A Concurrent Constraint Handling Rules Implementation in Haskell with Software Transactional Memory

Edmund S. L. Lam
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
lamsoonl@comp.nus.edu.sg

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

Abstract

Constraint Handling Rules (CHR) is a concurrent committed-choice constraint logic programming language to describe transformations (rewritings) among multi-sets of constraints (atomic formulae). CHR is widely used in a range of applications spanning from type system design to artificial intelligence. However, none of the existing CHR implementations we are aware of exploits concurrency or parallelism explicitly. We give a concurrent CHR implementation using GHC (Glasgow Haskell Compiler) with support for Software Transactional Memory. We achieve some significant performance improvements compared to a single-threaded CHR implementation. We obtain a further speed-up, in some cases nearly close to the optimum of 100%, when running programs under under a dual-core processor architecture. Our results show that CHR can be implemented efficiently on a multi-core architecture.

1. Introduction

Constraint Handling Rules (CHR) [5] is a concurrent committed-choice constraint logic programming language to exhaustively transform (rewrite) multi-sets of constraints (atomic formulae) into simpler ones. Initially, developed to specify incremental constraint solvers, CHR are now used as a general purpose concurrent constraint programming language. CHR is also used in a multitude of other applications such as type system design, agent specification and planning etc [7].

CHR naturally support concurrent programming. Conjunction of constraints can be regarded as interacting collections of multiple asynchronous agents or processes. Their interaction is specified via transformation rules which can be applied simultaneously if the transformation rules do not interfere. Hence, one would expect to run CHR faster by executing transformation rules in parallel on a multi-core machine architecture. To the best of our knowledge, all existing CHR implementations [2] are single threaded and neither exploit concurrency nor parallelism.

In this paper, we present our initial studies of a concurrent CHR implementation in Haskell [10] that makes use of Software Transactional Memory (STM) [9] as supported by GHC [8]. Specifically, we make the following contributions:

- We discuss a concurrent implementation of CHR where each transformation rule is implemented by a single thread. (Section 4). We use STM operations to avoid inconsistencies of CHR programs, i.e. set of transformation rules, manipulating a shared constraint store. Our implementation is very intuitive and matches precisely the declarative semantics of CHR [5].
- The common CHR implementations adopt the refined CHR semantics [3]. We give a concurrent formulation of the refined semantics and discuss our implementation in Haskell STM. In contrast to the standard declarative semantics where we have n threads for n transformation rules, in the concurrent refined semantics we have n active constraints (threads) which seek for the missing partner constraints such that a transformation rule applies (Section 5).
- We provide some initial experimental results of the execution time of CHR programs compiled with GHC 6.6. under a dual core architecture. (Section 6). Our results show that we obtain some significant performance improvements compared to a single threaded implementation under a single core architecture.

In the next section, we give an overview of the key ideas of our work. In Section 3, we review the syntax and declarative semantics of CHR. In Section 4 and 5, we describe the concurrent implementation of the declarative and refined semantics of CHR respectively. Section 6 summarizes our preliminary experiment results and we conclude in Section 7. Throughout the paper, we assume that the reader has some basic knowledge of Haskell and STM as supported by GHC.

2. Overview

We start off with a simple example to explain the basic concepts behind CHR. We introduce two CHR *simplification* rules to compute the greatest common divisor among a set of numbers.

```
rule Gcd(0) <==> True
rule Gcd(m),Gcd(n) <==> m>=n && n>0 | Gcd(m-n),Gcd(n)
```

Each CHR rule has one or more constraints on its left-hand and right-hand side. The constraint `True` represents the always true constraint. Guard constraints such as `m>=n && n>0` test whether rules are applicable. We can apply a CHR rule if we find constraints in the constraint store which match the left-hand side of the rule. Then, we replace (simplify) these constraints with the right-hand side. In case the CHR rule has a guard constraint the rule only applies if the guard constraints hold.

