Compiling Constraint Handling Rules with Lazy and Concurrent Search Techniques

Martin Sulzmann and Edmund S. L. Lam

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

Abstract. Constraint Handling Rules [5] (CHR) is a concurrent commited-choice constraint programming language to describe transformations (rewritings) among multi-sets of constraints. One of the main CHR execution tasks is to search for constraints from the store which match the constraints in the CHR rule head. In this paper, we demonstrate that laziness and concurrency are highly useful features to implement this search task efficiently.

1 Introduction

Constraint Handling Rules [5] (CHR) is a high level concurrent commited-choice constraint programming language to describe transformations (rewritings) among multi-sets of constraints (atomic formulae). Originally, CHR was designed to write incremental constraint solvers, but CHR is now used in a wide range of other applications such as type system design and agent specification. The semantics of CHR and its compilation is a fairly well studied subject. [2, 7, 10] Traditionally, CHR are compiled into logic languages such as Prolog and HAL and a number of efficient implementations exist. CHR is also implemented in Java [1] as well as the functional programming language Haskell [4]. One of the main compilation tasks is to search for constraints from the store which match the constraints in the CHR rule head.

In this paper, we demonstrate that laziness and concurrency are highly useful features to implement this search task efficiently. Lazy evaluation is the technique of delaying computations until the time where the result of the computation is required, thus allowing us to define operations like building the match tree in a purely declarative manner, without the worry of any runtime performance penalties. Concurrency on the other hand allow us to specify search tasks in parallel, hence possibly improving the performance of CHR execution. For this paper, we demostrate an implementation of the lazy and concurrent CHR match searching in Haskell [6], which is a lazy functional programming language with powerful concurrency abstractions [9].

Specifically, we make the following contributions:

- We first show how laziness can be used to implement the task of searching for matching constraints declaratively and thus more elegantly (Section 4). Existing CHR optimizations such as optimal join ordering and early guard scheduling can be easily integrated into our approach.

- Next, we employ concurrency to implement the search task more efficiently (Section 5). As in the previous case, existing CHR optimization methods can be integrated easily.
- We also develop a hybrid approach which combines laziness and concurrency (Section 6).
- We review a number of practical CHR examples which benefit from our approach (Section 7).

We continue in Section 2 where we highlight the key ideas of our lazy search method. Section 3 reviews background material on CHR. We conclude in Section 8 where we also discuss related work.

2 A Motivating Example

A CHR program is essentially a set of rules that describes rewritings among a multiset of constraints (atomic formulae) until a fixed-point is reached. A simple example of a CHR program is

$$A, B, C \Leftrightarrow D \qquad A, B, C \Rightarrow D \qquad A \setminus B, C \Leftrightarrow D$$

This example shows the three main types of CHR rules. The first rule is a *simplification* rule which says "if constraints A,B and C are in the store, then delete them and add D" while the second (middle), a *propagation* rule says "if constraints A,B and C are in the store, then add D". The right most rule is known as a *simpagation* rule which is a mix between simplification and propagation rules: Constraints before the '\' (A) are not deleted, while constraint after (B and C) are. Hence a significant part of CHR execution is the searching for sets of constraints in a constraint store that match rule heads (left-hand-side of rules). Particularly propagation rules need *all* possible matches, while simplification rules *only one*. This search space can be abstractly viewed as a *match tree*, which is an n-ary tree where each node contain a numbered constraint (constraints paired with unique integers to distinguish multiple copies) from the constraint store. For instance, given the following constraint store S,

$$S \equiv \{A\#1, B\#2, B\#3, C\#4, C\#5\}$$

we have a search space represented by the following match tree, assuming that we search in a left to right ordering of the rule heads:

$$\begin{tikzcd}
 & & & A\#1 \arrow[dll] \arrow[drr] & & & \\
 & B\#2 \arrow[dl] \arrow[dr] & & & & B\#3 \arrow[dl] \arrow[dr] & \\
C\#4 \arrow[d] & & C\#5 \arrow[d] & & C\#4 \arrow[d] & & C\#5 \arrow[d] \\
M1 & & M2 & & M3 & & M4
\end{tikzcd}$$

With lazy functional programming, we can implement the search for matching constraints elegantly as follows:

```
data Cons = A | B | C | D
data MTree = MTree Cons [MTree]

buildMTree :: [Cons] -> [Cons] -> [MTree]
buildMTree (c:cs) store =
  let candidates = filter (==c) store
  in map (\c' -> MTree c' (buildMTree cs store)) candidates
buildMTree [] _ = []

propMatches :: [MTree] -> [[Cons]]
propMatches ((MTree c mts'):mts) =
  (map (c:) (propMatches mts')) ++ (propMatches mts)
propMatches [] = []

simpMatch :: [MTree] -> [Cons]
simpMatch mts = head (propMatches mts)
```

Functions `propMatches` and `simpMatch` are essentially defined by the same code. The first argument (`store`) represents the constraint store, while the second represents the left hand side of the rule. Lazy evaluation ensures that when `findOneMatch` is called, only the first match is evaluated, thus never suffers from unneccessary runtime computations of unrequired matches. Note that lazy evaluation also allow us to separately define the search space construction `buildMTree` and the searching routines declaratively without performance penalty. The matching tree may seem redundant here when defined only for simplification and propagation matches, however when we introduce simpagation rule matches (Section 4.2), where an efficient implementation involves pruning techniques, its purpose will be more significant.

