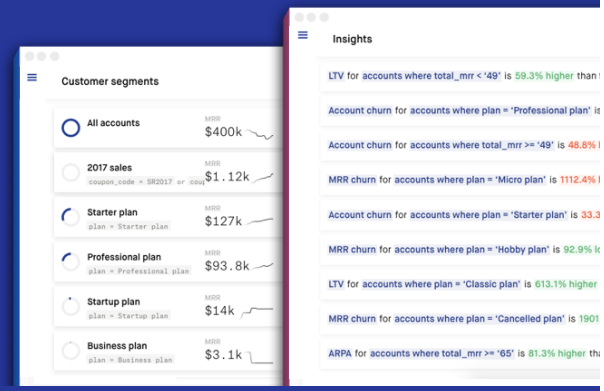
Bright

The smart founder's copilot

Functional Programming in Industry



A note about me

SaaS data analytics
Haskell for analytics,
Rails for presentation

Graduated Engineering Physics at LTH

Bdellium | Fugue | Bright

Retirement Plan Optimization
Data Analytics
Haskell for analytics,
Rails for presentation

Declarative Configuration DSL
Compiler, Haskell only

Why Haskell at Bright?

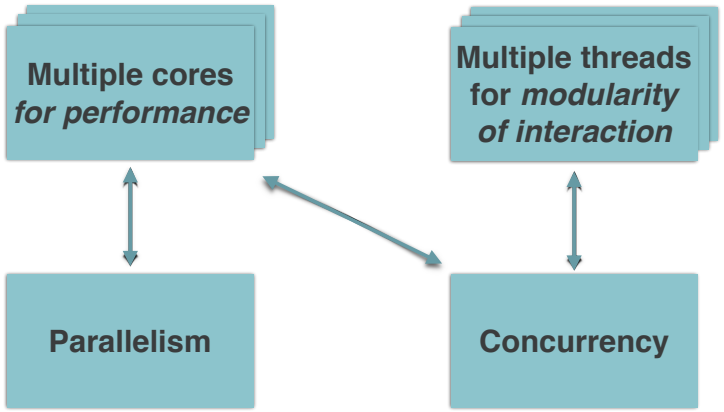
Bright connects to your company's existing data sources to identify strategic risks and opportunities. Data sources like Stripe (covering versions of data back to 2010), Google Adwords, Facebook, free-form data via API and Javascript snippets.

- Lots of different complex data sources means lots of places things can go wrong, algebraic data types and a compiler to the rescue!
- Very good support for easy parallelism and concurrency.
- Good community (IRC, Stack Overflow, Reddit), easy to hire for!

Parallelism and concurrency

- "Parallel and Concurrent Programming in Haskell" by Simon Marlow
- Parallelism and concurrency is more important than ever.
- There are a lot of different ways to do both parallelism and concurrency in Haskell.

Parallelism and concurrency



Pure, deterministic

Has side-effects

Parallelism

- Eval Monad, rpar and rseq
- Strategies
- Par Monad
- Data Parallel Programming with Repa
- GPU Programming with Accelerate

Concurrency

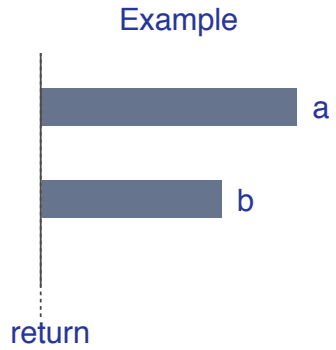
- Threads and MVars
- Software Transactional Memory
- Distributed Concurrency with Cloud Haskell (like Erlang)

Parallelism - rpar and rseq as building blocks

```
do
  a' <- rpar a
  b' <- rpar b
  return (a',b')
```

```
do
  a' <- rpar a
  b' <- rseq b
  return (a',b')
do
  a' <- rpar a
  b' <- rseq b
  rseq a'
  return (a',b')

do
  a' <- rpar a
  b' <- rpar b
  rseq a'
  rseq b'
  return (a',b')
```

