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:

$$\begin{aligned}\mathbf{data}\ \mathit{Thread} &= \mathit{Print}\ \mathit{Char}\ \mathit{Thread}\\ &\mid \mathit{Fork}\ \mathit{Thread}\ \mathit{Thread}\\ &\mid \mathit{End}\end{aligned}$$

A Concurrency Monad (1)

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

$$\begin{aligned}\textbf{data}\ Thread &= Print\ Char\ Thread \\ &\mid\ Fork\ Thread\ Thread \\ &\mid\ End\end{aligned}$$

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.

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)

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

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

Running a Concurrent Computation (2)

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

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

Running a Concurrent Computation (3)

Selects next $Thread$ to run, if any.

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

Running a Concurrent Computation (3)

Selects next $Thread$ to run, if any.

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

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

Example: Concurrent Processes

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

$$main = print\ (runCM\ p3)$$

Result: `aAbc1Bd2e3f4g5h6i7j890`

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

Incremental Output

Incremental output:

$$runCM :: CM\ a \rightarrow Output$$
$$runCM\ m = dispatch\ [\,]\ (thread\ m)$$

$$\begin{aligned} dispatch\ &::\ ThreadQueue \rightarrow Thread \rightarrow Output \\ dispatch\ rq\ (Print\ c\ t) &= c : schedule\ (rq +\!\!+ [t]) \\ dispatch\ rq\ (Fork\ t1\ t2) &= schedule\ (rq +\!\!+ [t1, t2]) \\ dispatch\ rq\ End &= schedule\ rq \end{aligned}$$

$$\begin{aligned} schedule &:: ThreadQueue \rightarrow Output \\ schedule\ [\,] &= [\,] \\ schedule\ (t : ts) &= dispatch\ ts\ t \end{aligned}$$

Example: Concurrent processes 2

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

$$main = print\ (runCM\ p3)$$

Result: $aAbc1Bd * * * Exception : Prelude.undefined$

Any Use?

- Illustrates the flexibility offered by monads for introducing new control abstractions, including on top of basic concurrency primitives (cf. $Control.Concurrent.Asynch$).

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

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.

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:

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

$MVars$

- 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.

Example: Basic Synchronization (1)

$$
\begin{array}{l}
\textbf{module}\ \mathit{Main}\ \textbf{where} \\
\textbf{import}\ \mathit{Control.Concurrent} \\
\mathit{countFromTo} :: \mathit{Int} \to \mathit{Int} \to \mathit{IO}\ () \\
\mathit{countFromTo}\ m\ n \\
\quad \mid m > n \qquad\ \ = \mathit{return}\ () \\
\quad \mid \mathit{otherwise} = \textbf{do} \\
\qquad\quad \mathit{putStrLn}\ (\mathit{show}\ \ m) \\
\qquad\quad \mathit{countFromTo}\ (m+1)\ n
\end{array}
$$

Example: Basic Synchronization (2)

$$
\begin{array}{l}
main = \textbf{do} \\
\quad start \leftarrow newEmptyMVar \\
\quad done \leftarrow newEmptyMVar \\
\quad forkIO \; \$ \; \textbf{do} \\
\quad\quad takeMVar \; start \\
\quad\quad countFromTo \; 1 \; 10 \\
\quad\quad putMVar \; done \; () \\
\quad putStrLn \; \texttt{"Go!"} \\
\quad putMVar \; start \; () \\
\quad takeMVar \; done \\
\quad countFromTo \; 11 \; 20 \\
\quad putStrLn \; \texttt{"Done!"}
\end{array}
$$