For example, we find that

$$\begin{array}{ll} & \texttt{Gcd}(3),\texttt{Gcd}(9) \\ \rightarrowtail & \texttt{Gcd}(3),\texttt{Gcd}(6) \\ \rightarrowtail & \texttt{Gcd}(3),\texttt{Gcd}(3) \\ \rightarrowtail & \texttt{Gcd}(3),\texttt{Gcd}(0) \\ \rightarrowtail & \texttt{Gcd}(3) \end{array} \tag{2.1}$$

We write $C \rightarrowtail C'$ to denote a CHR *derivation* step where we apply a CHR rule on the constraint store C resulting in the constraint store C'. We say a constraint store is *final* if no further CHR rules can be applied. In the above, we apply the second CHR three times before applying the first CHR once which leads to the final constraint store `Gcd(3)`. This shows that `3` is the greatest common divisor of `3` and `9`. Similarly, we find that

$$\texttt{Gcd}(4),\texttt{Gcd}(8) \rightarrowtail^* \texttt{Gcd}(4) \tag{2.2}$$

where $\rightarrowtail^*$ denotes zero or more derivation steps.

CHR satisfy a monotonicity property. That is, we can straightforwardly combine two derivations as long as they do not interfere. Hence, we find that

$$\mathtt{Gcd}(3), \mathtt{Gcd}(9), \mathtt{Gcd}(4), \mathtt{Gcd}(8) \rightarrowtail^* \mathtt{Gcd}(3), \mathtt{Gcd}(4) \rightarrowtail^* \mathtt{Gcd}(1)$$

In terms of the operational behavior of CHRs this means that we can build derivations 2.1 and 2.2 concurrently. Under a dual core architecture, we can then run both derivations in parallel. According to Früwirth [6] this form of CHR concurrency/parallelism is referred to as *weak* concurrency/parallelism. At the end of this section, we will briefly discuss *strong* concurrency/parallelism.

In Sections 4, we give a concurrent implementation of the declarative semantics where each concurrent thread corresponds to a CHR rule seeking constraints in the constraint store matching its left-hand side. Section 5 presents a concurrent variant of the refined semantics which is commonly found in CHR implementations. The execution of CHR is driven by active constraints which seek for missing partners such that a CHR applies.

In both cases, we model the constraint store as a shared linked list and employ STM operations to prevent inconsistencies when applying CHR rules. Each CHR application is split into a sequence of search and insert/delete STM operations. A subtle point is that making the entire search an atomic operation easily leads to interleaving of CHR derivation steps. For example, the concurrent execution of

$$\begin{array}{l} \mathtt{Gcd}(3), \mathtt{Gcd}(9) \rightarrowtail \mathtt{Gcd}(3), \mathtt{Gcd}(6) \quad \| \\ \mathtt{Gcd}(4), \mathtt{Gcd}(8) \rightarrowtail \mathtt{Gcd}(4), \mathtt{Gcd}(4) \end{array}$$

requires us to spawn two concurrent threads scanning the constraint store `Gcd(3),Gcd(9),Gcd(4),Gcd(8)` for constraints matching the left-hand side of the second CHR rule. If both threads start scanning the store from left to right, only one of the threads will be successful but the other thread will become invalid and has to "retry" in the STM sense. Specifically, say the first thread searches for `Gcd(3),Gcd(9)` and the second thread searches for `Gcd(4),Gcd(8)`. The first thread will be faster and will then replace `Gcd(3),Gcd(9)` by `Gcd(3),Gcd(6)`. But this will invalidate the transaction log of the second thread which contains `Gcd(3),Gcd(9)` because of the left to right search strategy.

Therefore, we use a "small step" search strategy where only the individual (but not the entire) search steps between neighboring nodes are atomic. Of course, we need to re-check the found constraints before the actual rule application. We also use different entry points to avoid "overlap" among the search threads. In essence, we use a shared circular list.

CHR also support a *stronger* form of concurrency/parallelism [6] which we will ignore here for brevity. Briefly, the two CHR derivations

$$\mathtt{Gcd}(3), \mathtt{Gcd}(9) \rightarrowtail^* \mathtt{Gcd}(3)$$

and

$$\mathtt{Gcd}(3), \mathtt{Gcd}(12) \rightarrowtail^* \mathtt{Gcd}(3)$$

share the common constraint `Gcd(3)`. In our current (naive) implementation, `Gcd(3)` is removed and added again each time the second CHR applies. Hence, we cannot concurrently execute the above CHR derivations. The important point to note that is that there is only one copy of `Gcd(3)`. However, both derivations do not alter `Gcd(3)`. When applying

```
rule Gcd(m),Gcd(n) <==> m>=n && n>0 | Gcd(m-n),Gcd(n)
```

we only remove the first constraint `Gcd(m)` by `Gcd(m-n)` but the second constraint `Gcd(n)` remains unchanged. Hence, there should not be any problem executing both derivations concurrently. Our implementation can be adjusted to deal with such cases but we postpone a discussion till a later version of this paper.

