The Haxl Project at Facebook

Simon Marlow
Jon Coens
Louis Brandy
Jon Purdy
& others
Use case: fighting spam

www (PHP)

Business Logic

Is this thing spam?

YES/NO

Databases

Other back-end services
Use case: fighting spam

www (PHP) -> Business Logic

Site-integrity engineers
push new rules hundreds of times per day

Databases

Other back-end services
Data dependencies in a computation
Code wants to be structured hierarchically

- abstraction
- modularity
Code wants to be structured hierarchically

- abstraction
- modularity
Code wants to be structured hierarchically

- abstraction
- modularity
Code wants to be structured hierarchically

- abstraction
- modularity
Execution wants to be structured horizontally

- Overlap multiple requests
- Batch requests to the same data source
- Cache multiple requests for the same data

**Database**

**Thrift**

**Memcache**
• Furthermore, each data source has different characteristics
  • Batch request API?
  • Sync or async API?
  • Set up a new connection for each request, or keep a pool of connections around?

• Want to abstract away from all of this in the business logic layer
But we know how to do this!
But we know how to do this!

- Concurrency.
  - Threads let us keep our abstractions & modularity while executing things at the same time.
  - Caching/batching can be implemented as a service in the process
    - as we do with the IO manager in GHC
But we know how to do this!

• Concurrency.
  • Threads let us keep our abstractions & modularity while executing things at the same time.
  • Caching/batching can be implemented as a service in the process
    • as we do with the IO manager in GHC

• But concurrency (the programming model) isn’t what we want here.
But we know how to do this!

• Concurrency.
  • Threads let us keep our abstractions & modularity while executing things at the same time.
  • Caching/batching can be implemented as a service in the process
    • as we do with the IO manager in GHC

• But concurrency (the programming model) isn’t what we want here.

• Example...
• x and y are Facebook users
• suppose we want to compute the number of friends that x and y have in common
• simplest way to write this:

```
length (intersect (friendsOf x) (friendsOf y))
```
Brief detour: TAO

• TAO implements Facebook’s data model
  • most important data source we need to deal with
• Data is a graph
  • Nodes are “objects”, identified by 64-bit ID
  • Edges are “assocs” (directed; a pair of 64-bit IDs)
• Objects and assocs have a type
  • object fields determined by the type
• Basic operations:
  • Get the object with a given ID
  • Get the assocs of a given type from a given ID
• Back to our example

```
length (intersect (friendsOf x) (friendsOf y))
```

• \((\text{friendsOf } x)\) makes a request to TAO to get all the IDs for which there is an assoc of type FRIEND \((x,\_).\)

• TAO has a multi-get API; very important that we submit \((\text{friendsOf } x)\) and \((\text{friendsOf } y)\) as a single operation.
Using concurrency

• This:

```plaintext
length (intersect (friendsOf x) (friendsOf y))
```
Using concurrency

• This:

    length (intersect (friendsOf x) (friendsOf y))

• Becomes this:

    do
    m1 <- newEmptyMVar
    m2 <- newEmptyMVar
    forkIO (friendsOf x >>= putMVar m1)
    forkIO (friendsOf y >>= putMVar m2)
    fx <- takeMVar m1
    fy <- takeMVar m2
    return (length (intersect fx fy))
OH GOD
MY EYES
• Using the async package:

```haskell
do
    ax <- async (friendsOf x)
    ay <- async (friendsOf y)
    fx <- wait ax
    fy <- wait ay
    return (length (intersect fx fy))
```
• Using Control.Concurrent.Async.concurrently:

```
{do
  (fx,fy) <- concurrently (friendsOf x) (friendsOf y)
  return (length (intersect fx fy))
```
Why not concurrency?

• `friendsOf x` and `friendsOf y` are
  • obviously independent
  • obviously both needed
  • “pure”
Why not concurrency?

- `friendsOf x` and `friendsOf y` are
  - obviously independent
  - obviously both needed
  - “pure”

- Caching is not just an optimisation:
  - if `friendsOf x` is requested twice, we *must* get the same answer both times
  - *caching is a requirement*
Why not concurrency?

