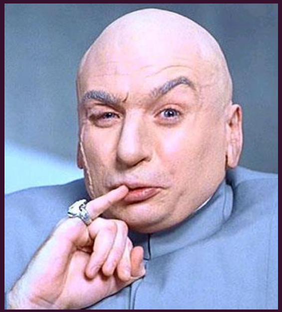
# Haxl:
# Haskell at Facebook

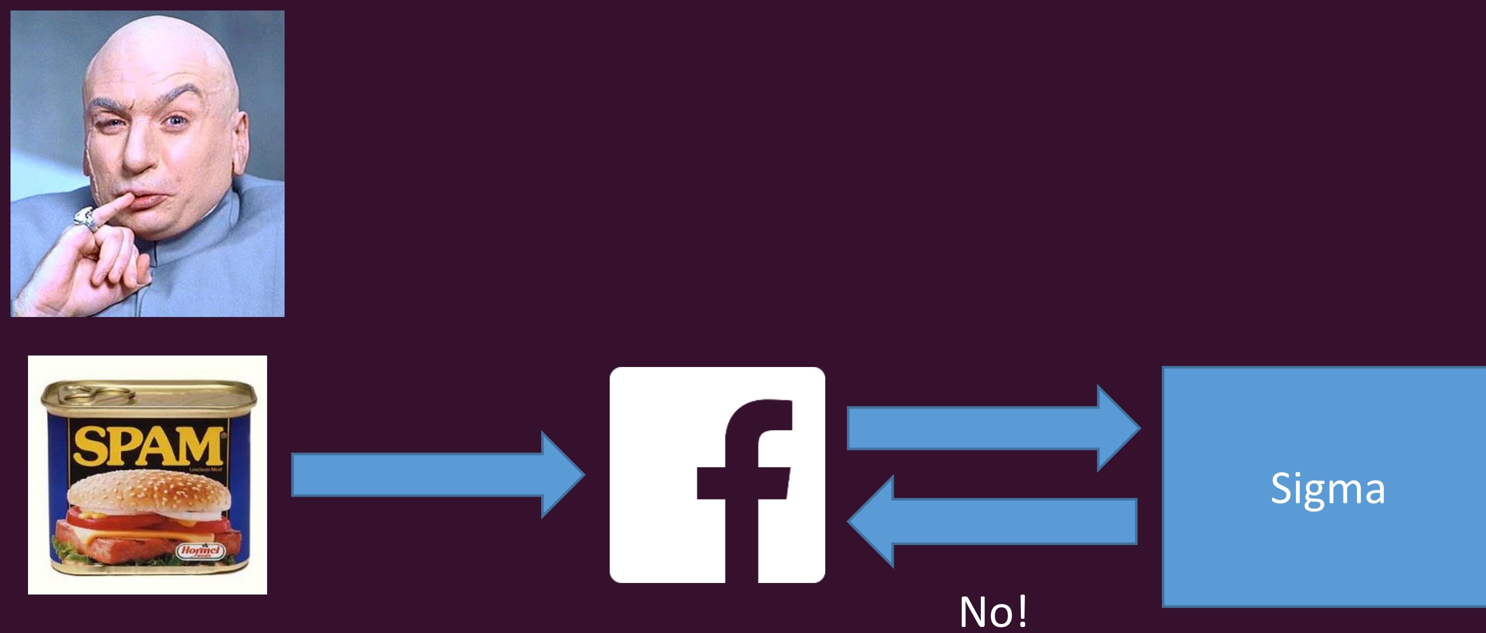
Simon Marlow
Jon Coens
Louis Brandy
Bartosz Nitka
Jon Purdy
Aaron Roth
& others

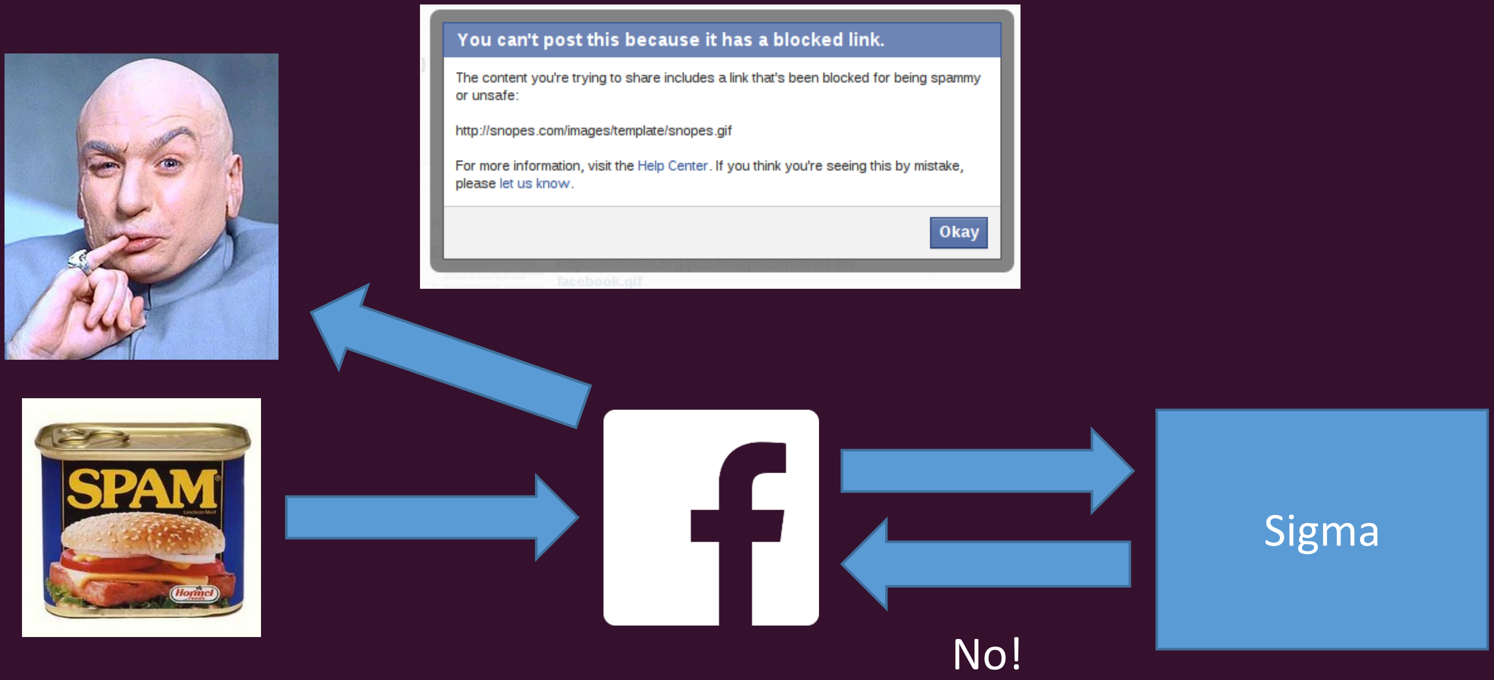
# What’s in this talk

- The Haxl project: ~12 months later, where are we
- Haxl published at ICFP’14!
- Haxl open source release!
  - walking through an example data source




SPAM


SPAM
Hormel
Sigma
No!


You can't post this because it has a blocked link.
The content you're trying to share includes a link that's been blocked for being spammy or unsafe:
http://snopes.com/images/template/snopes.gif
For more information, visit the Help Center. If you think you're seeing this by mistake, please let us know.
Okay
SPAM
Sigma
No!

