Higher Order Combinators for Join Patterns using STM

Satnam Singh
Microsoft
One Microsoft Way
Redmond WA 98052, USA
+1 425 705 8208

satnams@microsoft.com
http://research.microsoft.com/~satnams

ABSTRACT

Join patterns provide a higher level concurrent programming construct than the explicit use of threads and locks and have typically been implemented with special syntax and run-time support. This paper presents a strikingly simple design for a small number of higher order combinators which can be composed together to realize a powerful set of join patterns as a library in an existing language. The higher order combinators enjoy a lock free implementation that uses software transactional memory (STM). This allows joins patterns to be implemented simply as a library and provides a transformational semantics for join patterns.

1. INTRODUCTION

Join patterns provide a way to write concurrent programs that provide a programming model which is higher level than the direct invocation of threads and the explicit use of locks in a specific order. This programming model has at its heart the notion of atomically consuming messages from a group of channels and then executing some code that can use the consumed message values. Join patterns can be used to easily encode related concurrency idioms like actors and active objects [1][14] as shown by Benton et. al. in [4]. Join patterns typically occur as language-level constructs with special syntax along with a sophisticated implementation for a state machine which governs the atomic consumption of messages. The contribution of this paper is to show how join patterns can be modeled using a small but powerful collection of *higher order combinations* which can be implemented in a lock free style using software transactional memory. The combinators are higher order because they take functions (programs) as arguments and return functions (programs as result) which glue together the input programs to form a resulting composite program which allows us to make a domain specific language for join patterns. All of this is achieved as a library in an existing language without requiring any special syntax or run-time code. The complete implementation appears in this paper.

Join patterns emerged from a desire to find higher level concurrency and communication constructs than locks and threads for concurrent and distributed programs [13][6]. For example, the work of Fournet and Gonthier on *join calculus* [10][11] provides a process calculi which is amenable to direct implementation in a distributed setting. Related work on JoCaml [8] and Funnel [20] present similar ideas in a functional setting. An adaptation of join-calculus to an object-oriented setting is found in Comega (previously known as Polyphonic C#) [4] and similar extensions have also been reported for Java [16].

Concurrent programming using join patterns promises to provide useful higher level abstractions compared with asynchronous message passing programs that directly manipulate ports. Comega adds new language features to C# to implement join patterns. Adding concurrency features as language extensions has many advantages including allowing the compiler to analyze and optimize programs and detect problems at compile time. This paper presents a method of introducing a flexible collection of join operations which are implemented solely as a library. We do assume the availability of software transactional memories (STM) which may be implemented as syntactic language extensions or introduced just as a library. In this paper we use the lazy functional programming language Haskell as our host language for join patterns implemented in terms of STM because of the robust implementation which provides composable memory transactions [13] which also exploits the type system to statically forbid side effecting operations inside STM. In Haskell the STM functionality is made available through a regular library. We make extensive use of the *composable* nature of Haskell's STM implementation to help define join pattern elements which also possess good compensability properties. Other reasons for using Haskell include it support for very lightweight threads which allows us to experiment with join pattern programs with vastly more threads than is practical using a language in which threads are implemented directly with operating system threads.

The remainder of this paper briefly presents the salient features of Comega and STM in Haskell and then goes on to show how join patterns can be added as a library using STM. This paper contains listings for several complete Comega and Haskell programs and the reader is encouraged to compile and execute these programs.

2. JOIN PATTERNS IN COMEGA

The polyphonic extensions to C# comprise just two new concepts: (i) *asynchronous methods* which return control to the caller immediately and execute the body of the method concurrently; and (ii) *chords* (also known as 'synchronization patterns' or 'join patterns') which are methods whose execution is predicated by the prior invocation of some null-bodied asynchronous methods.

2.1 ASYNCHRONOUS METHODS

The code below is a complete Comega program that demonstrates an asynchronous method.

```
using System ;

public class MainProgram
{ public class ArraySummer
  { public async sumArray (int[] intArray)
    { int sum = 0 ;
      foreach (int value in intArray)
        sum += value ;
      Console.WriteLine ("Sum  = " + sum) ;
    }
  }

  static void Main()
  { Summer = new ArraySummer () ;
    Summer.sumArray (new int[] {1, 0, 6, 3, 5}) ;
    Summer.sumArray (new int[] {3, 1, 4, 1, 2}) ;
    Console.WriteLine ("Main method done.") ;
  }
}
```

Comega introduces the **async** keyword to identify an asynchronous method. Calls to an asynchronous method return immediately and asynchronous methods do not have a return type (they behave as if their return type is **void**). The `sumArray` asynchronous method captures an array from the caller and its body is run concurrently with respect to the caller's context. The compiler may choose a variety of schemes for implementing the concurrency. For example, a separate thread could be created for the body of the asynchronous method or a work item could be created for a thread pool or, on a multi-processor or multi-core machine, the body may execute in parallel with the calling context. The second call to the `sumArray` does not need to wait until the body of the `sumArray` method finishes executing from the first call to `sumArray`.

