> LVar a -> IO a
killsolver :: CHRSolver c => CHRHandler c -> IO ()
```

Fig. 13. Parallel CHR Solver Interfaces

simplified constraints will only be involved in one successful atomic revalidation, slower solver threads competing for it will fail this revalidation step and hence search for other alternatives.

5.3 Parallel CHR Solver Interfaces

Figure 13 shows the basic interfaces of our parallel CHR solver. A `CHRHandler` (inspired by spy handlers of the cloak and dagger world) is essentially the set of references to a CHR solver, in which an external process can issue orders to manipulate it. The `CHRSolver` class provides the interface for a new solver to be created given the number of solver threads to be spawned and a initial set of constraints. Note that CHR solvers are parameterized by the CHR constraints, as it is reasonable that each CHR constraint type are associated with only one type of solver. `tell` simply adds a new constraint to the solver, while `ask` retrieves a *logical variable* value from the solver. Note that logical variables `LVar a` are essentially just type synonyms for transactional memory variables (`TVar (Maybe a)`) which either contains nothing (uninstantiated) or some value (grounded). `newLVar` is an external interfaces to create new logical variables. `killsolver` simply issues the command to kill all solver threads.

`tell` and `ask` are actually basic interfaces provided by most CHR systems. The main difference is that `tell` here is a asynchronous function call, which simply adds the given constraint to the solver and proceeds immediately, assuming that a solver thread will eventually process the constraint. `ask` on the other hand, is a synchronous function which returns only when the queried variable has been instantiated by the solver. In the next section, we shall demonstrate how these CHR interfaces are used to implement asynchronous and synchronous operations of join-patterns.

6 Haskell-Join-Rules: Guarded Join Patterns in Haskell

In this section, we describe our prototype language extension that adds guarded join-patterns to Haskell, known as *Haskell-Join-Rules*. Currently introduced as

a source to source compiler, Haskell-Join-Rules introduces guarded join-patterns by means of two new forms of declarations, namely *join operation* declarations and *join pattern* declarations. The following example introduces these new constructs:

Selective Buffers Example in Haskell-Join-Rules

```
joinoperation SelectBuffer where
    sync getGt :: Int -> Join SelectBuffer Int
    sync getLt :: Int -> Join SelectBuffer Int
    async put :: Int -> Join SelectBuffer ()

joinpattern x@(getGt i) & (put j) | i > j where
    x = return j

joinpattern x@(getLt i) & (put j) | i < j where
    x = return j
```

The `joinoperation` declaration declares a new class of join operations of type `SelectBuffer`. `SelectBuffer` consist of three functions of the `Join` monad type. Synchronous and asynchronous join operations are defined by annotations `sync` and `async`. `getGt` and `getLt` are synchronous join operations that takes an integer and return another, while `put` is an asynchronous operation that simply takes an integer and proceed immediately. Join-pattern declarations define the synchronization patterns between join operations and hence the monadic `Join` operations that evaluates the return values of synchronous join operations. These example shows two join-patterns for the `getGt` and `getLt` Join operations. Note the 'at' pattern binding (@) which behaves different from haskell's 'at' pattern binding: here, `x` is a output variable which represents the monadic outputs of the respective synchronous join operation. With respect to CHR notations, we shall refer to individual function calls in join-patterns as heads of the join-pattern. When a particular join-pattern is successfully matched (eg. `getGt 4, put 2`) the output variable `x` is binded with the monadic operation specified by the join-pattern declaration. Note that a single join-pattern declaration can contain more than one such bindings, hence can synchronize more than one synchronous operation. This mechanism is similar to JoCaml's continuation passing style and the so called futures in Mercury.

The `joinoperation` and `joinpattern` clauses are translated into haskell codes which implement the synchronous/asynchronous join operations and the parallel CHR compilations. For instance, the select buffer join declarations are translated to the following (for simplicity, we show the respective CHR rules rather than their actual compilation in haskell):

CHR and Join Interfaces

```
type ContVar c a = LVar (Join c a)

data SelectBuffer = GetGt Int (ContVar SelectBuffer Int)
                  | GetLt Int (ContVar SelectBuffer Int)
```

```
                  | Put Int

instance CHRConstraint SelectBuffer
instance CHRSolver SelectBuffer
instance JoinSolver SelectBuffer
```

Compilation of Join Operations

