Chris Dornan

Why Functional Programming *Really* Matters

Why FP Matters

- Slides & Code
  - github.com/cdornan/birmingham
- Why Functional Programming Matters
  - John Hughes, 1990
  - Focuses on Functional Programming!
  - Building on David Turner & Robin Milner

Warning

Some of the types have been simplified on the following slides.

Functional Programming

- All x in X s.t. *x* is not a factor of p

$$\{\ x \in X \mid x \ \mathbf{mod}\ p \neq 0\ \}$$

- All `x` in `xs` s.t. `x` is not a factor of p

```
[ x | x<-xs, x `mod` p /= 0]
```

Sieve of Eratosthenes

```
primes = sieve [2..]
  where
    sieve (p:xs) =
        p : sieve [x | x<-xs, x `mod` p/=0]
```

Sieve of Eratosthenes

```
primes :: [Integer]
primes = sieve [2..]
  where
    sieve (p:xs) =
        p : sieve [x | x<-xs, x `mod` p/=0]
```

Partitioning Lists

- Partitioning two lists on p

```
[ x | x<-xs,       x<q   ]
[ x | x<-xs, not(x<q) ]
```

Sort

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (q:xs) =
    qsort [ x | x<-xs, x<q ]
         ++ [q]
         ++ qsort [ x | x<-xs, not(x<q) ]
```

Generalized Sort

```
qsort' :: (a->a->Bool) -> [a] -> [a]
qsort' (<) []      = []
qsort' (<) (q:xs) =
    sort' (<) [ x | x<-xs, x<=q ]
           ++ [q]
           ++ sort' (<=) [ x | x<-xs, not(x<q) ]
```

Sort Quiz

```
qsort :: Ord a => [a] -> [a]
qsort []      = []
qsort (q:xs) =
    qsort [ x | x<-xs, x<q ]
         ++ [q]
         ++ qsort [ x | x<-xs, not(x<q) ]
```

Why x<q instead of x<=q?

An Overview of Miranda

- An Overview of Miranda
  - David Turner, SIGPLAN Notices 1986

Doing Things

```
main = putStr “Hello World”
```

Doing Things

```
main :: IO ()
main = putStr “Hello World”
```

Doing Things

- `IO a`
  - Type of an I/O procedure that returns a value of type `a`
  - `IO ()` is **`void`**

Doing Many Things

- IO Composition Operators

```
(>>)    :: IO () -> IO () -> IO ()
return :: a -> IO a


(>>=)   :: IO a -> (a->IO b) -> IO b
```

Doing Many Things

```
main :: IO ()
main = putStr “Hello “
              >> putStr “World!”
```

Doing Many Things

```
main :: IO ()
main =
     do putStr “Hello “
        putStr “World!”
```

A Looping Operator

```
mapM :: (a->IO ()) -> [a] -> IO ()
mapM_ _ []     = return ()
mapM_ p (x:xs) = do p x; mapM_ p xs
```

Looping in Action

```
main :: IO ()
main = mapM_ print primes
```

```
    print :: a -> IO ()
    -- print a on standard output
```

Primes Output

```
2
3
5
7
11
13
17
19
23
…
```

Bounded Primes

```
main :: IO ()
main =
     do [w] <- getArgs
        print_ps (read w)

print_ps :: Int -> IO ()
print_ps sz = mapM_ print (take sz primes)
```

Error Handling

```
fcatch :: IO () -> IO () -> IO ()
```

Error handling operators have simple functional definitions and semantics.

Final Primes Program

```
main :: IO ()
main = fcatch hdl $
   do [w] <- getArgs
      print_primes (read w)
    where
      hdl =
          do putStrLn "usage: primes <num>”
             exitWith (ExitFailure 1)
```

Distributed Scheduler

- Open Source Haskell Package
- Runs Arbitrary (Haskell) Jobs
- Can be distributed over many nodes
- Currently two applications (FFmpeg)
- Web & C/L Interface
- Can be reconfigured on the fly
- Schedules FIFO but:
  - Can be per-node load limited
  - Depends upon absolute/time of week
  - Prioritized by job tag

Concurrency

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

wait    :: Async a -> IO a


waitAny :: [Async a] -> IO (Async a, a)
```

Server Loop

```
server :: WorkPackage p => S p -> IO ()
server s =
   do service_queue s
      gc s
      flushLogS s
      ev <- wait_event s
      ex <- case ev of
              JobE wi tr job ok -> const False `fmap` job_complete s wi tr job ok
              TmrE              -> const False `fmap` timer_trigrd s
              ReqE rt           ->                    req_received s rt
      when (not ex) $
          server s
```