We have made a few simplifying assumptions in the above framework: First, constraints are just propositional, hence matching here is simple equality. Also, CHR rules traditionally include guard conditions which must be checked before a match can be commited. We will address these extensions Section 4.

3 Constraint Handling Rules

3.1 Syntax and Semantics

We review the syntax and refined operational semantics of CHR. CHR describes multi-set rewriting of constraints, which are either builtin constraints or CHR constraints. These rewritings are specified by a set of CHR simpagation rules of the form:

$$r \ @ \ H_1 \setminus H_2 \Leftrightarrow g \mid C$$

We call H_1 the propagation heads and H_2 the simplification heads of the rule, each of which are sequences of CHR constraints. g is a builtin constraint referred to as the guard and C, a sequence of CHR and builtin constraints is the body of the rule. We assign each constraint in H_1 and H_2 a unique integer, known as the *occurrence number*. We shall denote a simpagation rule with empty propagation head (simplification rule) as $r \ @ \ H \Leftrightarrow C$ and a simpagation rule with empty simplification head (propagation rule) as $r \ @ \ H \Rightarrow C$. We denote the empty list as ϵ and the empty set as $\emptyset$. Multiset union is denoted by $\uplus$ and we sometimes treat sequences (lists) as multisets if ordering is unimportant.

To differentiate among copies of the same CHR constraints in the multiset store, we introduce *numbered CHR constraint*, which is a CHR constraint c paired with an integer n denoted by $c\#n$. We will refer to this integer n as the *identifier* of the constraint c. For convinence, we define auxiliary functions $Cons(c\#n)$ and $Id(c\#n)$ which returns c and n respectively, and overload them to lists and multisets of constraints in the obvious manner. A CHR execution state is the tuple:

$$\langle G, S, B, T\rangle_n$$

where G is a multiset of constraints known as the *goals*, S is a multiset of CHR constraints (*constraint store*), B is a builtin constraint *builtin store*, T is the propagation history which is a set of sequences of constraint identifiers appended with a rule name. Finally, n is an integer representing the next free identifier.

Informally the refined CHR semantics defines the exhaustive application of CHR rules, triggered by a multiset of goal constraints. The declarative semantics is based on the following 3 transitions, which maps execution states to execution states C to C', denoted $C \rightarrowtail C'$.

$$\boxed{\langle G, S, B, T\rangle_n \rightarrowtail \langle G, S, B, T\rangle_n}$$

1. **Solve**
$$\langle \{b\} \uplus G, S, B, T\rangle_n \rightarrowtail \langle G, S, b \wedge B, T\rangle_n$$
where b is a built-in constraint.
2. **Introduce**
$$\langle \{c\} \uplus G, S, B, T\rangle_n \rightarrowtail \langle G, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$$
where c is a CHR constraint.
3. **Apply**
$$\langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n \rightarrowtail \langle \theta(C) \uplus G, H_1 \uplus S, B, T'\rangle_n$$
where
$$\begin{array}{l} \exists(r \;@\; H_1' \setminus H_2' \Leftrightarrow g \mid C) \wedge \\ \exists\theta, \text{ a matching substitution such that:} \\ \qquad Cons(H_1) \equiv \theta(H_1') \\ \qquad Cons(H_2) \equiv \theta(H_2') \\ \qquad CT \models_S B \rightarrow \exists_r(\theta \wedge g) \\ Id(H_1) + +Id(H_2) + +[r] \notin T \\ T' \equiv \{Id(H_1) + +Id(H_2) + +[r]\} \cup T \end{array}$$

The **Solve** transition passes a new built-in constraint to the built-in store, while the **Introduce** transition does the same for CHR constraints to the constraint store. The **Apply** transition non-deterministically picks a rule and a subset of the constraint store that contains constraints matching the heads of the rule and applies the rewrite specified by the rule, provided the guard is entailed from the constraint theory CT. The history list T keeps track of all rule combinations that has fired, so as to ensure that each set of constraints matching a rule head fires a rule at most once.