```
getGt :: Int -> Join SelectBuffer Int
getGt i = do { x <- newLVar
             ; jhandler <- getHandler
             ; tell jhandler (GetGt i x)
             ; op <- ask jhandler x
             ; op }

getLt :: Int -> Join SelectBuffer Int
getLt i = do { x <- newLVar
             ; jhandler <- getHandler
             ; tell jhandler (GetLt i x)
             ; op <- ask jhandler x
             ; op }

put :: Int -> Join SelectBuffer ()
put i = do { jhandler <- getHandler
           ; tell jhandler (Put i) }
```

CHR Compilation of Join-Patterns

```
GetGt(i,x),Put(j) ⟺ i > j | x = return j
GetLt(i,x),Put(j) ⟺ i < j | x = return j
```

A continuation variable (or 'future') is represented by the type synonym `ContVar` The `SelectBuffer` data type defines the CHR constraints `GetGt`, `GetLt` and `Put` which represent the respective three join operations. Note that synchronous operations `getGt` and `getLt` are represented by constraints `GetGt` and `GetLt` which have an additional logical variable argument. These are essentially continuation variables in which bodies of successfully matched join-patterns are written into. Also accompanying are the `CHRSolver` and `JoinSolver` instances for `SelectBuffer`.

Join operations `getGt`, `getLt` and `put` are compiled into haskell functions. Before we explain the how this operations are implemented, we introduce the interfaces used by these join operations, shown in Figure 14. The `JoinHandler` type synonym and `JoinSolver` type class lifts the CHR interfaces into the join world. The `Join` monad is essentially an environment monad that hides the passing of the join handler across compositions of join monadic operations. `getHandler` simply retrieves the join handler from the monad, while `runJoin` runs a join operation with a given join handler, on the `IO` monad. The compilation schemes of join operations are straightforward: The `put` asynchronous join operations simply `tell` the current join handler of a new `Put` constraint and proceed. Synchronous calls like `getGt` (or `getLt`) first creates a new logical variable (which

```
type JoinHandler c = CHRHandler c

class CHRSolver c => JoinSolver c where
    newJoin :: Int -> IO (JoinHandler c)

data Join c a
getHandler :: Join c (JoinHandler c)
(.||.) :: Join c a -> Join c b -> Join c (a,b)

runJoin :: CHRConstraint c => JoinHandler c -> Join c a -> IO a
```

Fig. 14. Haskell-Join-Rules Library Interfaces

is treated as a single assignment continuation variable) and appends it with a new `GetGt` constraint. This constraint is added to the current join handler and the operation blocks until the continuation variable is instantiated (via the `ask` CHR solver interface). The synchronous call is completed by executing the join-pattern computation embedded in the continuation variable.

The following demonstrates how higher level join operations can be implemented and invoked via the basic join library interfaces shown in Figure 14:

```
producer :: Int -> Join SelectBuffer ()
producer 0 = return ()
producer n = do { (producer (n-1)) .||. (put n)
                ; return () }

consumer :: Int -> Join SelectBuffer ()
consumer 0 = return ()
consumer n = do { (consumer (n-1)) .||. (do { x <- getGt n
                                            ; -- do something with x }
                ; return () }

main :: IO ()
main = do { buffer <- newJoin 2
          ; runJoin buffer ((producer 10) .||. (consumer 10))
          ; return () }
```

The `main` function simply invokes a new buffer join handler with 2 solver threads (via `newJoin`). Next, the `producer` and `consumer` are invoked via the `runJoin` interface on the newly created buffer join handler, each creating a parallel composition (`.||.`) of 10 join operations executing `put` and `getGt`.

6.1 Other features of Haskell-Join-Rules

Haskell-Join-Rules introduces more than just guarded join-patterns to Haskell. Because we essentially use a parallel CHR solver to compute matchings of guarded join-patterns, we get other CHR features as well, for instance, consider the following:

Authorized Only Buffers Example in Haskell-Join-Rules