# How does Sigma know what’s spam?

- FXL

```
JaffaCakeSpam =
  MessageContains(“Jaffa Cakes”) &&
  Let
    LikesJaffaCakes(X) = Likes(JaffaCakes,X)
  In
  Length(Filter(LikesJaffaCakes, FriendsOf(SourceId))) < 3
```

- + machine learning.

```
SpamMessage = SpamScore > 0.99
```

  - Where do the inputs to the ML come from?
    - FXL expressions.

# What can you do in FXL?

- Fetch data from the Facebook graph:

```
FriendCount(uid) = AssocCountByType(uid, AssocFriends)
```

- Fetch data from any of the other 18 data sources
- Run machine learning classifiers
- Perform simple computations

```
RatioFriendsSourceIdOver20 =
  If FriendCountSourceId > 0
    Then Ratio(CountFriendsSourceIdOver(20),
               FriendCountSourceId)
    Else 0.0;
```

# What’s good about FXL?

- Clean syntax
  - SI engineers concentrate on fighting spam, not the language
- Static typing
  - cannot push type-incorrect code
- We can push changes *fast*
  - a couple of minutes from commit to production

# What’s not so good?

# What’s not so good?

- Limited abstractions

    We’re building larger systems in FXL now

# What’s not so good?

- Limited abstractions

  We’re building larger systems in FXL now

- Design quirks and hysterical raisins

# What’s not so good?

- Limited abstractions

    We’re building larger systems in FXL now

- Design quirks and hysterical raisins

  Static typing is limited

  - Only a few types: int, double, string, vector, map, JSON
  - No user-defined types
  - Type system doesn’t catch as many errors as it could

# What’s not so good?

- Limited abstractions

    We’re building larger systems in FXL now

- Design quirks and hysterical raisins

  Static typing is limited

  - Only a few types: int, double, string, vector, map, JSON
  - No user-defined types
  - Type system doesn’t catch as many errors as it could

- Slow (it’s an interpreter)

# Why are we switching to Haskell?

- Expressivity
- Learning resources available
- Lots of libraries
- Faster (it's compiled)
- Better implementation (error messages etc.)
- Chance to redesign the whole system
  - guaranteed replayability

# Technical challenges

1. Implicit concurrency
2. Implement all the FXL functionality in Haskell
3. Translate all the FXL code
4. Figure out how to compile+push all the Haskell code to all of the machines in a few minutes

- Status summary: we're mostly done with 1,2,3 and experimenting with a solution for 4.
- Now: testing, bug fixing and optimisation.

# Implicit concurrency

- In FXL you can write this:

```
NumCommonFriends(x, y) =
   Length(Intersect(FriendsOf(x), FriendsOf(y)));
```

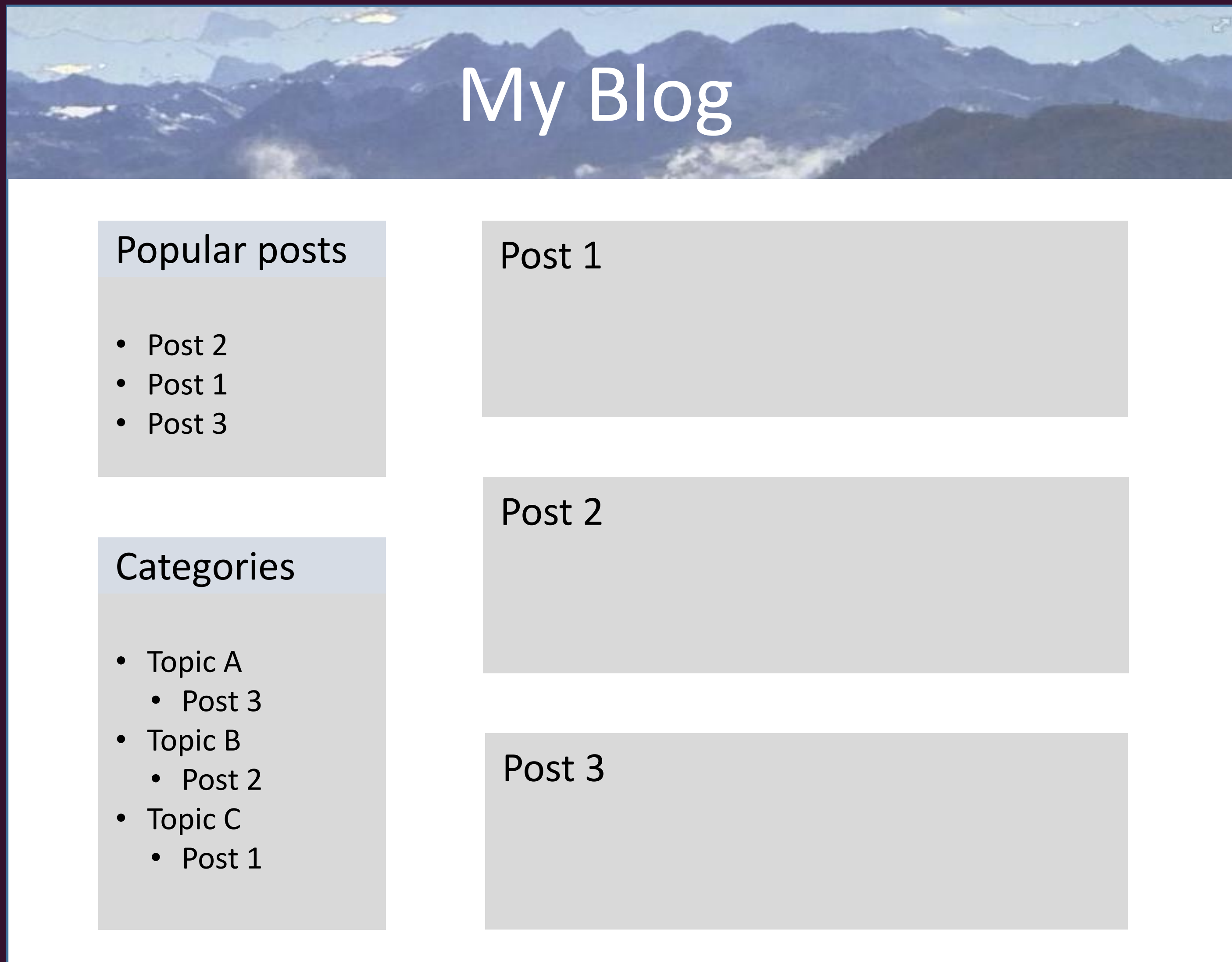
- And Sigma automatically batches the two requests together.
- With existing languages & frameworks you have to specify the concurrency explicitly...

