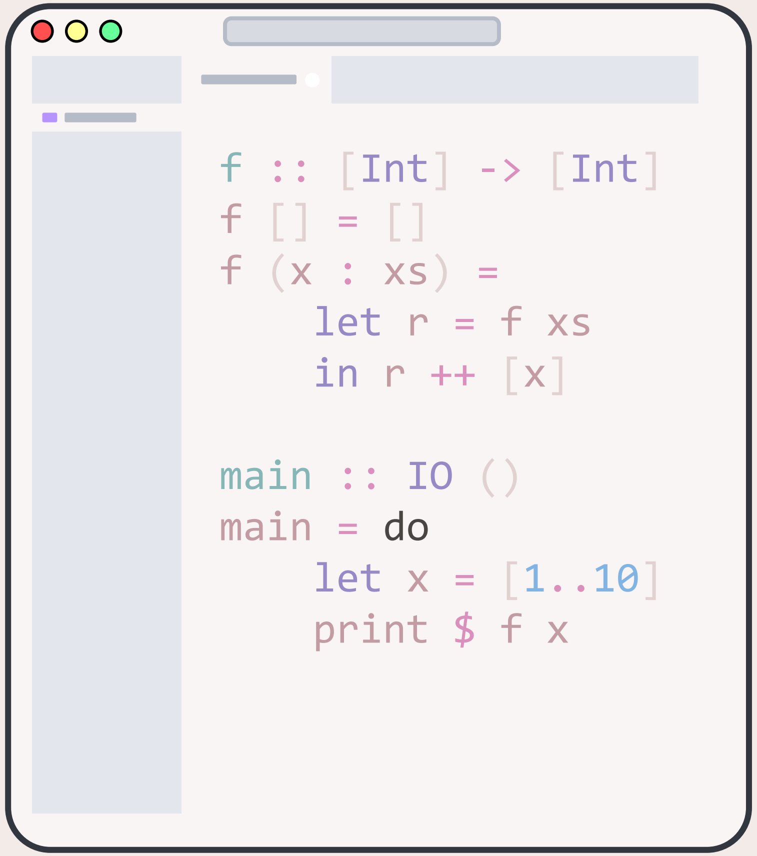
IT5100A

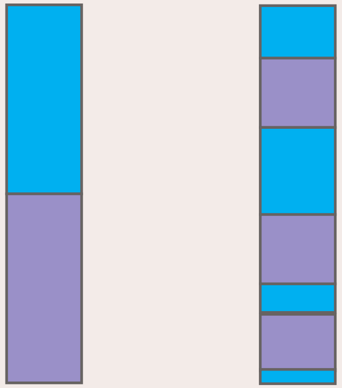
Industry Readiness:
Typed Functional Programming

Concurrent & Parallel Programming

Foo Yong Qi
yongqi@nus.edu.sg



Concurrent Programming



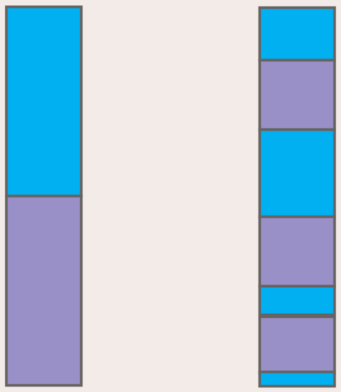
Multiple tasks done "at the same time"

Parallel Programming

One task split into smaller tasks and done "at the same time"

Concurrent Programming

Concurrent Programming



Each task can be provided as a single **thread** of control

- Use `forkIO` to create new threads of control
- Threads are nondeterministic

```
ghci> import Control.Concurrent
ghci> :t forkIO
forkIO :: IO () -> IO ThreadId
ghci> import System.Directory
ghci> forkIO (writeFile "it5100a-notes.md" "Hello World!")
          >> doesFileExist "it5100a-notes.md"
False
```

