COMP4075/G54RFP: Lecture 9

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:

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

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 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)

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

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

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

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

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)

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

Running a Concurrent Computation (2)

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

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

Running a Concurrent Computation (3)

Selects next $Thread$ to run, if any.

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

Running a Concurrent Computation (3)

Selects next $Thread$ to run, if any.

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

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

Example: Concurrent Processes

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

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

Result: `aAbc1Bd2e3f4g5h6i7j890`

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

Incremental Output

Incremental output:

$$
\begin{aligned}
&runCM :: CM\ a \to Output\\
&runCM\ m = dispatch\ [\,]\ (thread\ m)
\end{aligned}
$$

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

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

Example: Concurrent processes 2

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

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

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

Any Use?

- 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 $\mathit{Control.Concurrent}$. Excerpts:

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

$MVars$

- 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)

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

Example: Unbounded Buffer (1)

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

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

$$\begin{array}{l}\textbf{newtype}\ Buffer\ a = \\ \quad Buffer\ (MVar\ (Either\ [a]\ (Int, MVar\ a)))\end{array}$$

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

Example: Unbounded Buffer (2)

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

Example: Unbounded Buffer (3)

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

Example: Unbounded Buffer (4)

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

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.:

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

Example: Unbounded Buffer (5)

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

Compositionality? (1)

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

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

Would the following work?

$$x1 \leftarrow \mathit{readBuffer}\ b$$
$$x2 \leftarrow \mathit{readBuffer}\ b$$

Compositionality? (2)

What about this?

$$
\begin{array}{l}
mutex \leftarrow newMVar\ () \\
\ldots \\
takeMVar\ mutex \\
x1 \leftarrow readBuffer\ b \\
x2 \leftarrow readBuffer\ b \\
putMVar\ mutex\ ()
\end{array}
$$

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:

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

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)

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

Example: Buffer Revisited (3)

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

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***:

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

Example: Buffer Revisited (5)

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

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:

$$
\begin{aligned}
&atomically \ \$ \ \mathbf{do} \\
&\quad x1 \leftarrow readBuffer \ b \\
&\quad x2 \leftarrow readBuffer \ b \\
&\quad \ldots
\end{aligned}
$$

Composition (2)

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

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

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:

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

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.