In this program the two calls to the `sumArray` method of the `Summer` object behave as if the body of `sumArray` was forked off as a separate thread and control returns immediately to the main program. When this program is compiled and run it will in general write out the results of the two summations and the `Main method done` text in arbitrary orders. The Comega compiler can be downloaded from: http://research.microsoft.com/Comega/

2.2 CHORDS

The code below is a complete Comega program that demonstrates how a chord can be used to make a buffer.

```
using System ;
public class MainProgram
{ public class Buffer
  { public async Put (int value) ;
    public int Get () & Put(int value)
    { return value ; }
  }

  static void Main()
  { buf = new Buffer () ;
    buf.Put (42) ;
    buf.Put (66) ;
    Console.WriteLine (buf.Get() + " " +
                       buf.Get()) ;
  }
}
```

The & operator groups together methods that form a join pattern in Comega. A join pattern that contains only asynchronous methods will concurrently execute its body when all of the constituent methods have been called. A join pattern may have one (but not more) synchronous method which is identified by a return type other than **async**. The body for a synchronous join pattern fires when all the constituent methods (including the synchronous method) are called. The body is executed in the caller's context (thread). The Comega join pattern behaves like a join operation over a collection of ports (e.g. in JoCaml) with the methods taking on a role similar to ports. The calls to the `Put` method are similar in spirit to performing an asynchronous message send (or post) to a port. In this case the port is identified by a method name (i.e. `Put`). Although the asynchronous posts to the `Put` 'port' occur in series in the main body the values will arrive in the `Put` 'port' in an arbitrary order. Consequently the program shown above will have a non-deterministic output writing either "`42 66`" or "`66 42`".

3. STM IN CONCURRENT HASKELL

Software Transactional Memory (STM) is a mechanism for coordinating concurrent threads. We believe that STM offers a much higher level of abstraction than the traditional combination of locks and condition variables, a claim that this paper should substantiate. The material in this section is largely borrowed directly from [2]. We briefly review the STM idea, and especially its realization in concurrent Haskell; the interested reader should consult [9] for much more background and details.

Concurrent Haskell [21] is an extension to Haskell 98, a pure, lazy, functional programming language. It provides explicitly-forked threads, and abstractions for communicating between them. These constructs naturally involve side effects and so, given the lazy evaluation strategy, it is necessary to be able to control exactly when they occur. The big breakthrough came from using a mechanism called *monads* [22] . Here is the key idea: a value of type `IO a` is an "I/O action" that, when performed may do some input/output before yielding a value of type a. For example, the functions `putChar` and `getChar` have types:

```
putChar :: Char -> IO ()
getChar :: IO Char
```

That is, `putChar` takes a Char and delivers an I/O action that, when performed, prints the string on the standard output; while `getChar` is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete program must define an I/O action called main;

executing the program means performing that action. For example:

```
main :: IO ()
main = putChar 'x'
```

I/O actions can be glued together by a *monadic bind* combinator. This is normally used through some syntactic sugar, allowing a C-like syntax. Here, for example, is a complete program that reads a character and then prints it twice:

```
main = do { c <- getChar; putChar c; putChar c }
```

Threads in Haskell communicate by reading and writing *transactional variables*, or **`TVars`**. The operations on TVars are as follows:

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

All these operations all make use of the *STM monad*, which supports a carefully-designed set of transactional operations, including allocating, reading and writing transactional variables. The `readTVar` and `writeTVar` operations both return STM actions, but Haskell allows us to use the same `do {...}` syntax to compose STM actions as we did for I/O actions. These STM actions remain tentative during their execution: in order to expose an STM action to the rest of the system, it can be passed to a new function atomically, with type

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

It takes a memory transaction, of type STM a, and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. For example, one might say:

```
main = do { ...; atomically (getR r 3); ... }
```