Concurrent programs “hide latency”

```haskell
import Data.List

writeContents :: String -> String -> IO ()
writeContents file_name contents = do
  let c = intercalate "\n" $
          replicate 1000000
          (intercalate "" $ replicate 100 contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"
```

Concurrent programs "hide latency"

```haskell
main :: IO ()
main = do
  putStrLn "Enter filename:"
  x <- getLine
  if x == "exit"
  then return ()
  else do putStrLn "Enter file contents:"
          y <- getLine
          forkIO $ writeContents x y
          main
```

MVars: synchronizing mutable variables

```
ghci> import Control.Concurrent.MVar
ghci> :t putMVar
putMVar :: MVar a -> a -> IO ()
ghci> :t takeMVar
takeMVar :: MVar a -> IO a
```

Can use `MVar` for obtaining/storing information shared between threads

```haskell
writeContents :: String -> String -> MVar () -> IO ()
writeContents file_name contents lock = do
  takeMVar lock
  putStrLn $ "Write to " ++ file_name ++ " started"
  let !c = intercalate "\n" $
            replicate 1000000 (intercalate "" $
               replicate 500 contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"
  putMVar lock ()
```

- Use the `MVar` as a lock, only exit when **all** writing threads are done
- Haskell runtime wakes up sleeping threads in FIFO order

```haskell
mainLoop :: MVar () -> IO ()
mainLoop lock = do
  putStrLn "Enter filename:"
  x <- getLine
  if x == "exit"
  then do takeMVar lock
          return ()
  else do putStrLn "Enter file contents:"
          y <- getLine
          forkIO $ writeContents x y lock
          mainLoop lock
```

Chans: one-way communication channels

```
ghci> import Control.Concurrent.Chan
ghci> :t writeChan
writeChan :: Chan a -> a -> IO ()
ghci> :t readChan
readChan :: Chan a -> IO a
```

Can use Chan to pass information from main thread to writing thread

```haskell
writeContents :: Chan String -> IO ()
writeContents chan = do
  file_name <- readChan chan
  contents  <- readChan chan
  putStrLn $ "Write to " ++ file_name ++ " started"
  let c = intercalate "\n" $
          replicate 1000000 (
            intercalate "" $ replicate 500 contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"
  writeContents chan
```

Can use Chan to pass information from main thread to writing thread

```haskell
mainLoop :: Chan String -> IO ()
mainLoop chan = do
  putStrLn "Enter filename:"
  x <- getLine
  if x == "exit"
  then return ()
  else do putStrLn "Enter file contents:"
          y <- getLine
          writeChan chan x
          writeChan chan y
          mainLoop chan
```

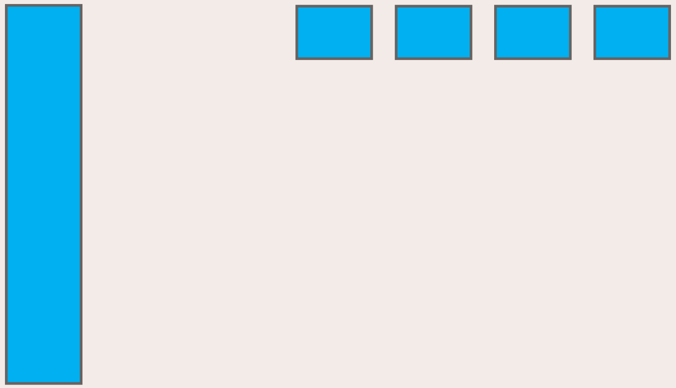
Concurrency doesn't need multiple cores, but can certainly help!
For Haskell, include **threaded runtime**

```
ghc Main.hs -threaded
./Main
4
```

```haskell
import Control.Concurrent
main :: IO ()
main = do setNumCapabilities 4
          print numCapabilities
```

Parallel Programming

Parallel Programming



Can be done with concurrency features, but Haskell is unique due to **non-strict evaluation**

