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

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)
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)
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
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.

\[
\text{schedule} :: \text{State} \rightarrow \text{Either (State, Thread)}
\]

Output

\[
\begin{align*}
\text{schedule} (o, []) & = \text{Right } o \\
\text{schedule} (o, t:ts) & = \text{Left } ((o, ts), t)
\end{align*}
\]
Example: Concurrent Processes

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

main = print (runCM p3)

Result: aAbc1Bd2e3f4g5h6i7j890
Note: As it stands, the output is only made available after all threads have terminated.)
```
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
```
Example: Concurrent processes 2

```haskell
p1 :: CM ()
p2 :: CM ()
p3 :: CM ()
p1 = do
  cPrint 'a'
  cPrint 'b'
  ...
  cPrint 'j'

p2 = do
  cPrint '1'
  undefined
  ...
  cPrint '0'

p3 = do
  cFork p1
  cPrint 'A'
  cFork p2
  cPrint 'B'

main = print (runCM p3)
```

Result: aAbc1Bd*** Exception: Prelude.undefined
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.
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`
The fundamental synchronisation mechanism is the \textit{MVar} ("em-var").
MVars

- The fundamental synchronisation mechanism is the \textit{MVar} ("em-var").
- An \textit{MVar} is a "one-item box" that may be \textit{empty} or \textit{full}.
The fundamental synchronisation mechanism is the \texttt{MVar} ("em-var").

An \texttt{MVar} is a “one-item box” that may be \textit{empty} or \textit{full}.

Reading (\texttt{takeMVar}) and writing (\texttt{putMVar}) are \textit{atomic} operations:
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.
The fundamental synchronisation mechanism is the \texttt{MVar} ("em-var").

An \texttt{MVar} is a “one-item box” that may be \textit{empty} or \textit{full}.

Reading (\texttt{takeMVar}) and writing (\texttt{putMVar}) are \textit{atomic} operations:
- Writing to an empty \texttt{MVar} makes it full.
- Writing to a full \texttt{MVar} blocks.
**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.
The fundamental synchronisation mechanism is the \textsf{MVar} ("em-var").

An \textsf{MVar} is a "one-item box" that may be \textit{empty} or \textit{full}.

Reading (\texttt{takeMVar}) and writing (\texttt{putMVar}) are \textit{atomic} operations:

- Writing to an empty \textsf{MVar} makes it full.
- Writing to a full \textsf{MVar} blocks.
- Reading from an empty \textsf{MVar} blocks.
- Reading from a full \textsf{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)
        countFromTo (m+1) n
Example: Basic Synchronization (2)

```haskell
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!"
```
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)
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
```
Example: Unbounded Buffer (3)

Why isn’t `Buffer` simply defined as

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

?
Example: Unbounded Buffer (3)

Why isn’t `Buffer` simply defined as

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

```haskell
main = do
  b <- newBuffer
  forkIO (writer b)
  forkIO (writer b)
  forkIO (reader b)
  forkIO (reader b)
  ...
```
Example: Unbounded Buffer (6)

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

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

That is, *sequential composition*.

Would the following work?

```plaintext
x1 <- readBuffer b
x2 <- readBuffer b
```
What about this?

```haskell
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.*
Compositionality? (3)

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

That is, composing alternatives.

Hmmm. How do we even begin?
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.
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.
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.
Software Transactional Memory (STM) is a new promising approach to facilitate writing correct and performant concurrent code.

Inspired by the notion of *database transactions*.
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*.
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.
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 \textit{atomic} w.r.t. other transactions.
Software Transactional Memory (2)

- Transactions thus *atomic* w.r.t. other transactions.
- Failed transactions are automatically *retried* until they succeed.
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.
Software Transactional Memory (2)

- Transactions thus \textit{atomic} w.r.t. other transactions.
- Failed transactions are automatically \textit{retried} until they succeed.
- \textit{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.
- \textit{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.
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.
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:

\[
\begin{align*}
\text{newTVar} & : a \rightarrow \text{STM} \ (\text{TVar} \ a) \\
\text{writeTVar} & : \text{TVar} \ a \rightarrow a \rightarrow \text{STM} \ () \\
\text{readTVar} & : \text{TVar} \ a \rightarrow \text{STM} \ a \\
\text{retry} & : \text{STM} \ a \\
\text{atomically} & : \text{STM} \ a \rightarrow \text{IO} \ a
\end{align*}
\]
Example: Buffer Revisited (1)

Let us rewrite the unbounded buffer using the STM monad:

```haskell
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)

```haskell
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])
```
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*:

```haskell
main = do
    b <- \textit{atomically} newBuffer
    forkIO (writer b)
    forkIO (reader b)
    forkIO (reader b)
...
```

MGS 2012: FUN Lecture 5 -- p.35/40
Example: Buffer Revisited (4)

\[
\text{reader :: Buffer Int -> IO ()}
\]
\[
\text{reader n b = rLoop}
\]
\[
\text{where}
\]
\[
\text{rLoop = do}
\]
\[
\text{x <- atomically (readBuffer b)}
\]
\[
\text{when (x > 0) $ do}
\]
\[
\text{putStrLn (n ++ " : " ++ show x)}
\]
\[
\text{rLoop}
\]

Why shouldn't \text{atomically} be part of the definition of \text{readBuffer}?
**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`:

```haskell
atomically $ do
  x1 <- readBuffer b
  x2 <- readBuffer b
  ...
```
Example, composing alternatives: reading from one of two buffers \( b_1 \) and \( b_2 \):

\[
x \leftarrow \text{atomically } \$
\begin{align*}
\text{readBuffer } b_1 \\
\text{`orElse`} \text{ readBuffer } b_2
\end{align*}
\]

The buffer operations thus composes nicely. No need to change the implementation of any of the operations!
Reading (1)

• Peter Bright. Transactional memory going mainstream with Intel Haswell. February 2012.