Another issue which we will ignore here is confluence. We say a set of CHR rules is *confluent* iff different derivations starting from the same point can always be brought together again. The above CHR rules are clearly confluent, because their left-hand side do not overlap. The situation changes if we drop the guard `n>0` from the second rule.

```
rule Gcd(0) <==> True
rule Gcd(m),Gcd(n) <==> m>=n | Gcd(m-n),Gcd(n)
```

Then, both rules are applicable on `Gcd(0),Gcd(0)`, however, the set of CHR rules remains confluent. We can check for confluence by checking that all "critical pairs" are joinable [1]. This confluence check is decidable if the CHR rules are terminating. We say a set of CHR rules are *terminating* iff for each initial constraint there exists a final constraint store. In theory, many CHR rules are non-confluent. But non-confluence does not arise in practice because the critical pair state can never be reached. We refer to [6, 4] for a discussion.

Terms	$\mathtt{s},\mathtt{t}$	::=	$x \mid \mathtt{0} \mid \mathtt{1} \mid -\mathtt{1} \mid ...$
			$\mathtt{t}-\mathtt{t} \mid \mathtt{t}+\mathtt{t} \mid ...$
			$(t,...,t) \mid \mathtt{K}\ \mathtt{t}...\mathtt{t}$
Guards	$\mathtt{g}$	::=	$\mathtt{g} \vert\vert \mathtt{g} \mid \mathtt{g}\&\&\mathtt{g} \mid \mathtt{t} > \mathtt{t} \mid ...$
Atoms	$\mathtt{at}$	::=	$\mathtt{P}\ \mathtt{t}$
CHR	$\mathtt{r}$	::=	$\mathtt{rule}\ \mathtt{P}_1\ \mathtt{t}_1\ ...\ \mathtt{P}_n\ \mathtt{t}_n \Longleftrightarrow$
			$\mathtt{g} \mid \mathtt{Q}_1\ \mathtt{s}_1\ ...\ \mathtt{Q}_m\ \mathtt{s}_m$

Figure 1. CHR Syntax

3. Syntax and Semantics of CHR

We review the syntax and declarative semantics of CHR. For our purposes, we assume CHR simplification rules of the form described in Figure 1. CHR also support simpagation and propagation rules and builtin constraints such as equality which we ignore for brevity here. The syntax of CHR in Figure 1 follows closely our implementation which is embedded into Haskell. We assume that `K` refers to a value constructor of a user-definable data type. The term and guard language is essentially the same as in Haskell. We adopt Haskell syntax and assume that variable names start with a lowercase letter and predicate and constructor names start with a uppercase letter. In the standard CHR syntax it is exactly the other way around.

As usual, we refer to the left-hand side of a CHR rule as the rule *head* and the right-hand side (excluding the guard) as the rule *body*. The guard is assumed to be true if omitted.

We assume that CHR rules are *variable-restricted*, i.e. all variables occurring on the right-hand side already occur on the left-hand side.

The declarative semantics of CHR is explained by exhaustive application of CHR rules on an initial constraint store until we reach a final constraint store where no further CHR rules are applicable. A constraint *store* is simply a multi-set of atoms to which we often refer to as constraints.

A single derivation step is formally defined as follows. A (renamed) CHR rule $\mathtt{r}$ of the form $\mathtt{rule}\ \mathtt{P}_1\ \mathtt{t}_1\ ...\ \mathtt{P}_n\ \mathtt{t}_n \Longleftrightarrow \mathtt{g} \mid \mathtt{Q}_1\ \mathtt{s}_1\ ...\ \mathtt{Q}_m\ \mathtt{s}_m$ applies to a multi-set C of atoms iff there exist $\mathtt{P}_1\ \mathtt{t'}_1\ ...\ \mathtt{P}_n\ \mathtt{t'}_n$ in C such that $\phi(\mathtt{t}_i) = \mathtt{t'}_i$ for $i = 1,...,n$ for some substitution ϕ. Then, we replace $\mathtt{P}_1\ \mathtt{t'}_1\ ...\ \mathtt{P}_n\ \mathtt{t'}_n$ in C by $\mathtt{Q}_1\ \phi(\mathtt{s}_1)\ ...\ \mathtt{Q}_m\ \phi(\mathtt{s}_m)$ if $\phi(\mathtt{g})$ evaluates to true. We refer to the resulting constraint as C' and write $C \rightarrowtail_r C'$ to denote the application of CHR $\mathtt{r}$ to C with result C'.