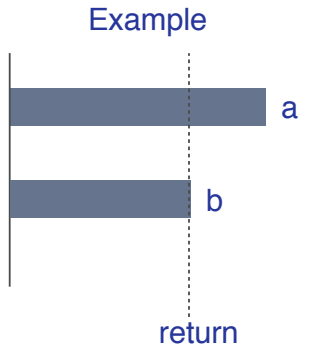


Parallelism - rpar and rseq as building blocks

```
do
  a' <- rpar a
  b' <- rpar b
  return (a',b')
```

```
do
  a' <- rpar a
  b' <- rseq b
  return (a',b')
do
  a' <- rpar a
  b' <- rseq b
  rseq a'
  return (a',b')

do
  a' <- rpar a
  b' <- rpar b
  rseq a'
  rseq b'
  return (a',b')
```



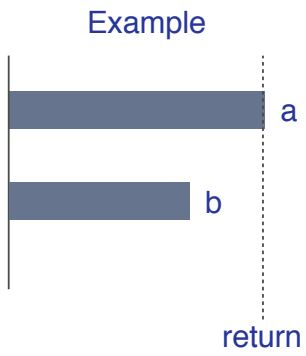
Parallelism - rpar and rseq as building blocks

```haskell
do
  a' <- rpar a
  b' <- rpar b
  return (a',b')

do
  a' <- rpar a
  b' <- rseq b
  return (a',b')

do
  a' <- rpar a
  b' <- rseq b
  rseq a'
  return (a',b')

do
  a' <- rpar a
  b' <- rpar b
  rseq a'
  rseq b'
  return (a',b')
```



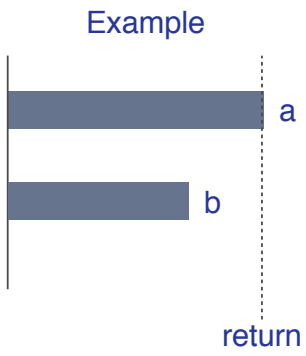
Parallelism - rpar and rseq as building blocks

```haskell
do
  a' <- rpar a
  b' <- rpar b
  return (a',b')

do
  a' <- rpar a
  b' <- rseq b
  return (a',b')

do
  a' <- rpar a
  b' <- rseq b
  rseq a'
  return (a',b')

do
  a' <- rpar a
  b' <- rpar b
  rseq a'
  rseq b'
  return (a',b')
```



Parallelism - Control.Parallel.Strategies

- Tuple Strategies
  seqPair, parPair, …
- General Traversals
  seqTraverse, parTraverse
- List strategies
  seqList, parList, parMap, parListChunk, …
- Relatively easy to build own strategies

Parallelism - Actual Real-world Example

Sequential version

```haskell
-- | Create forecasts for every participant
assignForecasts :: Inputs -> [Participant]
assignForecasts ins = map create (ins^.participants)
```

Parallel version

```haskell
-- | Create forecasts for every participant
assignForecasts :: Inputs -> [Participant]
assignForecasts ins = map create (ins^.participants) `using` parList rdeepseq
```

Concurrency - The building blocks

```
forkIO       :: IO () -> IO ThreadId
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

MVars are single-value communication channels. A box than can be full or empty.

Example from Simon Marlow's book

```
do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Shovel"
    putMVar m1 r
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Spade"
    putMVar m2 r
  r1 <- takeMVar m1
  r2 <- takeMVar m2
  return (r1,r2)
```

Concurrency - Actual Real-world Example

```
fetchRange :: Text -> Text -> Vector Day -> Bright IO ()
fetchRange stripeAcctId stripeToken range = do
  chan <- liftIO $ atomically $ newTBMChan 20
  st <- ask
  let f 0 x = stripeFetcher chan stripeToken Nothing x Nothing
      f i x = stripeFetcher chan stripeToken (range V.!? (i - 1)) x Nothing
  _ <- V.imapM_ f range
  runResourceT $ sourceTBMChan chan $$ processingSink stripeAcctId (length range) 0
  return ()
```