```
type Auth = String

joinoperation AuthBuffer where
    sync aget :: Auth -> Join SelectBuffer Int
    async put :: Int -> Join SelectBuffer ()
    async auth :: Auth -> Join SelectBuffer ()

joinpattern (auth a) \  x@(aget a) & (put j) where
    x = return j
```

This example illustrates two inherited CHR features: *propagation* and *non-linear patterns*. The authorized buffer is a variant of the buffer join program which only allows access to authorized `aget` calls. This authorized get operation is modelled by the join-pattern: an `aget a` call matches with a `put j` call, only if we can find a matching `auth a` call. Firstly, let's focus on the propagated call `auth a` (appearing before the '\'). Propagated calls are not removed when the join-pattern is triggered, hence they represent the calls which act only as catalysts (undeleted ingredients) to a join-pattern. The following is an example of parallel derivation of authorized buffer join-patterns with 2 solver threads (note that constraint symbols are capitalized versions of their associated join operations):

$$
\begin{aligned}
&\langle \{(t, n, \{\texttt{Aget("a",x)}\}), (t', n', \{\texttt{Aget("a",y)}\}), \emptyset \\
&\quad \{\texttt{Aget("a",x),Aget("a",y),Auth("a"),Put(1),Put(2)}\}, \emptyset \rangle \\
&\rightarrowtail \langle \{(t, n, \emptyset), (t', n', \emptyset), \{\texttt{x = return 1}, \texttt{y = return 2}\}, \{\texttt{Auth("a")}\}, \emptyset \rangle \\
&\rightarrowtail \langle \{(t, n, \{\texttt{x = return 1}\}), (t', n', \texttt{y = return 2}), \emptyset, \{\texttt{Auth("a")}\}, \emptyset \rangle \\
&\rightarrowtail \langle \{(t, n, \emptyset), (t', n', \emptyset), \emptyset, \{\texttt{Auth("a")}\}, \{\texttt{x = return 1}, \texttt{y = return 2}\} \rangle
\end{aligned}
$$

This illustrates two `aget` join operations executed in parallel, sharing the same `auth "a"` call. Note that this is only possible when the `auth "a"` call is declared as a propagated call, hence propagated calls are more than just syntactic sugar, but plays an active part in improving concurrent behaviour.

Note we also have a non-linear pattern here, since `a` appears in two distinct locations of the join-pattern (`auth a` and `aget a`). Non-linear patterns are convinent short hands for simple equality guards. For example, this join-pattern can be rewritten to the following:

```
joinpattern (auth a1) \  x@(aget a2) & (put j) | a1 == a2 where
    x = return j
```

More importantly, standard CHR compilation schemes create hash indexes on stored constraints involved in such non-linear patterns for efficient enumeration of matching constraints. For instance, starting from `aget a` the task of retrieving matching candidates of `auth a` would be of constant time complexity if we store the constraints in the right indexes. Such index locations can be determined

by analysis techniques on the CHR rules (join-patterns in our context) well documented in the CHR literature [3, 21].

Another feature of Haskell-Join-Rules is join-patterns with multiple synchronous calls. Consider the following example:

Swap Example in Haskell-Join-Rules

```
joinoperation Swap where
    sync swap :: Int -> Join Swap Int

joinpattern x@(swap i) & y@(swap j) where
    x = return j
    y = return i
```

This join-pattern coordinates the synchronization between two `swap` join operations called by different and concurrently running processes. Note that this can be considered a generalization of polyphonic $C\#$ join-patterns: the number of body definitions in a join-pattern must be equal to the number of simplified synchronous call. A successful matching would result in the exchange of their input integer. Such join-patterns are not allowed in Polyphonic $C\#$, but also available in JoCaml.

The last example we show here is parallel union find. Originally described and implemented in CHR [9], the following shows a parallel implementation of union find in Haskell-Join-Rules:

Parallel Union Find Example in Haskell-Join-Rules

```
joinoperation UnionFind where
    sync make   :: Char -> Join UnionFind ()
    sync union  :: Char -> Char -> Join UnionFind ()
    sync find   :: Char -> Join UnionFind Char
    sync link   :: Char -> Char -> Join UnionFind ()
    async root  :: Char -> Join UnionFind ()
    async edge  :: Char -> Char -> Join UnionFind ()