Readers familiar with Prolog will notice the two important differences between Prolog and CHR. CHR applies forward reasoning whereas Prolog uses backward reasoning. In CHR, we seek for matchings where in Prolog we use unification to fire rules.

Derivation d_1	**Derivation** d_2
Atomic Search and Rewrite Read C_1: `Gcd(0)` Rewrite C_1 with $True$ Commit: Success	Atomic Search and Rewrite Read C_1: `Gcd(0)` Read C_2: `Gcd(6)` Read C_3: `Gcd(3)` Rewrite C_2, C_3 with `Gcd(3),Gcd(3)` Commit: Fail

Figure 2. "Big step" search

Derivation d_1	**Derivation** d_2
Atomic Hop 1: Read C_1: `Gcd(0)` Commit: Success	Atomic Hop 1: Read C_1: `Gcd(0)` Commit: Success
Atomic Rewrite: Verify C_1 contains `Gcd(0)` Rewrite C_1 with $True$ Commit: Success	Atomic Hop 2: Read C_2: `Gcd(6)` Commit: Success
	Atomic Hop 3: Read C_3: `Gcd(3)` Commit: Success
	Atomic Rewrite: Verify C_2, C_3 contains `Gcd(6),Gcd(3)` Rewrite C_2, C_3 with `Gcd(3),Gcd(3)` Commit: Success

Figure 3. "Small step" search

4. A Concurrent Implementation of the Declarative Semantics

In this section, we discuss a concurrent implementation of the declarative CHR semantics. The declarative CHR semantics does not specify a specific order in which we apply the CHR rules. Hence, we can assume the simultaneous application of CHR rules on a shared constraint store. In our implementation, we spawn a new thread for each CHR rule. We could of course have several threads for the same CHR rule but we ignore this possibility here. Each thread/CHR rule searches the constraints in the store that match the rule head and replaces them with the rule body if the guard constraints hold. If there is no interference among CHR rule applications, we can execute the CHR rules concurrently as the following example shows.

$$
\begin{tikzcd}
 & & \mathtt{Gcd(0), Gcd(6), Gcd(3)} \arrow[dl] \arrow[dr] & \\
\mathtt{Gcd(0)} \arrow[r, "\mathtt{r}_1"] & \emptyset \arrow[dr] \quad \| \quad \mathtt{Gcd(6), Gcd(3)} \arrow[rr, "\mathtt{r}_2"] & & \mathtt{Gcd(3), Gcd(3)} \arrow[dl] \\
 & & \mathtt{Gcd(3), Gcd(3)} &
\end{tikzcd}
$$

To guarantee consistency we demand that any computation of the concurrent execution of rules must be obtainable by a sequential execution of the same rules. We enforce consistency by implementing the shared constraint store as a shared linked list stored using Haskell with Software Transactional Memory (STM). CHR rule applications are implemented by composing STM read/write operations on the linked list. Then, consistency is guaranteed by the atomicity of the STM operations. Haskell with STM uses lock free optimistic synchronization, where each STM operation accumulates a local transaction log recording all read and write operations. This transaction log is validated at the end of the operation and the runtime system determines if the operation can commit or must retry.

Consistency can be easily enforced by composing the *entire* CHR derivation as a single atomic STM operation. First, the constraint store is searched, say from the head of the shared linked list, for matching constraints. Then, we apply a CHR rule if a match is found. Unfortunately, this strategy will likely force most derivations to be interleaved. Each search operation will accumulate reads of a prefix of the shared constraint store. Hence, writes to any constraint in the prefix will force the (atomic) derivation to retry. In general, atomic derivations are interleaved as long as their search operations over-lap.