Operationally, atomically takes the tentative updates and actually applies them to the `TVars` involved, thereby making these effects visible to other transactions. The atomically function and all of the STM-typed operations are built over the software transactional memory. This deals with maintaining a per-thread transaction log that records the tentative accesses made to `TVars`. When atomically is invoked the STM checks that the logged accesses are *valid* — i.e. no concurrent transaction has committed conflicting updates. If the log is valid then the STM *commits* it atomically to the heap. Otherwise the memory transaction is re-executed with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees: (i) only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot; (ii) no STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a `TVar` without the protection of atomically. Of course, one can always write atomically (`readTVar v`) to read a `TVar` in a trivial transaction, but the call to atomically cannot be omitted. As an example, here is a procedure that atomically increments a `TVar`:

```
incT :: TVar Int -> IO ()
incT v = atomically (do x <- readTVar v
                        writeTVar v (x+1))
```

The implementation guarantees that the body of a call to atomically runs atomically with respect to every other thread; for example, there is no possibility that another thread can read v between the `readTVar` and `writeTVar` of `incT`.

A transaction can block using retry:

```
retry :: STM a
```

The semantics of retry is to abort the current atomic transaction, and re-run it after one of the transactional variables has been updated. For example, here is a procedure that decrements a `TVar`, but blocks if the variable is already zero:

```
decT :: TVar Int -> IO ()
decT v = atomically (do x <- readTVar v
                        when (x == 0)
                           retry
                        writeTVar v (x-1))
```

The when function examines a predicate (here the text to see if `x` is 0) and if it is true it executes a monadic calculation (here `retry`).

Finally, the `orElse` function allows two transactions to be tried in sequence: (`s1 `orElse` s2`) is a transaction that first attempts `s1`; if it calls retry, then `s2` is tried instead; if that retries as well, then the entire call to `orElse` retries. For example, this procedure will decrement `v1` unless `v1` is already zero, in which case it will decrement `v2`. If both are zero, the thread will block:

```
decPair v1 v1 :: TVar Int -> TVar Int -> IO ()
decPair v1 v2
  = atomically (decT v1 `orElse` decT v2)
```

In addition, the STM code needs no modifications at all to be robust to exceptions. The semantics of atomically is that if an exception is raised inside the transaction, then no globally visible state change whatsoever is made.

An example of how a concurrent data structure from the Java JSR-166 [18] collection can be written using STM in Haskell appears in [2].

4. IMPLEMENTING JOINS WITH STM

4.1 TRANSACTED CHANNELS

To help make join patterns out of the STM mechanism in Haskell we shall make use of an existing library which provides transacted channels:

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

A new transacted channel is created with a call to `newTChan`. A value is read from a channel by `readTChan` and a value is written by `writeTChan`. These are tentative operations which occur inside the STM monad and they have to be part of an STM expression which is the subject of a call to `atomically` in order to actually execute and commit.

4.2 SYNCHRONOUS JOIN PATTERNS

A first step towards trying to approach a join pattern like feature of Comega is to try and capture the notion of a synchronous join pattern. We choose to model the methods in Comega as channels in Haskell. We can then model a join pattern by atomically reading from multiple channels. This feature can be trivially implemented using an STM as shown in the definition of `join2` below.

```
module Main
where

import Control.Concurrent
import Control.Concurrent.STM

join2 :: TChan a -> TChan b -> IO (a, b)
join2 chanA chanB
  = atomically (do a <- readTChan chanA
                   b <- readTChan chanB
                   return (a, b)
                )

taskA :: TChan Int -> TChan Int -> IO ()
taskA chan1 chan2
  = do (v1, v2) <- join2 chan1 chan2
       putStrLn ("taskA got: " ++ show (v1, v2))

main
  = do chanA <- atomically newTChan
       chanB <- atomically newTChan
       atomically (writeTChan chanA 42)
       atomically (writeTChan chanB 75)
       taskA chanA chanB
```

Assuming this program is saved in a file called `Join2.hs` it can be compiled using the commands shown below. The Glasgow Haskell compiler can be downloaded from http://www.haskell.org/ghc/

```
$ ghc --make -fglasgow-exts Join2.hs -o join2.exe
Chasing modules from: Join2.hs
Compiling Main             ( Join2.hs, Join2.o )
Linking ...
$ ./join2.exe
taskA got: (42,75)
```

In this program the `join2` function takes two channels and returns a pair of values which have been read from each channel. If either or both of the channels are empty then the STM aborts and retries. Using this definition of `join2` we still do not have a full chord yet and we have to piece together the notion of synchronizing on the arrival of data on several channels with the code to execute when the synchronization fires. This is done in the function `taskA`.

The implementation of the join mechanism in other languages might involve creating a state machine which monitors the arrival of messages on several ports and then decides which handler to run. The complexity of such an implementation is proportional to the number of ports being joined. Exploiting the STM mechanism in Haskell gives a join style synchronization almost for free but the cost of this implementation also depends on the size of the values beings joined because these values are copied into a transaction log.

