

MGS 2012: FUN Lecture 5

Concurrency

Henrik Nilsson

University of Nottingham, UK

MGS 2012: FUN Lecture 5 – p.1/40

This Lecture

- A concurrency monad (adapted from Claessen (1999))
- Traditional, lock-based concurrent programming in Haskell
- Review of issues with lock-based concurrent programming
- Software Transactional Memory (STM monad)
- Why pure functional programming and STM is a great fit

MGS 2012: FUN Lecture 5 – p.2/40

A Concurrency Monad (1)

Demonstration that the notion of concurrent computation can be captured by a monad, and interesting example of a monad.

A `Thread` represents a process: a stream of primitive **atomic** operations:

```
data Thread = Print Char Thread
            | Fork Thread Thread
            | End
```

Note that a `Thread` represents the **entire rest** of a computation.

MGS 2012: FUN Lecture 5 – p.3/40

A Concurrency Monad (2)

Introduce a monad representing “interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.

How can `Threads` be constructed sequentially? The only way is to parameterize thread prefixes on the rest of the `Thread`. This leads directly to **continuations**.

MGS 2012: FUN Lecture 5 – p.4/40

A Concurrency Monad (3)

```
newtype CM a = CM ((a -> Thread) -> Thread)
```

```
fromCM :: CM a -> ((a -> Thread) -> Thread)
fromCM (CM x) = x
```

```
thread :: CM a -> Thread
thread m = fromCM m (const End)
```

```
instance Monad CM where
  return x = CM (\k -> k x)
  m >=> f = CM $ \k ->
    fromCM m (\x -> fromCM (f x) k)
```

MGS 2012: FUN Lecture 5 – p.5/40

A Concurrency Monad (4)

Atomic operations:

```
cPrint :: Char -> CM ()
cPrint c = CM (\k -> Print c (k ()))
```

```
cFork :: CM a -> CM ()
cFork m = CM (\k -> Fork (thread m) (k ()))
```

```
cEnd :: CM a
cEnd = CM (\_ -> End)
```

MGS 2012: FUN Lecture 5 – p.6/40

Running a Concurrent Computation (1)

Running a computation:

```
type Output = [Char]
type ThreadQueue = [Thread]
type State = (Output, ThreadQueue)
```

```
runCM :: CM a -> Output
runCM m = runHlp ("", []) (thread m)
```

where

```
runHlp s t =
```

```
  case dispatch s t of
```

```
    Left (s', t) -> runHlp s' t
```

```
    Right o     -> o
```

MGS 2012: FUN Lecture 5 – p.7/40

Running a Concurrent Computation (2)

Dispatch on the operation of the currently running Thread. Then call the scheduler.

```
dispatch :: State -> Thread
          -> Either (State, Thread) Output
```

```
dispatch (o, rq) (Print c t) =
  schedule (o ++ [c], rq ++ [t])
```

```
dispatch (o, rq) (Fork t1 t2) =
  schedule (o, rq ++ [t1, t2])
```

```
dispatch (o, rq) End =
  schedule (o, rq)
```

MGS 2012: FUN Lecture 5 – p.8/40

Running a Concurrent Computation (3)

Selects next Thread to run, if any.

```
schedule :: State -> Either (State, Thread)
                                Output
schedule (o, []) = Right o
schedule (o, t:ts) = Left ((o, ts), t)
```

MGS 2012: FUN Lecture 5 – p.9/40

Example: Concurrent Processes

```
p1 :: CM ()      p2 :: CM ()      p3 :: CM ()
p1 = do          p2 = do          p3 = do
  cPrint 'a'     cPrint '1'       cFork p1
  cPrint 'b'     cPrint '2'       cPrint 'A'
  ...           ...               cFork p2
  cPrint 'j'     cPrint '0'       cPrint 'B'
```

```
main = print (runCM p3)
```

Result: aAbc1Bd2e3f4g5h6i7j890

Note: As it stands, the output is only made available after *all* threads have terminated.)

