Microsoft Research

Cambridge, UK

Multi-way Rendezvous in Haskell+STM

Nalini Vasudevan

Satnam Singh

Objectives

- Goal: trying to encode various kinds of concurrency idioms in STM Haskell.
- Deterministic parallelism.
- Par/seq?
- Multi-way rendezvous (SHIM).
- Can this be implemented adequately as a library in Haskell with MVars and STM?
- Is it sensible to try and encode concurrency idioms with STM?

Comega Join Patterns

```
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()) ;
  }
}
```

One Shot Synchronous Join

```
(&) :: 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))
```

Biased Choice

```
(|+|) :: (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
```

```
 (chan1 & chan2 & chan3,
    \ ((a,b),c) -> putStrLn (show (a,b,c)))
|+|
 (chan1 & chan2,
    \ (a,b) -> putStrLn (show (a,b)))
```

Conditional Joins

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

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

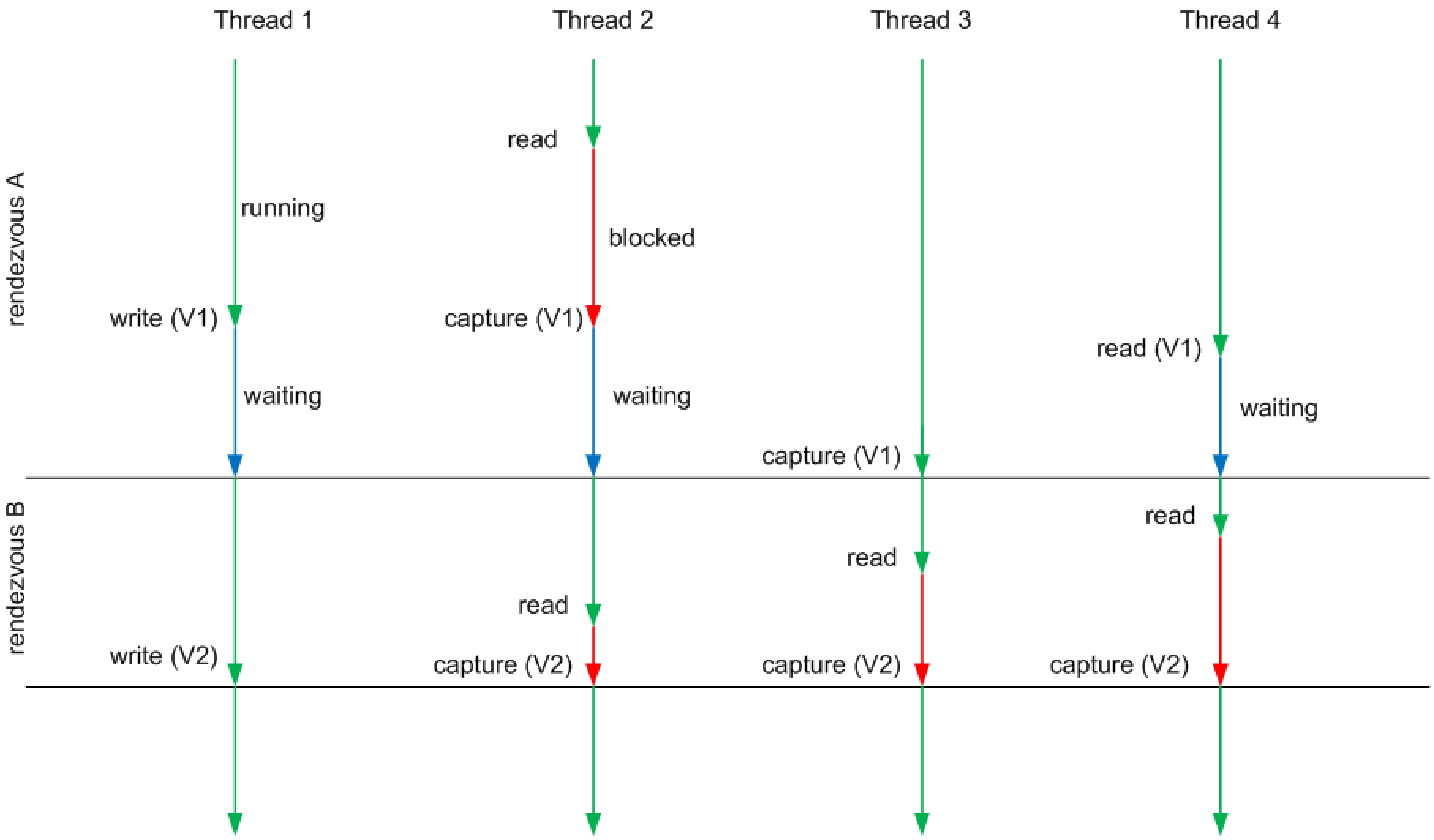
SHIM

```
void f(int a, int &b) {
  while (true) {
     b = a + 1;
     next b; // sends b since b is passed by reference
     next a; // receives a since a is passed by value
  }
}

void g(int b, int &c) {
  while (true) {
    next b; // receives
    c = b;
    next c; // sends
   }
}

void main() {
  int a; a = 0; int b; int c;
  f(a, b); par g(b, c); par g(c, a);
}
```

SHIM

```
void fifo(int i, int &o, int n)
{
  int c; int m; m = n- 1;
  if (m) {
    g(i, c); par fifo(c, o, m);
  } else {
    g(i, o);
  }
}
```