- e.g using Haskell asyncs:

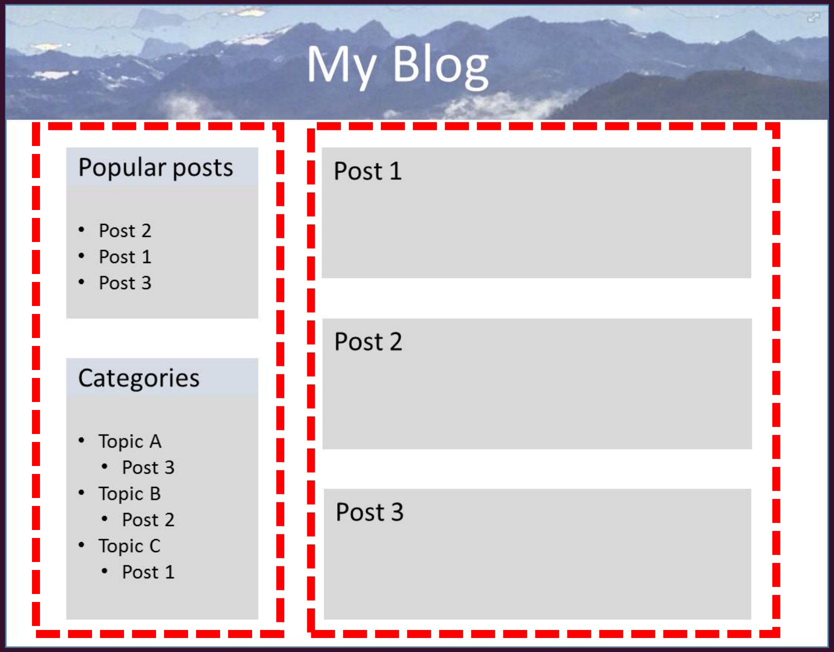
```
do
      ax <- async (friendsOf x)
      ay <- async (friendsOf y)
      fx <- wait ax
      fy <- wait ay
      return (length (intersect fx fy))
```

- Too verbose
- Prone to false dependencies

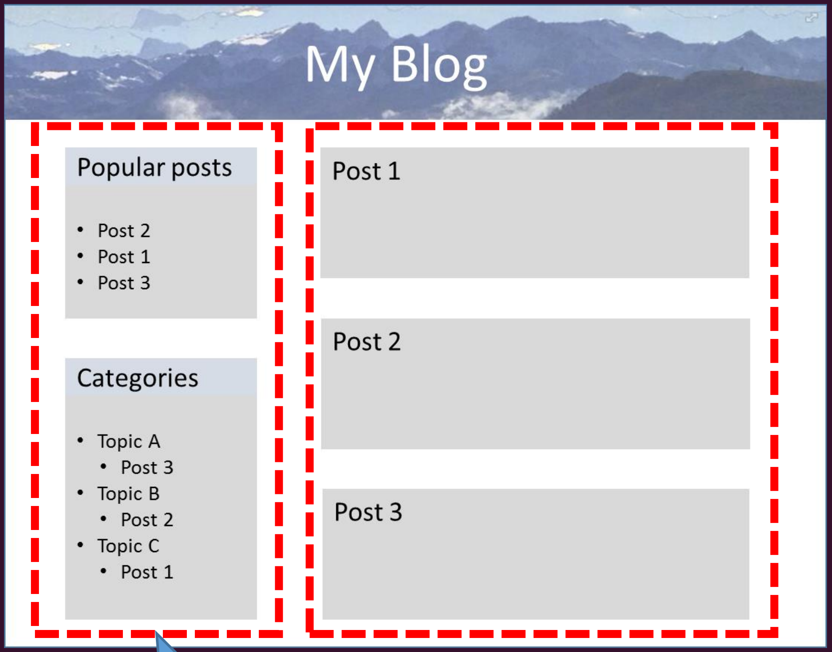
# Larger example: a blog server



# Another example: a blog server

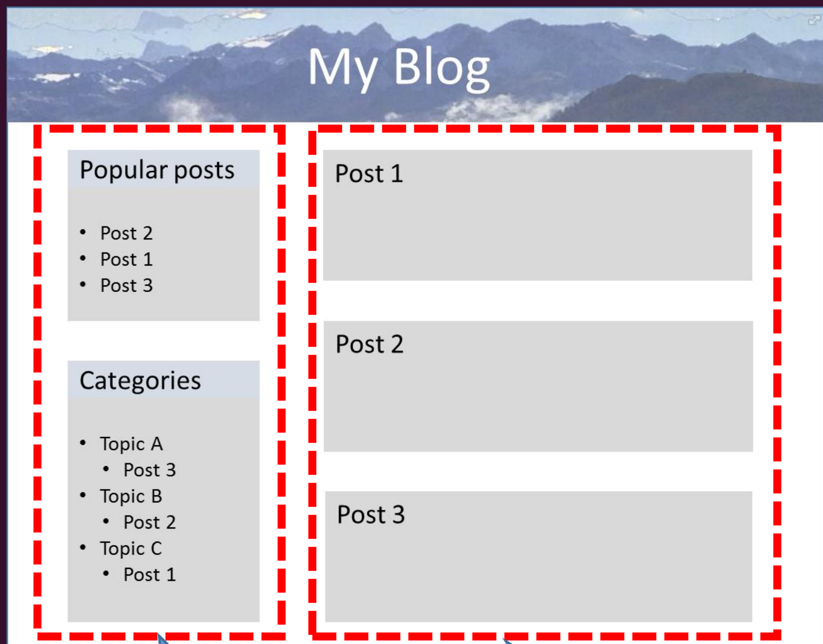


# Another example: a blog server




We want modular code – code each pane independently

# Another example: a blog server




We want modular code – code each pane independently

We want it to execute efficiently.

- Concurrent data fetches
- No repeated data fetches

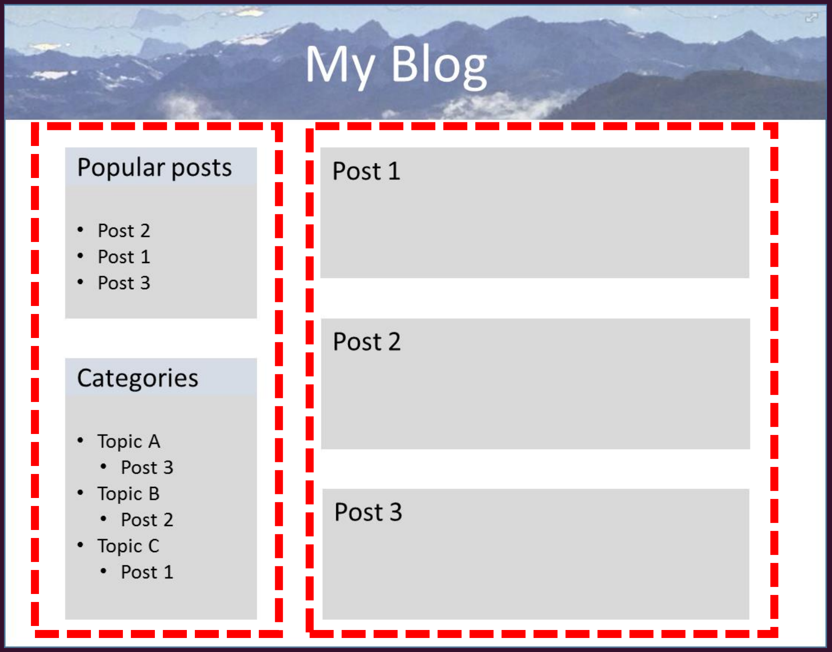
```haskell
data PostId      -- identifies a post
data Date        -- a calendar date
data PostContent -- the content of a post

data PostInfo = PostInfo
  { postId    :: PostId
  , postDate  :: Date
  , postTopic :: String
  }

-- data-fetching operations
getPostIds     :: Haxl [PostId]
getPostInfo    :: PostId -> Haxl PostInfo
getPostContent :: PostId -> Haxl PostContent
getPostViews   :: PostId -> Haxl Int

-- rendering functions
renderPosts    :: [(PostInfo,PostContent)] -> Html
renderPage     :: Html -> Html -> Html
...
```

```haskell
blog :: Haxl Html
blog = renderPage <$> leftPane <*> mainPane

getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = mapM getPostInfo =<< getPostIds

mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered =
        take 5 $
        sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```

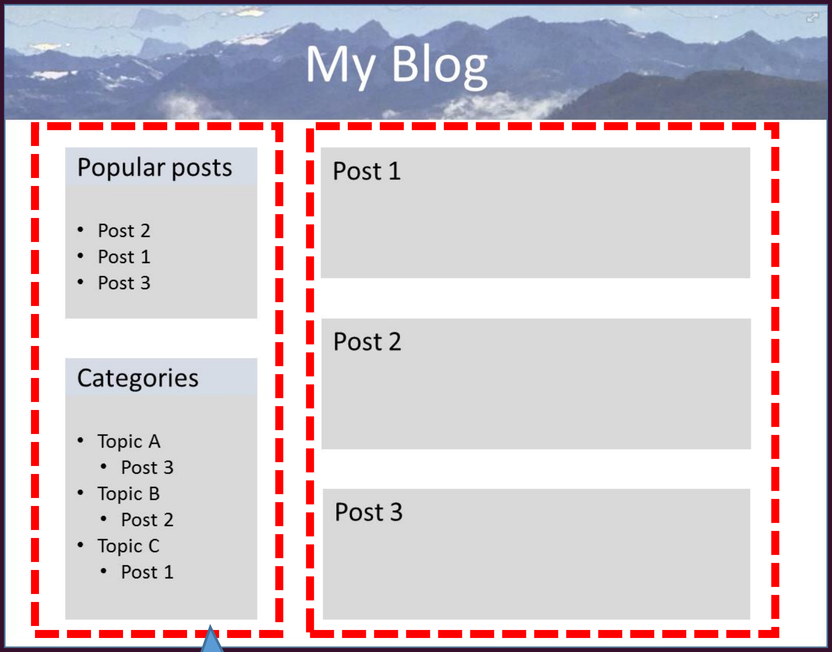
```haskell
leftPane :: Haxl Html
leftPane = renderSidePane <$> popularPosts <*> topics