MGS 2012: FUN Lecture 5 – p.10/40

Incremental Output

Incremental output:

```
runCM :: CM a -> Output
runCM m = dispatch [] (thread m)

dispatch :: ThreadQueue -> Thread -> Output
dispatch rq (Print c t) = c : schedule (rq ++ [t])
dispatch rq (Fork t1 t2) = schedule (rq ++ [t1, t2])
dispatch rq End          = schedule rq

schedule :: ThreadQueue -> Output
schedule [] = []
schedule (t:ts) = dispatch ts t
```

MGS 2012: FUN Lecture 5 – p.11/40

Example: Concurrent processes 2

```
p1 :: CM ()      p2 :: CM ()      p3 :: CM ()
p1 = do          p2 = do          p3 = do
  cPrint 'a'     cPrint '1'       cFork p1
  cPrint 'b'     undefined        cPrint 'A'
  ...           ...               cFork p2
  cPrint 'j'     cPrint '0'       cPrint 'B'
```

```
main = print (runCM p3)
```

Result: aAbc1Bd*** Exception:
Prelude.undefined

MGS 2012: FUN Lecture 5 – p.12/40

Any Use?

- A number of libraries and embedded languages use similar ideas, e.g.
 - Fudgets
 - Yampa
 - FRP in general
- Studying semantics of concurrent programs.
- Aid for testing, debugging, and reasoning about concurrent programs.

MGS 2012: FUN Lecture 5 – p.13/40

Concurrent Programming in Haskell

Primitives for concurrent programming provided as operations of the IO monad (or “sin bin” :-). They are in the module `Control.Concurrent`. Excerpts:

```
forkIO      :: IO () -> IO ThreadId
killThread  :: ThreadId -> IO ()
threadDelay :: Int -> IO ()
newMVar     :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
putMVar     :: MVar a -> a -> IO ()
takeMVar    :: MVar a -> IO a
```

MGS 2012: FUN Lecture 5 – p.14/40

MVars

- The fundamental synchronisation mechanism is the *MVar* (“em-var”).
- An *MVar* is a “one-item box” that may be *empty* or *full*.
- Reading (`takeMVar`) and writing (`putMVar`) are *atomic* operations:
 - Writing to an empty *MVar* makes it full.
 - Writing to a full *MVar* blocks.
 - Reading from an empty *MVar* blocks.
 - Reading from a full *MVar* makes it empty.

MGS 2012: FUN Lecture 5 – p.15/40

Example: Basic Synchronization (1)

Traditional lock-based synchronization: *MVars* used as semaphores.

```
module Main where

import Control.Concurrent

countFromTo :: Int -> Int -> IO ()
countFromTo m n
  | m > n      = return ()
  | otherwise = do
    putStrLn (show m)
    countFromTo (m+1) n
```

MGS 2012: FUN Lecture 5 – p.16/40

Example: Basic Synchronization (2)

```
main = do
  start <- newEmptyMVar
  done <- newEmptyMVar
  forkIO $ do
    takeMVar start
    countFromTo 1 10
    putMVar done ()
  putStrLn "Go!"
  putMVar start ()
  takeMVar done
  (countFromTo 11 20)
  putStrLn "Done!"
```

MGS 2012: FUN Lecture 5 – p.17/40

Example: Unbounded Buffer (2)

```
readBuffer :: Buffer a -> IO a
readBuffer (Buffer b) = do
  bc <- takeMVar b
  case bc of
    Left (x : xs) -> do
      putMVar b (Left xs)
      return x
    Left [] -> do
      w <- newEmptyMVar
      putMVar b (Right (1,w))
      takeMVar w
    Right (n,w) -> do
      putMVar b (Right (n + 1, w))
      takeMVar w
```

MGS 2012: FUN Lecture 5 – p.19/40

Example: Unbounded Buffer (1)

```
module Main where

import Control.Monad (when)
import Control.Concurrent

newtype Buffer a =
  Buffer (MVar (Either [a] (Int, MVar a)))

newBuffer :: IO (Buffer a)
newBuffer = do
  b <- newMVar (Left [])
  return (Buffer b)
```

MGS 2012: FUN Lecture 5 – p.18/40

Example: Unbounded Buffer (3)

Why isn't Buffer simply defined as

```
newtype Buffer a = Buffer [a]
```

?

