COMP4075: Lecture 10

Concurrency

Henrik Nilsson

University of Nottingham, UK

This Lecture

- A concurrency monad (adapted from Claessen (1999))
- Basic concurrent programming in Haskell
- Software Transactional Memory (the STM monad)

A Concurrency Monad (1)

A $Thread$ represents a (branching) process: a stream of primitive ***atomic*** operations:

$$\mathbf{data}\ Thread = Print\ Char\ Thread \mid Fork\ Thread\ Thread \mid End$$

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

Note also that a $Thread$ can spawn other $Thread$s (so we get a tree, if you prefer).

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 $Thread$s 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)

$$\textbf{newtype}\ CM\ a = CM\ ((a \to Thread) \to Thread)$$

$$fromCM :: CM\ a \to ((a \to Thread) \to Thread)$$
$$fromCM\ (CM\ x) = x$$

$$thread :: CM\ a \to Thread$$
$$thread\ m = fromCM\ m\ (const\ End)$$

$$\textbf{instance}\ Monad\ CM\ \textbf{where}$$
$$\quad return\ x = CM\ (\lambda k \to k\ x)$$
$$\quad m \gg\!= f\ = CM\ \$\ \lambda k \to$$
$$\qquad fromCM\ m\ (\lambda x \to fromCM\ (f\ x)\ k)$$

A Concurrency Monad (4)

Atomic operations:

$$cPrint :: Char \to CM\ ()$$
$$cPrint\ c = CM\ (\lambda k \to Print\ c\ (k\ ()))$$

$$cFork :: CM\ a \to CM\ ()$$
$$cFork\ m = CM\ (\lambda k \to Fork\ (thread\ m)\ (k\ ()))$$

$$cEnd :: CM\ a$$
$$cEnd = CM\ (\backslash\_ \to End)$$

Running a Concurrent Computation (1)

$$\textbf{type}\ Output = [Char]$$
$$\textbf{type}\ ThreadQueue = [Thread]$$
$$\textbf{type}\ State = (Output, ThreadQueue)$$

$$runCM :: CM\ a \to Output$$
$$runCM\ m = runHlp\ (\texttt{""}, [\,])\ (thread\ m)$$
$$\quad \textbf{where}$$
$$\qquad runHlp\ s\ t =$$
$$\qquad\quad \textbf{case}\ dispatch\ s\ t\ \textbf{of}$$
$$\qquad\qquad Left\ (s', t) \to runHlp\ s'\ t$$
$$\qquad\qquad Right\ o \to o$$

Running a Concurrent Computation (2)

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

$$dispatch :: State \to Thread$$
$$\quad \to Either\ (State, Thread)\ Output$$
$$dispatch\ (o, rq)\ (Print\ c\ t) =$$
$$\quad schedule\ (o +\!\!+ [c], rq +\!\!+ [t])$$
$$dispatch\ (o, rq)\ (Fork\ t1\ t2) =$$
$$\quad schedule\ (o, rq +\!\!+ [t1, t2])$$
$$dispatch\ (o, rq)\ End =$$
$$\quad schedule\ (o, rq)$$

Running a Concurrent Computation (3)

Selects next Thread to run, if any.

$$\begin{aligned}
&\mathit{schedule} :: \mathit{State} \to \mathit{Either}\ (\mathit{State}, \mathit{Thread})\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Output}\\
&\mathit{schedule}\ (o, [\,])\quad\ = \mathit{Right}\ o\\
&\mathit{schedule}\ (o, t : \mathit{ts}) = \mathit{Left}\ ((o, \mathit{ts}), t)
\end{aligned}$$

This all amounts to a ***topological sorting*** of the nodes in the Thread-tree.

Example: Concurrent Processes

$$\begin{array}{lll}
p1 :: \mathit{CM}\ () & p2 :: \mathit{CM}\ () & p3 :: \mathit{CM}\ ()\\
p1 = \mathbf{do} & p2 = \mathbf{do} & p3 = \mathbf{do}\\
\quad \mathit{cPrint}\ \texttt{'a'} & \quad \mathit{cPrint}\ \texttt{'1'} & \quad \mathit{cFork}\ p1\\
\quad \mathit{cPrint}\ \texttt{'b'} & \quad \mathit{cPrint}\ \texttt{'2'} & \quad \mathit{cPrint}\ \texttt{'A'}\\
\quad \ldots & \quad \ldots & \quad \mathit{cFork}\ p2\\
\quad \mathit{cPrint}\ \texttt{'j'} & \quad \mathit{cPrint}\ \texttt{'0'} & \quad \mathit{cPrint}\ \texttt{'B'}
\end{array}$$