The declarative semantics is highly non-deterministic. This is because it declaratively specifies the operational behaviour of CHR programs, but do not impose explicitly an order in which goals are processed and an order in which rule matches are tried. Most CHR systems implement the refined operational semantics [3] which defines a more deterministic execution model of CHR programs. Informally, the refined semantics execute CHR goals in left-to-right depth first order, which essentially means treating the *goals* G of a CHR execution state as a stack: only the left most constraint (*active*) may be executed for any given state, and newly added constraints are added to the left (head) of G. Further more, an *active* goal constraint must be matched with rule heads in a specific order, normally assumed to be the top down and left-to-right order of appearence of the CHR rule heads. Note that the matching tree abstraction is also compatible with the refined semantics, simply by only allowing match trees to be construction from the active goal constraint as the root.

Consider the abstract CHR program shown in Figure 1 which consists of 3 rules: $r1$ is a *simplification* rule (no propagate heads), $r2$ a *propagation* rule (no simplify heads) and $r3$ a *simpagation* rule. The following shows a constraint store S from the CHR program in figure 1, and a corresponding match tree containing matching heads of the $r3$ rule under the active constraint $a(5, W)$.

Constraint Store S
$S \equiv \{a(5,R)\#1, c(R,3)\#2, c(R,2)\#3, b(2)\#4, b(8)\#5\}$

Match Tree of $r3$ @ $a(X,Y) \setminus b(W), c(Y,Z) \Leftrightarrow X > Z \mid d(X,W,Y,Z)$

$$
\begin{array}{lcccccccc}
\mathit{Prop}\ a(X,Y): & & & & a(5,R)\#1 & & & & \\
 & & & \swarrow & & \searrow & & & \\
\mathit{Simp}\ c(Y,Z): & & c(R,3)\#2 & & & & c(R,2)\#4 & & \\
 & \swarrow & & \searrow & & \swarrow & & \searrow & \\
\mathit{Simp}\ b(W): & b(2)\#5 & & & b(8)\#6 \quad b(2)\#5 & & & & b(8)\#6 \\
 & \downarrow & & & \downarrow \qquad\quad \downarrow & & & & \downarrow \\
\mathit{Match}: & \theta_1 & & & \theta_2 \qquad\quad \theta_3 & & & & \theta_4
\end{array}
$$

where $\theta_1 \equiv \{X \mapsto 5, Y \mapsto R, Z \mapsto 3, W \mapsto 2\}$ $\quad\theta_2 \equiv \{X \mapsto 5, Y \mapsto R, Z \mapsto 3, W \mapsto 8\}$
$\theta_3 \equiv \{X \mapsto 5, Y \mapsto R, Z \mapsto 2, W \mapsto 2\}$ $\quad\theta_4 \equiv \{X \mapsto 5, Y \mapsto R, Z \mapsto 2, W \mapsto 8\}$

We assume for now that the order in which partner constraints are searched for is arbitrally chosen. In practice, optimal join ordering is determined by analysis techniques detailed in [2]. A CHR rule head *match* is a pair consisting of a set of matching constraints and the matching substitution that matches it to the CHR rule head. Note that for the above matching tree, we will fire the matches $\{(\theta_1, \{\#1, \#2, \#5\}), (\theta_4, \{\#1, \#4, \#6\})\}$ as each contains their own unique simplification heads ($\{\#2, \#5\}$ and $\{\#4, \#6\}$ respectively). Another alternative set of matches that may fire is $\{(\theta_2, \{\#1, \#2, \#6\}),\ (\theta_3, \{\#1, \#4, \#5\})\}$. If we consider the simplification rule $r1$ instead we can only fire a single match, while in the other extreme case (propagation rule $r2$), we can fire all matches (assuming that none has been fired before).

4 Searching For Matching Constraints Lazily

We refine the approach from Section 2 to handle the full CHR semantics, by including matching substitutions and guard constraints. First, we consider the

$$r1 \ @ \ a(X,Y), b(W), c(Y,Z) \Leftrightarrow X > Z \mid d(X,W,Y,Z)$$
$$r2 \ @ \ a(X,Y), b(W), c(Y,Z) \Rightarrow X > Z \mid d(X,W,Y,Z)$$
$$r3 \ @ \ a(X,Y) \setminus b(W), c(Y,Z) \Leftrightarrow X > Z \mid d(X,W,Y,Z)$$

Fig. 1. An Example CHR Program

```
data Term     = Value String | Var String
data Cons     = Cons { symbol::String , args::[Term] }
              | NumCons { nsymbol::String , nargs::[Term] , nid::Int }
data RuleHead = RuleHead { htype::HeadType , cons::Cons }
data HeadType = Simp | Prop
type Subst    = [(Term,Term)]

apply   :: Subst -> Cons -> Cons
compose :: Subst -> Subst -> Subst
match   :: Cons -> Cons -> Maybe Subst
```

Fig. 2. CHR Data Representation & Matching Interfaces

special cases of simplification and propagation CHR in Section 4.1. As observed earlier, in case of a simplification CHR we only require to find *one* match whereas for a propagation CHR we want to find *all* matches. The situation gets more complicated in case of simpagation CHR rules, which we will address in Section 4.2.

