The Haxl Project at Facebook

Simon Marlow
Jon Coens
Louis Brandy
Jon Purdy
& others
Business Logic service API

Databases

Other back-end services
Use case: fighting spam

www (PHP) ➔ Business Logic ➔ Is this thing spam? ➔ YES/NO ➔ Databases ➔ Other back-end services
Use case: fighting spam

Site-integrity engineers push new rules hundreds of times per day.
Data dependencies in a computation

- database
- thrift
- memcache
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
• 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 programing 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

\[ \text{length (intersect (friendsOf x) (friendsOf y))} \]

- \((\text{friendsOf x})\) 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:

```text
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:

```plaintext
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:

```haskell
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`?

```
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 fetches
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

```
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?
Start with a concurrency monad

```haskell
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)
```
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 $
    \text{case } \text{unHaxl } m \text{ of }
    \text{Done } a \rightarrow \text{unHaxl } (k \ a)
    \text{Blocked } r \rightarrow \text{Blocked } (r >>= k)
• 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
newtype Haxl a = Haxl { unHaxl :: Responses -> Result a }

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

instance Monad Haxl where
  return a = Haxl $ \_ -> Done a
  Haxl m >>= k = Haxl $ \resps ->
    case m resps of
      Done a -> unHaxl (k a) resps
      Blocked reqs r -> Blocked reqs (r >>= k)

addRequest :: Request a -> Requests -> Requests
emptyRequests :: Requests

fetchResponse :: Request a -> Responses -> a

dataFetch :: Request a -> Haxl a
dataFetch req = Haxl $ \_ ->
  Blocked (addRequest req emptyRequests) $ Haxl $ \resps ->
  Done (fetchResponse req resps)
• 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 $ \resps ->
        case f resps of
            Done f' ->
                case a resps of
                    Done a'       -> Done (f' a')
                    Blocked reqs a'       -> Blocked reqs (f' <$> a')
            Blocked reqs f' ->
                case a resps of
                    Done a'       -> Blocked reqs (f' <*> return a')
                    Blocked reqs' a'     -> Blocked (reqs <> reqs') (f' <*> a')
```

\( <*> :: \text{Applicative } f => f (a \to b) \to f a \to f b \)
• 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.

\[
\text{do}
\begin{align*}
fs & \leftarrow \text{friendsOf } x \\
\text{if } \text{simon `elem` } fs & \\
\text{then } \ldots & \\
\text{else } \ldots
\end{align*}
\]

• 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
  • provide APIs that batch correctly
  • translate do-notation into Applicative where possible
    • (forthcoming GHC extension)
• We really want bulk operations to benefit from batching.

```haskell
friendsOfFriends id =
  concat <$> (mapM friendsOf =<< friendsOf id)
```

• 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`

• Will be fixed once `Applicative` is a superclass of `Monad`
Implementation

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

- Experience
- Status etc.
Data Source Abstraction

• We want to structure the system like this:

  ![Diagram]

  - 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

```haskell
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.
• Clean data source abstraction

• Means that we can plug in any set of data sources at runtime
  • e.g. mock data sources for testing and experimentation
  • 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...
• 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
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
Questions?