# Haskell-Join-Rules

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

There are numerous calculi and programming languages to support concurrent programming. A particular fruitful model appears to be the join calculus [3] which provides the basis for the concurrency abstractions found in JoCaml [9] and Polyphonic C# [1]. The join pattern concurrency abstractions provide for a more declarative and higher-level model of concurrency compared to the traditional model based on threads and locks, promising programmers a more simple and intuitive model of concurrent programming.

***Join patterns in Polyphonic C#:*** Here is a simple example in Polyphonic C#:

```
public class Buffer {
  public async Put(Int i);
  public string Get() & Put(Int i) { return i; }
}
```

The buffer class represents a simple unbounded communication buffer with 2 method calls. `Put(i)` is a asynchronous method call, which returns immediately to the caller. `Get()` is a synchronous method call, which will block and only returns to the caller if we find a partner `Put(i)` call, during which returns `s`. Join patterns (Chords) like this (`Get() & Put(i)`) defines useful synchronization patterns by associating a method body with one or more method calls.

Unfortunately, implementations of join patterns (Polyphonic C# and JoCaml) do not yet provide explicit support for general guard conditions, as efficient handling of guards is a non-trivial problem [1]. This is because triggering of guarded join patterns involves checking possibly all combinators of queued method calls.

***Modelling join patterns with Constraint Handling Rules (CHR):*** Interestingly, a similar model of concurrency has been developed independently in the context of logic and constraint programming. Constraint Handling Rules (CHR) [4, 5] is a concurrent committed-choice constraint logic programming language to exhaustively transform (rewrite) multi-sets of constraints. Initially, CHR were developed to specify incremental constraint solvers, CHR are now used in a multitude of other applications, from agent planning to type systems [6, 17, 16].

Constraints in a CHR multi-set (known as the store) can be viewed as asynchronous method calls. For instance, we can write the buffer example shown earlier, as follows:

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

This CHR simply states that if we find constraints matching the simplify heads $\{Get(x), Put(y)\}$, we simplify (delete and replace) it with $\{x = 1\}$. Note that the `Get` constraint has a argument `x` that models the return value of the method call. Most importantly CHR rules also consist of propagation heads and general guard conditions [8, 2, 14] where efficient optimization techniques have been widely studied. Hence if we compile join patterns into CHR rules, we get propagation and guards for free. Unfortunately, standard CHR semantics has no notion of synchronous constraints, or any form of control structures since it only defines exhaustive rewritings of a multi-set of constraints.

***Introducing Haskell-Join-Rules:*** Haskell-Join-Rules [11] combines the best of both worlds: We employ CHR for the coordination of concurrent computations where the underlying computations are carried out in a host langauge, specifically in this case Haskell [13]. From join calculus we adopt the concept of synchronous method calls and join patterns, from CHR we adopt a well established search technique for matching constraints that handles propagation and general guard conditions. We choose Haskell as our host language because Haskell is well-suited to develop domain-specific language extensions. Furthermore, we can easily take advantage of our concurrent implementation of CHR in Haskell [10] which we have built using the Glasgow Haskell Compiler (GHC) [7].

The buffer example is defined in Haskell-Join-Rules by the following:

```
Chord Get(x) & Put(y) = x `is` y

Get :: JRChan a -> UCons
Put :: JRChan a -> UCons
is  :: JRChan a -> JRChan a -> Join ()
```

We introduce chords where the left-hand effectively corresponds to the left-hand side of a CHR. The right-hand side, which will be executed if we find a match of the left-hand, yields a join computation. Thus, a chord like this is easily compiled into a CHR rule. Constraint symbols are treated like (data) constructors and their arguments are channels. The join computation ``x `is` y`` instantiates `x` by the value of `y`. `Get(x)` and `Put(y)` are essentially asynchronous constraints. To define the respective method calls, we provide interfaces to manipulate the CHR store:

```
chan :: a -> JRChan a
newEmptyJRChan :: Join (JRChan a)
readJRChan :: JRChan a -> Join a
addToCHR :: UCons -> Join ConsId
waitUntilDeleted :: ConsId -> Join ()
```