```
stripeFetcher :: TBMChan (ByteString, Bool) -> Text -> Maybe Day -> Day -> Maybe ByteString -> Bright IO ()
stripeFetcher chan stripeToken beginning end startingAfter = do
  request <- …
  manager <- liftIO $ newManager tlsManagerSettings
  response <- httpLbs request manager
  let body = LBS.toStrict $ responseBody response
  when (isNothing startingAfter) $
    $(logTM) DebugS $ "Fetching " <> showLS (fromMaybe 0 $ body ^? key "total_count" .
_Integer) <> " events"

  if stripeHasMore body
  then do
    yield (body, False) $$ sinkTBMChan chan False
    stripeFetcher chan stripeToken beginning end (getStartingAfterId body)
  else do
    yield (body, True) $$ sinkTBMChan chan False
    return ()
```

Notes on Library Support

Very good library coverage for almost all things you want to do, available on Hackage. Most of the time, documentation is somewhat lacking.

Commonly used libraries

- Lens – Makes most things much more convenient
- Pipes / Conduit – Solves "The Lazy IO problem"
- Wreq – For dealing with web requests to other services
- Aeson – JSON serialization
- wai / Scotty / Servant – Building web services

Lenses

- A concept that's implemented in multiple libraries, the most popular being the `lens` package by Edward Kmett.
- Typically used to make dealing with records a bit more pleasant
- Provides a lot of convenience in working with traversals of generic data structures

Lenses

```haskell
data Person = Person { name :: String
                     , addr :: Address
                     , salary :: Int }

data Address = Address { road :: String
                       , city :: String
                       , postcode :: String }
```

```haskell
getName :: Person -> String
getName p = name p

get :: (s -> a) -> s -> a
get property structure = property structure

getCity :: Person -> String
getCity p = city (addr p)

getdeep :: (s1 -> s2) -> (s2 -> a) -> s1 -> a
getdeep prop1 prop2 structure = prop2 (prop1 structure)

>>> getdeep addr city == get (addr.city)
```

Lenses

```haskell
data Person = Person { name :: String
                     , addr :: Address
                     , salary :: Int }

data Address = Address { road :: String
                       , city :: String
                       , postcode :: String }
```

```haskell
setName :: String -> Person -> Person
setName n p = p { name = n }

setPostcode :: String -> Person -> Person
setPostcode pc p = p { addr = addr p { postcode = pc }} -- UGLY!

set :: (s -> a) -> a -> s -> s
set prop n p = p { prop = n }

LensExample.hs:15:20:
   `prop' is not a (visible) constructor field name
```

Lenses

- Example of nested data structures making heavy use of the accessor methods.

```haskell
p^.assets
p^.forecast.retiresAt
p^.forecast.current.gapAnalysis
p^.forecast.optimized.projectedFunding.income.percent
```

- Making traversals trivial

```haskell
-- sum of all unconstSalary records
sumOf (accs.traverse.unconstSalary) paccs

-- maximum forecastSalary in the first 36 months
maximumOf (accs.taking 36 traverse.forecastSalary) paccs

-- a list of the RRR for each participant under the current plan
design
ps^..folded.forecast.current.projectedFunding.income.percent
```

Pipes / Conduit

- Libraries for stream programming in Haskell.
- Dealing with long-running, complex or Lazy IO in Haskell can be very difficult.
- Provides a clean and simple API to provide effectful, streaming, and composable programming.

Pipes / Conduit

- Lazy IO is especially hard to get right

```
withFile "hello.txt" ReadMode hGetContents >>= print
>>> ""
withFile "hello.txt" ReadMode (hGetContents >=> print)
>>> "Hello world!"
```

- IO is difficult to decompose and re-use. Most attempts result in some sort of pipeline, whether explicit or not.

Example of decomposing “echo” program

