LiU-FP2010 Part II: Lecture 7

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 process: a stream of primitive ***atomic*** operations:

```
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)

```
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:

```
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
```


Running a Concurrent Computation (2)

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

```
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.

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


Example: Concurrent Processes

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

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

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

main = print (runCM p3)
```

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

Any Use?

- A number of libraries and embedded langauges 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:

```
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
```


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)

```
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
        Right (n,w)   -> do
            putMVar b (Right (n + 1, w))
            takeMVar w
```


Example: Unbounded Buffer (3)

```
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 [])
```

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.

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


Example: Unbounded Buffer (5)

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


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
```


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!

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)

- ***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 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 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***:

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


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:

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


Composition (2)

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

```
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

- 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.