4.3 ASYNCHRONOUS JOIN PATTERNS

In the code above `taskA` is an example of a synchronous join pattern which runs in the context of the caller. We can also program a recurring asynchronous join with a recursive call:

```
module Main
where

import Control.Concurrent
import Control.Concurrent.STM

join2 :: TChan a -> TChan b -> IO (a, b)
join2 chanA chanB
  = atomically (do a <- readTChan chanA
                   b <- readTChan chanB
                   return (a, b)
                )

asyncJoin2 chan1 chan2 handler
 = forkIO (asyncJoinLoop2 chan1 chan2 handler)

asyncJoinLoop2 chan1 chan2 handler
  = do (v1, v2) <- join2 chan1 chan2
```

```
       forkIO (handler v1 v2)
       asyncJoinLoop2 chan1 chan2 handler



taskA :: Int -> Int -> IO ()
taskA v1 v2
  = putStrLn ("taskA got: " ++ show (v1, v2))

taskB :: Int -> Int -> IO ()
taskB v1 v2
  = putStrLn ("taskB got: " ++ show (v1, v2))

main
  = do chanA <- atomically newTChan
       chanB <- atomically newTChan
       chanC <- atomically newTChan
       atomically (writeTChan chanA 42)
       atomically (writeTChan chanC 75)
       atomically (writeTChan chanB 21)
       atomically (writeTChan chanB 14)
       asyncJoin2 chanA chanB taskA
       asyncJoin2 chanB chanC taskB
       threadDelay 1000
```

`asyncJoin2` here is different from `join2` in two important respects. First, the intention is that the join should automatically re-issue. This is done by recursively calling `asyncJoinLoop2`. Second, this version concurrently executes the body (handler) when the join synchronization occurs (this corresponds to the case in Comega when a chord only contains asynchronous methods). This example spawns off two threads which compete for values on a shared channel.

When either thread captures values from a join pattern it then forks off a handler thread to deal with these values and immediately starts to compete for more values from the ports it is watching. Here is a sample execution of this program:

```
> ./main
taskA got: (42,21)
taskB got: (14,75)
```

4.4 HIGHER ORDER JOIN COMBINATORS

Haskell allows the definition of infix symbols which can help to make the join patterns much easier to read. This section presents some type classes which in conjunction with infix symbols provide a convenient syntax for join patterns.

A synchronous join pattern can be represented using one infix operator to identify channels to be joined and another operator to apply the handler. The infix operators are declared to be left associative and are given binding strengths. The purpose of the `&` combinator is to compose together the elements of a join pattern which identify when the join should fire (in this case it identifies channels). The purpose of the synchronous `>>>` combinator is to take a join pattern and execute a handler when it fires. The result of the handler expression is the result of the join pattern. We use a lambda expression to bind names to the results of the join pattern although we could also have used a named function. A sample join pattern is shown in the definition of the function `example`.

```
module Main
where

import Control.Concurrent
import Control.Concurrent.STM

infixl 5 &
infixl 3 >>>

(&) :: TChan a -> TChan b -> STM (a, b)
(&) chan1 chan2
  = do a <- readTChan chan1
       b <- readTChan chan2
       return (a, b)

(>>>) :: STM a -> (a -> IO b) -> IO b
(>>>) joinPattern handler
   = do results <- atomically joinPattern
        handler results

example chan1 chan2
  = chan1 & chan2 >>> \ (a, b) -> putStrLn (show
(a, b))

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       atomically (writeTChan chan1 14)
       atomically (writeTChan chan2 "wombat")
       example chan1 chan2
```

This program writes “`(14, “wombat”)`”. We can define an operator for performing replicated asynchronous joins in a similar way, as shown below.

```
…
(>!>) :: STM a -> (a -> IO ()) -> IO ()
(>!>) joins cont
   = do forkIO (asyncJoinLoop joins cont)
        return () -- discard thread ID

asyncJoinLoop joinPattern handler
  = do results <- atomically joinPattern
       forkIO (handler results)
       asyncJoinLoop joinPattern handler
```

```
example chan1 chan2
  = chan1 & chan2 >!> \ (a, b) -> putStrLn (show
((a, b)))

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       atomically (writeTChan chan1 14)
       atomically (writeTChan chan2 "wombat")
       atomically (writeTChan chan1 45)
       atomically (writeTChan chan2 "numbat")
       example chan1 chan2
       threadDelay 1000
```

The continuation associated with the joins on `chan1` and `chan2` is run each time the join pattern fires. A sample output is:

```
(14,"wombat")
(45,"numbat")
```

The asynchronous pattern `>!>` runs indefinitely or until the main program ends and brings down all the other threads. One could write a variant of this join pattern which gets notified when it becomes indefinitely blocked (through an exception). This exception could be caught and used to terminate `asyncJoinLoop`. We choose to avoid such asynchronous finalizers.