Figure 2 shows a possible accumulation of transaction logs which causes the above CHR derivations to interleave. We assume that the constraint store contains three shared memory locations C_1, C_2 and C_3 containing `Gcd(0)`, `Gcd(6)` and `Gcd(3)` respectively. We see that derivation d_2 fails to commit because of d_1's write into C_1. This is undesirable as d_2 has over-accumulated its transaction log with reads to locations (C_1) containing unrelated constraints. Our idea is to perform a "small step" search via a series of atomic "hops" within the shared constraint store. Since the small step search is unsafe, we must atomically re-verify (read C_2 and c_3 again) that all matching constraints found during the search are still present before we execute the CHR rule. Figure 3 illustrates a possible concurrent execution of CHR derivations using the small step search method.

We summarize. The execution of a CHR derivation step is split into a small-step search step followed by a verification step which if successful allows us to actually apply the CHR rule by deleting the left-hand side constraints and inserting the right-hand side constraints. As a further optimization, we perform in-place update rather then delete followed by insert if the number of left-hand side constraints is smaller than the number of right-hand side constraints. In practice, the declarative semantics is usually too inefficient. The common approach is to implement the refined semantics which we discuss next.

5. A Concurrent Implementation of the Refined Semantics

We first review the refined semantics [3] where we process (execute) constraints depth-first from left to right similar to Prolog. Standard implementations use a single execution stack. We give a concurrent variant of the refined semantics using multi-threaded execution stacks and discuss some of the implementation issues.

5.1 The Refined Semantics

The refined operational semantics [3] introduces *active* constraints to guide the search for constraints matching the CHR rule head. A constraint remains active until the CHR rule corresponding to a particular matching has fired. Newly introduced constraints are added to an execution stack E where active constraints are processed repeatedly until the stack is empty. Hence, the constraint store is split into $\langle E \mid C \rangle$ where E is an execution stack and C is the constraint store. We write $\emptyset$ to denote the empty constraint store and ϵ for the empty execution stack. For example, we find the following derivation steps in the refined semantics.

$$
\begin{array}{ll}
 & \langle [\mathtt{Gcd(0)}\#1, \mathtt{Gcd(4)}\#2, \mathtt{Gcd(4)}\#3] \mid \emptyset \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(0)}\#1, \mathtt{Gcd(4)}\#2, \mathtt{Gcd(4)}\#3] \mid \{\mathtt{Gcd(0)}\#1\} \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(4)}\#2, \mathtt{Gcd(4)}\#3] \mid \emptyset \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(4)}\#2, \mathtt{Gcd(4)}\#3] \mid \{\mathtt{Gcd(4)}\#2\} \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(4)}\#3] \mid \{\mathtt{Gcd(4)}\#2, \mathtt{Gcd(4)}\#3\} \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(0)}\#4, \mathtt{Gcd(4)}\#5] \mid \emptyset \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(0)}\#4, \mathtt{Gcd(4)}\#5] \mid \{\mathtt{Gcd(0)}\#4\} \rangle \\
\rightarrowtail & \langle [\mathtt{Gcd(4)}\#5] \mid \emptyset \rangle \\
\rightarrowtail & \langle \epsilon \mid \{\mathtt{Gcd(4)}\#5\} \rangle
\end{array}
$$

Notice that each constraint is attached with a unique identifier $\#n$ to disambiguate multiple occurrences of the same constraint in the (multi-set) constraint store. We refer to $c\#n$ as a numbered constraint. Initially, all constraints are in the execution stack. Then,

$$\boxed{\text{Interleaving Semantics: } \langle E \mid C\rangle_i^x \rightarrowtail \langle E \mid C\rangle_i^x}$$

Simplification:

$$\text{(Simp)} \quad \frac{\begin{array}{c}\exists (H' \Leftrightarrow g \mid B') \quad \exists \theta \text{ such that } \theta(H') = (\{c\} \uplus Cons(H)) \quad \text{such that} \\ \theta(g) \text{ evals to } True \quad B \equiv Label(\theta(B'), x, i)\end{array}}{\langle (c\#x'_j) : E \mid \{c\#x'_j\} \uplus H \uplus C\rangle_i^x \rightarrowtail \langle B{+}{+}E \mid B \uplus C\rangle_{(i+|B|)}^x}$$

