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

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:

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

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

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.

A Concurrency Monad (3)

```haskell
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))
```

A Concurrency Monad (4)

Atomic operations:

```haskell
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)
```

Running a Concurrent Computation (1)

Running a computation:

```haskell
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
```

Running a Concurrent Computation (2)

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

```haskell
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)
```

Running a Concurrent Computation (3)

Selects next Thread to run, if any.

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

\[
\begin{align*}
\text{p1 :: CM ()} & \quad \text{p2 :: CM ()} & \quad \text{p3 :: CM ()} \\
\text{p1 = do} & \quad \text{p2 = do} & \quad \text{p3 = do} \\
\text{cPrint 'a'} & \quad \text{cPrint '1'} & \quad \text{cFork p1} \\
\text{cPrint 'b'} & \quad \text{cPrint '2'} & \quad \text{cPrint 'A'} \\
\text{...} & \quad \text{...} & \quad \text{cFork p2} \\
\text{cPrint 'j'} & \quad \text{cPrint '0'} & \quad \text{cPrint 'B'}
\end{align*}
\]

main = print (runCM p3)

Result: \text{aAbc1Bd2e3f4g5h6i7j890}

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

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.

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:

\[
\begin{align*}
\text{forkIO : : IO ()} & \quad \text{-> IO ThreadId} \\
\text{killThread : : ThreadId -> IO ()} \\
\text{threadDelay : : Int -> IO ()} \\
\text{newMVar : : a -> IO (MVar a)} \\
\text{newEmptyMVar : : IO (MVar a)} \\
\text{putMVar : : MVar a -> a -> IO ()} \\
\text{takeMVar : : MVar a -> IO a}
\end{align*}
\]

MVars

- The fundamental synchronisation mechanism is the \text{MVar} ("em-var").
- An MVar is a "one-item box" that may be \text{empty} or \text{full}.
- Reading (\text{takeMVar}) and writing (\text{putMVar}) are \text{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.

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)
    putMVar done ()
    countFromTo (m+1) n

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!"

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!"

Example: Basic Synchronization (3)

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!"

Example: Unbounded Buffer (1)

module Main where

import Control.Monad (when)
import Control.Concurrent

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

newBuffer :: IO (Buffer a)
newBuffer = do
  Buffer (newMVar (Left []))
Example: Unbounded Buffer (2)

```haskell
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
```

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?

Example: Unbounded Buffer (4)

```haskell
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 [])
```

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
```

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)
  ...
```

Compositionality? (2)

What about this?
```
mutex <- newMVar ()
...
takeMVar mutex
x1 <- readBuffer b
x2 <- readBuffer b
putMVar mutex ()
```

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!

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).
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.

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.)

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.

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?)

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

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)
```

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 x
      return x
```

Example: Buffer Revisited (3)

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

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?
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`:

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

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

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

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

---

**Reading (1)**


---

**Reading (2)**

- Peter Bright. Transactional memory going mainstream with Intel Haswell. February 2012.
  
  [Link](http://arstechnica.com/business/news/2012/02/transactional-memory-going-mainstream-with-intel-haswell.ars)