Example: Unbounded Buffer (1)

$$\textbf{module}\ \mathit{Main}\ \textbf{where}$$

$$\textbf{import}\ \mathit{Control.Monad}\ (\mathit{when})$$
$$\textbf{import}\ \mathit{Control.Concurrent}$$

$$\textbf{newtype}\ \mathit{Buffer}\ a =$$
$$\quad \mathit{Buffer}\ (\mathit{MVar}\ (\mathit{Either}\ [a]\ (\mathit{Int}, \mathit{MVar}\ a)))$$

$$\mathit{newBuffer} :: \mathit{IO}\ (\mathit{Buffer}\ a)$$
$$\mathit{newBuffer} = \textbf{do}$$
$$\quad b \leftarrow \mathit{newMVar}\ (\mathit{Left}\ [\,])$$
$$\quad \mathit{return}\ (\mathit{Buffer}\ b)$$

Example: Unbounded Buffer (2)

$$
\begin{array}{l}
readBuffer :: Buffer\ a \rightarrow IO\ a \\
readBuffer\ (Buffer\ b) = \mathbf{do} \\
\quad bc \leftarrow takeMVar\ b \\
\quad \mathbf{case}\ bc\ \mathbf{of} \\
\quad\quad Left\ (x : xs) \rightarrow \mathbf{do} \\
\quad\quad\quad putMVar\ b\ (Left\ xs) \\
\quad\quad\quad return\ x \\
\quad\quad Left\ [\,] \rightarrow \mathbf{do} \\
\quad\quad\quad w \leftarrow newEmptyMVar \\
\quad\quad\quad putMVar\ b\ (Right\ (1, w)) \\
\quad\quad\quad takeMVar\ w
\end{array}
$$

Example: Unbounded Buffer (3)

$$\ldots$$

$$\begin{array}{l} Right\ (n, w) \rightarrow \mathbf{do} \\ \quad putMVar\ b\ (Right\ (n+1, w)) \\ \quad takeMVar\ w \end{array}$$

Example: Unbounded Buffer (4)

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

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.:

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

Example: Unbounded Buffer (5)

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

Compositionality? (1)

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

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

Would the following work?

$$x1 \leftarrow readBuffer\ b$$
$$x2 \leftarrow readBuffer\ b$$

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***.

Compositionality? (3)

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

That is, ***composing alternatives***.

Hmmm. How do we even begin?

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.

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:

$$\textbf{module}\ \mathit{Main}\ \textbf{where}$$

$$\textbf{import}\ \mathit{Control.Monad}\ (\mathit{when})$$
$$\textbf{import}\ \mathit{Control.Concurrent}$$
$$\textbf{import}\ \mathit{Control.Concurrent.STM}$$

$$\textbf{newtype}\ \mathit{Buffer}\ a = \mathit{Buffer}\ (\mathit{TVar}\ [a])$$

$$\mathit{newBuffer} :: \mathit{STM}\ (\mathit{Buffer}\ a)$$
$$\mathit{newBuffer} = \textbf{do}$$
$$\quad b \leftarrow \mathit{newTVar}\ [\,]$$
$$\quad \mathit{return}\ (\mathit{Buffer}\ b)$$

Example: Buffer Revisited (2)

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

Example: Buffer Revisited (3)

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

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***:

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

Example: Buffer Revisited (5)

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

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:

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

Composition (2)

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

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

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 $TVar$s:

$$TMVar\ a \approx TVar\ (Maybe\ a)$$

Some operations:

- $newTMVar :: a \to STM\ (TMVar\ a)$
- $newEmptyTMVar :: STM\ (TMVar\ a)$
- $putTMVar :: TMVar\ a \to a \to STM\ ()$
- $takeTMVar :: TMVar\ a \to STM\ a$
- $readTMVar :: TMVar\ a \to STM\ a$
- $swapTMVar :: TMVar\ a \to a \to STM\ a$

Further STM Functionality (2)

Some non-blocking operations:

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

Further STM Functionality (3)

Other process communication and synchronization facilities:

- $TChan\ a$: Unbounded FIFO channel
- $TQueue\ a$: Variation of $TChan$ with faster (amortised) throughput.
- $TBQueue\ a$: 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.