Drop 1 (No Rules Apply):

$$\text{(D1)} \quad \frac{\begin{array}{c}\forall (H' \Leftrightarrow g \mid B') \quad \neg \exists H \subseteq C \quad \text{such that} \\ \exists \theta \text{ such that } \theta(H') = (\{c\} \uplus Cons(H)) \quad \theta(g) \text{ evals to } True\end{array}}{\langle (c\#x'_j) : E \mid C\rangle_i^x \rightarrowtail \langle E \mid C\rangle_i^x}$$

Drop 2 (Active Constraint Absent):

$$\text{(D2)} \quad \frac{c\#x_j \notin C}{\langle (c\#x_j) : E \mid C\rangle_i^x \rightarrowtail \langle E \mid C\rangle_i^x}$$

$$\boxed{\text{Concurrent Semantics: } \langle E, E \mid C\rangle_{[i,i]}^{[x,x]} \rightarrowtail \langle E, E \mid C\rangle_{[i,i]}^{[x,x]}}$$

Interleaving Left:

$$\text{(IL)} \quad \frac{\langle E_1 \mid C\rangle_i^1 \rightarrowtail \langle E'_1 \mid C'\rangle_{i'}^1}{\langle E_1, E_2 \mid C\rangle_{[i,j]}^{[1,2]} \rightarrowtail \langle E'_1, E_2 \mid C'\rangle_{[i',j]}^{[1,2]}}$$

Interleaving Right:

$$\text{(IR)} \quad \frac{\langle E_2 \mid C\rangle_j^2 \rightarrowtail \langle E'_2 \mid C'\rangle_{j'}^2}{\langle E_1, E_2 \mid C\rangle_{[i,j]}^{[1,2]} \rightarrowtail \langle E_1, E'_2 \mid C'\rangle_{[i,j']}^{[1,2]}}$$

Weak Concurrency:

$$\text{(WC)} \quad \frac{\begin{array}{c}\langle E_1 \mid C_1 \uplus C_2 \uplus D\rangle_i^1 \rightarrowtail \langle E'_1 \mid C'_1 \uplus C_2 \uplus D\rangle_{i'}^1 \\ \langle E_2 \mid C_1 \uplus C_2 \uplus D\rangle_j^2 \rightarrowtail \langle E'_2 \mid C_1 \uplus C'_2 \uplus D\rangle_{j'}^2\end{array}}{\langle E_1, E_2 \mid C_1 \uplus C_2 \uplus D\rangle_{[i,j]}^{[1,2]} \rightarrowtail \langle E'_1, E'_2 \mid C'_1 \uplus C'_2 \uplus D\rangle_{[i',j']}^{[1,2]}}$$

Figure 4. The CHR Concurrent Refined Semantics

the head of the stack is activated and introduced to the store, followed by the search for matching rules. The activated constraint is dropped once the search is completed and the CHR rule has fired.

5.2 The Concurrent Refined Semantics

In a concurrent setting, we wish to have n execution stacks operating simultaneously on the shared constraint store. Here is sample derivation using two execution stacks.

$$\begin{array}{ll} & \langle [\texttt{Gcd(0)}\#1, \texttt{Gcd(4)}\#2], [\texttt{Gcd(4)}\#3] \mid \\ & \{\texttt{Gcd(0)}\#1, \texttt{Gcd(4)}\#2, \texttt{Gcd(4)}\#3\}\rangle \\ \rightarrowtail & \langle [\texttt{Gcd(4)}\#2], [\texttt{Gcd(0)}\#4, \texttt{Gcd(4)}\#5] \mid \\ & \{\texttt{Gcd(0)}\#4, \texttt{Gcd(4)}\#5\}\rangle \\ \rightarrowtail & \langle \epsilon, [\texttt{Gcd(4)}\#5] \mid \{\texttt{Gcd(4)}\#5\}\rangle \\ \rightarrowtail & \langle \epsilon, \epsilon \mid \{\texttt{Gcd(4)}\#5\}\rangle \end{array}$$

In contrast to the single-threaded refined semantics, we assume that all constraints from the execution stacks are already copied into the store. Making all constraints visible in the store immediately allows possible matches to be found earlier. For the above example, if $\texttt{Gcd(4)}\#2$ is not immediately added to the store, the second execution stack is not able to make progress and hence drops $\texttt{Gcd(4)}\#3$. The first execution stack has then to do all the remaining work. There is obviously the issue of a "fair" distribution among the execution stacks which we will ignore here.