Hint: What would happen if e.g. an attempt is made to read from an empty buffer?

MGS 2012: FUN Lecture 5 – p.20/40

Example: Unbounded Buffer (4)

```
writeBuffer :: Buffer a -> a -> IO ()
writeBuffer (Buffer b) x = do
  bc <- takeMVar b
  case bc of
    Left xs ->
      putMVar b (Left (xs ++ [x]))
    Right (n,w) -> do
      putMVar w x
      if n > 1 then
        putMVar b (Right (n - 1, w))
      else
        putMVar b (Left [])
```

MGS 2012: FUN Lecture 5 – p.21/40

Example: Unbounded Buffer (5)

The buffer can now be used as a channel of communication between a set of “writers” and a set of “readers”. E.g.

```
main = do
  b <- newBuffer
  forkIO (writer b)
  forkIO (writer b)
  forkIO (reader b)
  forkIO (reader b)
  ...
```

MGS 2012: FUN Lecture 5 – p.22/40

Example: Unbounded Buffer (6)

```
reader :: Buffer Int -> IO ()
reader n b = rLoop
  where
    rLoop = do
      x <- readBuffer b
      when (x > 0) $ do
        putStrLn (n ++ ": " ++ show x)
        rLoop
```

MGS 2012: FUN Lecture 5 – p.23/40

Compositionality? (1)

Suppose we would like to read two *consecutive* elements from a buffer b?

That is, *sequential composition*.

Would the following work?

```
x1 <- readBuffer b
x2 <- readBuffer b
```

MGS 2012: FUN Lecture 5 – p.24/40

Compositionality? (2)

What about this?

```
mutex <- newMVar ()
...
takeMVar mutex
x1 <- readBuffer b
x2 <- readBuffer b
putMVar mutex ()
```

MGS 2012: FUN Lecture 5 – p.25/40

Compositionality? (3)

Suppose we would like to read from **one of two** buffers.

That is, **composing alternatives**.

Hmmm. How do we even begin?

- No way to attempt reading a buffer without risking blocking.
- We have to change or enrich the buffer implementation. E.g. add a `tryReadBuffer` operation, and then repeatedly poll the two buffers in a tight loop. Not so good!

MGS 2012: FUN Lecture 5 – p.26/40

Locks Are Pessimistic

- In practice, it is often the case that conflicts that would lead to actual harm are rare.
- Lock-based synchronisation thus tends to limit concurrency unnecessarily, potentially harming performance in particular on parallel hardware (such as multi-core processors).

MGS 2012: FUN Lecture 5 – p.27/40

Software Transactional Memory (1)

- Software Transactional Memory (STM) is a new promising approach to facilitate writing correct and performant concurrent code.
- Inspired by the notion of **database transactions**.
- Operations on shared mutable variables grouped into **transactions**.
- Transactions **optimistically** executed concurrently.
- Each transaction succeeds or fails in its **entirety**, depending on if there **actually** was a problem.

MGS 2012: FUN Lecture 5 – p.28/40

Software Transactional Memory (2)

- Transactions thus **atomic** w.r.t. other transactions.
- Failed transactions are automatically **retried** until they succeed.
- **Transaction logs**, which records reading and writing of shared variables, maintained to enable transactions to be validated, partial transactions to be rolled back, and to determine when worth trying a transaction again.
- **No locks!** (At the application level.)

MGS 2012: FUN Lecture 5 – p.29/40

Software Transactional Memory (3)

- Transactional memory poised to go mainstream with the arrival of hardware support in mainstream multi-core processors; e.g., Intel's upcoming (2013) Haswell architecture.