joinpattern o@(make a) where o = root a
joinpattern o@(union a b) where
    o = do { (x,y) <- (find a) .||. (find b)
           ; link x y }
joinpattern (edge a b) \  x@(find a) where x = find b
joinpattern (root a) \  x@(find a) where x = return a
joinpattern x@(link a a) where x = return ()
joinpattern (root a) \  x@(link a b) & (root b) where x = edge b a
```

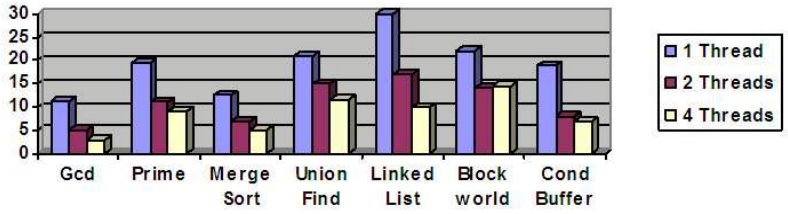
Union find defines a data structure that maintains the union relation over disjoint sets, via two operations `union` and `find`. As shown in the second join-pattern, we optimize by conducting two `find` operations in parallel (via the `(.||.)` operator). Note that this example uses propagation and non-linear patterns (hence guards) which are not expressible in any existing join system implementation.

6.2 Parallelism in Haskell-Join-Rules

Using a parallel implementation of CHR is a cruicial feature of Haskell-Join-Rules. No doubt we could have simply used a standard off-the-shelf implementation of CHR, this unfortunately would not likely scale well as it forces all synchronous calls to an instance of a join handler to be interleaved, since the multi-headed pattern matching of join-patterns (CHR rules) must occur sequentially. Interestingly, standard join-pattern compilation schemes follows a similar approach: matching of join-patterns are conducted by a state-machine based model. Triggering of join-patterns are hence strictly sequential. However, the task of triggering basic join-patterns (no guards, propagation, etc..) is trivial (just pop messages from a queue) and highly optimized in this compilation schemes, hence parallelism within a join-pattern instance is not important. Triggering of guarded join-patterns however is a entirely different and more complex problem, dealing with a combinatorial search as opposed to simple dequeuing of join method calls.

Preliminary experimental results on some Haskell-Join-Rules example supports this conjecture. The main objectives of these preliminary experiments are to prove that implementing guarded join-patterns via parallel CHR will scale with the number of solver threads assigned to the parallel CHR solver.

Experiments were conducted on a Intel Pentium4 Xeon with 4 2.8GHz processors and 2.5Gb of RAM. Measurements were taken in seconds and of several examples of varying complexity: *Gcd*, *Prime* and *Merge-sort* are basic CHR examples reformulated in Haskell-Join-Rules, *Union-find* and *Linked-list* are implementations of parallel data structures. *Blockworld* is a simple agent simulation program of concurrent threads moving 'blocks' in a shared state world and finally, *Cond-buffer* is a collection of simulations over the various buffer examples shown in this paper. These examples are available for download together with our Haskell-Join-Rules prototype in [**?**]. We tested the programs over increasing number of system threads (1, 2 and 4) with generally conclusive and positive results. The following illustrates these results:



The Haskell-Join-Rule examples scale well in general as more active threads are running. Unfortunately, significant resistence to performance improvement can be seen as number of active threads is raised above 2. An interesting observation is that examples with non-linear patterns (eg. `Union-find`, `linked-list` ,`block world`) seem to improve less with more threads, compared to others. Our current implementation of Haskell-Join-Rules do not generate hash-indexes

for more efficient non-linear pattern matching, suggesting that this may be an important future work to be examined.

7 Future Works and Timeline

This section highlights our targeted future works and timeline of completion. We have now identified two main fronts of our research, namely the semantics and implementation of Parallel CHR and language design and implementation of our prototype system Haskell-Join-Rules. While our works described in Section 4, 5 and 6 have yielded a prototype of parallel CHR system and our CHR-based compilation of guarded join-patterns, there are a number of issues which still needs to be addressed. The following sub sections highlights the various issues that we wish to address for the two crucial fronts of our research.

7.1 Towards a more Deterministic and Efficient Parallel CHR

As mentioned in Section 4.1, our parallel CHR operational semantics is necessarily under-specified (goal ordering, rule ordering, etc ..) for efficiency reasons. The parallel CHR operational semantics does not dictate that a specific ordering in which goals, rules and matching candidates are processed hence allow maximum parallelism (in theory) between multiple solver threads. It is important to note that the general opinion of the join-pattern community supports such indeterminism among join-patterns, hence the parallel CHR operational semantics on it's own fits perfectly preferred indeterministic Unfortunately, there are times where this indeterminism may make programming difficult, for instance, consider the following example:

```
type Key = Int       type Item = String       data Elem = Elem Key Item

joinoperation MultiSet where
    sync find :: Key -> Join MultiSet (Maybe Item)
    async slot :: Key -> Item -> Join MultiSet ()

joinpattern (slot k i) \  o@(find k) where o = return (Just i)
joinpattern o@(find k) where = o = return Nothing
```

This example attempts to implement a parallel multiset data structure, where a `slot` asynchronous call simply adds a new string `Item` tagged with an integer `Key`, while `find` call returns an item if the given key exists, otherwise nothing. Unfortunately it does not behave as expected: note that the program only works with the assumption that texture order of the join-patterns would dictate that the first join-pattern (`(slot k i) \  o@(find k)`) would be triggered exhaustively and the second will only trigger if the first is unsuccessful. This would be a reasonable assumption for a Haskell programmer, since pattern matching on multiple function definitions works this way. The parallel CHR operational semantics however does not enforce such rule ordering, hence it would be wrong

for the programmer to make such assumptions. In practice, our implementation would not enforce such rule orderings when the join handler is instantiated with more than one solver threads. We wish to investigate into the possibility of enforcing such determinism (rule priority) without compromising too much of parallelism, hence guarded join-pattern programs such as the above can be written in Haskell-Join-Rules without suffering the penalties of interleaved rule execution.

We believe that one possible approach is to parallelize existing CHR operational semantics like the refined operational semantics [3] or the rule priority semantics [**?**], which are single threaded in nature. This approach is unfortunately highly complex and preliminary investigation suggests that prioritizing rule executions in general ultimately leads to interleaved rule executions, this of course, needs to be proven with proven with empirical results. One promising compromise that can probably be derived from this approach is soft rule priorities where rule priorities are not absolute but quantitative (higher priority rules fire more frequently than lower ones).

On the implementation front of the parallel CHR semantics, much work is still required in optimizing our parallel CHR solver. For instance, our current prototype does not use indexing for non-linear multi-headed pattern matching, as well as optimal join ordering of rule head matching. Most existing local optimization techniques (optimal join ordering, early guard scheduling, non-linear indexing) are immediately applicable in the parallel CHR setting, hence the only hurdle here is the selection and implementation of suitable non-blocking shared data structures (eg. shared Hashtable). Global CHR optimizations (eg. passive constraints, never stored) however, are not directly applicable, hence we must investigate into whether each can be specifically adapted to suit the parallel setting.

7.2 Haskell-Join-Rules Language Design and Empirical Results

So far, we have left out defining any static checks for our Haskell-Join-Rules compiler. It is important that we impose some syntactic restrictions to ensure that only correctly defined programs are accepted. For instance consider the following program:

```
joinoperation StrangeBuffer where
    sync sget :: Join StrangeBuffer Int
    async put :: Int -> Join StrangeBuffer ()

joinpattern o@sget \(put i) where o = return i
```

This Haskell-Join-Rule program is problematic as we have a join-pattern with a synchronous propagated call (`o@sget`) with a body definition for it's output continuation variable, which should be treated as single assignment variables. This is because a single synchronous `sget` call is never deleted after the join-pattern is triggered and it is free to match with unbounded numbers of `put i`, making the continuation `o` no longer a single assignment variable. Other static

checks like type checking is also ignored in current implementation and will be considered for future works.

We are also looking into other language features like including haskell style polymorphism into the definitions of join operations. For instance, we can define parallel unbounded buffers for any generic haskell types:

```
joinoperation Buffer a where
    sync get :: Join (Buffer a) a
    sync put :: a -> Join (Buffer a) ()

joinpattern o@get & (put i) where o = return i
```

Just like in haskell type classes type parameters in join operation definitions should be able to be restricted by type context, adding to more control over polymorphic types:

```
joinoperation Ord a => Buffer a where
    sync getGt :: a -> Join (Buffer a) a
    async put :: a -> Join (Buffer a) ()

joinpattern o@(getGt i) & (put j) | i > j where
    o = return j
```

The type context `Ord a` simply states that the type `a` here must be an instance of the `Ord` type class, which defines ordering and comparison between values of `a`. Hence in the join-patterns, we are free to use overloaded comparison functions (in this case, `>`). We believe that introducing haskell language features like this adds to the usability of our language extension and should not be overlooked.

Our currently Haskell-Join-Rules prototype implements the language constructs described in Section 6. Join-patterns are allowed to have as many body definitions as it has simplified synchronous heads. Each body definition binds a output continuation variable (which a synchronous call blocks on) to a join operation of the matching type. This design describes multi way synchronizations in a simple and intuitive way: synchronous calls block until they are involved in a complete join-pattern match, during which, each synchronous call receives a unique body computation which it is to execute. In a way, it resembles a generalization of polyphonic $C\#$'s join-patterns in that it allows more than one synchronous head for each join-pattern, but does not spawn any new execution threads upon triggering of the join-pattern.

Unfortunately, there is one weakness in this design: purely asynchronous join-patterns (ie. join-patterns with only asynchronous heads) can have no body definitions. This is because all heads are asynchronous and even if we have body computations, we have not continuation variables to bind them to and no synchronous threads to execute them. This is illustrated by the following:

```
joinoperation MergingBuffer where
    sync get :: Join MergingBuffer [Int]
```

```
reply :: JoinHandler c => LVar (Join c a) -> Join c a -> IO ()
```

Fig. 15. `reply` interface for alternative Haskell-Join-Rules

```
    async put :: [Int] -> Join MergingBuffer ()

joinpattern o@get & (put i) where o = return i
joinpattern (put is) & (put js) where
    ? = put (is++js)
```

The `MergingBuffer` implements a buffer variant where `put` calls are merged into one, and `get` calls are allowed to atomically retrieve all values within a merged `put` call. This mergal is implemented by the second join-pattern. The body computation of this join-pattern (`put (is++js)`) is suppose to replace the two `put` calls with their union. However, we cannot correctly specify this since there are no synchronous call blocking and waiting to execute this body computation.

We are currently also exploring other alternative language designs for Haskell-Join-Rules that can avoid the short-comings illustrated above. This alternative variant of Haskell-Join-Rules forces that each join-pattern have strictly one body computation. A surrogate thread is explicitly spawned to execute this computation every time a join-pattern is successfully triggered. For example, the `MergingBuffer` can now be correctly implemented by the following:

```
joinoperation MergingBuffer where
    sync get :: Join MergingBuffer [Int]
    async put :: [Int] -> Join MergingBuffer ()

joinpattern o@get & (put i) = reply o (return i)
joinpattern (put is) & (put js) = put (is++js)
```

Similar to the previous approach, we still retain the use of continutation variables, but in a different style. This binding is now specified as a monadic operation `reply` shown in Figure 15. This allows us to pass continuations to synchronous threads blocking on the join-pattern as before. The main difference is that we now have a explicit thread of computation dedicated to executing the join-pattern body, hence we can execute join operations in the body which are not passed to any synchronous heads of the join-pattern, as shown in the second join-pattern of the `MergingBuffer`.

This alternative Haskell-Join-Rule language design subsumes the original. As an example, multiple way synchronization still can be easily expressed:

Swap Example in Haskell-Join-Rules Alternative