$$\mathit{main} = \mathit{print}\ (\mathit{runCM}\ p3)$$

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

Incremental Output

Incremental output:

$$\begin{aligned}
&\mathit{runCM} :: \mathit{CM}\ a \to \mathit{Output}\\
&\mathit{runCM}\ m = \mathit{dispatch}\ [\,]\ (\mathit{thread}\ m)\\
&\mathit{dispatch}\ :: \mathit{ThreadQueue} \to \mathit{Thread} \to \mathit{Output}\\
&\mathit{dispatch}\ \mathit{rq}\ (\mathit{Print}\ c\ t)\quad = c : \mathit{schedule}\ (\mathit{rq} +\!\!+ [t])\\
&\mathit{dispatch}\ \mathit{rq}\ (\mathit{Fork}\ t1\ t2) = \mathit{schedule}\ (\mathit{rq} +\!\!+ [t1, t2])\\
&\mathit{dispatch}\ \mathit{rq}\ \mathit{End}\qquad\quad = \mathit{schedule}\ \mathit{rq}\\
&\mathit{schedule} :: \mathit{ThreadQueue} \to \mathit{Output}\\
&\mathit{schedule}\ [\,]\qquad = [\,]\\
&\mathit{schedule}\ (t : \mathit{ts}) = \mathit{dispatch}\ \mathit{ts}\ t
\end{aligned}$$

Example: Concurrent processes 2

$$\begin{array}{lll}
p1 :: \mathit{CM}\ () & p2 :: \mathit{CM}\ () & p3 :: \mathit{CM}\ ()\\
p1 = \mathbf{do} & p2 = \mathbf{do} & p3 = \mathbf{do}\\
\quad \mathit{cPrint}\ \texttt{'a'} & \quad \mathit{cPrint}\ \texttt{'1'} & \quad \mathit{cFork}\ p1\\
\quad \mathit{cPrint}\ \texttt{'b'} & \quad \mathit{undefined} & \quad \mathit{cPrint}\ \texttt{'A'}\\
\quad \ldots & \quad \ldots & \quad \mathit{cFork}\ p2\\
\quad \mathit{cPrint}\ \texttt{'j'} & \quad \mathit{cPrint}\ \texttt{'0'} & \quad \mathit{cPrint}\ \texttt{'B'}
\end{array}$$

$$\mathit{main} = \mathit{print}\ (\mathit{runCM}\ p3)$$

Result: $\mathit{aAbc1Bd} *** \mathit{Exception} : \mathit{Prelude.undefined}$

Any Use?

- Illustrates the flexibility offered by monads for introducing new control abstractions, including on top of basic concurrency primitives (cf. $Control.Concurrent.Asynch$).
- A number of libraries and embedded langauges use similar ideas, e.g.
  - Fudgets: A GUI library
  - Yampa: A FRP library
- 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. They are in the module $Control.Concurrent$. Excerpts:

$$
\begin{array}{ll}
forkIO & :: IO\ () \to IO\ ThreadId \\
killThread & :: ThreadId \to IO\ () \\
threadDelay & :: Int \to IO\ () \\
newMVar & :: a \to IO\ (MVar\ a) \\
newEmptyMVar & :: IO\ (MVar\ a) \\
putMVar & :: MVar\ a \to a \to IO\ () \\
takeMVar & :: MVar\ a \to IO\ a
\end{array}
$$

$MVar$s

- 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.

Example: Basic Synchronization (1)

```
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)

```
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 [a] (Int, MVar a)))
newBuffer :: IO (Buffer a)
newBuffer = do
  b ← newMVar (Left [])
  return (Buffer b)
```


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
```


Example: Unbounded Buffer (3)

```
...
Right (n, w) → do
  putMVar b (Right (n + 1, w))
  takeMVar w
```

Example: Unbounded Buffer (4)

$$
\begin{array}{l}
\mathit{writeBuffer} :: \mathit{Buffer}\ a \rightarrow a \rightarrow \mathit{IO}\ ()\\
\mathit{writeBuffer}\ (\mathit{Buffer}\ b)\ x = \mathbf{do}\\
\quad \mathit{bc} \leftarrow \mathit{takeMVar}\ b\\
\quad \mathbf{case}\ \mathit{bc}\ \mathbf{of}\\
\quad\quad \mathit{Left}\ \mathit{xs} \rightarrow\\
\quad\quad\quad \mathit{putMVar}\ b\ (\mathit{Left}\ (\mathit{xs} +\!\!+ [x]))\\
\quad\quad \mathit{Right}\ (n, w) \rightarrow \mathbf{do}\\
\quad\quad\quad \mathit{putMVar}\ w\ x\\
\quad\quad\quad \mathbf{if}\ n > 1\\
\quad\quad\quad \mathbf{then}\ \mathit{putMVar}\ b\ (\mathit{Right}\ (n-1, w))\\
\quad\quad\quad \mathbf{else}\ \ \mathit{putMVar}\ b\ (\mathit{Left}\ [\,])
\end{array}
$$