getPostDetails :: PostId -> Haxl (PostInfo, PostContent)
getPostDetails pid =
 (,) <$> getPostInfo pid <*> getPostContent pid

popularPosts :: Haxl Html
popularPosts = do
  pids <- getPostIds
  views <- mapM getPostViews pids
  let ordered =
        take 5 $ map fst $
        sortBy (flip (comparing snd)) (zip pids views)
  content <- mapM getPostDetails ordered
  return $ renderPostList content
```

```haskell
topics :: Haxl Html
topics = do
  posts <- getAllPostsInfo
  let topiccounts =
        Map.fromListWith (+) [ (postTopic p, 1) | p <- posts ]
  return $ renderTopics topiccounts
```

# So how did we do?

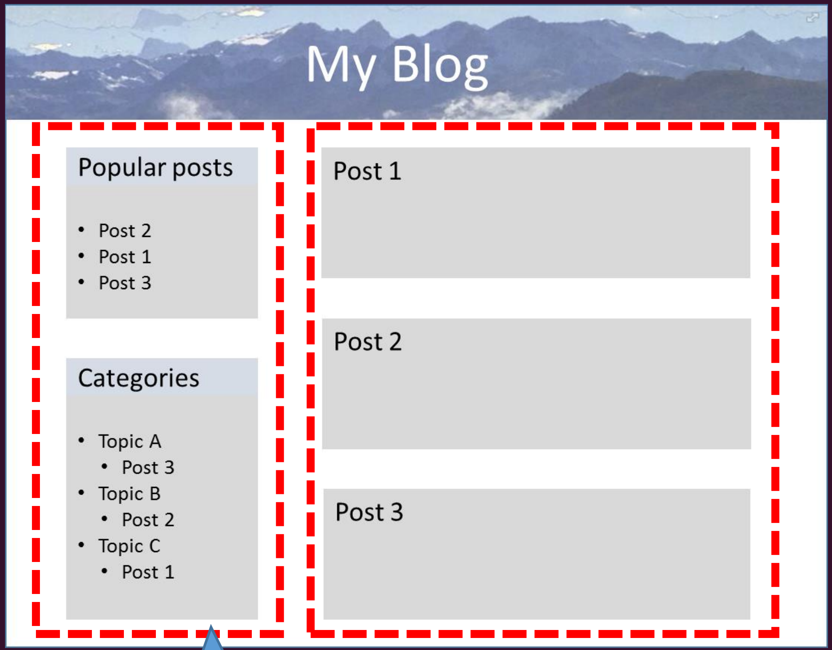


# So how did we do?



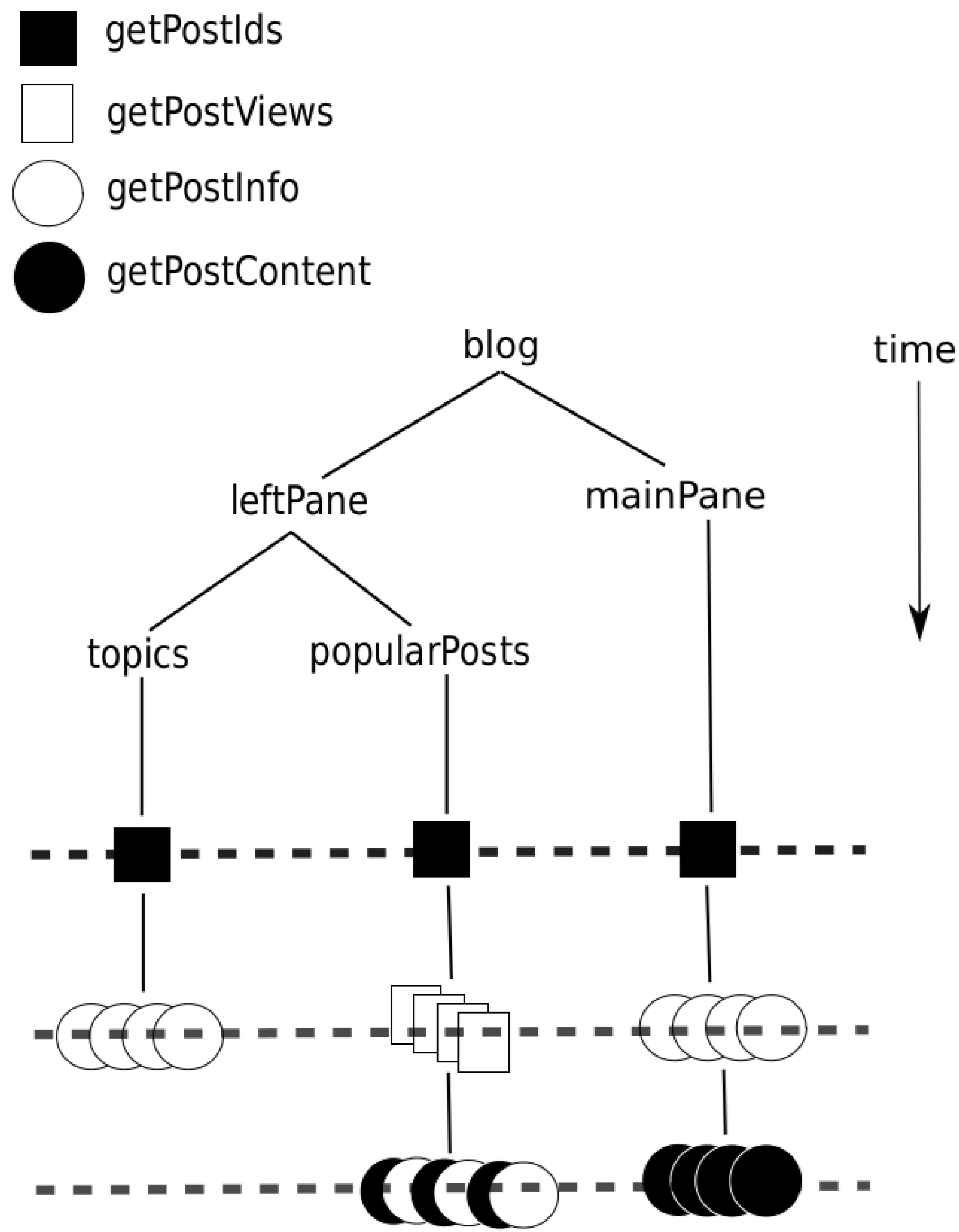

Code is clean and modular

# So how did we do?




Code is clean and modular

But does it execute efficiently?


getPostIds
getPostViews
getPostInfo
getPostContent
blog
time
leftPane
mainPane
topics
popularPosts

# Demo

# Implementation

- Start with a concurrency monad