```
joinoperation Swap where
    sync swap :: Int -> Join Swap Int

joinpattern x@(swap i) & y@(swap j) = do
```

```
{ reply x (return j)
; reply y (return i) }
```

Note that this alternative resembles more closely to join-patterns in JoCaml. One disadvantage of this alternative is that the triggering of each join-pattern will now invoke the spawning of a new thread. While this is not really a problem in Haskell (as threads in Haskell are very light-weight), this may not be acceptable when the language extension is considered for other programming languages.

Still, whether light-weight or not, spawning threads would incur some overheads in the runtime system. To avoid such overheads, we can let a CHR solver thread execute this body computation instead (note that body computations can be modelled as built-in constraints to be executed in the CHR 'world') as oppose to creating a new 'surrogate' thread. This however, will require that join-pattern bodies are non-blocking as it would be pretty awful to have CHR solver threads blocking while executing these join computations. It is clear that there is a wide range of design decisions to be considered. Exploring them would be an important subject for future works.

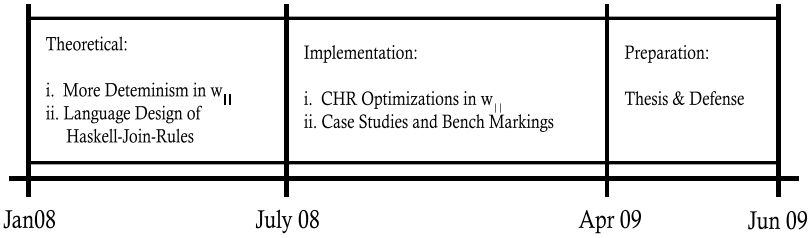
Last but not least in the agenda is empirical results for Haskell-Join-Rules. Currently, our experimental results are still preliminary: We have so far tested our prototype with simple programs and a 4-core system. In future, we intend to derive more conclusive results by first defining a larger test suite comprising of a wider range of parallel applications from simple ones to larger and practical ones. We also wish to conduct study on larger scale multicore systems to test the scalability of Haskell-Join-Rules. Our empirical studies should also include comparing performances between the proposed alternative language designs of Haskell-Join-Rules.

7.3 Timeline

The following summarizes the proposed future works together with their estimated time required:

- More determinism in Parallel CHR semantics $w_{||}$ (eg. soft rule priorities) (3 months).
- Adapting and implementation CHR optimizations for Parallel CHR (5 months).
- Language design issues of Haskell-Join-Rules (4 months).
- Empirical results (Case studies and benchmarking) (4 months).

The following highlights our proposed timeline for completion of my thesis within the next 18 months, future works are partitioned into 2 groups: theoretical and implementational, the former addresses issues related to the formal and theoretical models (Parallel CHR and CHR compilation of guarded join-patterns) of our approach and the latter addresses practical issues of the Haskell-Join-Rules system.



Note that this timeline is strictly an estimate and may be subjected to change as our work progresses.

8 Conclusion

Join-calculus is a promising approach to high level concurrency, but existing implementations [2, 24, 1, 20] do not implement efficient compilations of guarded join-patterns. While various extensions to join-patterns have been studied [**?**,23, **?**], the problem of handling general guard constraints in join-patterns still remains an open issue, as it is non-trivial and deals with parallel combinatorial matching.

In this proposal, we introduce our novel solution to efficient high level concurrency and parallelism abstraction: Guarded join-patterns via Constraint Handling Rules (CHR). We have shown that constraint handling rules in probably the best candidate to solve this problem. CHR operational Semantics define a goal directed way of computing multi-headed pattern matching with guards on demand, a strategy which can be directly adopted to trigger guarded join-patterns. We have also motivated the need for parallelism in the triggering of guarded join-patterns and hence a parallel semantics and implementation of CHR, as an interleaving semantics simply would not scale well.

We have implemented a prototype system, Haskell-Join-Rules which introduces guarded join-patterns to Haskell. Join-patterns are compiled into CHR rules which are executed by a parallel implementation of CHR in Haskell. Preliminary experimental results show that our approach scales relatively well with the number of processors.

We have identified possible future works that would further concretize our approach. The theoretical aspects of these future works include investigating into a more deterministic parallel CHR semantics and identifying the design choices of our language extension to Haskell, as well as their implications concerning expressiveness and efficiency. Practical aspects of these future works include introducing CHR optimizations to our parallel CHR implementation and a stronger empirical study of our implementation which includes case studies of larger parallel application and a wider range of bench marking suites.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

2. 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.
3. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, The University of Melbourne, 2005.
4. 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.
5. 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.
6. 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.
7. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
8. T. Frühwirth. Constraint Handling Rules. *Lecture Notes in Computer Science*, (910):90–107, 1995.
9. Thom W. Frühwirth. Parallelizing union-find in constraint handling rules using confluence analysis. In *ICLP*, pages 113–127, 2005.
10. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.
11. 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.
12. Edmund S. L. Lam and Martin Sulzmann. Haskell-join-rules. Technical report, 2007. Presented in IFL 2007, Freiburg.
13. Edmund S.L. Lam and Martin Sulzmann. Towards agent programming in CHR. Technical Report CW 452, Katholieke Universteit Leuven, 2006. Informal Proc. of CHR 2006, Third Workshop on Constraint Handling Rules.
14. Edmund S.L. Lam and Martin 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.
15. Edmund S.L. Lam and Martin Sulzmann. Compiling constraint handling rules with lazy and concurrent search techniques. Technical report, July 2007. Manuscript.
16. Edmund S.L. Lam and Martin Sulzmann. A concurrent constraint handling rules semantics and its implementation with software transactional memory. Technical report, June 2007. Manuscript.
17. Edmund S.L. Lam and Martin Sulzmann. Specifying and controlling agents in haskell. Technical report, September 2006. Manuscript.
18. S. Peyton Jones. Beautiful concurrency. http://research.microsoft.com/ simonpj.
19. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
20. C. Russo. The Joins concurrency library. In *Proc. of PADL'07*, volume 4354 of *LNCS*, pages 260–274. Springer-Verlag, 2007.
21. 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.

22. 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.
23. 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.
24. G. Stewart v. Itzstein and D. Kearney. The expression of common concurrency patterns in join java. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2004.
25. 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.