Bunfight at the language corral

Laziness, strong types, and library design

Bryan O'Sullivan

Lean startups for old farts

- Approximate definition of a "lean startup"?
  - *Achieve the impossible on a negligible budget*
- An immersive way to spend a few years of your early 20s
- But what if you're a bit older, have an established life, and still feel that itch?
  - *(Assumption: you want to sustain some sort of life)*

Architectures of least surprise

- The last thing you want during a hard evening building cushion forts with the kids?
  - *TYPE: Cannot access database on mysql.example.com HOST:appl3 STATE:CRITICAL*
- A small company has proportionally *more* need of robust services than a large one...

  ... because it's always *you* in harm's way

Making choices

- Want a reliable, low-maintenance architecture?
  - *This has pervasive consequences, from the ground up*

Reliable storage

- We have the typical love/hate relationship with MySQL
- On the one hand, it's fast and pretty reliable
- On the other, it's complicated to keep it both healthy and fast in a somewhat unreliable shared hosting environment
  - *Write master fails?*
  - *Read slaves fall behind?*
  - *EBS RAID volume vanishes?*
  - *You've got a long evening ahead*

Avoiding single points of failure

- There aren't many fault tolerant DB-like systems out there at a startup's budget
- Main contenders?
  - *Cassandra*
  - *Voldemort*
  - *Riak*
- All three provide distributed, fault tolerant key/value storage
- If something goes wrong, there's a fair chance you can deal with it the next morning

Why choose Riak?

- Compared to Cassandra and Voldemort, Riak is dramatically easier to deal with on several fronts:
  - *Initial learning curve*
  - *Casual programming (`curl` ftw!)*
- Once you get deeper, there are fewer truly significant differences, but the ease of that initial getting-started period is /important/

The next layer up the stack

- Not surprisingly, our "business logic" is in Haskell
- It's fast, compact, safe, easy to deploy, and has great libraries
- These are very practical concerns, *not* rooted in some abstract philosophy of purity
    - *but those supposedly abstract notions have some lovely concrete benefits*

Your toddler is right: sharing is hard

- One tricky aspect of data storage:
  - *Concurrent updates to shared data*
- The traditional database approach:
  - *Wrap it in a transaction*
- The distributed key/value store approach:
  - *Use vector clocks to signal conflicts*

Transactions

- Think of transactions as like fire sprinklers
- If you remember to use them, you get a high degree of automatic protection
- Under load, things can get distinctly soggy

Vector clocks

- A vector clock tells *you* *only* that something is inconsistent in your data
- Think of it as like a fire alarm
- It's still *your* job to put the fire out!

Everything sucks

- With transactions:
  - *The performance model*
- With vector clocks:
  - *The programming model*

Easing the pain

- So if we're choosing Riak, and we're clever, we must have some sort of trick up our sleeves, right?
- Well, sort of

The Haskell Riak client library

- A layered library, built by MailRank engineering (aka me)
- At the lowest level, hammer away at the bare bones of Riak's APIs
- At the highest level, write conflict resolution code once, and have it work automatically forever after

Performance is important

- Performance is a big deal to us
  - *Riak's HTTP API is pleasant, but slow*
  - *It provides a much faster protocol buffers API*
- The Haskell client library uses protocol buffers

Yep, performance is important

- The library is built around request pipelining
- We've tested with thousands of requests in flight at once, with responses being received while requests are still being sent

Correctness is also important

- The high-level APIs automatically perform conflict resolution during reads and writes
- Conflict resolution is also pipelined, so in a truly demanding application, you get to resolve many conflicts quickly, concurrently, and completely automatically

What *is* a conflict, anyway?

- If you read the vector clock literature, you'll find out about partial orderings, semilattices, and other algebraic terms that aren't very enlightening (maybe unless you're a Haskeller)
- Basically: when I say the thing named *foo* has value X, and you say it has value Y, we have to come to an agreement about what its value really is

Conflict resolution in Haskell

```
class (Eq a, Show a) => Resolvable a where
    resolve :: a -> a -> a
```

- What's this mean?
- There's a class *a* of types for which I can call a `resolve` function
- Given two conflicting values, `resolve` tells me what the "real" value should be

The typical salesman's example

- The easiest example of conflict resolution is with a set of values
  - *I think the set contains* `(1,2,3)`
  - *You say it contains* `(2,4,6)`
- An obvious way to resolve our conflict is to choose the *union* of our two sets
  - `(1,2,3,4,6)`
- We can easily express this in Haskell:

```
instance Resolvable (Set a) where
    resolve = union
```

So...a salesman's example?

- Conflict resolution looks easy, right?
- If you pay attention, you'll find that people who talk about vector clocks always choose something that looks like a set as their example (e.g. a shopping cart)
- Why is this? Shouldn't something like incrementing an integer be even easier?
  - *Nope—it's actually far harder*
- Conflict resolution is, in general, *really* tough
  - *The folks who want to sell you on distributed key/value stores are rarely as quick as they should be to admit this*

Operational concerns

- The overall story is indeed mixed
- Yes, programming with distributed key/value stores is awkward
- The ability to lose part of a cluster without it being a disaster is a big deal
- Performance and convenience are *far* lower than with MySQL

What aspects of Haskell help us?

- Performance
  - *We can ship many tens of MB/sec of JSON around with a single CPU*
- Purity
  - *The design of Haskell libraries makes it far easier to build important features like pipelining and connection pooling*
- Static and dynamic assurance
  - *Whole classes of bugs are eliminated*
  - *Features such as automatic conflict resolution are made safe and (relatively) easy*
  - *We can refactor quickly and with confidence*

Static assurance: the type system

- Encode knowledge about our data that is enforced by a theorem prover
  - *"This data was provided by a user and cannot be trusted"*
  - *"This function never performs I/O of any kind"*
- Specify behaviour that we care about in easily reproduced ways
  - *"for any type in the* `Resolvable` *class, I always know how to resolve conflicts"*

Dynamic assurance: testability

- The QuickCheck library, combined with the type system, makes testing Haskell code a breeze

- Here's how I generate arbitrary compressed data for testing with Google's Snappy compression library:

```
instance Arbitrary (Compressed B.ByteString) where
    arbitrary = (Compressed . B.compress) <$> arbitrary
```

- And here's how I ensure that a scatter-gather I/O vector of little chunks always gets decompressed to the same result as a single buffer containing the same compressed data, no matter what size the chunks are or what the data is:

```
decompress_eq n (Compressed bs) =
    L.fromChunks [B.decompress bs] == L.decompress (rechunk n bs)
```

Get the code, get hacking

- If you want to try this stuff out, get the easy-to-install Haskell Platform: hackage.haskell.org/platform
- All the code I've talked about is open source and installable with a single command
- Riak client
    - *cabal install riak*
- Fast, easy JSON support (60MB/sec+)
    - *cabal install aeson*
- Protocol buffers
    - *cabal install protocol-buffers*
- Snappy (compress at 250 MB/sec, decompress at 500 MB/sec)
    - *cabal install snappy*

Thanks!

- Want these slides? bitbucket.org/bos/erlang-factory-2011
- I ramble on Twitter: @bos31337