- `friendsOf x` and `friendsOf y` are
  - obviously independent
  - obviously both needed
  - “pure”

- Caching is not just an optimisation:
  - if `friendsOf x` is requested twice, we *must* get the same answer both times
  - *caching is a requirement*

- we don’t want the programmer to have to ask for concurrency here
• Could we use `unsafePerformIO`?

```haskell
length (intersect (friendsOf x) (friendsOf y))
friendsOf = unsafePerformIO ( .. )
```

• we could do caching this way, but not concurrency. Execution will stop at the first data fetch.
Central problem

- Reorder execution of an expression to perform data fetching optimally.
- The programming model has no side effects (other than reading)
What we would like to do:

- explore the expression along all branches to get a set of data fetched
What we would like to do:

- submit the data fetches
What we would like to do:

• wait for the responses
What we would like to do:

• now the computation is unblocked along multiple paths
• ... explore again
• collect the next batch of data fetches
• and so on
• Facebook’s existing solution to this problem: FXL
• Lets you write

    \[ \text{Length(Intersect(FriendsOf(X), FriendsOf(Y)))} \]

• And optimises the data fetching correctly.
• But it’s an interpreter, and works with an explicit representation of the computation graph.
• We want to run compiled code for efficiency
• And take advantage of Haskell
  • high quality implementation
  • great libraries for writing business logic etc.

• So, how can we implement the right data fetching behaviour in a Haskell DSL?
newtype Haxl a = Haxl { unHaxl :: Result a }

data Result a = Done a
               | Blocked (Haxl a)

instance Monad Haxl where
  return a = Haxl (Done a)
  m >>= k = Haxl $
               case unHaxl m of
               Done a    -> unHaxl (k a)
               Blocked r -> Blocked (r >>= k)
• Old idea:
  • Scholz 1995 *A Concurrency Monad Based on Constructor Primitives*
  • Claessen’s *A Poor Man’s Concurrency Monad* (JFP 1999)
  • Called the “Resumption Monad” in Harrison’s *Cheap (But Functional) Threads* (JFP 2004)
  • Used to good effect in Li & Zdancewic’s PLDI’07 paper *Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives*
  • Nowadays it’s a Free Monad
• The concurrency monad lets us run a computation until it blocks, do something, then resume it
• But we need to know what it blocked on...
• Could add some info to the **Blocked** constructor
• We choose to put it in IO instead:

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

data Result a = Done a
  | Blocked (Haxl a)

instance Monad Haxl where
  return a = Haxl (return (Done a))
  m >>= k = Haxl $ do
    a <- unHaxl m
    case a of
      Done a -> unHaxl (k a)
      Blocked r -> return (Blocked (r >>= k))

dataFetch :: Request a -> Haxl a
dataFetch r = do
  addRequest r
  Haxl (return (Blocked (Haxl (getResult r)))))
```
• Ok so far, but we still get blocked at the first data fetch.

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

Blocked here
• To explore multiple branches, we need to use **Applicative**

```haskell
instance Applicative Haxl where
    pure = return
    Haxl f <*> Haxl a = Haxl $ do
        r <- f
        case r of
            Done f' -> do
                ra <- a
                case ra of
                    Done a' -> return (Done (f' a'))
                    Blocked a' -> return (Blocked (f' <$> return a'))
            Blocked f' -> do
                ra <- a
                case ra of
                    Done a' -> return (Blocked (f' <*> return a'))
                    Blocked a' -> return (Blocked (f' <*> a'))
```
• This is precisely the advantage of **Applicative** over **Monad**:
  
  • **Applicative** allows exploration of the structure of the computation

• Our example is now written:

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

• Or:

```haskell
numCommonFriends x y =
  length <$> common (friendsOf x) (friendsOf y)
  where common = liftA2 intersect
```
• Note that we still have the Monad!
• The Monad allows us to make decisions based on values when we need to.

```haskell
do
  fs <- friendsOf x
  if simon `elem` fs
    then ...
   else ...
```

• Batching will not explore the then/else branches
  • exactly what we want.
• But it does mean the programmer should use Applicative composition to get batching.

• This is suboptimal:

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

• So our plan is to guide programmers away from this by using a combination of:
  • APIs that discourage it (e.g. providing lifted operations)
  • Code analysis and automatic feedback
  • Profiling
We really want bulk operations to benefit from batching.

But this doesn’t work: `mapM` uses `Monad` rather than `Applicative` composition.

This is why `traverse` exists:

```haskell
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
```

So in our library, we make `mapM = traverse`

Also: `sequence = sequenceA`
fpFriends :: Id -> Haxl [Text]
fpFriends id = do
  fs <- friendsOf id
  fp <- filterM (memberOfGroup functionalProgramming) fs
  mapM getName fp
Return the names of all the friends of the user that are members of the Functional Programming group.

