

Networking and concurrency

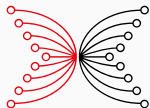
Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP

Dr. Lars Brünjes, IOHK

Dr. Polina Vinogradova, IOHK

2019-01-25



INPUT | OUTPUT

Goals

- Introduce `forkIO`.
- Explain the typical server pattern.
- Software Transactional Memory.

Concurrency vs. Parallelism

Concurrency

Language constructs that support structuring a program as if it had many independent threads of control.

Language constructs that support structuring a program as if it had many independent threads of control.

- Necessarily involves some amount of side effects.
- Pre-dates and is independent of multicore.
- Can be used to implement parallelism, but is not the only way to do so.

Parallelism

Running (parts of) programs in parallel on multiple cores (or nodes), in order to speed up the program.

Parallelism

Running (parts of) programs in parallel on multiple cores (or nodes), in order to speed up the program.

- Primary goal is speed, not structure.
- Does not make sense without multiple cores / parallel hardware.
- Does not conceptually require side effects.
- In fact, deterministic results are preferable.

Today

We will only talk about concurrency, primarily in the context of implementing a network server.

Concurrency

Forking lightweight threads

```
import Control.Concurrent  
data ThreadId -- abstract  
forkIO :: IO () -> IO ThreadId
```

- Given computation is started in a separate (Haskell) thread.
- Haskell threads are lightweight and very cheap, *much* cheaper than OS threads.
- Haskell threads can use multiple cores if they are available, but thousands of threads can happily run concurrently on a single core.
- The `ThreadId` is a handle on the thread that can be used to send signals to the thread.

The `-threaded` flag

GHC gives you the choice between two versions of the run-time system:

- By default, you will get a simpler RTS that is restricted to a single **OS thread** and therefore, a single core.
- If you pass `-threaded` to GHC, you will get a more complex RTS that can use many OS threads and **many cores**.

Both run-time systems support `forkIO` just fine.

Example

```
thread :: Int -> IO ()
thread n = forever $ print n

main :: IO ()
main = do
  mapM_ (forkIO . thread) [1..10]
  thread 0
```

Example

```
thread :: Int -> IO ()
thread n = forever $ print n

main :: IO ()
main = do
  mapM_ (forkIO . thread) [1..10]
  thread 0
```

- Numbers are printed in nondeterministic order depending on scheduling decisions of the run-time system.
- All threads are killed if the main thread is killed.
- In GHCi, the main thread is the thread running GHCi itself!

Delaying a thread

```
second :: Int
second = 1000000  -- delays measured in microseconds

thread :: Int -> IO ()
thread n = forever $ do
    print n
    threadDelay (second `div` 10)

main :: IO ()
main = do
    mapM_ (forkIO . thread) [1..10]
    threadDelay (5 * second)
```

Waiting does not keep the CPU busy.

Networking

A “shouting” server

```
main :: IO ()
main = do
  s <- listenOn (PortNumber 8765)
  forever $ do
    (h, _, _) <- accept s
    forkIO $ handleClient h
handleClient :: Handle -> IO ()
handleClient h = do
  hSetBuffering h LineBuffering
  forever $ do
    line <- hGetLine h
    hPutStrLn h (map toUpper line)
```

Can be tested using `telnet` or `nc`.

The server pattern

```
main :: IO ()
main = do
  ...
  s <- listenOn ...
  forever $ do
    (h, _, _) <- accept s
    forkIO $ handleClient h ...
handleClient :: Handle -> ... -> IO ()
handleClient = ...
```

- Endless `accept` loop.
- Fork a new thread for every client.

Threads and exceptions

```
handleClient :: Handle -> IO ()
handleClient h = do
  hSetBuffering h LineBuffering
  forever $ do
    line <- hGetLine h
    hPutStrLn h (map toUpper line)
```

Once the client closes the connection, `hGetLine` will fail with an exception:

- default exception handler will print the exception,
- but only that thread will be terminated.

A corresponding client

```
main :: IO ()
main = do
  h <- connectTo "127.0.0.1" (PortNumber 8765)
  hSetBuffering h LineBuffering
  forkIO $ copyByLine h stdout -- receiving lines
  copyByLine stdin h -- sending lines

copyByLine :: Handle -> Handle -> IO ()
copyByLine from to = forever $ do
  line <- hGetLine from
  hPutStrLn to line
```

Communication between threads

Communication methods

Shared memory

- All threads can access the same memory and communicate by modifying and inspecting mutable variables.
- Very convenient (if large amounts of data are shared).
- Potentially risky (race conditions, ...).

