Haskell ❤️ Business Rules

Modeling your stuff with glorious, chewy, types

Business Rules

- What fields are linked?
- What values are optional (important, etc.)?
- What constraints are there?
- Domain logic?

Other things

- Have to communicate with stake holders/domain experts
  - Cucumber?
- Express ourselves clearly
  - conversations
  - but also code

Represent

```
data Contact = Contact
    { firstName     :: String
    , middleInitial :: String
    , lastName      :: String
    , emailAddress  :: String
    , isVerified    :: Bool
    } deriving (Show, Eq)
```

Represent

```
data Contact = Contact
    { firstName     :: String
    , middleInitial :: String
    , lastName      :: String
    , emailAddress  :: String
    , isVerified    :: Bool
    } deriving (Show, Eq)
```

I'm going to omit this later

Well...

These are really linked...

```
data Contact = Contact
    { firstName     :: String
    , middleInitial :: String
    , lastName      :: String

    , emailAddress  :: String
    , isVerified    :: Bool
    }
```

Well...

```
data Contact = Contact
    { firstName     :: String
    , middleInitial :: String
    , lastName      :: String

    , emailAddress  :: String
    , isVerified    :: Bool
    }
```

As are these

There are lots of constraints lurking

- Teasing these out can be real work!
- Communicating with the domain expert is essential
- BUT! FP has great answers for many situations

Factor things out

```
data PersonalName = PersonalName
    { firstName     :: String
    , middleInitial :: String
    , lastName      :: String
    }

data Contact = Contact
    { name          :: PersonalName
    , emailAddress :: String
    , isVerified   :: Bool
    }
```

This is now separate

Business Rule++

You don't have to have a middle name

```
data PersonalName = PersonalName
    { firstName     :: String
    , middleInitial :: Maybe String
    , lastName      :: String
    }

data Contact = Contact
    { name          :: PersonalName
    , emailAddress :: String
    , isVerified   :: Bool
    }
```

"We got some empty names"

```
data PersonalName = PersonalName
    { firstName     :: String
    , middleInitial :: Maybe String
    , lastName      :: String
    }

data Contact = Contact
    { name          :: PersonalName
    , emailAddress  :: String
    , isVerified    :: Bool
    }
```

Do we really want ANY string?

Smart constructors

A do-nothing wrapper

```
newtype NonEmptyStr = NES String

mkNonEmptyString :: String
                 -> Maybe NonEmptyStr
mkNonEmptyString s
    | length s > 0 = Just (NES s)
    | otherwise    = Nothing
```

The smart constructor guards creation of our type

Names: yes!

```
data PersonalName = PersonalName
    { firstName     :: NonEmptyStr
    , middleInitial :: Maybe String
    , lastName      :: NonEmptyStr
    }

data Contact = Contact
    { name          :: PersonalName
    , emailAddress :: String
    , isVerified   :: Bool
    }
```

Note lack of "Maybe" at this point, we've already checked

"We got some bad email addresses"

```
data PersonalName = PersonalName
    { firstName     :: NonEmptyStr
    , middleInitial :: Maybe String
    , lastName      :: NonEmptyStr
    }

data Contact = Contact
    { name          :: PersonalName
    , emailAddress :: String
    , isVerified    :: Bool
    }
```

We can do better

Make illegal states unrepresentable

```
newtype Email = Email String

mkEmail :: String -> Maybe Email
mkEmail s = do
    match <- s =~~ ".*@example.com$"
    return (Email match)
```

More smart constructors! this would be the only function exported for creating 'Email's

Email: yes!

```
data PersonalName = PersonalName
    { firstName     :: NonEmptyStr
    , middleInitial :: Maybe String
    , lastName      :: NonEmptyStr
    }

data Contact = Contact
    { name          :: PersonalName
    , emailAddress  :: Email
    , isVerified    :: Bool
    }
```

We now know
this is an Email

Business Rule++

```
data PersonalName = PersonalName
    { firstName     :: NonEmptyStr
    , middleInitial :: Maybe String
    , lastName      :: NonEmptyStr
    }

data Contact = Contact
    { name          :: PersonalName
    , phone         :: String
    , emailAddress  :: Email
    , isVerified    :: Bool
    }
```

Phone plz.

"Well they don't **have** to have a phone, but we want to contact them"

```
data PersonalName = PersonalName
    { firstName     :: NonEmptyStr
    , middleInitial :: Maybe String
    , lastName      :: NonEmptyStr
    }

data Contact = Contact
    { name          :: PersonalName
    , phone         :: Maybe String
    , emailAddress  :: Email
    , isVerified    :: Bool
    }
```

Phone plz?