```haskell
fPFriends :: Id -> Haxl [Text]
fPFriends id = do
  fs <- friendsOf id
  fp <- filterM (memberOfGroup functionalProgramming) fs
  mapM getName fp
```
•But we need a different filterM. The standard one:

```haskell
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
filterM _ [] = return []
filterM p (x:xs) = do
  flg <- p x
  ys <- filterM p xs
  return (if flg then x:ys else ys)
```
•But we need a different filterM. The standard one:

\[
\text{filterM} :: (\text{Monad } m) \Rightarrow (a \rightarrow m \text{ Bool}) \rightarrow [a] \rightarrow m [a]
\]

\[
\text{filterM } [] \quad = \quad \text{return } []
\]

\[
\text{filterM } p \ (x:xs) \quad = \quad \text{do}
\]

\[
\text{flg} \quad <- \quad p \ x
\]

\[
\text{ys} \quad <- \quad \text{filterM } p \ xs
\]

\[
\text{return } (\text{if } \text{flg} \quad \text{then } x:ys \quad \text{else } ys)
\]

•and the one that batches the data fetches correctly:

\[
\text{filterM} :: (\text{Applicative } f, \text{Monad } f) \Rightarrow (a \rightarrow f \text{ Bool}) \rightarrow [a] \rightarrow f [a]
\]

\[
\text{filterM } \text{pred } \text{xs} \quad = \quad \text{do}
\]

\[
\text{bools} \quad <- \quad \text{mapM } \text{pred } \text{xs}
\]

\[
\text{return } [ \ x \mid (x,\text{True}) \quad <- \quad \text{zip } \text{xs } \text{bools} ]
\]
• Let’s write the same example in a slightly different way:

```haskell
nameOfFPFriend :: Id -> Haxl (Maybe Text)
nameOfFPFriend id = do
  b <-memberOfGroup functionalProgramming id
  if b then Just <$> getName id
    else return Nothing

fpFriends2 :: Id -> Haxl [Text]
fpFriends2 id = do
  fs <- friendsOf id
  name_maybes <- mapM nameOfFPFriend fs
  return (catMaybes name_maybes)
```
• Before there were obviously 3 rounds.
• Before there were obviously 3 rounds.
• But in the new version, we packaged up two fetches in `nameOfFPFriend`
• Before there were obviously 3 rounds.
• But in the new version, we packaged up two fetches in `nameOfFPFriend`
• The monad still executes everything in 3 rounds.
• But maybe we don’t like all this `mapM/filterM` stuff
• But maybe we don’t like all this `mapM/filterM` stuff
• A query-style API might be nicer.
• But maybe we don’t like all this `mapM/filterM` stuff
• A query-style API might be nicer.

```haskell
newtype HaxlQuery a = HaxlQuery { query :: Haxl [a] }
  -- basically `ListT Haxl`, but `ListT` doesn't work!
  -- instances of Monad, MonadPlus

selectFrom :: Haxl [a] -> HaxlQuery a
selectFrom = HaxlQuery

liftH :: Haxl a -> HaxlQuery a
liftH m = HaxlQuery (return <$> m)

suchThat :: Haxl Bool -> HaxlQuery ()
suchThat m = liftH m >>= guard
```
But maybe we don’t like all this `mapM/filterM` stuff
A query-style API might be nicer.

```haskell
newtype HaxlQuery a = HaxlQuery { query :: Haxl [a] }
  -- basically ListT Haxl, but ListT doesn’t work!
  -- instances of Monad, MonadPlus

selectFrom :: Haxl [a] -> HaxlQuery a
selectFrom = HaxlQuery

liftH :: Haxl a -> HaxlQuery a
liftH m = HaxlQuery (return <$> m)

suchThat :: Haxl Bool -> HaxlQuery ()
suchThat m = liftH m >>= guard

fpFriends3 :: Id -> Haxl [Text]
fpFriends3 id =
  query [ name | f <- selectFrom (friendsOf id),
          _ <- suchThat (memberOfFPGroup f)
           name <- liftH (getName f) ]
```
DSL tricks

```haskell
instance Num a => Num (Haxl a) where
  (+) = liftA2 (+)
  (\_) = liftA2 (\_)
-- etc.