Communication methods

Shared memory

- All threads can access the same memory and communicate by modifying and inspecting mutable variables.
- Very convenient (if large amounts of data are shared).
- Potentially risky (race conditions, ...).

Message passing

- Communication via messages sent between threads.
- More overhead if large amounts of data are shared.
- Less risky.

Haskell in principle supports both models.

Today, we'll look at shared memory.

Excursion: mutable variables

```
import Data.IORef

data IORef a  -- abstract

newIORef      :: a -> IO (IORef a)
readIORef     :: IORef a -> IO a
writeIORef    :: IORef a -> a -> IO ()
```

Note: `IORef`s can store delayed computations.

Modifying an `IORef`

```
modifyIORef :: IORef a -> (a -> a) -> IO ()  
modifyIORef ref f = do  
  old <- readIORef ref  
  writeIORef ref (f old)
```

(This function is also in the library.)

Example

```
GHCi> r <- newIORef 3
GHCi> readIORef r
3
GHCi> modifyIORef r (+ 1)
GHCi> readIORef r
4
GHCi> writeIORef r 7
GHCi> readIORef r
7
```

Laziness example

```
GHCi> r <- newIORef 3
GHCi> modifyIORef r (`div` 0)
GHCi> readIORef r
*** Exception: divide by zero
```

Modifying an `IOWRef` strictly

```
modifyIOWRef' :: IOWRef a -> (a -> a) -> IO ()  
modifyIOWRef' ref f = do  
  old <- readIOWRef ref  
  writeIOWRef ref $! f old
```

```
($!) :: (a -> b) -> a -> b  
f $! (!x) = f x  
infixr 0 $!
```

Laziness example revisited

```
GHCi> r <- newIORef 3
GHCi> modifyIORef' r (`div` 0)
*** Exception: divide by zero
```

Communicating via `IORef`s is dangerous

```
thread :: IORef Int -> Int -> IO ()
thread var n = forever $ do
  writeIORef var n
  x <- readIORef var
  when (x /= n) $ print (x, n)

main :: IO ()
main = do
  var <- newIORef 0
  mapM_ (forkIO . thread var) [1..10]
  threadDelay (5 * second)
```

At least with `-threaded -N2`, this will produce output.

Atomic modification

```
atomicModifyIORef' ::  
  IORef a -> (a -> (a, b)) -> IO b
```

- Function is applied to current value.
- First component of result will become new value.
- Second component of result is returned.
- Whole operation is performed atomically, without other threads interfering.
- Function forces the value in the `IORef` and the value returned to WHNF.

- The presence of `atomicModifyIORef'` allows modifications of a single `IORef` in a predictable way.
- But operations involving several `IORef`s at once will always be unpredictable.

Classic example: transfer money

Transfer function:

```
transfer ::  
    IORef Integer  
-> IORef Integer  
-> Integer  
-> IO ()  
transfer from to amount = do  
    atomicModifyIORef' from $ \ x ->  
        (x - amount, ())  
    atomicModifyIORef' to $ \ x ->  
        (x + amount, ())
```

Stress-testing the example

```
main :: IO ()
main = do
  accs <- mapM newIORef [1000, 2500]
  total <- getTotal accs
  print total
  forkIO $ monitor total accs
  replicateM_ 100000
    (forkIO (randomTransfer accs))
  threadDelay (5 * second) -- horrible!!
```

Stress-testing the example (contd.)

```
getTotal :: [IORef Integer] -> IO Integer
getTotal accs = sum <$> mapM readIORef accs
```

```
randomTransfer :: [IORef Integer] -> IO ()
randomTransfer xs = do
  let maxIndex = length xs - 1
  from    <- randomRIO (0, maxIndex)
  to      <- randomRIO (0, maxIndex)
  amount <- randomRIO (- 100, 100)
  transfer (xs !! from) (xs !! to) amount
```

Stress-testing the example (contd.)

```
monitor :: Integer -> [IORef Integer] -> IO ()
monitor expected accs = forever $ do
  actual <- getTotal accs
  when (actual /= expected) $
    print $ "INVALID STATE: expected "
      ++ show expected
      ++ ", but have "
      ++ show actual
```

- The traditional solution to the problem we have just seen is *locks*.
- You can use this approach in Haskell, too, by using `MVar`'s from module `Control.Concurrent.MVar`.
- We will (probably) look at `MVar`s in much more detail later. For the time being, however, we will look at *Software Transactional Memory* instead.