"...and we'd prefer a phone number, but having both is OK"

```
data PersonalName = PersonalName
    { firstName     :: String
    , middleInitial :: Maybe String
    , lastName      :: String
    }

data Contact = Contact
    { name          :: PersonalName
    , phone         :: Maybe String
    , emailAddress  :: Maybe Email
    , isVerified    :: Bool
    }
```

Hmm, this won't work

And this could be fixed up

Algebraic Data Types

and a smart constructor...

```
newtype Phone = Phone String
```

3 ways to construct a value of type *ContactInfo*

```
data ContactInfo =
        EmailOnly Email
      | PhoneOnly Phone
      | EmailAndPhone Email Phone
```

Make Email more coherent

```
newtype VerifiedEmail =
    VerifiedEmail Email
```

Wrap *Email* in a type that captures the *isVerified* constraint

```
verifyEmail :: Email -> VerifiedEmail

data ContactInfo =
      EmailOnly VerifiedEmail
    | PhoneOnly Phone
    | EmailPhone VerifiedEmail Phone
```

Only way to get a *VerifiedEmail*

Use it!

Much better

```
data PersonalName = ...

data ContactInfo =
       EmailOnly VerifiedEmail
     | PhoneOnly Phone
     | EmailPhone VerifiedEmail Phone

data Contact = Contact
    { name          :: PersonalName
    , contact       :: ContactInfo
    }
```

Same as before

"Not all contacts are of the same reliability..."

```
data PersonalName = ...

data ContactInfo =
       EmailOnly VerifiedEmail
     | PhoneOnly Phone
     | EmailPhone VerifiedEmail Phone

data Contact = Contact
    { name          :: PersonalName
    , contact       :: ContactInfo
    }
```

Want a 0 to 5 rating

"Not all contacts are of the same reliability..."

smart constr.

```
mkStars :: InBounds n => n -> Stars n
mkStars _ = Stars

addStar :: InBounds (S n) => Stars n
        -> Stars (S n)
addStar _ = Stars

removeStar :: InBounds n => Stars (S n)
           -> Stars n
removeStar _ = Stars

data Stars n = Stars
```

All the action is at the type level. There are no useful values around!

"n" is a *phantom type*

Stars: yes!

```
data PersonalName = ...
data ContactInfo = ...

data Contact = Contact
    { name    :: PersonalName
    , contact :: ContactInfo
    , rating  :: Stars Zero
    }
```

Must start at zero

*[Some type-level hackery omitted]

- We can enforce 0 to 5 stars at *compile* time
- This isof questionable "worth-it-ness"
  - But 100% cool
- Actually *can't* be tested!
  - Any code testing an illegal out-of-bounds condition *won't compile* (as intended)

won't *compile?*

This test is actually happening in the type system

```
main = hspec $ do
  describe "Stars" $ do
    it "Adding stars works" $ do
      addStar (mkStars s4) `shouldBe` (mkStars s5)

    -- it "Can't compile this test" $ do
      -- addStar (mkStars s5) `shouldBe` undefined
```

This failing test *can't even be written*

Review: went from this...

```
data Contact = Contact
    { firstName     :: String
    , middleInitial :: String
    , lastName      :: String
    , emailAddress  :: String
    , isVerified    :: Bool
    } deriving (Show, Eq)
```

Review: to this.*

```
data PersonalName = PersonalName
    { firstName     :: NonEmptyStr
    , middleInitial :: Maybe String
    , lastName      :: NonEmptyStr
    }

data ContactInfo =
      EmailOnly VerifiedEmail
    | PhoneOnly Phone
    | EmailAndPhone VerifiedEmail Phone

data Contact = Contact
    { name        :: PersonalName
    , contactInfo :: ContactInfo
    , rating      :: Stars Zero
    }
```

This could reasonably be shown to a domain expert

*(smart constructors & etc. omitted)

Review

- Overall: 180 lines
  - ~20 lines of type definitions
  - ~40 lines of smart constructors
  - ~40 lines of custom show functions (can be automatically written for you)
  - ~50 lines of comments/whitespace
  - ~30 lines gratuitous type hackery (this was just for fun)

Review

- Gained a lot of clarity into the business domain
  - We can talk very precisely with the client using *concrete* and *specific* data types
- Refactorable, testable, and intention-revealing
- Unvalidated data is *excluded* from the application before it ever enters
  - If we have a *Contact* we know it's good (been validated)

Thanks!

- Based **heavily** on: Domain Driven Design, F# and Types
  - Okay, it's basically just a translation of the above (excellent) talk into Haskell
- Unit testing isn't enough
- http://www.haskell.org/haskellwiki/Smart_constructors
- http://okmij.org/ftp/Haskell/eliminating-array-bound-check.lhs (<- not for the faint of heart)

Me ❤️

- Chris Wilson
- @twopoint718
- chris@bendyworks.com