simonFriends id =
  filterM (isCalled "Simon") =<< friendsOf id

numCoolFriends id =
  length <$> fpFriends id + length <$> simonFriends id
```

duplicate data fetching between the two arguments
Implementation

• DataSource abstraction
• Replaying requests
• Scaling
• Hot-code swapping

• Experience
• Status etc.
Data Source Abstraction

- We want to structure the system like this:

  ![Diagram showing core and data sources]

  - Core code includes the monad, caching support etc.
  - Core is *generic*: no data sources built-in
How do we arrange this?

- Three ways that a data source interacts with core:
  - issuing a data fetch request
  - persistent state
  - fetching the data

- Package this up in a type class

```hs
class DataSource req where
...
```

- Let’s look at requests first...
Example Request type

```haskell
data ExampleReq a where
  CountAardvarks :: String -> ExampleReq Int
  ListWombats   :: Id     -> ExampleReq [Id]
 deriving Typeable
```

- Core has a single way to issue a request

```haskell
dataFetch :: DataSource req => req a -> Haxl a
```

- Note how the result type matches up.
Fetching requests

- It is Core’s job to keep track of requests submitted via `dataFetch`
- When the computation is blocked, we have to fetch the batch of requests from the data source

```haskell
class DataSource req where
    fetch :: [BlockedFetch req] -> IO ()

data BlockedFetch req
    = forall a . BlockedFetch (req a) (MVar a)
```
How do we store the requests?

- Core doesn’t know which data sources are in use
- Need to use dynamic typing (Typeable) to implement the store of pending requests and the cache.

```haskell
class (Typeable1 req, Hashable1 req, Eq1 req) => DataSource req where ...

class Eq1 req where
eq1 :: req a -> req a -> Bool
```

unfortunate that we need this
Data Source State

Core keeps track of the state for each data source
Passes it to fetch

```
class (Typeable1 req, Hashable1 req, Eq1 req)
  => DataSource req where

  data DataState req

  fetch :: DataState req
    -> [BlockedFetch req]
    -> IO ()
```

The state for a data source is an associated datatype
• Clean data source abstraction

• Means that we can plug in any set of data sources at *runtime*
  • e.g. mock data sources for testing
  • core code can be built & tested independently
Replayability

• The Haxl monad and the type system give us:
  • Guarantee of no side effects, except via dataFetch
  • Guarantee that everything is cached
  • The ability to replay requests...
user code
user code → Haxl Core
• The data sources change over time
• But if we *persist the cache*, we can re-run the user code and get the same results
• Great for
  • testing
  • fault diagnosis
  • profiling
Scaling

• Each server has lots of cores, pounded by requests from other boxes constantly.
Hot code swapping

- 1-2K machines, new code pushed many times per day
- Use GHC’s built-in linker
  - Had to modify it to unload code
  - GC detects when it is safe to release old code
- We can swap in new code while requests are still running on the old code
Haskell Extensions?
Haskell Extensions?

• Other people on the team are Haskell beginners
Haskell Extensions?

• Other people on the team are Haskell beginners
• Goal: keep the code simple, use few extensions
Haskell Extensions?

- Other people on the team are Haskell beginners
- Goal: keep the code simple, use few extensions

```haskell
{-# LANGUAGE ExistentialQuantification,
    TypeFamilies,
    GADTs,
    RankNTypes,
    ScopedTypeVariables,
    DeriveDataTypeable,
    StandaloneDeriving,
    MultiParamTypeClasses,
    FlexibleInstances
#-}
```
Haskell Extensions?

• Other people on the team are Haskell beginners
• Goal: keep the code simple, use few extensions

{-# LANGUAGE ExistentialQuantification, TypeFamilies, GADTs, RankNTypes, ScopedTypeVariables, DeriveDataTypeable, StandaloneDeriving, MultiParamTypeClasses, FlexibleInstances #-}

• But we’re doing ok.
Status

• Prototyped most features (including hot code swapping & scaling)
• Core is written
• We have a few data sources, more in the works
• Busy hooking it up to the infrastructure
• Can play around with the system in GHCi, including data sources
Key points

• Haxl Monad:
  • Implicit concurrency with Applicative
  • Automatic batching of data fetches

• Haskell’s type system guarantees the user can’t break the rules
  • ... and gives us guaranteed replayability

• Clean separation of core from datasources

• Scaling & hot-code swapping
Questions?