We can use Haskell's multi-parameter type class mechanism to overload the definition of `&` to allow more than two channels to be joined. Here we define a type class called `Joinable` which allows us to overload the definition of `&`. There instances are given: one for the case where both arguments are transacted channels; one for the case where the second argument is an STM expression (resulting from another join pattern); and one for the case where the left argument is an STM expression. A fourth instance for the case when both arguments are STM expressions has been omitted but is straight forward to define.

…

```
class Joinable t1 t2 where
  (&) :: t1 a -> t2 b -> STM (a, b)

instance Joinable TChan TChan where
  (&) = join2

instance Joinable TChan STM where
  (&) = join2b

instance Joinable STM TChan where
  (&) a b = do (x,y) <- join2b b a
               return (y, x)

join2b :: TChan a -> STM b -> STM (a, b)
join2b chan1 stm
  = do a <- readTChan chan1
       b <- stm
       return (a, b)
```

```
example chan1 chan2 chan3
  = chan1 & chan2 & chan3 >>> \ ((a, b), c) ->
putStrLn (show [a,b,c])

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       chan3 <- atomically newTChan
       atomically (writeTChan chan1 14)
       atomically (writeTChan chan2 75)
       atomically (writeTChan chan3 11)
       example chan1 chan2 chan3
```

One problem with this formulation is that the `&` operator is not associative. The `&` was defined to be a left-associated infix operator which means that different shapes of tuples are returned from the join pattern depending on how the join pattern is bracketed. For example:

```
example1 chan1 chan2 chan3
  = (chan1 & chan2) & chan3 >>> \ ((a, b), c) ->
putStrLn (show [a,b,c])
example2 chan1 chan2 chan3
  = chan1 & (chan2 & chan3) >>> \ (a, (b, c)) ->
putStrLn (show [a,b,c])
```

It would be much more desirable to have nested applications of the `&` operator return a flat structure. We can address this problem in various ways. One approach might be to use type classes again to provide overloaded definitions for `>>>` which fix-up the return type to be a flat tuple. This method is brittle because it requires us to type in instance declarations that map every nested tuple pattern to a flat tuple and we can not type in all of them. Other approaches could exploit Haskell's dynamic types or the template facility for program splicing to define a meta-program that re-writes nested tuples to flat tuples. We do not go into the details of these technicalities here and for clarity of exposition we stick with the nested tuples for the remainder of this paper.

4.5 JOINS ON LISTS OF CHANNELS

Joining on a list of channels is easily accomplished by mapping the channel reading operation on each element of a list. This is demonstrated in the one line definition of `joinList` below.

…

```
joinList :: [TChan a] -> STM [a]
joinList = mapM readTChan

example channels chan2
  = joinList channels & chan2 >>> \ (a, b) ->
putStrLn (show (a, b))

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       chan3 <- atomically newTChan
       atomically (writeTChan chan1 14)
```

```
        atomically (writeTChan chan2 75)
        atomically (writeTChan chan3 11)
        example [chan1, chan2] chan3
```

This program writes out "`([14,75],11)`". One can define a join over arrays of ports in a similar way. For greater generality one could define a type class to introduce a method for mapping a type `T (TChan a)` to the isomorphic type `T a` by performing `readTChan` operations on each channel. One could also look into ways of overloading `&` to operate directly over lists and arrays but applying the `joinList` function as shown above seems to work well and interacts well as an expression in a join pattern.

4.6 CHOICE

The biased choice combinator allows the expression of a choice between two join patterns. The choice is biased because it will always prefer the first join pattern if it can fire. Each alternative is represented by a pair which contains a join pattern and the action to be executed if the join pattern fires.

```
(|+|) :: (STM a, a -> IO c) ->
         (STM b, b -> IO c) ->
         IO c
(|+|) (joina, action1) (joinb, action2)
  = do io <- atomically
             (do a <- joina
                 return (action1 a)
             `orElse`
              do b <- joinb
                 return (action2 b))
       io
```

Here the `orElse` combinator is used to help compose alternatives. This combinator tries to execute the first join pattern (`joina`) and if it succeeds a value is bound is the variable `a` and this is used as input to the IO action called `action1`. If the first join pattern can not fire the first argument of `orElse` performs a retry and then the second alternative is attempted (using the pattern `joinb`). This will either succeed and the value emitted from the `joinb` pattern is then supplied to `action2` or it will fail and the whole STM express will perform a retry.