Figure 2 shows the representation of terms and constraints in Haskell data types `Term` and `Cons`. The `RuleHead` and `RuleType` datatypes describe rule heads, which are essentially constraints tagged with a head type indicating if it is a simplification `Simp` or propagation `Prop` head. These are followed by interfaces of substitution and matching over terms and constraints in 3 functions `apply`, `compose` and `match` with their obvious meanings. An *occurrence rule head compilation* is thus represented by a list of `RuleHead` datatypes.

4.1 Lazy Simplification and Propagation Match Search

We show now how simplification (one match) and propagation matches (all matches) can be implemented lazily. Figure 3 shows the matching tree data structure `MTree` and a naive implementation of the CHR constraint store `ListCHRStore`, which is simply a list of constraints. The function `getCandidates` retrieves all matching constraints from a constraint store given the constraint to be matched. The `SearchTask` data type represents the matching search sequence for each CHR rule occurrence. A search task `Lookup c` represents the task of branching the search with constraints in the store matching `c`, while `Guard g` means that the guard `g` can be schedule at the current node (hence pruning away nodes that do not pass the guard). Hence for a CHR rule $r \ @ \ A, B, C \Rightarrow g \mid D$, we assume that its head and guard is compiled into 3 basic matching search sequence (as-

```
type ListCHRStore   = [Cons]
data MTree          = MNode RuleHead [MTree] | MLeaf Subst

getCandidates :: Cons -> ListCHRStore -> [(Subst,Cons)]
getCandidates p (c:cs) =
    case match p c of
        Just sub -> (sub,c):(getCandidates p cs)
        Nothing -> getCandidates p cs
getCandidates _ [] = []

data SearchTask = Lookup RuleHead
                | Guard (Subst -> Bool)

buildMTree :: Subst -> [SearchTask] -> ListCHRStore -> [MTree]
buildMTree sub ((Guard g):ts) st =
  if g sub then buildMTree sub ts st else []
buildMTree sub ((Lookup r):ts) st =
  let RuleHead h c = r
      ms = getCandidates (apply sub c) st
  in map (\(sub',c') -> MTree (RuleHead h c') (buildMTree (compose sub sub') ts st)) ms
buildMTree sub [] _ = [MLeaf sub]

data Match = Match { subst::Subst , heads::[RuleHead] }

propMatches :: [RuleHead] -> [MTree] -> [Match]
propMatches rs ((MNode r mts'):mts) = (propMatches (rs++[r]) mts')++(propMatches rs mts)
propMatches rs ((MLeaf sub):mts) = (Match sub rs):(propMatches rs mts)
propMatches _ [] = []

simpMatch :: [RuleHead] -> [MTree] -> Match
simpMatch rs mts = head (propMatches rs mts)
```

Fig. 3. MTree Builder & Simplification/Propagation Rule Head Match Search

suming left to right ordering of rule heads), one for the search sequence starting from each rule head occurrence:

$$\begin{aligned}
occurrenceA &\equiv [Lookup\ A, Lookup\ B, Lookup\ C, Guard\ g]\\
occurrenceB &\equiv [Lookup\ B, Lookup\ A, Lookup\ C, Guard\ g]\\
occurrenceC &\equiv [Lookup\ C, Lookup\ A, Lookup\ B, Guard\ g]
\end{aligned}$$

This compilation scheme of CHR rule heads is almost similiar to existing schemes. Note that basic CHR optimizations [7, 2, 10] can be applied to this compilation. The function `buildMTree` returns all possible matching trees given a constraint store and a list of `SearchTask`. Note that we can use the refined semantics to 'drive' the search by supplying an active constraint as the root of a `MTree`. An important point is that the `buildMTree` declaratively specifies the entire match tree that represent the search space of finding rule head matches from a constraint store. However, in a lazy language like Haskell subtrees of the

```
data TVar a
newTVar   :: STM (TVar a)       writeTVar   :: TVar a -> a -> STM ()
readTVar  :: TVar a -> STM a    atomically  :: STM a -> IO a
```

Fig. 4. Haskell Transactional Memory Interfaces

mtree are only ever computed when their values are explicitly required. This applies also to the `getCandidate` function.

The data type `Match` represents a match pair consisting of matching substitution of type `Subst` and the set of matching rule heads of type `[RuleHead]`. Given a list of `MTree`, *propagation* and *simplification* matches can be extracted by the the `propMatches` (returns all matches from a `MTree`) and `simpMatch` (returns left most match from a `MTree`) functions. The `simpMatch` function is defined as the head (left most) element of the result of `propMatches` and thanks to laziness, the tail of the list is never evaluated.