Multi-Way Rendezvous



DVar

```
data DVar a
 = DVar
  { dval :: TVar (Maybe a), -- This is the value of the DVar variable (if it has one)

    dname :: String, -- This is the name of the DVar

    writerRegistered :: TVar Bool, -- Writer registered?

    numReaders :: TVar Int, -- The number of registered readers

    numReadsSoFar :: TVar Int, -- The number of reads that have occurred

    allReadsDone :: TVar Bool -- True if all the reads on a dVar have been performed
}
```

writeDVar

```haskell
writeDVar :: DVar a -> a -> IO ()
writeDVar dVar value
  = do -- First perform the write
       atomically $ writeTVar (dval dVar) (Just value)
                    writeTVar (allReadsDone dVar) False

       -- Now wait for all reads to occcur
       atomically $ do allDone <- readTVar (allReadsDone dVar)
                       if not allDone then
                         retry
                        else
                         return ()
```

waitOnValue

```
waitOnValue :: TVar (Maybe a) -> STM a
waitOnValue maybeT
  = do jv <- readTVar maybeT
       let Just v = jv
       if isNothing jv then
         retry
        else
         return v
```

readDVar

```
readDVar :: DVar a -> IO a
readDVar dVar
  = do v <- atomically $ do v <- waitOnValue (dval dVar)
                            -- Indicate that we have read it
                            nrRead <- readTVar (numReadsSoFar dVar)
                            writeTVar (numReadsSoFar dVar) (nrRead+1)
                            -- See if all the reads have occured
                            nrReaders <- readTVar (numReaders dVar)
                            when (nrRead+1 == nrReaders)
                            -- Release waiting writer
                              $ writeTVar (allReadsDone dVar) True
                            return v
       atomically $ do -- Wait until all reads have occured
                       allDone <- readTVar (allReadsDone dVar)
                       when (not allDone)
                         retry
                       nrRead <- readTVar (numReadsSoFar dVar)
                       writeTVar (numReadsSoFar dVar) (nrRead-1)
                       when (nrRead == 1)
                         $ writeTVar (dval dVar) Nothing
       return v
```

dPar

```
dPar :: IO a  -> IO b  -> IO (a, b)
dPar function1 function2
  = do done1 <- newEmptyMVar
       done2 <- newEmptyMVar
       forkIO (do res <- function1
                  putMVar done1 res
              )
       forkIO (do res <- function2
                  putMVar done2 res
              )
       res1 <- takeMVar done1
       res2 <- takeMVar done2
       return (res1, res2)
```

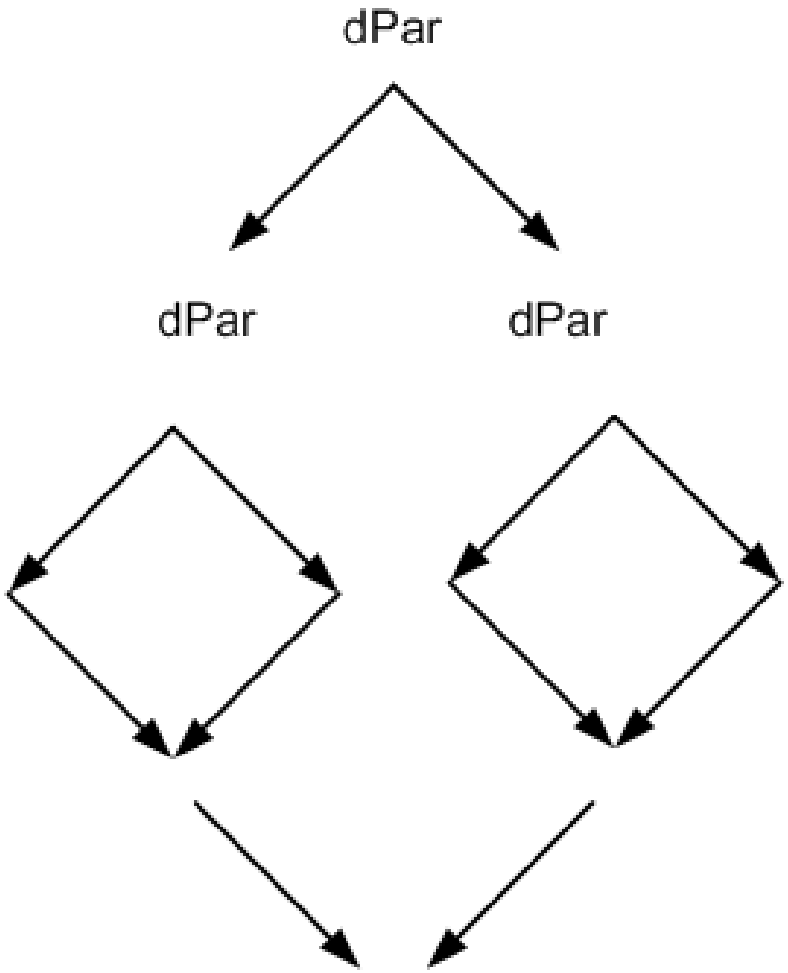
registerWriter

```
registerWriter :: DVar a -> IO ()
registerWriter dVar
  = -- Has someone already registered write interest
    atomically $ do anyWriters <- readTVar (writerRegistered dVar)
                    if anyWriters then
                      error "Too many writers."
                     else
                       -- Record that fact that this dVar now has a writer
                       writeTVar (writerRegistered dVar) True
```

TwoReaders

- (Emacs)

Dynamically created dPars



Question

- In SHIM the compiler can tell by analysis how many reading and writing threads are acting on a DVar.
- If we want to embed a DPar like mechanism in Haskell is it possibly to statically check for programs with too many writers?