MGS 2012: FUN Lecture 5 – p.30/40

STM and Pure Declarative Languages

- STM perfect match for **purely declarative languages**:
 - reading and writing of shared mutable variables explicit and relatively rare;
 - most computations are pure and need not be logged.
- Disciplined use of effects through monads a **huge** payoff: easy to ensure that **only** effects that can be undone can go inside a transaction.
(Imagine the havoc arbitrary I/O actions could cause if part of transaction: How to undo? What if retried?)

MGS 2012: FUN Lecture 5 – p.31/40

The STM monad

The software transactional memory abstraction provided by a monad STM. **Distinct from IO!**
Defined in `Control.Concurrent.STM`.

Excerpts:

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

MGS 2012: FUN Lecture 5 – p.32/40

Example: Buffer Revisited (1)

Let us rewrite the unbounded buffer using the STM monad:

```
module Main where

import Control.Monad (when)
import Control.Concurrent
import Control.Concurrent.STM

newtype Buffer a = Buffer (TVar [a])

newBuffer :: STM (Buffer a)
newBuffer = do
  b <- newTVar []
  return (Buffer b)
```

MGS 2012: FUN Lecture 5 – p.33/40

Example: Buffer Revisited (3)

The main program and code for readers and writers can remain unchanged, except that STM operations must be carried out **atomically**:

```
main = do
  b <- atomically newBuffer
  forkIO (writer b)
  forkIO (writer b)
  forkIO (reader b)
  forkIO (reader b)
  ...
```

MGS 2012: FUN Lecture 5 – p.35/40

Example: Buffer Revisited (2)

```
readBuffer :: Buffer a -> STM a
readBuffer (Buffer b) = do
  xs <- readTVar b
  case xs of
    []      -> retry
    (x : xs') -> do
      writeTVar b xs'
      return x

writeBuffer :: Buffer a -> a -> STM ()
writeBuffer (Buffer b) x = do
  xs <- readTVar b
  writeTVar b (xs ++ [x])
```

MGS 2012: FUN Lecture 5 – p.34/40

Example: Buffer Revisited (4)

```
reader :: Buffer Int -> IO ()
reader n b = rLoop
  where
    rLoop = do
      x <- atomically (readBuffer b)
      when (x > 0) $ do
        putStrLn (n ++ ": " ++ show x)
        rLoop
```

Why shouldn't atomically be part of the definition of readBuffer?

MGS 2012: FUN Lecture 5 – p.36/40

Composition (1)

STM operations can be **robustly composed**. That's the reason for making `readBuffer` and `writeBuffer` STM operations, and leaving it to client code to decide the scope of atomic blocks.

Example, sequential composition: reading two consecutive elements from a buffer `b`:

```
atomically $ do
  x1 <- readBuffer b
  x2 <- readBuffer b
  ...
```

MGS 2012: FUN Lecture 5 – p.37/40

Composition (2)

Example, composing alternatives: reading from one of two buffers `b1` and `b2`:

```
x <- atomically $
  readBuffer b1
  `orElse` readBuffer b2
```

The buffer operations thus composes nicely. No need to change the implementation of any of the operations!

MGS 2012: FUN Lecture 5 – p.38/40

Reading (1)

- Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3), 1999.
- Wouter Swierstra and Thorsten Altenkirch. Beauty in the Beast: A Functional Semantics for the Awkward Squad. In *Proceedings of Haskell'07*, 2007.
- Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. Composable Memory Transactions. In *Proceedings of PPOPP'05*, 2005
- Simon Peyton Jones. Beautiful Concurrency. Chapter from *Beautiful Code*, ed. Greg Wilson, O'Reilly 2007.

MGS 2012: FUN Lecture 5 – p.39/40

Reading (2)

- Peter Bright. Transactional memory going mainstream with Intel Haswell. February 2012.
<http://arstechnica.com/business/news/2012/02/transactional-memory-going-mainstream-with-intel-haswell.ars>

MGS 2012: FUN Lecture 5 – p.40/40