4.2 Lazy Simpagation Match Search

Simplification and propagation are special cases where we must retrieve one or all of the matches, tasks which requires no search. In this section we show how an efficient match search routine for simpagation rules can be implemented. Since, the former rules can be treated as special simpagation rules, this implementation subsumes that of the previous subsection. Consider the following matching tree, from the example in Section 3.1:

$$
\begin{array}{llllllll}
\mathit{Prop}\ a(X,Y): & & & & a(5,R)\#1 & & & \\
 & & & \swarrow & & \searrow & & \\
\mathit{Simp}\ c(Y,Z): & & c(R,3)\#2 & & & & c(R,2)\#4 & \\
 & \swarrow & & \searrow & & \swarrow & & \searrow \\
\mathit{Simp}\ b(W): & b(2)\#5 & & b(8)\#6 & b(2)\#5 & & & b(8)\#6 \\
 & \downarrow & & \downarrow & \downarrow & & & \downarrow \\
\mathit{Match}: & \theta_1 & & \theta_2 & \theta_3 & & & \theta_4
\end{array}
$$

We wish to find the matches $\{(\theta_1, \{\#1, \#2, \#5\}), (\theta_4, \{\#1, \#4, \#6\})\}$ as they each have unique simplification heads. A naive way to do this is to extract all matches and linear filter away overlapping matches. However we wish to do this efficiently, by pruning away overlapping matches in the `MTree`. Note that commiting to θ_1 not only invalidates θ_2 (a sibling), it also invalidates θ_3 (non-sibling). Hence to prune away invalidate matches, we can 'thread' through a list of constraints (possibly indexed by constraint id) representing the constraints which already has been commited. This solution however introduce the additional burden of keeping track of and indexing a simplification head 'history' list.

A better solution would be to allow the search procedures to immediately physically commit to the match (delete simplification heads from the constraint store) after verifying that all its simplification heads are still in the store. If simplification heads cannot be verified (already deleted), the match is dropped. Thus we do not need to keep track of any additional data structures, while the match search procedure additionally contains the *side-effects* of deleting simplification head constraints from the store (a task which must anyway be done during execution of CHR). However, representing the store as a pure list `ListCHRStote` and the pure functions in Figure 3 do not allow us to specify side effects in Haskell. Thus we look to *monads* and mutable references in Haskell. *Monads* provides the Haskell programmer a mechanism to define operations with side-effects, as well as an interface to define imperative style operations. The key idea of monads is as follows: a type of `IO a` where `IO` is a monad, represents an `IO` operation that performs some action (which may contain side-effects) and produces a value of type `a`.

Figure 4 shows the interface for the `TVar` data type which is a mutable transactional memory reference and some monadic operations. A type `TVar a` is a reference to a value of type `a`. `newTVar`, `readTVar` and `writeTVar` provides creation, read and write interfaces to transaction memory. Note that transactional memory operations are executed only on the `STM` monad. The `atomically` operation simply executes an `STM` operation as an `IO` operation. For reasons that will be clearer in the next section (Section 5) we have chosen to use transactional memory as our store references. However, for the purpose of this section, it is suffice to treat `TVar`'s as standard mutable reference. Monads also introduces strictness (as oppose to laziness), hence by default a monadic operation `IO a` must be executed in an imperative sequence. However, the Haskell libraries do provide primitives to allow programmers to write *non-strict* monadic operations. For instance, we can write a lazy `map IO` monad operation as follows:

```
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 []
```

The `unsafeInterleaveIO :: IO a` $\rightarrow$ `IO a` library operation takes an `IO` action of type `IO a`, but delays the execution of this action until the value of `a` is required. Figure 5 illustrates the interfaces of an implementation of the CHR Store on shared memory. For brevity we only show the interfaces of the reference CHR store with the assumption that CHR store is implemented on transactional memory. `getCandidatesIO` and `buildMTreeIO` are the monadic versions of the `MTree` building operations. Note that we define them lazily with the help of the `unsafeInterleaveIO` operation. `removeAllOrNone`, whose purpose will be clear soon, is an STM operation which works as follows: check that a given list of constraints is in the shared store and if so remove all from the store and return true, otherwise just return false.

