Introduction to Opaleye

Jack Kelly

July 25, 2017

Opaleye

- Typesafe wrapper around `postgresql-simple`
- That means your `INSERT`s are safe
- . . . and your `UPDATE`s
- . . . and your `DELETE`s
- . . . and your queries

Data Structures - First Cut

```
data Post
  = Post
  { postId :: Int
  , postAuthorId :: Int
  , postTitle :: Text
  , postBody :: Text
  }
```

Data Structures - Second Cut

```
data PostPoly id author title body
  = Post
  { _id :: id
  , _author :: author
  , _title :: title
  , _body :: body
  }

type Post = PostPoly Int Int Text Text
type NewPost = PostPoly () Int Text Text
```

Data Structures - Describing Tables

- Opaleye gives us `Column PGFoo` types
- We can describe the underlying table with these

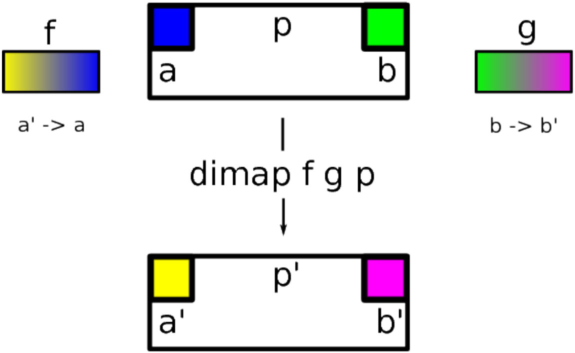
```
data PostPoly id author title body
  = Post { ... }

type PostW = PostPoly
  (Maybe (Column PGInt4))
  (Column PGInt4)
  (Column PGText)
  (Column PGText)

type PostR = PostPoly
  (Column PGInt4)
  (Column PGInt4)
  (Column PGText)
  (Column PGText)
```

Table Mapping

- We have cool types.
- Q: How do we map between haskell and postgres?
- A: Product Profunctors.

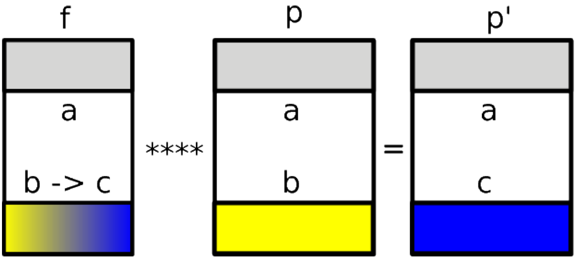
Profunctors

```
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
  lmap :: (a -> b) -> p b c -> p a c
  rmap :: (b -> c) -> p a b -> p a c
```



Product Profunctors

```
class ProductProfunctor p where
  purePP :: b -> p a b
  (****) :: p a (b -> c) -> p a b -> p a c
```



Default Product Profunctors

```
class Default p a b where
  def :: p a b

instance (Default p a c, Default p b d)
 => Default p (a, b) (c, d)
-- And many, many others
```

- Meaning: "There's a canonical way to get a `p` from `a` to `b`"
- There are a bunch of profunctor-agnostic instances that build on each other
- Two important ones in opaleye:
  - `Constant`: Map from haskell values to postgres values
  - `QueryRunner`: Map from postgres values to haskell values

makeAdaptorAndInstance

```
import Data.Profunctor.Product.TH

data PostPoly id author title body
  = Post { ... }

$(makeAdaptorAndInstance "pPost" ''PostPoly)
```

- Defines a product profunctor “adaptor”...

```
pPost :: PostPoly (p a0 b0) (p a1 b1)
                  (p a2 b2) (p a3 b3)
      -> p (PostPoly a0 a1 a2 a3)
           (PostPoly b0 b1 b2 b3)
```

makeAdaptorAndInstance

```
import Data.Profunctor.Product.TH

data PostPoly id author title body
  = Post { ... }

$(makeAdaptorAndInstance "pPost" ''PostPoly)
```

- ... and a Default instance

```
instance (ProductProfunctor p,
          Default p a0 b0,
          Default p a1 b1,
          Default p a2 b2,
          Default p a3 b3)
  => Default p (PostPoly a0 a1 a2 a3)
               (PostPoly b0 b1 b2 b3)
```

Mapping the DB Table

```haskell
Table :: String
      -> TableProperties w r
      -> Table w r

required :: String
         -> TableProperties
            (Column a) (Column a)

optional :: String
         -> TableProperties
            (Maybe (Column a)) (Column a)

postTable :: Table PostW PostR
postTable = Table "post" . pPost $ Post
  (optional "id")
  (required "author_id")
  (required "title")
  (required "body")
```

INSERT

```
conn :: Connection
runInsert :: Connection
          -> Table w r
          -> [w]
          -> IO Int64

runInsert conn postTable
  [constant $ Post 3 4 "Title" "Lorem..."]

constant :: Default Constant h c => h -> c
```

DELETE

```
runDelete :: Connection
          -> Table x r
          -> (r -> Column PGBool)
          -> IO Int64

runDelete conn postTable $
   \p -> _id p .== constant (3 :: Int32)

(.==) :: Column a -> Column a -> Column PGBool
```

UPDATE

```
runUpdate :: Connection
          -> Table w r
          -> (r -> w)
          -> (r -> Column PGBool)
          -> IO Int64

runUpdate conn postTable retitle $
   \p -> _id p .== constant (4 :: Int32)

retitle p = p { _id = Just (_id p)
              , _title = constant "Renamed!"
              }
```

SELECT

- `SELECT` shouldn't be too bad, right?
- Sadly, no.
- Let's talk about arrows.

Arrows

```
class Category a where
  id :: a b b
  (.) :: a c d -> a b c -> a b d

class Category a => Arrow a where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  -- Some equivalent operations omitted
```

Opaleye's Query Functions

```haskell
newtype QueryArr a b = -- ...
type Query b = QueryArr () b

runQuery :: Default QueryRunner c h
         => Connection
         -> Query c
         -> IO [h]
```

Querying with Arrow Expressions

```haskell
{-# LANGUAGE Arrows #-}

import Control.Arrow (returnA)

postById :: Int -> IO [Post]
postById postId = runQuery conn $ proc () -> do
    post <- queryTable postTable -< ()
    restrict -< _id post .== constant postId
    returnA -< post
```

What Rocks?

- Agnostic to DB structure
- Type-safe queries:
  - `INSERT`
  - `UPDATE`
  - `DELETE`
  - `SELECT` (simple ones)
- Composable queries!
- Drop back to SQL when you need to

What Sucks?

- No `UPSERT`
- Hard to call stored procedures
- Lots of boilerplate
- Easy to return too many results
- `LEFT JOIN` is painful
- No native transaction support

What Next?

- You ask questions!