Software Transactional Memory

A lock-free approach to concurrency

Haskell's `stm` package offers an appealing approach to concurrency:

- `transactions` are guaranteed to be run atomically;
- the type system guarantees that transactions can be safely restarted;
- there are no locks, hence no danger of deadlocks;
- transactional computations are easy to compose, unlike classic lock-based approaches.

Control.Concurrent.STM interface

```
data STM a  -- abstract
instance Monad STM
data TVar a  -- abstract
    -- transactional variables
newTVar      :: a -> STM (TVar a)
newTVarIO    :: a -> IO (TVar a)
readTVar     :: TVar a -> STM a
writeTVar    :: TVar a -> a -> STM ()
    -- running a transaction
atomically  :: STM a -> IO a
```

Note that `STM` is a restricted form of `IO`.

Classic example: transfer money

Library helper function:

```
modifyTVar :: TVar a -> (a -> a) -> STM ()  
modifyTVar var f = do  
  x <- readTVar var  
  writeTVar var (f x)
```

Transfer function:

```
transfer :: Num a => TVar a -> TVar a -> a -> STM ()  
transfer from to amount = do  
  modifyTVar from (\ x -> x - amount)  
  modifyTVar to   (\ x -> x + amount)
```

Stress-testing the example

```
main :: IO ()
main = do
  accs <- mapM newTVarIO [1000, 2500]
  total <- atomically $ getTotal accs
  print total
  forkIO $ monitor total accs
  replicateM_ 100000
    (forkIO (randomTransfer accs))
  threadDelay (5 * second) -- horrible!!
```

Stress-testing the example (contd.)

```
getTotal :: [TVar Integer] -> STM Integer
getTotal accs = sum <$> mapM readTVar accs
```

```
randomTransfer :: [TVar Integer] -> IO ()
randomTransfer xs = do
  let maxIndex = length xs - 1
  from    <- randomRIO (0, maxIndex)
  to      <- randomRIO (0, maxIndex)
  amount <- randomRIO (- 100, 100)
  atomically $
    transfer (xs !! from) (xs !! to) amount
```

Stress-testing the example (contd.)

```
monitor :: Integer -> [TVar Integer] -> IO ()
monitor expected accs = forever $ do
  actual <- atomically $ getTotal accs
  when (actual /= expected) $
    print $ "INVALID STATE: expected "
      ++ show expected
      ++ ", but have "
      ++ show actual
```

Associating **IO** with transactions

We cannot do IO within a transaction, but we can perform IO after a transaction:

- Compute the data necessary to perform the IO within the transaction and return that from the transaction.

Associating **IO** with transactions

We cannot do IO within a transaction, but we can perform IO after a transaction:

- Compute the data necessary to perform the IO within the transaction and return that from the transaction.
- Because **IO** is first-class data in Haskell, we can even compute the action itself.

Example

```
transfer ::  
  TVar Integer -> TVar Integer  
  -> Integer -> STM (IO ())  
transfer from to amount = do  
  current <- readTVar from  
  if current < amount  
    then return $ putStrLn "not ok"  
    else do  
      modifyTVar from (\ x -> x - amount)  
      modifyTVar to   (\ x -> x + amount)  
      return $ putStrLn $ "ok: " ++ show amount
```

Note that `return` is used on something of type `IO ()` here.

Example (contd.)

```
randomTransfer :: [TVar Integer] -> IO ()
randomTransfer xs = do
  let maxIndex = length xs - 1
  from    <- randomRIO (0, maxIndex)
  to      <- randomRIO (0, maxIndex)
  amount  <- randomRIO (- 100, 100)
  log     <- atomically $
              transfer (xs !! from) (xs !! to) amount
  log    -- execute the logging action after the transaction
```


Another useful monad combinator

From `Control.Monad`:

```
join :: Monad m => m (m a) -> m a
join mma = do
  ma <- mma
  ma
```

Rewriting the example using `join`

```
randomTransfer :: [TVar Integer] -> IO ()
randomTransfer xs = do
  let maxIndex = length xs - 1
  from    <- randomRIO (0, maxIndex)
  to      <- randomRIO (0, maxIndex)
  amount <- randomRIO (- 100, 100)
  join $ atomically $
    transfer (xs !! from) (xs !! to) amount
```

Retrying or combining transactions

```
retry    :: STM a  
orElse  :: STM a -> STM a -> STM a
```

Retrying or combining transactions

```
retry    :: STM a  
orElse   :: STM a -> STM a -> STM a
```

A `retry` does not actually rerun the transaction unless some of the inputs have changed.