```haskell
data Result a

    -- we're finished, here's the result
  = Done a

    -- the computation blocked...
  | Blocked
      (Seq BlockedRequest) -- requests to perform
      (Haxl a)             -- continuation
```

# Start with a concurrency monad

```
data Result a
  = Done a
  | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }

instance Monad Haxl where
  return a = Haxl $ return (Done a)

  Haxl m >>= k = Haxl $ do
    r <- m
    case r of
      Done a         -> unHaxl (k a)
      Blocked br c -> return (Blocked br (c >>= k))
```

# Add an Applicative instance

```haskell
instance Applicative Haxl where
  pure = return

  Haxl f <*> Haxl x = Haxl $ do
    f' <- f
    x' <- x
    case (f',x') of
      (Done g,         Done y        ) -> return (Done (g y))
      (Done g,         Blocked br c  ) -> return (Blocked br (g <$> c))
      (Blocked br c,   Done y        ) -> return (Blocked br (c <*> return y))
      (Blocked br1 c, Blocked br2 d) -> return (Blocked (br1 <> br2) (c <*> d))
```

# Fetching Data

```
dataFetch :: Request a -> Haxl a
```

GADT

Type parameter is the result type of the request

# Fetching Data

```
dataFetch :: Request a -> Haxl a
```

GADT

Type parameter is the result type of the request

```
data Request a where
  FetchPosts       :: Request [PostId]
  FetchPostInfo    :: PostId -> Request PostInfo
  FetchPostContent :: PostId -> Request PostContent
  FetchPostViews   :: PostId -> Request Int
```

# Fetching Data

```
dataFetch :: Request a -> Haxl a
```

GADT

Type parameter is the result type of the request

```
data Request a where
  FetchPosts       :: Request [PostId]
  FetchPostInfo    :: PostId -> Request PostInfo
  FetchPostContent :: PostId -> Request PostContent
  FetchPostViews   :: PostId -> Request Int
```

```
data FetchStatus a = NotFetched | FetchSuccess a

data BlockedRequest =
  forall a . BlockedRequest (Request a) (IORef (FetchStatus a))
```

- We can implement dataFetch:

```
dataFetch :: Request a -> Haxl a
dataFetch request = Haxl $ do
  box <- newIORef NotFetched
  let br = BlockedRequest request box
  let cont = Haxl $ do
        FetchSuccess a <- readIORef box
        return (Done a)
  return (Blocked (singleton br) cont)
```

- To fetch data, we need

```
fetch :: [BlockedRequest] -> IO ()
```

Application-specific data-fetching function.

Batches multiple requests, uses concurrency, etc.

- To run a computation to completion, we need a loop:

```haskell
runHaxl :: Haxl a -> IO a
runHaxl (Haxl h) = do
  r <- h
  case r of
    Done a -> return a
    Blocked br cont -> do
      fetch (toList br)
      runFetch cont
```

- Done!

# We also want caching

- Reader monad passes an IORef DataCache around
- Complication:
  - cache maps Request a to a
  - can't do this with Data.Map alone

```
newtype DataCache =
  DataCache (forall a . Map (Request a) (IORef (FetchStatus a)))
```

ICFP'14

# There is no Fork: an Abstraction for Efficient, Concurrent, and Concise Data Access


Simon Marlow
Facebook
smarlow@fb.com

Louis Brandy
Facebook
ldbrandy@fb.com

Jonathan Coens
Facebook
jon.coens@fb.com

Jon Purdy
Facebook
jonp@fb.com



## Abstract

We describe a new programming idiom for concurrency, based on Applicative Functors, where concurrency is implicit in the Applicative `<*>` operator. The result is that concurrent programs can be written in a natural applicative style, and they retain a high degree of clarity and modularity while executing with maximal concurrency. This idiom is particularly useful for programming against external data sources, where the application code is written without the use of explicit concurrency constructs, while the implementation is able to batch together multiple requests for data from the same source, and fetch data from multiple sources concurrently. Our abstraction uses a cache to ensure that multiple requests for the same data return the same result, which frees the programmer from having to arrange to fetch data only once, which in turn leads to greater modularity.

While it is generally applicable, our technique was designed with a particular application in mind: an internal service at Facebook that identifies particular types of content and takes actions based on it. Our application has a large body of business logic that fetches data from as many as 15 different external sources. The framework described in this paper enables the business logic to execute efficiently by automatically fetching data concurrently; we present some preliminary results.


efficiency in this setting: accessing multiple remote data sources efficiently requires *concurrency*, and that normally requires the programmer to intervene and program the concurrency explicitly.

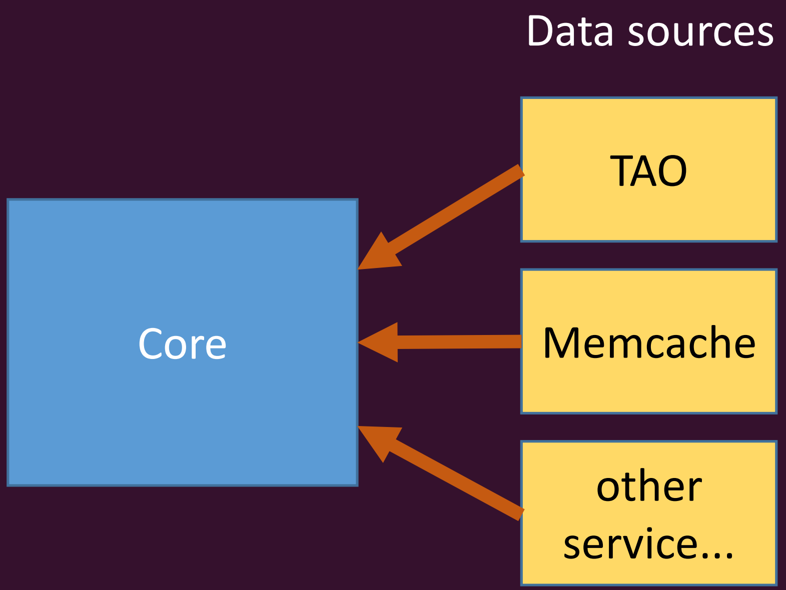
When the business logic is only concerned with *reading* data from external sources and not *writing*, the programmer doesn't care about the order in which data accesses happen, since there are no side-effects that could make the result different when the order changes. So in this case the programmer would be entirely happy with not having to specify either ordering or concurrency, and letting the system perform data access in the most efficient way possible. In this paper we present an embedded domain-specific language (EDSL), written in Haskell, that facilitates this style of programming, while automatically extracting and exploiting any concurrency inherent in the program.

Our contributions can be summarised as follows:

- We present an `Applicative` abstraction that allows implicit concurrency to be extracted from computations written with a combination of `Monad` and `Applicative`. This is an extension of the idea of concurrency monads [9], using Applicative `<*>` as a way to introduce concurrency (Section 4). We then develop the idea into an abstraction that supports concurrent access to remote data (Section 5), and failure (Section 8).
- We show how to add a *cache* to the framework (Section 6).

# But…

- The Request type was wired into the monad
- How can we make the monad independent of the data source(s)?




- Core code includes the monad, caching support etc.
- Core is *generic:* no data sources built-in