Figure 6 shows the `simpagateMatchesIO` operation which implements the search for matches with non-overlapping simplification heads. It is summarized by the following: If the current node is an `MNode` search it only if the constraint it represents is still in store, by returning one match (simplification) or all matches (propagation). If it is an `MLeaf`, delete all simplification heads from the store and return the match. Note that `deleteAllOrNone` seemingly would never fail and always return true. However, this is not true when we consider a concurrent search technique (Section 6).

```
data RefCHRStore
deleteFromStore   :: SharedCHRStore -> Cons -> STM ()
isStored          :: SharedCHRStore -> Cons -> STM Bool
getStoredContents :: SharedCHRStore -> STM [Cons]

getCandidatesIO :: Cons -> SharedCHRStore -> IO [(Subst,Cons)]
getCandidatesIO c st = do
  ls <- getStoredContents st
  return (getCandidates c ls)

buildMTree :: Subst -> [SearchTask] -> ListCHRStore -> [MTree]
buildMTree sub ((Guard g):ts) st =
  if g sub then buildMTree sub ts st else []
buildMTree sub ((Lookup r):ts) st =
  let RuleHead h c = r
      ms = getCandidates (apply sub c) st
  in map (\(sub',c') -> MTree (RuleHead h c') (buildMTree (compose sub sub') ts st)) ms
buildMTree sub [] _ = [MLeaf sub]

buildMTreeIO :: Subst -> [SearckTask] -> RefCHRStore -> IO [MTree]
buildMTreeIO sub ((Guard g):ts) st = do
  case g sub of
    True   -> buildMTreeIO sub ts st
    False  -> return [] buildMTreeIO sub ((Lookup r):rs) st = do
  let RuleHead h c = r
  ms <- getCandidatesIO (apply sub c) st
  mapIOLazily (buildMTreeIO' sub r rs st) ms
  where
    buildMTreeIO' sub (RuleHead h c) rs st (sub',c') = do
      mts <- buildMTreeIO (compose sub sub') rs st
      return (MTree (RuleHead h c') mts)
buildMTreeIO sub [] _ = return [MLeaf sub]

removeAllOrNone :: RefCHRStore -> [Cons] -> STM Bool
removeAllOrNone st cs = do
  bs <- mapM (isStored st) cs
  case and bs of
    True   -> do deleteFromStore st cs
                 return True
    False  -> return False
```

Fig. 5. Shared CHR Store & Monadic MTree Operations

5 Searching for Matching Constraints Concurrently

The previous section highlights the lazy approach of implementing CHR simpagation rule head matching. The match search function is essentially a tree search algorithm through the `MTree`, hence it is possible and beneficial to search

```
simpagateMatchesIO :: [RuleHead] -> RefCHRStore -> [MTree] -> IO [Match]
simpagateMatchesIO rs st ((MNode r mts):mts') = do
  isvalid <- atomically (isStored st (cons r))
  if isvalid then do
    mcs <- simpagateMatchesIO (rs++[r]) st mts
    mcs' <- unsafeInterleaveIO (simpagateMatchesIO rs st mts')
    case (htype r) of
      Simp -> do case mcs of
                   (m:_) -> return (m:mcs')
                   []    -> return mcs'
      Prop -> return (mcs++mcs')
  else simpagateMatchesIO rs st mts')
simpagateMatchesIO rs st ((MLeaf sub):mts) = do
  let simpheads = map cons (filter ((==Simp).htype) rs)
  succ <- atomically (deleteAllOrNone st simpheads)
  if succ then do
    mcs <- unsafeInterleaveIO (simpagateMatchesIO rs st mts)
    return ((Match sub rs):mcs)
  else simpagateMatchesIO rs st mts)
simpagateMatchesIO _ _ [] = return []
```

Fig. 6. Simpagation Rule Head Match Searching

```
data TChan a
newTChan     :: STM (TChan a)      writeTChan :: TChan a -> a -> STM ()
readTChan    :: TChan a -> STM a   forkIO     :: IO () -> IO ThreadId
```

Fig. 7. Transactional Channels & IO Forking Operation

branches of the tree concurrently. In this section, we present an alternative to the lazy match search of Section 4: concurrent matching searching.

Up to now, we have only used the `STM` transaction memory as standard mutable memory references. Yet in fact the STM monad actually provides a Haskell programmer a concurrency abstraction that supports composable operations and communication channels `TChan` between concurrent processes. Show in Figure 7, a `TChan` act as a channel of communication, which can be created, read and written into by the operations `newTChan`, `readTChan` and `writeTChan`. Recall that `STM` operations are executed from the `IO` monad with operation `atomically` (Figure 4). This execution of an `STM` operation is guaranteed to be *atomic* (An operation is fully executed as a single atomic action) and *isolated* (Effects of an uncompleted operation cannot be observed by other concurrent operations, until it has completed/commited).