- When expressions are defined, Haskell puts a thunk
- Thunks are evaluated **on-demand** up to **Head Normal Form (HNF)**

```
ghci> x = [1..]
ghci> case x of { [] -> 0; (x:xs) -> x }
1
ghci> let x = [1..]; y = sum x in 1 + 2
3
```

Non-strict evaluation means some times we accidentally run computations on the main thread, not on forked threads

```haskell
expensive :: MVar String -> IO ()
expensive var = do
    putMVar var expensivelyComputedString

main :: IO ()
main = do
    var <- newEmptyMVar
    forkIO $ expensive var
    whatever
    result <- takeMVar var
    print result
```

`seq` introduces an artificial demand on first argument for evaluation on second argument

```
ghci> :t seq
seq :: a -> b -> b
```

```
ghci> let x = [1..]; y = sum x in y `seq` 1 + 2
-- ... does not terminate
ghci> let x = [1..] in x `seq` 1 + 2
3 -- ... only evaluates x to HNF
```

Example: create new evaluation strategy that evaluates deeply

```
ghci> :{
ghci| deepSeq :: [a] -> b -> b
ghci| deepSeq [] x = x
ghci| deepSeq (x:xs) y = x `seq` deepSeq xs y
ghci| :}
ghci> x = [1..]
ghci> x `seq` 1
1
ghci> x `deepSeq` 1
-- does not terminate
```

Parallel programming can be done by using **parallel evaluation strategies**

```
ghci> import GHC.Conc
ghci> :t par
par :: a -> b -> b
ghci> :t pseq
pseq :: a -> b -> b
```

```haskell
x `par` (f x y)
```

Evaluate `x` in parallel with `f x y`; what if evaluation of `f x y` happens first which evaluates `x` first? No parallelism happens

```haskell
x `par` (y `pseq` f x y)
```

Evaluate `x` in parallel, evaluate `y` before `f x y` on the main thread

Parallel fibonacci

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = n1 `par` (n2 `pseq` (n1 + n2))
  where n1 = fib (n - 1)
        n2 = fib (n - 2)
```

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
1134903170

________________________________________________________
Executed in    3.29 secs    fish           external
   usr time   53.14 secs  319.00 micros   53.14 secs
   sys time    0.47 secs  129.00 micros    0.47 secs
```

```
> time cabal run playground -- +RTS -N1
Number of cores: 1
1134903170

________________________________________________________
Executed in   12.93 secs    fish           external
   usr time   12.61 secs  418.00 micros   12.61 secs
   sys time    0.08 secs  171.00 micros    0.08 secs
```

At some point, parallel evaluation overhead outweighs performance gains, put a threshold

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
-- sequential for small n
fib n | n <= 10 = fib (n - 1) + fib (n - 2)
-- parallel for large n
fib n = n1 `par` (n2 `pseq` (n1 + n2))
  where n1 = fib (n - 1)
        n2 = fib (n - 2)
```

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
1134903170

________________________________________________________
Executed in  892.37 millis    fish           external
   usr time   13.01 secs    646.00 micros   13.00 secs
   sys time    0.18 secs      0.00 micros    0.18 secs
```

```
> time cabal run playground -- +RTS -N1
Number of cores: 1
1134903170

________________________________________________________
Executed in    6.81 secs    fish           external
   usr time    6.71 secs  453.00 micros    6.71 secs
   sys time    0.03 secs    0.00 micros    0.03 secs
```

Parallel merge sort

```haskell
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort ls
  | n < 100 = merge left' right'
  | otherwise = par left' $ pseq right' $ merge left' right'
    where n = length ls `div` 2
          merge [] ys = ys
          merge xs [] = xs
          merge (x:xs) (y:ys)
            | x <= y = x : merge xs (y : ys)
            | otherwise = y : merge (x:xs) ys
          (left, right) = splitAt n ls
          left' = mergesort left
          right' = mergesort right
```