`chan` simply lifts a value into a channel type, `newEmptyJRChan` creates a new channel. `readJRChan` reads the value of a given channel and blocks while it is still empty. `addToCHR` adds a constraint to the CHR store and returns a constraint id. `waitUntilDeleted` checks whether the constraint has been removed from the store and will block otherwise. Thus, we can implement an asynchronous put operation and synchronous get operation:

```
sync :: UCons -> Join ()              async :: UCons -> Join ()
sync c = do { id <- addToCHR c        async c = do { addToCHR c
            ; waitUntilDeleted id }                ; return () }
```

```
get :: Join a                      put :: a -> Join ()
get = do { x <- newEmptyJRChan     put x = async (Put(chan x))
         ; sync (Get(x))
         ; readJRChan x}
```

Note that we effectively turn method calls into constraints via the sync and async interfaces. We summarize the useful features introduced by Haskell-Join-Rules join patterns:

***Exploiting guards:*** The conditional buffer example is an extension of the buffer example which illustrates the usefulness of guards. A condGet method call consists of a filter Join computation to select only items that passes this filter. The conditional buffer Haskell-Join-Rules program is illustrated by the following:

```
Chord CondGet(f,x) & Put(y) | f y = x `is` y

condGet :: (JRChan (a -> Join Bool),JRChan a) -> UCons
Put :: JRChan a -> UCons

condGet :: (a -> Join Bool) -> Join a
condGet f = do { x <- newEmptyJRChan
               ; sync (CondGet(chan f,x))
               ; readJRChan x }
```

***Exploiting propagation:*** Propagation adds the convenience of defining chords with method calls that act as 'catalysts': These method calls are necessary for activating the chord but are not consumed upon successful activation. The authorized buffer is an extension of the buffer example in which get calls (in this case authGet) will only be successful if they are called by authorized processes. The following highlights a simple implementation of the authorized buffer:

```
Chord Auth(id1) \ AuthGet(id2,x) & Put(y)
                       | id1 == id2 = x `is` y

Auth :: JRChan String -> UCons
AuthGet :: (JRChan String,JRChan a) -> UCons
Put :: JRChan a -> UCons

auth :: String -> Join ()
auth id = async (Auth(chan id))

authGet :: String -> Join a
authGet id = do { x <- newEmptyJRChan
                ; sync (AuthGet(chan id,x))
                ; readJRChan x }
```

Note that the Auth(id1) constraint is propagated (not deleted) if the chord is triggered. Propagation does not offer greater expressiveness, but it provides not only convenience, most importantly it provides better a concurrency behavior, since propagated constraints can be shared and need not be deleted and reintroduced unnecessarily.

***Exploiting Customized Synchronization:*** In Haskell-Join-Rules, we can customize our own synchronization patterns. We illustrate this in the following implementation of a n-way synchronization:

```
Chord Accept(x) & Entry(y) = y `is` (chan x) -- {2a}

Accept :: JRChan () -> UCons
Entry  :: JRChan (JRChan ()) -> UCons

accept :: Int -> Join ()
accept n =
    do { x <- newEmptyJRChan   -- {1}
       ; mapM (\_ -> sync (Accept(x))) [1..n] -- {2b}
       ; x `is` () }    -- {3}

entry :: Join ()
entry = do { y <- newEmptyJRChan
           ; sync (Entry(y))
           ; x <- readJRChan y  -- {4}
           ; readJRChan x }     -- {5}
```

n-way synchronization is implemented by using a single channel to synchronize with n entry calls and an accept call. This synchronizing channel is created by the accept call ({1}) and distributed to entry calls via the chord show above ({2a,2b}). Upon receiving this synchronization channel ({4}), each entry call blocks until the n-way synchronization is completed ({5}). accept completes the n-way after finding n entry calls, and writes the synchronization channel with a dummy value. ({3}).

Extending Haskell with join style patterns have been explored earlier in [15], which introduces Join patterns to Haskell as a higher order combinator library. In [12], an extension of JoCaml with pattern matching is introduced. This extension allows ML style patterns in chord heads to be compiled into standard join patterns. Yet Haskell-Join-Rules is the first which introduces rule based constraint programming features (propagation and guards) into join calculus base language.

## 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] G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, The University of Melbourne, 2005.

[3] 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.

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

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

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

[8] 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.

[9] Jocaml. http://jocaml.inria.fr/.

[10] E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory. In *Proc. of ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, pages 19–24, 2007.

[11] E.S.L. Lam and M. Sulzmann. Haskell-join-rules. Technical report.

[12] Q. Ma and L. Maranget. Compiling pattern matching in join-patterns. In *CONCUR*, pages 417–431, 2004.

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

[14] 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.

[15] 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.

[16] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[17] M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007.