Figure 8 shows a brute force implementation of the concurrent match search. The `concSimpagateMatchIO` operation computes a set of non-overlapping matches the brute force way: fork an IO thread for *every* branch of the `MTree`. Matches are

```
concSimpagateMatchIO :: [MTree] -> SharedCHRStore -> IO (TChan Match)
concSimpagateMatchIO mts st = do
  ms <- atomically newTChan
  mapM_ (\mt -> forkIO (concSimpagateMatchIO' [] ms st mt)) mts
  return ms
  where
    concSimpagateMatchIO' :: [RuleHead] -> TChan Match -> SharedCHRStore -> MTree -> IO ()
    concSimpagateMatchIO' rs tms st (MNode r mts) = do
      isvalid <- atomically (isStored st (cons r))
      if isvalid then do
        mapM_ (\mt -> forkIO (concSimpagateMatchIO' (rs++[r]) tms st mt)) mts
      else return ()
    concSimpagateMatchIO' rs tms st (MLeaf sub) = do
      let simpheads = map cons (filter ((==Simp).htype) rs)
      succ <- atomically (removeAllOrNone st simpheads)
      if succ then do
        atomically (writeTChan tms (Match sub rs))
      else return ()
```

Fig. 8. Brute Force Concurrent Match Search

gathered into a signal transactional channel which is returned as the final result. Each thread independently determine if their correspond match do not conflict with any other, simply by atomically verifying (`removeAllOrNone`) that their simplification heads have not been removed from the store and removing them if verification is successful. If this atomic operation is a success, the corresponding matching data can be placed in the transactional channel.

6 Lazy and Concurrent Match Search

The concurrent match search discuss in Section 5 effectively computes the entire `MTree`, ignoring any possible pruning of matches. This may mean that we will eventually compute many redundant matches, for instance only a single match is ever possible from a sub-tree with simplification root but the search of this sub-tree is still done by mutliple competing threads.

This can be highly inefficient at times and the concurrent match searching routine should be less aggressive when spawning threads to search sub-trees rooted by simplification nodes. The are several design decisions we can choose, one of which is to set a *trashhold* value for maximum number of threads for each branches once the search reach its first simplification node. A more interesting and balanced search strategy is to combine the lazy and concurrent search strategy, where concurrent search is performed up to the first simplification head from the top.

We provide, in Figure 9, the lazy and concurrent match search implementation in Haskell. Note that it prunes simplification nodes by executing a lazy search (`simpagateMatchesIO` of Figure 6) in place of the concurrent search. Also note that now the purpose of using `STM` transactional memory for the lazy search

```
lconcSimpagateMatchIO :: [MTree] -> SharedCHRStore -> IO (TChan Match)
lconcSimpagateMatchIO mts st = do
  ms <- atomically newTChan
  mapM_ (\mt -> forkIO (lconcSimpagateMatchIO' [] ms st mt)) mts
  return ms
  where
    lconcSimpagateMatchIO' :: [RuleHead] -> TChan Match -> SharedCHRStore -> MTree -> IO ()
    lconcSimpagateMatchIO' rs tms st (MNode r mts) = do
      isvalid <- atomically (isStored st (cons r))
      if isvalid then do
        case (htype r) of
          Simp -> do ms <- simpagateMatchesIO rs st [MNode r mts]
                     atomically (writeTChan tms (head ms))
          Prop -> mapM_ (\mt -> forkIO (lconcSimpagateMatchIO' (rs++[r]) tms st mt)) mts
      else return ()
    lconcSimpagateMatchIO' rs tms st (MLeaf sub) = do
      let simpheads = map cons (filter ((==Simp).htype) rs)
      succ <- atomically (removeAllOrNone st simpheads)
      if succ then do
        atomically (writeTChan tms (Match sub rs))
      else return ()
```

Fig. 9. Lazy and Concurrent Match Search Implementation

is clear as it allows us to integrated the lazy search with the concurrent search consistently (Using standard Haskell mutable memory (`MVars`) would not allow us to safely compose these two operations).

7 Practical Examples

In this section, we show some examples of actual CHR programs that may benefit from the lazy and concurrent search match strategy. Consider Greatest Common Divisor (GCD) CHR program with the initial store S

$$gcd1 \;@\; gcd(0) \Leftrightarrow True$$
$$gcd2 \;@\; gcd(n) \setminus gcd(m) \Leftrightarrow m > n \mid gcd(m-n)$$

$$S \equiv \{gcd(2)\#1, gcd(3)\#2, gcd(4)\#3, gcd(5)\#4, \}$$

Suppose we build a match tree for the second Gcd rule, with $gcd(2)\#1$ as the root matching with the propagation head $gcd(n)$:

$$\begin{array}{llll} Prop\ gcd(n): & & gcd(2)\#2 & \\ & \swarrow & \downarrow & \searrow \\ Simp\ gcd(m): & gcd(3)\#2 & gcd(4)\#3 & gcd(5)\#4 \\ & \downarrow & \downarrow & \downarrow \\ Match: & \theta_1 & \theta_2 & \theta_3 \end{array}$$

$$where\ \theta_1 \equiv \{n \mapsto 2, m \mapsto 3\}, \quad \theta_2 \equiv \{n \mapsto 2, m \mapsto 4\}, \quad \theta_3 \equiv \{n \mapsto 2, m \mapsto 5\}$$

The concurrent match search would spawn a thread for each branch to the simplfication heads matching $gcd(m)$, hence three variants of rule $gcd2$ can fire concurrently.