Figure 4 contains the formal description of our concurrent refined semantics. For simplicity, we only consider the case of two execution stacks. In the concurrent semantics, each state $\langle E_1, E_2 \mid C\rangle_{[i,j]}^{[x,y]}$ is indexed by the thread number x and y and the most recently activated number i and j in each thread. Thus, each thread can independently create unique identifiers when activating constraints in the interleaving semantics. The interleaving semantics describes the possible derivation steps for each single execution stack.

We first take a look at the interleaving semantics. In (Simp), firing of a CHR rule activates the right-hand side constraints B. As said before, we add B to the execution stack and constraint store. The auxiliary function $Cons$ returns an un-numbered constraint set and is defined as $Cons(c_1\#n_1, ..., c_m\#n_m) = \{c_1, ..., c_m\}$. Function $Label$ turns an un-numbered constraint into a numbered constraint and is defined as $Label(\{c_1, .., c_n\}, x, i) = \{c_1\#x_{1+i}, .., c_n\#x_{n+i}\}$. We write $c\#x_i$ to denote a (concurrent) numbered constraint with the identifier x_i. We write $\uplus$ to denote multi-set union. Execution stacks are represented by (Haskell) lists and we write $E_1 + +E_2$ to denote list concatenation and $c : E$ to denote an execution stack with the top element c and tail E.

Rule (D1) covers the case of an active constraint which does not trigger any CHR rule and therefore can be dropped (removed). Rule (D2) applies if the active constraint is not in the store and hence must have been consumed by an earlier derivation. We drop the active constraint then.

Next, we take a look at the concurrent semantics. Rules (IL) and (IR) describe the derivation step taken by using either execution stack. Rule (WC) shows the concurrent execution of derivation steps as long as they do not interfere.

Let us consider another example using two CHR rules $\texttt{rule}\ C \Longleftrightarrow B$ (R1) and $\texttt{rule}\ A, B \Longleftrightarrow D$ (R2). We assume that A, B and C are some primitive constraints. In the interleaving semantics, we find that

$$T_1 : \langle [A\#2] \mid \{C\#1, A\#2\}\rangle \rightarrowtail_{D1} \langle \epsilon \mid \{C\#1, A\#2\}\rangle$$

and

$$T_2 : \langle [C\#1] \mid \{C\#1, A\#2\}\rangle \rightarrowtail_{R1} \langle [B\#3] \mid \{B\#3, A\#2\}\rangle$$

For simplicity, we drop the thread and number supply indices. Thread T_1 applies rule (D1) whereas thread T_2 applies the CHR rule (R1). It seems that there is the potential problem of failing to apply CHR rule (R2). However, constraint $B\#3$ is now active in thread T_2 and will eventually fire rule (R2). Hence, we can guarantee the exhaustive application of CHR rules in the current refined semantics.

5.3 Implementation Highlights

We have implemented the concurrent refined semantics (Figure 4) using the small step search strategy. In our implementation, the choice between the interleaving and weak concurrency steps

Multi-threaded Solver																		
Program	Gcd						Prime						Blockworld					
SMP Flags	Off			On			Off			On			Off			On		
Threads	1	2	8	1	2	8	1	2	8	1	2	8	1	2	8	1	2	8
Solver A (sec)	92.5	109.0	152.0	95.0	120.5	149.0	32.0	48.5	53.5	32.0	45.0	55.5	21.0	31.5	39.0	22.5	25.0	32.5
Solver B (sec)	93.0	21.0	13.0	95.5	37.0	29.0	33.5	34.0	36.0	31.5	31.0	31.0	23.0	23.5	33.0	22.0	21.5	34.5
Solver C (sec)	91.0	28.5	13.0	92.0	18.0	9.5	34.0	35.5	35.0	34.0	22.0	21.5	23.5	24.0	34.0	23.0	13.5	23.5

Figure 5. Experimental Results

is decided by the search strategy and the STM runtime system. Effectively, rule (WC) applies if the search yields non-overlapping matches and the STM validates consistency and commits of both derivation steps.