```
dataFetch :: Request a -> Haxl a
```

```haskell
dataFetch :: Request a -> Haxl a
```

```haskell
dataFetch :: (DataSource u req, Request req a)
          => req a
          -> GenHaxl u a
```

```
dataFetch :: (DataSource u req, Request req a)
          => req a
          -> GenHaxl u a
```

```
dataFetch :: (DataSource u req, Request req a)
          => req a
          -> GenHaxl u a
```

DataSource class: every data source implements this

```haskell
dataFetch :: (DataSource u req, Request req a)
          => req a
          -> GenHaxl u a
```

DataSource class: every data source implements this

Request class: just Eq, Hashable, Typeable, Show

```
dataFetch :: (DataSource u req, Request req a)
          => req a
          -> GenHaxl u a
```

DataSource class: every data source implements this

Request class: just Eq, Hashable, Typeable, Show

User state – passed around, can be accessed by data sources

# DataSource walk-through

- We'll walk through constructing a complete data source
- We'll make a data source for the Facebook Graph API
  - web API for querying the Facebook Graph
  - using Felipe Lessa's fb package to do the real work
  - our data source will perform requests concurrently up to a maximum number of threads

- Start with the request type:

```haskell
data FacebookReq a where
   GetObject       :: Id -> FacebookReq Object
   GetUser         :: UserId -> FacebookReq User
   GetUserFriends :: UserId -> FacebookReq [Friend]
  deriving Typeable
```

GADT, as before

- We also need some boilerplate:

```haskell
deriving instance Eq (FacebookReq a)
deriving instance Show (FacebookReq a)

instance Show1 FacebookReq where show1 = show

instance Hashable (FacebookReq a) where ...
```

- A data source has some state:

```
instance StateKey FacebookReq where
  data State FacebookReq =
    FacebookState
       { credentials :: Credentials
       , userAccessToken :: UserAccessToken
       , manager :: Manager
       , numThreads :: Int
       }
```

API keys

HTTP connection manager

Concurrency control

- Initialise the state:

```
initGlobalState
  :: Int
  -> Credentials
  -> UserAccessToken
  -> IO (State FacebookReq)

initGlobalState threads creds token = do
  manager <- newManager tlsManagerSettings
  return FacebookState
    { credentials = creds
    , manager = manager
    , userAccessToken = token
    , numThreads = threads
    }
```

- nothing surprising there.

- Make an instance of DataSource

```
class DataSourceName req where
  dataSourceName :: req a -> Text

class (DataSourceName req, StateKey req, Show1 req)
      => DataSource u req where
  fetch
    :: State req
    -> Flags
    -> u
    -> [BlockedFetch req]
    -> PerformFetch
```

```
instance DataSourceName FacebookReq where
 dataSourceName _ = "Facebook“

instance DataSource u FacebookReq where
  fetch = facebookFetch
```

- Implement fetch

```
data PerformFetch
  = SyncFetch  (IO ())
  | AsyncFetch (IO () -> IO ())
```

IO to do while the requests are in progress

```
facebookFetch
  :: State FacebookReq
  -> Flags
  -> ()
  -> [BlockedFetch FacebookReq]
  -> PerformFetch

facebookFetch FacebookState{..} _flags _user bfs =
  AsyncFetch $ \inner -> do
    sem <- newQSem numThreads
    asyncs <- mapM (async . fetchAsync credentials manager
                                  userAccessToken sem) bfs
    inner
    mapM_ wait asyncs
```

- Implement fetchAsync

```
fetchAsync
  :: Credentials -> Manager -> UserAccessToken -> QSem
  -> BlockedFetch FacebookReq
  -> IO ()

fetchAsync creds manager tok sem (BlockedFetch req rvar) =
  bracket_ (waitQSem sem) (signalQSem sem) $ do

    e <- Control.Exception.try $
         runResourceT $
         runFacebookT creds manager $
         fetchReq tok req

    case e of
      Left ex -> putFailure rvar (ex :: SomeException)
      Right a -> putSuccess rvar a
```

- fetchReq maps FacebookReq to FacebookT computations

```
fetchReq
  :: UserAccessToken
  -> FacebookReq a
  -> FacebookT Auth (ResourceT IO) a

fetchReq tok (GetObject (Id id)) =
  getObject ("/" <> id) [] (Just tok)

fetchReq _tok (GetUser id) =
  getUser id [] Nothing

fetchReq tok (GetUserFriends id) = do
  f <- getUserFriends id [] tok
  source <- fetchAllNextPages f
  source $$ consume
```

- Example

```haskell
main :: IO ()
main = do
  (creds, access_token) <- getCredentials
  facebookState <- initGlobalState 10 creds access_token
  env <- initEnv (stateSet facebookState stateEmpty) ()
  r <- runHaxl env $ do
    likes <- getObject "me/likes"
    mapM getObject (likeIds likes)
  print r
```

Many requests, performed concurrently.

# Back to our Haxl project...

- But do people have to learn <$>, <*>, etc?

```
numCommonFriends x y =
   length <$> (intersect <$> friendsOf x <*> friendsOf y)
```

- No, because this

```
numCommonFriends x y = do
   fx <- friendsOf x
   fy <- friendsOf y
   return (length (intersect fx fy))
```

- can be silently translated to the Applicative form in the compiler
  - (not implemented yet)

- Going further, we could write a pre-processor from this:

```
numCommonFriends :: Haxl Int
numCommonFriends =
  length (intersect (friendsOf sourceId) (friendsOf targetId))
```

- To this:

```
numCommonFriends :: Haxl Int
numCommonFriends
 = length <$>
    (intersect <$>
      (join (friendsOf <$> sourceId)) <*>
      (join (friendsOf <$> targetId)))
```

- straightforward with haskell-src-exts

# Dilemna: monads or no monads?

- Using a preprocessor
  - Advantages
    - Everything is monadic, but looks pure to the programmer
    - Easier to understand
  - Disadvantages
    - Can't write pure code
    - Hard to interpret error messages
    - Two languages adds complexity

- We decided not to go this route (for now)

# Technical challenges

1. Implicit concurrency
2. Implement all the FXL functionality in Haskell
3. Translate all the FXL code
4. Figure out how to compile+push all the Haskell code to all of the machines in a few minutes

# Implement all the FXL functionality in Haskell

- several data sources
- Each needs a Haskell/C++ FFI layer
- ~450 built-in functions
  - Ranging from easy (StrCmp) to really annoying (ParseActivityLog)
- We created lots of tasks
  - some done by the team
  - (we're working on our 4th iteration of the TAO layer)
  - others grabbed by interested people around Facebook: hack-a-month projects and bootcamp

- As of two weeks ago, we have everything implemented!

# Technical challenges

1. Implicit concurrency
2. Implement all the FXL functionality in Haskell
3. Translate all the FXL code
4. Figure out how to compile+push all the Haskell code to all of the machines in a few minutes

# Translate all the FXL code

- We have a *lot* of FXL code
  - impractical to translate it all by hand
- Wrote a translation tool
  - tricky bit is converting to do-syntax or Applicative where necessary, while keeping as much code as possible pure
- Auto-translated code will become the source