Next, consider the CHR program modelling an Authorized Access Only Buffer Protocol. Agents are allowed to execute two actions $get(Id, X)$ and $put(Id, X)$ to get and put integers into a shared channel $chan(Ch, Ids)$. However get and put actions of respective agents are only possible if it is authorized to access the channel.

$$\begin{array}{l} get\ @\ chan(Ch, Ids) \setminus get(Id, X), buf(Ch, Y) \Leftrightarrow Id \in Ids \mid X = Y \\ put\ @\ chan(Ch, Ids) \setminus put(Id, X) \Leftrightarrow Id \in Ids \mid buf(Ch, X) \end{array}$$

$$\begin{array}{ll} S \equiv \{ & chan(c1, [a1, a2])\#1, get(a1, M)\#2, get(a3, N)\#3, get(a2, P)\#4, \\ & buf(c1, 1)\#5, buf(c1, 2)\#6, ..., buf(c1, 100)\#104\} \end{array}$$

The store S consist of a single channel $c1$ which agents $a1$ and $a2$ has access, 3 get actions and 100 integers. Consider the match tree of the get rule with $chan(c1, [a1, a2])\#1$ as the root. We assume that the lazy and concurrent search strategy and early guard scheduling analysis has scheduled the guard $Id \in Ids$ right after the $get(Id, X)$ head matches are retrieved.

$$\begin{array}{lllll} Prop\ chan(Ch, Ids): & & chan(c1, [a1, a2])\#1 & & \\ & \swarrow & \downarrow & \searrow & \\ Simp\ get(Id, X): & get(a1, M)\#2 & get(a3, N)\#3 & get(a2, P)\#4 & \\ & \downarrow & & \downarrow & \searrow \\ Simp\ buf(Ch, Y): & buf(c1, 1)\#5 & & buf(c1, 1)\#5 & buf(c1, 2)\#6 \\ & \downarrow & & & \downarrow \\ Match: & \theta_1 & & & \theta_2 \end{array}$$

$$\begin{array}{ll} where & \theta_1 \equiv \{Ch \mapsto c1, Ids \mapsto [a1, a2], Id \mapsto a1, X \mapsto M, Y \mapsto 1\} \\ & \theta_2 \equiv \{Ch \mapsto c1, Ids \mapsto [a1, a2], Id \mapsto a2, X \mapsto P, Y \mapsto 2\} \end{array}$$

The above shows the sub-tree explored by the lazy and concurrent search strategy. Concurrent search is executed at the top level, hence the propagation node $chan(Ch, Ids)$ spawns a thread for each constraint matching $get(Id, X)$. Since the next nodes are simplification heads, lazy searches are executed from that point. This decision turns out to be a good one, since a brute force concurrent search would unneccessarily compute a match for each buffer and get action pairs, most of which have over-lapping simplification heads.

8 Conclusion and Related Works

We have highlighted the implementation of two search techniques for CHR rule head matching, namely lazy and concurrent search, which is compatiable with standard CHR compilation optimization techniques. Lazy evaluation allows us to define search routines declaratively while not forcing strict and unnecessary runtime computations. Concurrent searching allows us to explore the search space of rule head matches more efficiently. By combining the two approaches, we have a more well-balanced and practical search strategy.

Optimized CHR compilation techniques have been widely studied. An implementation of CHR in Haskell is also explored in [4]. CHR match search compilations discussed in [2] follows a strict order of processing CHR constraints similiar to Prolog style (left-to-right, depth first) evaluations, while our implementation is more closely related to [10] which allows multiple rule variants to fire from active constraints matching propagation rule heads. Yet to the best of our knowledge, we are the first to explicitly explore concurrent and lazy match search techniques. Local optimization techniques like optimal join-ordering and early guard scheduling discussed in [7, 2, 10] are compatible with our implementation.

In future, we intend to introduce these match searching techniques to our concurrent CHR implementation [8]. Other possible future works include integrating other existing CHR optimization techniques (eg. late storage, continuation optimizations, etc) into our frame work, as well as obtaining emprical results on the performance of various match search techniques discussed here.

References

1. K. u. leuven java constraint handling rules (jchr) system. http://www.cs.kuleuven.ac.be/~dtai/projects/CHR.
2. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, The University of Melbourne, 2005.
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*, volume 3132 of *LNCS*, pages 90–104. Springer-Verlag, 2004.
4. Gregory J. Duck. Haskell chr. http://www.cs.mu.oz.au/ gjd/haskellchr/.
5. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.
6. Haskell home page. http://www.haskell.org/.
7. 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.
8. E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory, 2007. Proc. of ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'07).
9. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96*, pages 295–308. ACM Press, 1996.
10. 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.