Launching Tasks

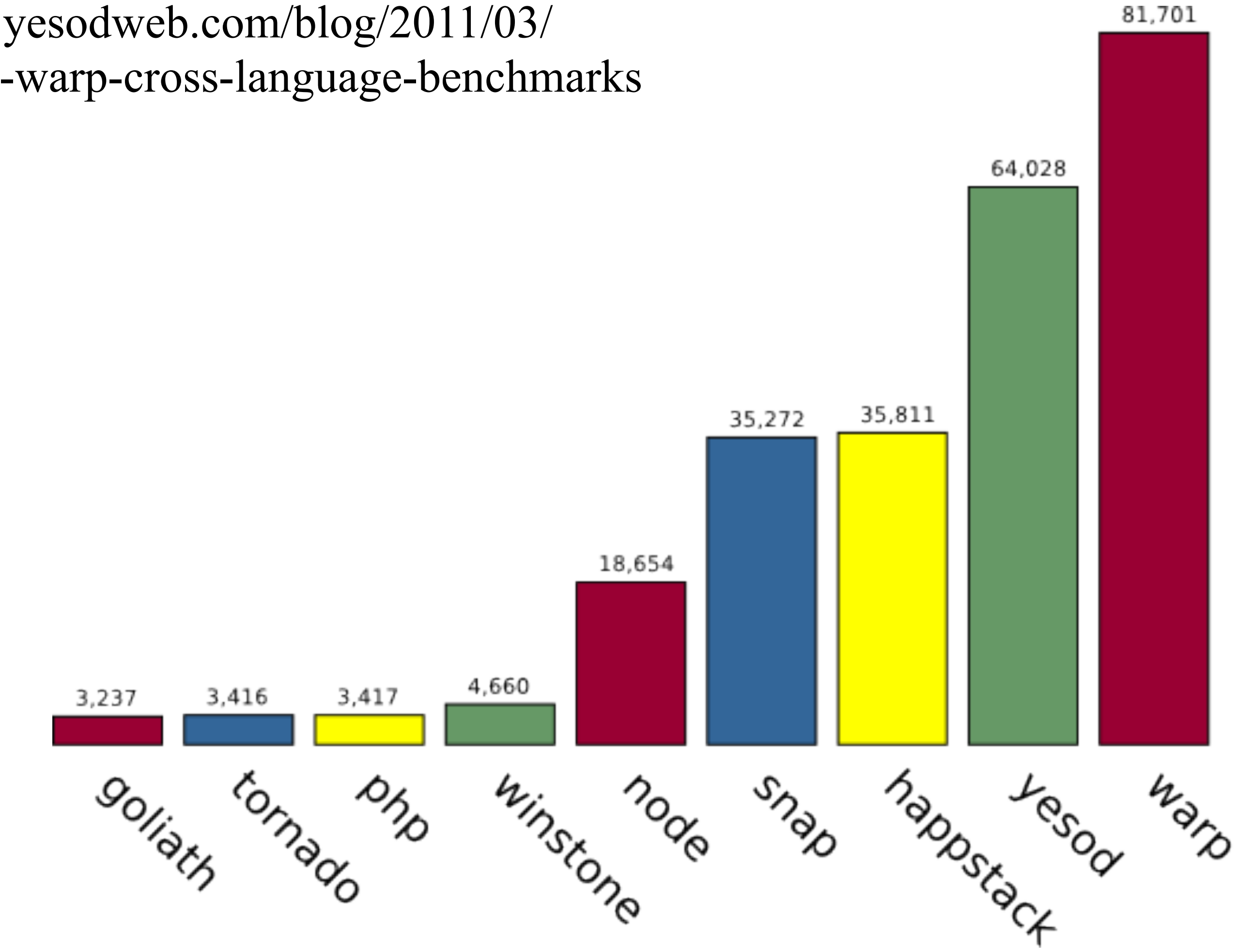
```
launch s job0 wc = flip E.catch hdl $
     do nw  <- now
        phr <- newIORef Nothing
        rmr <- newIORef Nothing
        a <- case mb_hn of
               Nothing -> async $ launch_job   s    phr job
               Just hn ->         launch_r_job s hn rmr job
        writeIORef (stateWC wc) $ Just $ TK job a phr rmr
        return True
      where
        ...
```

Primes Output

```
wait_event :: S p -> IO (Event p)
wait_event s =
     do jas <- jobs_as
        ta  <- timr_a
        ra  <- rqst_a
        ev  <- snd `fmap` waitAny (ra:ta:jas)
return ev
      where
        ...
```

Some Numbers

http://www.yesodweb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks


goliath
tornado
php
winstone
node
snap
happstack
yesod
warp