A fairer choice can be made by using a pseudo-random variable to dynamically construct an `orElse` expression which will either bias `joina` or `joinb`. Another option is to keep alternating the roles of `joina` and `joinb` by using a transacted variable to record which join pattern should be checked first.

4.7 DYNAMIC JOINS

Join patterns in Comega occur as declarations which make them a very static construct. Often one wants to dynamically construct a join pattern depending on some information that is only available at run-time. This argues for join patterns occurring as expressions or statements rather than as declarations. Since in our formulation join patterns are just expressions we get dynamic joins for free. Here is a simple example:

```
example numSensors numSensors chan1 chan2 chan3
  = if numSensors = 2 then
       chan1 & chan2 >!> \ (a, b)
         -> putStrLn (show ((a, b)))
    else
       chan1 & chan2 & chan3 >!> \ ((a, b), c)
         -> putStrLn (show ((a, b, c)))
```

In this example the value of the variable `numSensors` is used to determine which join pattern is executed. A more elaborate example would be a join pattern which used the values read from the patter to dynamically construct a new join pattern in the handler function. Another example would be a join pattern which returns channels which are then used to dynamically construct a join pattern in the handler function.

Statically defined joins enjoy more opportunities for efficient compilation and analysis than dynamically constructed joins.

4.8 CONDITIONAL JOINS

Sometimes it is desirable to predicate a join pattern to fire only when some guard conditions are met or only if the values that would be read by the join pattern satisfy a certain criteria.

We can avoid the complexities of implementing conditional join patterns through tricky concurrent programming and language extension by once again exploiting the STM library interface in Haskell. First we define guards that predicate the consumption of a value from a channel.

```
(?) :: TChan a -> Bool -> STM a
(?) chan predicate
  = if predicate then
      readTChan chan
    else
      retry


example cond chan1 chan2
  = (chan1 ? cond) & chan2 >>> \ (a, b) ->
putStrLn (show (a, b))
```

The guards expressed by `?` can only be boolean expressions and one could always have written a dynamically constructed join pattern instead of a guard. The implementation exploits the `retry` function in the Haskell STM interface to abort this transacted channel read if the predicate is not satisfied.

A more useful kind of conditional join would want to access some shared state about the system to help formulate the condition. Shared state for STM programs can only be accessed via the STM monad so we can introduce another overloaded version of `?` which takes a condition in the STM monad:

```
(?) :: TChan a -> STM Bool -> STM a
(?) chan predicate
  = do cond <- predicate
       if cond then
          readTChan chan
        else
          retry
```

Now the predicate can be supplied with transacted variables which can be used to predicate the consumption of a value from a channel. These conditions can also update shared state. Several guards can try to update the shared state at the same time and the

STM mechanism will ensure that only consistent updates are allowed.

This definition of `?` also allows quite powerful conditional expressions to be written which can depend on the values that would be read from other channels in the join pattern. The condition STM predicate can be supplied with the channels in the join pattern or other transacted variables to help form the predicate. This allows quite dynamic forms of join e.g. sometimes performing a join pattern on channels `chan1` and `chan2` and sometimes performing a join pattern on channels `chan1` and `chan3` depending on the value read from `chan1`.

A special case of the STM predicate version of `?` is a conditional join that tests to see if the value that would be read satisfies some predicate. The code below defines a function `??` which takes such a predicate function as one of its arguments. The example shows a join pattern which will only fire if the value read on `chan1` is greater than 3.

```
(??) :: TChan a -> (a -> Bool) -> STM a
(??) chan predicate
  = do value <- readTChan chan
       if predicate value then
          return value
        else
          retry

example chan1 chan2
  = (chan1 ?? \x -> x > 3) & chan2 >>> \ (a, b)
     -> putStrLn (show (a, b))
```

A conditional join pattern could be implemented in Comega by returning a value to a port if it does not satisfy some predicate. If several threads read from the same port and then return the values they read there is a possibility that the port will end up with values returned in a different order. Furthermore, other threads can make judgments based on the state of the port after the value has been read but before it has been returned. The conditional formulations that we present where atomically remove values from a port when a predicate is satisfied so they do not suffer from such problems.