An `orElse` tries the second computation only if the first retries.

Example

```
transfer ::  
  TVar Integer -> TVar Integer ->  
  Integer -> STM (IO ())  
transfer from to amount = do  
  current <- readTVar from  
  when (current < amount) retry  
  modifyTVar from (\ x -> x - amount)  
  modifyTVar to   (\ x -> x + amount)  
  return $ putStrLn $ "ok: " ++ show amount
```

A `retry` example

```
main :: IO ()
main = do
  accs@[a1, a2, a3] <-
    mapM newTVarIO [1000, 2500, 5000]
  join $ atomically $ transfer a1 a3 2000
  mapM_ ((>>= print) . readTVarIO) accs
```

This will block indefinitely!

A `retry` example (contd.)

```
transfer' ::  
    TVar Integer  
-> TVar Integer  
-> TVar Integer  
-> Integer  
-> STM (IO ())  
transfer' from from' to amount =  
    transfer from to amount `orElse`  
    transfer from' to amount
```

A `retry` example (contd.)

```
main :: IO ()
main = do
  accs@[a1, a2, a3] <-
    mapM newTVarIO [1000, 2500, 5000]
  join $ atomically $ transfer' a1 a2 a3 2000
  mapM_ ((>>= print) . readTVarIO) accs
```

This will work just fine!

Asynchronous computations

Revisiting the horrible hack

```
main :: IO ()
main = do
  accs <- mapM newTVarIO [1000, 2500]
  total <- atomically $ getTotal accs
  print total
  forkIO $ monitor total accs
  replicateM_ 100000
    (forkIO (randomTransfer accs))
  threadDelay (5 * second) -- horrible!!
```

We cannot easily wait for all threads to finish.

Why not?

Haskell threads are kept as lightweight as possible.

All additional functionality can and should be built on top when needed.

Transactions again?

We could solve this via more `TVar` s:

- Use a `TVar Bool` to indicate whether a thread is finished.
- Initialize all such `TVar` s to `False`.
- Let the thread set it to `True` when done.
- Have a transaction check all `TVar` s to be `True` and otherwise call `retry`.

Introducing async

Fortunately, all this is already done in the `async` package:

```
import Control.Concurrent.Async  
  
async :: IO a -> IO (Async a)  -- an improved forkIO  
wait  :: Async a -> IO a
```

Using `async` and `wait`

```
main :: IO ()
main = do
  accs <- mapM newTVarIO [1000, 2500]
  total <- atomically $ getTotal accs
  print total
  forkIO $ monitor total accs
  asyncs <- replicateM 100000
    (async (randomTransfer accs))
  mapM_ wait asyncs
```

Useful helper functions in the library

```
replicateConcurrently  :: Int -> IO a -> IO [a]  
replicateConcurrently_ :: Int -> IO a -> IO ()
```

Both of these block until all computations are done.

Using `replicateConcurrently_`

```
main :: IO ()
main = do
  accs <- mapM newTVarIO [1000, 2500]
  total <- atomically $ getTotal accs
  print total
  forkIO $ monitor total accs
  replicateConcurrently_ 100000
    (randomTransfer accs)
```


Asynchronous exceptions and exceptions

Recall:

- If an exception in a thread is triggered but not caught, the thread will be stopped, but the parent will not know.

Asynchronous exceptions and exceptions

Recall:

- If an exception in a thread is triggered but not caught, the thread will be stopped, but the parent will not know.

With `Async`:

- If an exception in a thread is triggered but not caught, the thread will be stopped and the exception will propagate to the parent once it calls `wait`.

More flexibility

- If the thread itself wants to handle exceptions, it can just catch them.
- The parent can choose to handle propagated exceptions in any way it likes.

More flexibility

- If the thread itself wants to handle exceptions, it can just catch them.
- The parent can choose to handle propagated exceptions in any way it likes.

The parent can also kill asynchronous computations explicitly:

```
cancel :: Async a -> IO ()
```

This will only return once the computation has actually stopped.