  - Try to produce readable code

# Technical challenges

1. Implicit concurrency
2. Implement all the FXL functionality in Haskell
3. Translate all the FXL code
4. Figure out how to compile+push all the Haskell code to all of the machines in a few minutes

# Compile time

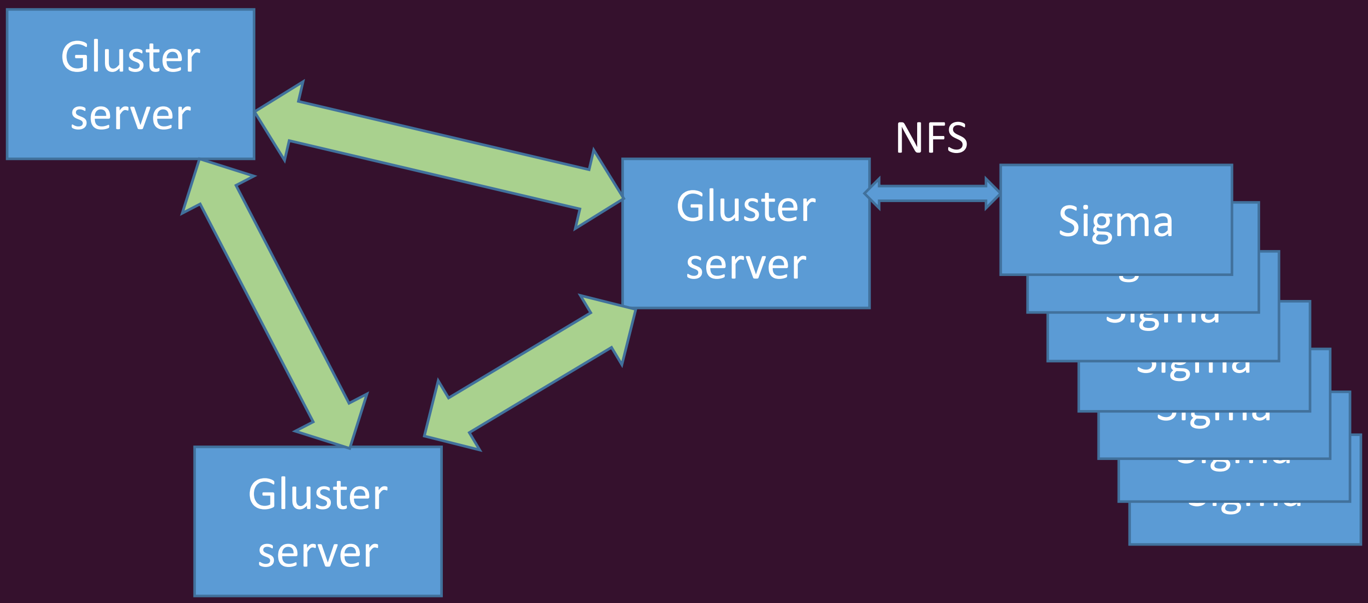
- At first, compiling the whole translated codebase took ~30 mins
- (FXL push currently takes ~2 mins)

# Reducing compile time

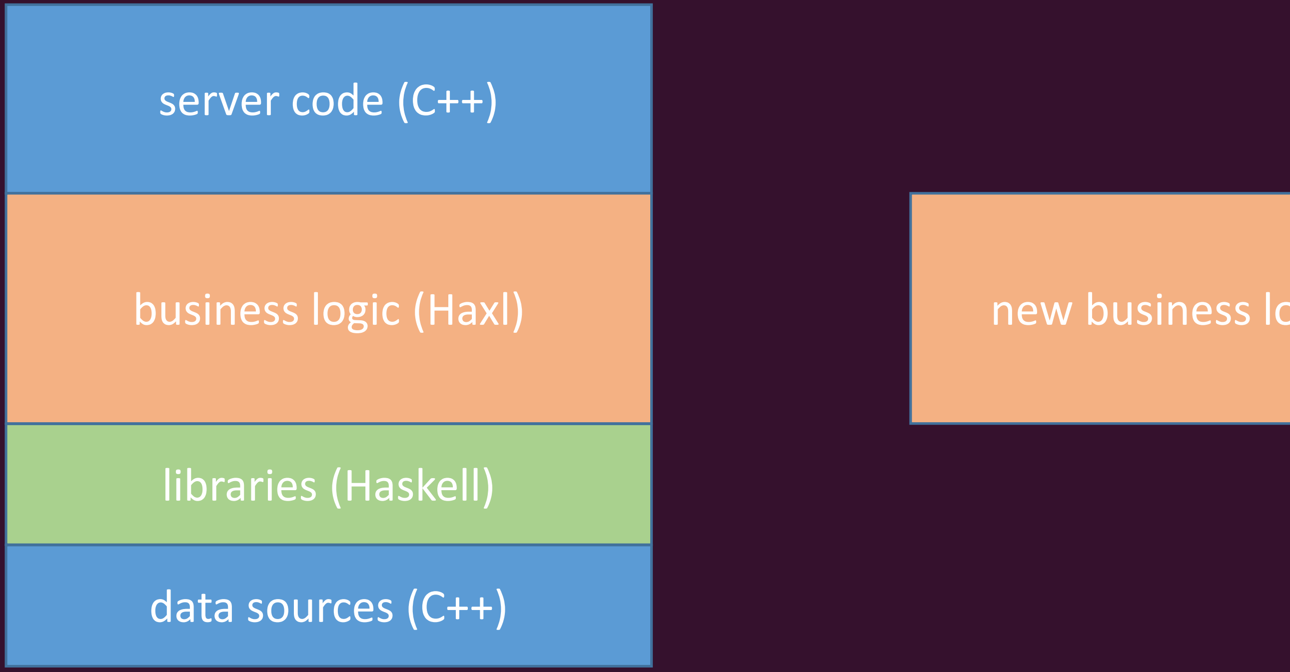
- Long laborious process to impose a sensible module structure
  - FXL source files now form a DAG
  - Compilation has some parallelism now
- We have full compile down to ~5 mins
  - incremental compile usually much faster (~2 mins)

# How to push to Sigma machines?

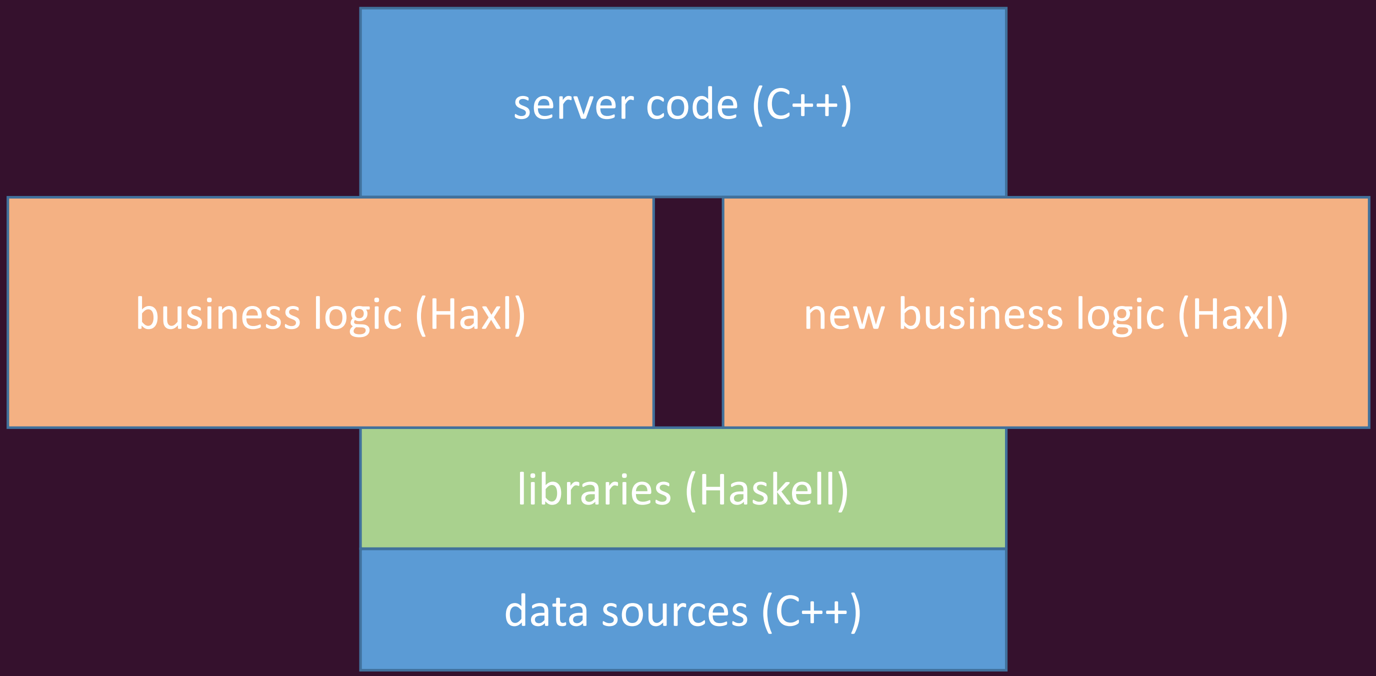
- We’re experimenting with GlusterFS for distribution.
  - seems good: all machines get a new object in <1 min



# Hot code swapping



# Hot code swapping



# Hot code swapping

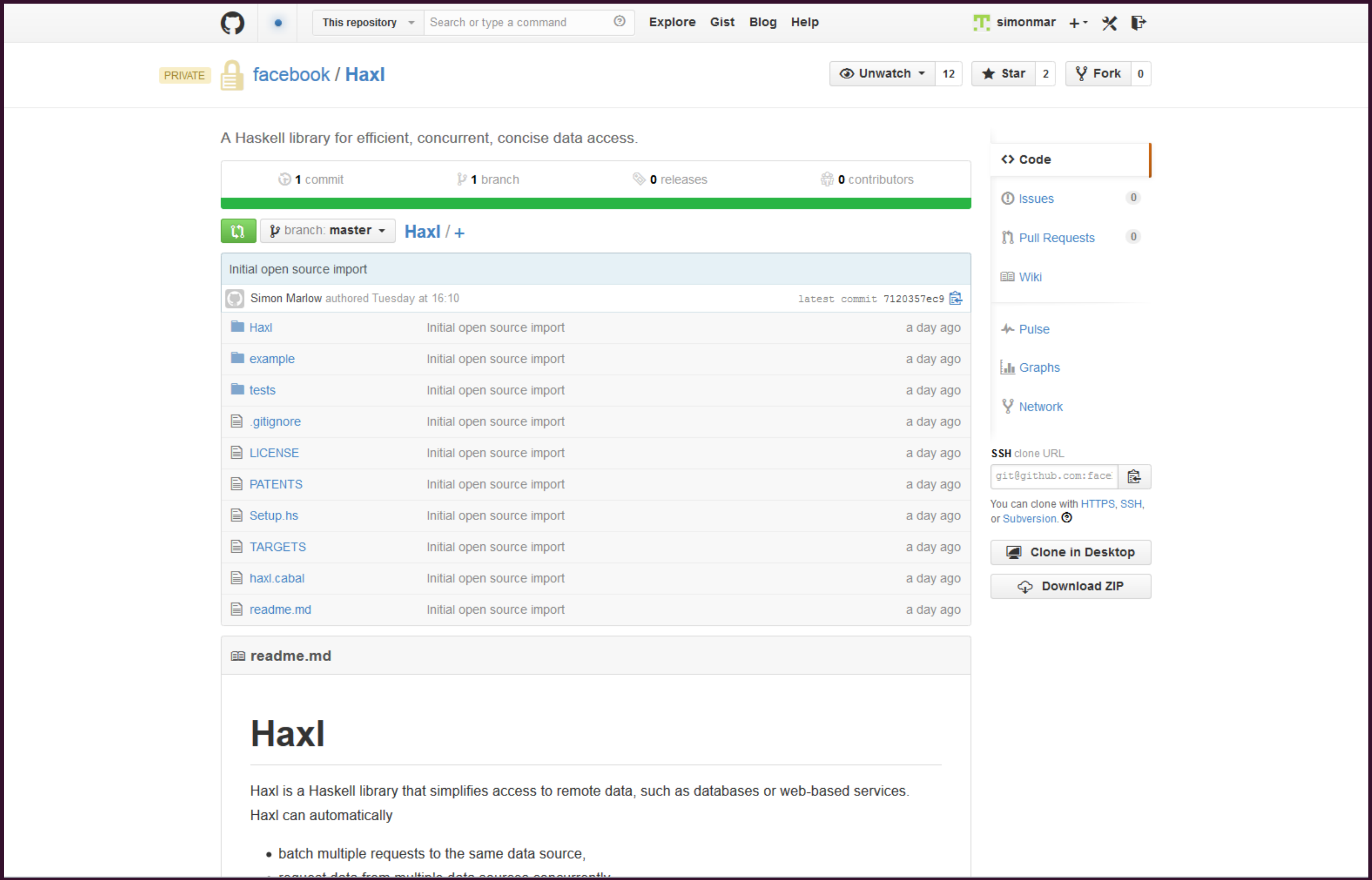
- Keep serving requests while we load new code

- Use GHC’s built-in linker
  - Had to modify it to unload code (shipped in GHC 7.8)
  - GC detects when it is safe to release old code
  - We can have multiple copies of the code running while existing requests drain

# Status

- Call graph complete
- Full FXL codebase translated
- Next goals:
  - achieve 99% correctness (100% hard due to random effects)
  - get performance up par with FXL
  - experiment with running production traffic
  - open source...

# Open Source! (coming next week)



# Haxl:
# Haskell at Facebook


Simon Marlow
Jon Coens
Louis Brandy
Bartosz Nitka
Jon Purdy
Aaron Roth
& others
**<your name here>**

# Questions?