4.9 Non-Blocking Variants

Non-blocking variants may be made by composing the blocking versions of join patterns using `orElse` with an alternative that returns negative status information. This is demonstrated in the definition of `nonBlockingJoin` below which returns a value wrapped in a Maybe type which has constructors `Just a` for a positive result and `Nothing` for a negative result.

```
nonBlockingJoin :: STM a -> STM (Maybe a)
nonBlockingJoin pattern
  = (do result <- pattern
        return (Just result))
    `orElse`
    (return Nothing)
```

4.10 Exceptions

Understanding how exceptions behave in this join pattern scheme amounts to understanding how exceptions behave in the Haskell STM interface. Exceptions can be thrown and caught as described in [13]. Our encoding of join patterns gives a default backward error recovery scheme for the implementation of the join pattern firing mechanism because if an error occurs in the handler code the transaction is restarted and any consumed values are returned to ports from which they were read. The handler code however does not execute in the STM monad so it may raise exception. This exception will require forward error recovery which may involve returning values to channels because this code is executed after the transacted consumption of values from channels has committed.

5. Related Work

A join pattern library for C# called CCR was recently announced [7] although the underlying model is quite different what is presented here. This model exposes ‘arbiters’ which govern how messages are consumed (or returned) to ports. These arbiters are the fundamental building blocks which are used to encode a variety of communication and synchronization constructs including a variant of join patterns. A significant difference is the lack of a synchronous join because all handler code for join patterns is asynchronously executed on a worker thread. This requires the programmer to explicitly code in a continuation passing style although the iterator mechanism in C# has been exploited by the CCR to effectively get the compiler make the continuation passing transform automatically for the user (in the style of CLU [17]).

One could imagine extending Haskell with JoCaml [11] style join patterns which are special language feature with special syntax. Here is an example of a composite join pattern from the JoCaml manual:

```
# let def apple! () | pie! ()
     = print_string "apple pie" ;
# or  raspberry! () | pie! ()
     = print_string "raspberry pie" ;
```

Three ports are defined: apple, pie and raspberry. The composite join pattern defines a synchronization pattern which contains two alternatives: one which is eligible to fire when there are values available on the ports apple and pie and the other when there are values available on raspberry and pie. When there is only one message on pie the system makes an internal choice e.g.

```
# spawn {apple () | raspberry () | pie ()}
# ;;

-> raspberry pie
```

Alternatively, the system could have equally well responded with `apple pie`. Expressing such patterns using the Haskell STM encoding of join patterns seems very similar yet this approach does not require special syntax or language extensions. However, making join patterns concrete in the language does facilitate compiler analysis and optimization.

6. Conclusions and Future Work

The main contribution of this paper is the realization in Haskell STM of join combinators which model join patterns that already exist in other languages. The embedding of Comega style join patterns into Haskell by exploiting a library that gives a small but powerful interface to an STM mechanism affords a great deal of

expressive power. Furthermore, the embedding is implemented solely as a library without any need to extend the language and modify the compiler. The entire source of the embedding is compact enough to appear in several forms in this paper along with examples.

Several reasons conspire to aid the embedding of join patterns as we have presented them. The very *composable* nature of STM in Haskell means that we can separately define the behavior of elements of join patterns and then compose them together with powerful higher order combinators like `&`, `>>>`, `>!>` and `?`. STM actions can be glued together and executed atomically which allows a good separation of concerns between what to do about a particular channel and what to do about the interaction between all the channels. The behavior of the exception mechanism also composes in a very pleasant way.

The type safety that Haskell provides to ensure that no side-effecting operations can occur inside an STM operation also greatly aids the production of robust programs. The ability to define symbolic infix operators and exploit the type class system for systematic overloading also help to produce join patterns that are concise. We also benefit from representing join patterns as expressions rather than as declarations in Comega.

The STM mechanism proves to be very effective at allowing us to describe conditional join patterns. These would be quite complicated to define in terms of lower level concurrency primitives. We were able to give very short and clear definitions of several types of conditional join patterns.

The ability to perform dynamic joins over composite data structures that contain ports (like lists) and conditional joins makes this library more expressive than what is currently implemented in Comega. Furthermore, in certain situations the optimistic concurrency of a STM based implementation may yield advantages over a more pessimistic lock-based implementation of a finite state machine for join patterns. Another approach for realizing join patterns in a lock free manner could involve implementing the state machine at the heart of the join machinery in languages like Comega using STM rather than explicit locks.

Even if an STM representation of join patterns is not the first choice of an implementer we think that the transformational semantics that they provide for join patterns is a useful model for the programmer. Many of the join patterns we have shown could have been written directly in the STM monad. We think that when synchronization is appropriately expressible as a join pattern then this is preferable for several reasons including the need for intimating the programmer's intent and also giving the compiler an opportunity to perhaps compile such join patterns using a more specialized mechanism than STM.