```
main = do
  eof <- isEOF
  unless eof $ do
    str <- getLine
    putStrLn (transform str)
    main

transform :: String -> String
transform = reverse
```

```
main = readData transform printer

readData :: (String -> String) -> (String -> IO ()) -> IO ()
readData t p = do
  eof <- isEOF
  unless eof $ do
    str <- getLine
    p $ t str
    readData t p

printer :: String -> IO ()
printer s = putStrLn s

transform :: String -> String
transform = reverse
```

Example of decomposing “echo” program

```
main = runEffect mainPipeline

mainPipeline :: Effect IO ()
mainPipeline = readData
           >-> transform
           >-> printer

readData :: Producer String IO ()
readData = forever $ do
  eof <- lift isEOF
  unless eof $ do
    str <- lift getLine
    yield str

transform :: Monad m => Pipe String String m ()
transform = forever $ do
  str <- await
  yield $ reverse str

printer :: Consumer String IO ()
printer = forever $ do
  str <- await
  lift $ putStrLn str
```

Talking to websites

- Several libraries depending on what “level” of interaction you want (high/low)
- Wreq is designed to be easy and simple to work with

```
>>> r <- get "http://httpbin.org/get"
>>> r ^. responseStatus . statusCode
200

>>> r <- post "http://httpbin.org/post" ["num" := 31337, "str" := "foo"]
>>> r ^? responseBody . key "form" . key "num"
Just (String "31337")
```

Dealing with JSON

Aeson and lens-aeson makes it easy to both produce and consume JSON

```
data Person = Person {
    name :: String
  , age  :: Int
  }

instance FromJSON Person where
    parseJSON = withObject "Person" $ \v -> Person
        <$> v .: "name"
        <*> v .: "age"

>>> decode "{\"name\":\"Joe\",\"age\":12}" :: Maybe Person
Just (Person {name = "Joe", age = 12})

instance ToJSON Person where
    toJSON (Person name age) =
        object ["name" .= name, "age" .= age]

>>> encode (Person {name = "Joe", age = 12})
"{\"name\":\"Joe\",\"age\":12}"
```

Dealing with JSON

Using some language extensions to make life easier

```
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

import GHC.Generics
import Data.Aeson

data Person = Person {
  name :: String,
  age  :: Int }
  deriving (Generic, ToJSON, FromJSON)
```

Dealing with JSON

Working with arbitrary JSON in an easy way with lens-aeson

```
>>> "[1, \"x\"]" ^? nth 0 . _Number
Just 1.0

>>> "{\"a\": \"xyz\", \"b\": true}" ^? key "a" . _String
Just "xyz"

>>> "{\"a\": \"xyz\", \"b\": true}" ^? key "b" . _String
Nothing

>>> "{\"a\": \"xyz\", \"b\": true}" ^? key "b"
Just (Bool True)
```

Creating web services

Many excellent libraries at various levels

- Yesod
- Scotty / Spock
- Servant
- wai

Creating web services

Spock example

```haskell
main :: IO ()
main =
  runSpock 8080 $ spockT id $ do
    get "/" $
      html "<a href='/calculator/313/+/3'>Calculate 313 + 3</a>"
    get ("hello" <//> ":name") $ do
      name <- param' "name"
      text $ "Hello " <> name <> "!"
    get ("calculator" <//> ":a" <//> "+" <//> ":b") $ do
      a <- param' "a"
      b <- param' "b"
      text $ pack $ show (a + b :: Int)
```

Creating web services

Wai example

```haskell
app :: Application
app _ respond = do
    putStrLn "I've done some IO here"
    respond $ responseLBS
        status200
        [("Content-Type", "text/plain")]
        "Hello, Web!"
```

Creating web services

Servant example
(http://haskell-servant.readthedocs.io/en/stable/tutorial/index.html)

```haskell
type UserAPI = "users" :> Get '[JSON] [User] -- GET /users

data User = User
  { name :: String
  , age  :: Int
  } deriving (Eq, Show, Generic, ToJSON)

-- From DB in real life
users :: [User]
users = [ User "Isaac Newton" 372, User "Albert Einstein" 136]

server :: Server UserAPI
server = return users

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server
```

Honorable mentions

- Elm
- Elixir
- Rust

Questions?

fredrik@bright.io