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
10000000

________________________________________________________
Executed in    3.58 secs    fish           external
   usr time   16.02 secs  381.00 micros   16.02 secs
   sys time    1.39 secs  159.00 micros    1.39 secs
```

```
> time cabal run playground -- +RTS -N1
Number of cores: 1
10000000

________________________________________________________
Executed in    6.11 secs    fish           external
   usr time    5.62 secs    0.00 micros    5.62 secs
   sys time    0.43 secs  586.00 micros    0.43 secs
```

Force full evaluation in other thread

```haskell
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort ls
  | n < 100 = merge left' right'
  | otherwise = par (deepSeq left') $ pseq right' $ merge left' right'
    where n = length ls `div` 2
          merge [] ys = ys
          merge xs [] = xs
          merge (x:xs) (y:ys)
            | x <= y = x : merge xs (y : ys)
            | otherwise = y : merge (x:xs) ys
          (left, right) = splitAt n ls
          left' = mergesort left
          right' = mergesort right

deepSeq :: [a] -> ()
deepSeq [] = ()
deepSeq (x:xs) = x `seq` deepSeq xs
```

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
10000000

________________________________________________________
Executed in    2.89 secs    fish           external
   usr time   18.04 secs  365.00 micros   18.04 secs
   sys time    0.68 secs  145.00 micros    0.67 secs
```

```
> time cabal run playground -- +RTS -N1
Number of cores: 1
10000000

________________________________________________________
Executed in    6.18 secs    fish           external
   usr time    5.59 secs  362.00 micros    5.59 secs
   sys time    0.46 secs  145.00 micros    0.46 secs
```

Can use `parallel` library to express parallel evaluation strategies easily

```haskell
otherwise = runEval $ do
    l <- rparWith rdeepseq left'
    r <- rseq right'
    return $ merge l r
```

```haskell
fib n = runEval $ do
    n1 <- rpar (fib (n - 1))
    n2 <- rseq (fib (n - 2))
    return $ n1 + n2
```

Separate algorithm from evaluation strategy

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n | n <= 10    = n1 + n2
      | otherwise = (n1 + n2) `using` strat
  where n1 = fib (n - 1)
        n2 = fib (n - 2)
        strat v = do { rpar n1; rseq n2; return v }
```

Software Transactional Memory

What problems can happen?

```haskell
swap a b chan = do
    x <- takeMVar a
    y <- takeMVar b
    putMVar a y
    putMVar b x
    writeChan chan ()
```

```haskell
addToMVar a b chan = do
    y <- takeMVar b
    x <- takeMVar a
    let z = x + y
    putMVar b z
    putMVar a x
    writeChan chan ()
```

Example transaction: taking two mutable variables; `STM` equivalent of `MVar` is `TMVar`

```haskell
takeBothTMVars :: TMVar a -> TMVar b -> STM (a, b)
takeBothTMVars a b = do
  x <- takeTMVar a
  y <- takeTMVar b
  return (x, y)
```

```haskell
takeBothMVars :: MVar a -> MVar b -> IO (a, b)
takeBothMVars a b = do
  x <- takeMVar a
  y <- takeMVar b
  return (x, y)
```

Both **`TMVars`** are taken in one fell swoop, no lock order inversion since atomic transaction is indivisible

```haskell
swap :: TMVar a -> TMVar a -> Chan () -> IO ()
swap a b chan = do
    (x, y) <- atomically $ takeBothTMVars a b
    -- ...
addToMVar :: Num a => TMVar a -> TMVar a
          -> Chan () -> IO ()
addToMVar a b chan = do
    (y, x) <- atomically $ takeBothTMVars b a
    -- ...
```

- Concurrency not very different in Haskell than other languages, yet not so applicable
- Parallelism easy due to non-strict evaluation in Haskell
- Haskell concurrency has same problems with other languages
- STM monad makes composing transactions easy

Thank you

Foo Yong Qi

yongqi@nus.edu.sg

https://yongqi.foo/