Example: Unbounded Buffer (4)

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

$$
\begin{array}{l}
\mathit{main} = \mathbf{do}\\
\quad b \leftarrow \mathit{newBuffer}\\
\quad \mathit{forkIO}\ (\mathit{writer}\ b)\\
\quad \mathit{forkIO}\ (\mathit{writer}\ b)\\
\quad \mathit{forkIO}\ (\mathit{reader}\ b)\\
\quad \mathit{forkIO}\ (\mathit{reader}\ b)\\
\quad \ldots
\end{array}
$$

Example: Unbounded Buffer (5)

$$
\begin{array}{l}
\mathit{reader} :: \mathit{Buffer}\ \mathit{Int} \rightarrow \mathit{IO}\ ()\\
\mathit{reader}\ n\ b = \mathit{rLoop}\\
\quad \mathbf{where}\\
\quad\quad \mathit{rLoop} = \mathbf{do}\\
\quad\quad\quad x \leftarrow \mathit{readBuffer}\ b\\
\quad\quad\quad \mathit{when}\ (x > 0)\ \$\ \mathbf{do}\\
\quad\quad\quad\quad \mathit{putStrLn}\ (n +\!\!+ \texttt{": "} +\!\!+ \mathit{show}\ x)\\
\quad\quad\quad\quad \mathit{rLoop}
\end{array}
$$

Compositionality? (1)

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

That is, ***sequential composition***.

Would the following work?

$$
\begin{array}{l}
\mathit{x1} \leftarrow \mathit{readBuffer}\ b\\
\mathit{x2} \leftarrow \mathit{readBuffer}\ b
\end{array}
$$

Compositionality? (2)

What about this?

$$mutex \leftarrow newMVar\ ()$$
$$\ldots$$
$$takeMVar\ mutex$$
$$x1 \leftarrow readBuffer\ b$$
$$x2 \leftarrow readBuffer\ b$$
$$putMVar\ mutex\ ()$$

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!

Software Transactional Memory (1)

- Operations on shared mutable variables grouped into ***transactions***.
- A transaction either succeeds or fails in its ***entirety***. I.e., ***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)

- ***Basic consistency requirement***: The effects of reading and writing within a transaction must be indistinguishable from the transaction having been carried out in isolation.
- ***No locks!*** (At the application level.)

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 of arbitrary I/O actions 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:

$$\begin{aligned}
&newTVar &&:: a \to STM\ (TVar\ a)\\
&writeTVar &&:: TVar\ a \to a \to STM\ ()\\
&readTVar &&:: TVar\ a \to STM\ a\\
&retry &&:: STM\ a\\
&atomically &&:: STM\ a \to IO\ a
\end{aligned}$$

Example: Buffer Revisited (1)

Unbounded buffer using the STM monad:

$$\begin{aligned}
&\mathbf{module}\ Main\ \mathbf{where}\\
&\mathbf{import}\ Control.Monad\ (when)\\
&\mathbf{import}\ Control.Concurrent\\
&\mathbf{import}\ Control.Concurrent.STM\\
&\mathbf{newtype}\ Buffer\ a = Buffer\ (TVar\ [a])\\
&newBuffer :: STM\ (Buffer\ a)\\
&newBuffer = \mathbf{do}\\
&\quad b \leftarrow newTVar\ [\,]\\
&\quad return\ (Buffer\ b)
\end{aligned}$$

Example: Buffer Revisited (2)

$$\begin{aligned}
&readBuffer :: Buffer\ a \to STM\ a\\
&readBuffer\ (Buffer\ b) = \mathbf{do}\\
&\quad xs \leftarrow readTVar\ b\\
&\quad \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \to retry\\
&\qquad (x : xs') \to \mathbf{do}\\
&\qquad\quad writeTVar\ b\ xs'\\
&\qquad\quad return\ x
\end{aligned}$$