To further minimize the amount of interference among CHR derivation steps, we refine our small step search strategy. We assume that each thread has its own distinct entry point to the shared linked list which represents the constraint store. For example, consider two threads

$$T_1 \text{ Active: } \texttt{Gcd(4)}\#2 \qquad \text{T}_2 \text{ Active: } \texttt{Gcd(6)}\#3$$

operating on the constraint store (assuming the below textual ordering among constraints)

$$\{\texttt{Gcd(2)}\#1, \texttt{Gcd(4)}\#2, \texttt{Gcd(6)}\#3, \texttt{Gcd(2)}\#4\}$$

We employ the CHR rules we have seen earlier

```
rule Gcd(0) <==> True
rule Gcd(m),Gcd(n) <==> m>=n && n>0 | Gcd(m-n),Gcd(n)
```

The constraint $\texttt{Gcd(2)}\#1$ is a common partner for both active constraints. Hence if both threads start their search from the same entry point to the store, the derivations will be interleaved. Therefore, we assume that each thread is given a distinct entry point (effectively, we use a circular linked list). Thus, thread T_1 finds the partner constraint $\texttt{Gcd(2)}\#1$ and T_2 finds the (distinct) partner constraint $\texttt{Gcd(2)}\#4$. Then, each thread can concurrently execute the second CHR rule.

6. Experimental Results

Figure 5 summarizes some preliminary experimental results of our implementation of the concurrent semantics. For measurement we used a Mac PC, 1.83 GHz Intel-Pentium Duo-Core system with 512 MB RAM running Mac OS X. For compilation we used GHC 6.6 with support for symmetric multi-processors (smp). The results are given in seconds and averaged over multiple tests.

Our experiments aim at measuring three quantities: effects of concurrency (1,2 or 8 threads), effects of parallelism (1 or 2 processors) and the effects of the various matching constraint search strategies represented by three solver settings. Solver A implements the big step search strategy where each execution thread uses the same entry point to the constraint store. Solver B uses the small step search strategy and but still assumes the same entry point for each thread to the constraint store. Solver C use the small step search strategy and distinct entry points for each execution thread. Our experiments clearly show that solver C benefits the most from concurrency and parallelism.

We run each solver on three test programs: *Gcd* (Greatest Common Divisor) finds the Gcd of the first 4000 multiples of n , *Prime* finds the first 4000 prime numbers and *Blockworld* is a simulation of two autonomous agents each moving 100 distinct blocks between non-overlapping locations.

The *Gcd* example benefits the most from concurrency and parallelism. Interestingly, the *Blockworld* example slows down in case of eight threads. A possible explanation is that the eight threads need to be distributed among two processors. Therefore, there might be some interference between the threads. It will be interesting to measure the performance when the number of threads corresponds to the number of processors.

An unexpected result uncovered by our experiments is the improved performance of the multi-threaded solver over the single threaded solver, even on a uniprocessor setting. This apparently counter-intuitive result is due to the nature of the match searching operation. Note that the cost of a match failure (complete search required) is greater than a match success (if one exist). In the single threaded solver, active constraints are processed strictly in a sequential order. In an n-threaded solver, we process *n* active constraints concurrently. Hence, even on a uniprocessor, progress made by a match success need not be delayed by failed matches of active constraints processed earlier (in a sequential ordering). Note that this result is highly program dependent, and does not apply in general as indicated by the experiment results.

7. Conclusion

We discussed two concurrent implementations of CHR. First, we gave a concurrent implementation of the declarative semantics. Then, we devised a concurrent variant of the refined semantics. A prototype of the concurrent refined semantics implemented in Haskell STM can be downloaded via

`http://taichi.ddns.comp.nus.edu.sg/taichiwiki/CCHR`

Our experimental results show that the run-time performance of CHR gains significantly from concurrency and we can exploit parallelism when executing CHR under a multi-core architecture.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[2] Common CHR implementations. http://www.cs.kuleuven.ac.be/~dtai/projects/CHR.

[3] G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proc of ICLP'04*, pages 90–104, 2004.

[4] G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable confluence for constraint handling rules. Technical Report CW 452, Katholieke Universteit Leuven, 2006. Proc. of CHR 2006, Third Workshop on Constraint Handling Rules.

[5] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[6] T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence analysis. In *Proc. of CLP*, volume 3668 of *LNCS*, pages 113–127. Springer-Verlag, 2005.

[7] T. Frühwirth. Constraint handling rules: the story so far. In *Proc. of PPDP '06*, pages 13–14. ACM Press, 2006.

[8] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[9] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP'05*, pages 48–60. ACM Press, 2005.

[10] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.