An interesting avenue of future work suggested by one of the anonymous reviewers is to consider the reverse experiment i.e. use an optimistic implementation of join-calculus primitives in conjunction with monitors and condition variables to try and implement the Haskell STM mechanism. Our intuition is such an approach would be much more complicated to implement. We believe the value of the experiment presented in this paper is not to do with the design of an efficient join pattern library but rather to show that STM may be a viable idiom for capturing various domain specific concurrency abstractions.

Although a Haskell based implementation is not likely to enjoy widespread use or adoption we do believe that the model we have presented provides a useful workbench for exploring how join patterns can be encoded using a library based on higher order combinators with a lock free implementation. Higher order combinators can be encoded to some extent in conventional languages using constructs like delegates in C#. Prototype implementations of STM are available for some mainstream languages e.g. Join Java [16] and SXM [12] for C#. When translating examples from the Haskell STM world into languages like C# which rely on heavyweight operating system threads one may need to introduce extra machinery like threadpools which are not required in Haskell because of its support for a large number of lightweight threads.

REFERENCES

[1] Agha, G. ACTORS: A model of Concurrent computations in Distributed Systems. The MIT Press, Cambridge, Mass. 1990.

[2] Discolo, A., Harris, T., Marlow, M., Peyton Jones, S., Singh, S. Lock Free Data Structures using STM Haskell. Eigth International Symposium on Functional and Logic Programing (FLOPS 2006). April 2006 (to appear).

[3] Appel, A. Compiling with Continuations. Cambridge University Press. 1992

[4] Benton, N., Cardelli, L., Fournet, C. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 26, Issue 5, 2004.

[5] Birrell, A. D. An Introduction to Programming with Threads. Research Report 35, DEC. 1989.

[6] Chaki, S., Rajamani, S. K., Rehof, J. Types as models: Model Checking Message Passing Programs. In Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2002.

[7] Chrysanthakopoulos, G., Singh, S. An Asynchronous Messaging Library for C#. Synchronization and Concurrency in Object-Oriented Languages (SCOOL). October 2005.

[8] Conchon, S., Le Fessant, F. JoCaml: Mobile agents for Objective-Caml. In First International Symposium on Agent Systems and Applications. (ASA'99)/Third International Symposium onMobile Agents (MA'99). IEEE Computer Society, 1999.

[9] Daume III, H. Yet Another Haskell Tutorial. 2004. Available at http://www.isi.edu/~hdaume/htut/ or via http://www.haskell.org/.

[10] Fournet, C., Gonthier, G. The reflexive chemical abstract machine and the join calculus. In Proceedings of the 23rd ACM-SIGACT Symposium on Principles of Programming Languages. ACM, 1996.

[11] Fournet, C., Gonthier, G. The join calculus: a language for distributed mobile programming. In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha, Sept. 2000, G. Barthe, P. Dybjer, , L. Pinto, J. Saraiva, Eds. Lecture Notes in Computer Science, vol. 2395. Springer-Verlag, 2000.

[12] Guerraoui, R., Herlihy, M., Pochon, S., Polymorphic Contention Management. Proceedings of the 19th International Symposium on Distributed Computing (DISC 2005), Cracow, Poland, September 26-29, 2005.

[13] Harris, T., Marlow, S., Jones, S. P., Herlihy, M. Composable Memory Transactions. PPoPP 2005.

[14] Hewitt, C. Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence 8, 3, 323–364. 1977.

[15] Igarashi, A., Kobayashi, K. Resource Usage Analysis. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 27 Issue 2, 2005.

[16] Itzstein, G. S, Kearney, D. Join Java: An alternative concurrency semantics for Java. Tech. Rep. ACRC-01-001, University of South Australia, 2001.

[17] Liskov, B. A History of CLU. ACM SIGPLAN Notices, 28:3, 1993.

[18] Lea, D. The java.util.concurrent Synchronizer Framework. PODC CSJP Workshop. 2004.

[19] Ousterhout, J. Why Threads Are A Bad Idea (for most purposes). Presentation at USENIX Technical Conference. 1996.

[20] Odersky, M. Functional nets. In Proceedings of the European Symposium on Programming. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, 2000.

[21] Peyton Jones, S., Gordon A., Finne S. Concurrent Haskell. In 23rd ACM Symposium on Principles of Programming Languages (POPL'96), pp. 295–308.

[22] Peyton Jones, S., Wadler, P. Imperative functional programming. In 20th ACM Symposium on Principles of Programming Languages (POPL'93), pp. 71–84.