Example: Buffer Revisited (3)

$$
\begin{array}{l}
\mathit{writeBuffer} :: \mathit{Buffer}\ a \to a \to \mathit{STM}\ () \\
\mathit{writeBuffer}\ (\mathit{Buffer}\ b)\ x = \mathbf{do} \\
\quad \mathit{xs} \leftarrow \mathit{readTVar}\ b \\
\quad \mathit{writeTVar}\ b\ (\mathit{xs} +\!\!+ [x])
\end{array}
$$

Example: Buffer Revisited (4)

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

$$
\begin{array}{l}
\mathit{main} = \mathbf{do} \\
\quad b \leftarrow \mathit{atomically}\ \mathit{newBuffer} \\
\quad \mathit{forkIO}\ (\mathit{writer}\ b) \\
\quad \mathit{forkIO}\ (\mathit{writer}\ b) \\
\quad \mathit{forkIO}\ (\mathit{reader}\ b) \\
\quad \mathit{forkIO}\ (\mathit{reader}\ b) \\
\quad \ldots
\end{array}
$$

Example: Buffer Revisited (5)

$$
\begin{array}{l}
\mathit{reader} :: \mathit{Buffer}\ \mathit{Int} \to \mathit{IO}\ () \\
\mathit{reader}\ n\ b = \mathit{rLoop} \\
\quad \mathbf{where} \\
\quad\quad \mathit{rLoop} = \mathbf{do} \\
\quad\quad\quad x \leftarrow \mathit{atomically}\ (\mathit{readBuffer}\ b) \\
\quad\quad\quad \mathit{when}\ (x > 0)\ \$\ \mathbf{do} \\
\quad\quad\quad\quad \mathit{putStrLn}\ (n +\!\!+ \texttt{": "} +\!\!+ \mathit{show}\ x) \\
\quad\quad\quad\quad \mathit{rLoop}
\end{array}
$$

Composition (1)

STM operations can be ***robustly composed***. That's the reason for making $\mathit{readBuffer}$ and $\mathit{writeBuffer}$ 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:

$$
\begin{array}{l}
\mathit{atomically}\ \$\ \mathbf{do} \\
\quad \mathit{x1} \leftarrow \mathit{readBuffer}\ b \\
\quad \mathit{x2} \leftarrow \mathit{readBuffer}\ b \\
\quad \ldots
\end{array}
$$

Composition (2)

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

$$
\begin{aligned}
&x \leftarrow \mathit{atomically}\ \$ \\
&\quad \mathit{readBuffer}\ b1 \\
&\quad \text{`}\mathit{orElse}\text{`}\ \mathit{readBuffer}\ b2
\end{aligned}
$$

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

Further STM Functionality (1)

TMVar: STM version of MVars for synchoronisation; built on top of TVars:

$$\mathit{TMVar}\ a \approx \mathit{TVar}\ (\mathit{Maybe}\ a)$$

Some operations:

- $\mathit{newTMVar} :: a \rightarrow \mathit{STM}\ (\mathit{TMVar}\ a)$
- $\mathit{newEmptyTMVar} :: \mathit{STM}\ (\mathit{TMVar}\ a)$
- $\mathit{putTMVar} :: \mathit{TMVar}\ a \rightarrow a \rightarrow \mathit{STM}\ ()$
- $\mathit{takeTMVar} :: \mathit{TMVar}\ a \rightarrow \mathit{STM}\ a$
- $\mathit{readTMVar} :: \mathit{TMVar}\ a \rightarrow \mathit{STM}\ a$
- $\mathit{swapTMVar} :: \mathit{TMVar}\ a \rightarrow a \rightarrow \mathit{STM}\ a$

Further STM Functionality (2)

Some non-blocking operations:

- $\mathit{isEmptyTMVar} :: \mathit{TMVar}\ a \rightarrow \mathit{STM}\ \mathit{Bool}$
- $\mathit{tryPutTMVar} :: \mathit{TMVar}\ a \rightarrow a \rightarrow \mathit{STM}\ \mathit{Bool}$
- $\mathit{tryTakeTMVar} :: \mathit{TMVar}\ a \rightarrow \mathit{STM}\ (\mathit{Maybe}\ a)$
- $\mathit{tryReadTMVar} :: \mathit{TMVar}\ a \rightarrow \mathit{STM}\ (\mathit{Maybe}\ a)$

Further STM Functionality (3)

Other process communication and synchronization facilities:

- $\mathit{TChan}\ a$: Unbounded FIFO channel
- $\mathit{TQueue}\ a$: Variation of TChan with faster (amortised) throughput.
- $\mathit{TBQueue}\ a$: Bounded FIFO channel
- TSem: Transactional counting semaphore

Reading

- 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.