Composable, type-safe SQL generation in Haskell

HaskellerZ Meetup

Zürich, May 19th 2016

Carlos D.

Agenda

1. Introduction
2. What this talk is about
3. Sample application and design
4. Current state of things
5. Haskell Relational Record (HRR)
6. Opaleye
7. Closing notes. Considerations.

Introduction

About me

I'm Carlos.

I've been paid to write Java, Scala and PHP.

I've been kindly asked to write Ruby, Python, Clojure, Objective-C, others...

I'm in (newfound) love with Haskell.

What this talk is about

Results oriented

We're concerned with results rather than performance.

Not about ORMs

Generating queries is not about mapping data types to database rows, we also have a need for aggregation, counting, reports...

If you're looking for an ORM, try Persistent.

PostgreSQL

The commonly-supported RDMBS between the libraries we'll be seeing is PostgreSQL. Another one would be sqlite*, which is a simpler database engine.

* Although to use sqlite we'd have to download an alternative version of Opaleye.

What this talk is about

What does it mean to be *composable*?

Abstraction, projection, and special operations as means of taking simple, well-tested queries to provide arbitrarily complex ones.

What does it mean to be *type-safe*?

Having pertinent data structures and type information for our queries to provide compile-time safety to our programs.

What is a *valid query*?

"It depends". Both HRR and Opaleye will help us avoid making trivial mistakes, but neither is perfect.

Sample application and design

We'll be looking at how to make an application that will allow me to add TODOs, read them, and complete or discard them.

Sample output

We want to see all our TODOs

```
$ todos list
1. Buy food             (due by: 2016-05-20) (priority: 5 )
2. Call parents         (due by: 2016-05-20) (priority: 7 )
3. Wash clothes         (due by: 2016-05-23) (priority: - )
4. Finish presentation (due by: 2016-05-19) (priority: 10)
5. Call boss            (due by: 2016-05-18) (priority: 20)
```

We want to find specific ones

```
$ todos find 1
Todo:     Buy food
Due by:   2016-05-20
Priority: 5
Hashtags: #independence #responsible
```

Sample application and design

Sample output

We want to complete a TODO

```
$ todos complete 4
   Completed 'Finish presentation #haskellerz'
```

Of course we also want to add a TODO

```
$ todos add 'Display presentation at Haskellerz!' \
            --due-by   '2016-05-19'               \
            --hashtags '#haskellerz, #fun'        \
            --priority 10
Added TODO with id 6
```

So far this is a basic CR~~U~~D. But what about more complex queries? No software is complete without its options. Plus, from `$ todos find 1` we saw that our app is aware of the concept of a "hashtag", as well as due dates.

Sample application and design

Sample output

List TODOs that are due by a certain date.

```
$ todos list --due-by '2016-05-20' --with-hashtags
1. Buy food     (due by: 2016-05-20) (priority: 5) #independence #responsible
2. Call parents (due by: 2016-05-20) (priority: 7) #good-son
```

List TODOs that are due on a certain date, and belong to a certain hashtag.

```
$ todos list --due-by '2015-05-20' --order-by-priority --hashtag "#responsible"
1. Buy food     (due by: 2015-05-20) (priority: 5) #independence #responsible
```

List TODOs that are already late...

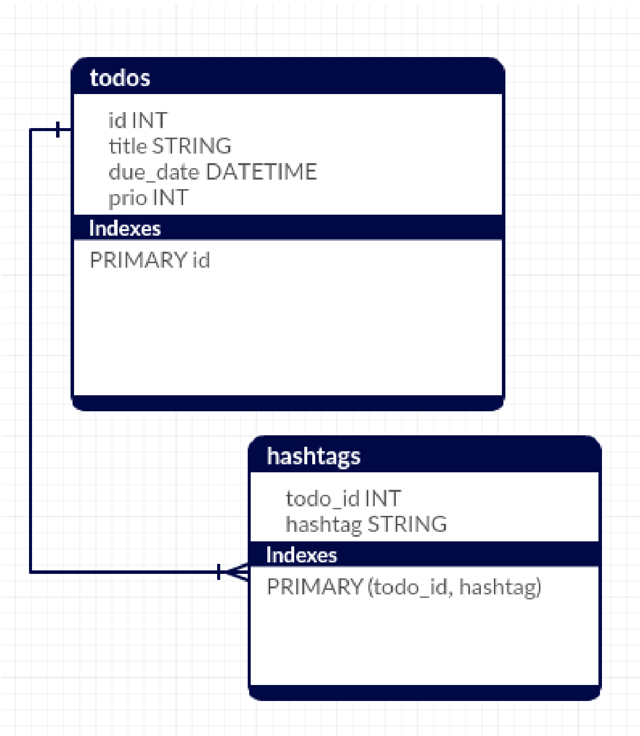
```
$ todos list --late --with-hashtags
5. Call boss (due by: 18 May 2016) (priority: 20) #good-employee
```

Finally, get some reports

```
$ todos report
# Reports of late todos, todos without hashtags, with multiple hashtags...
```

Sample application and design

Database design

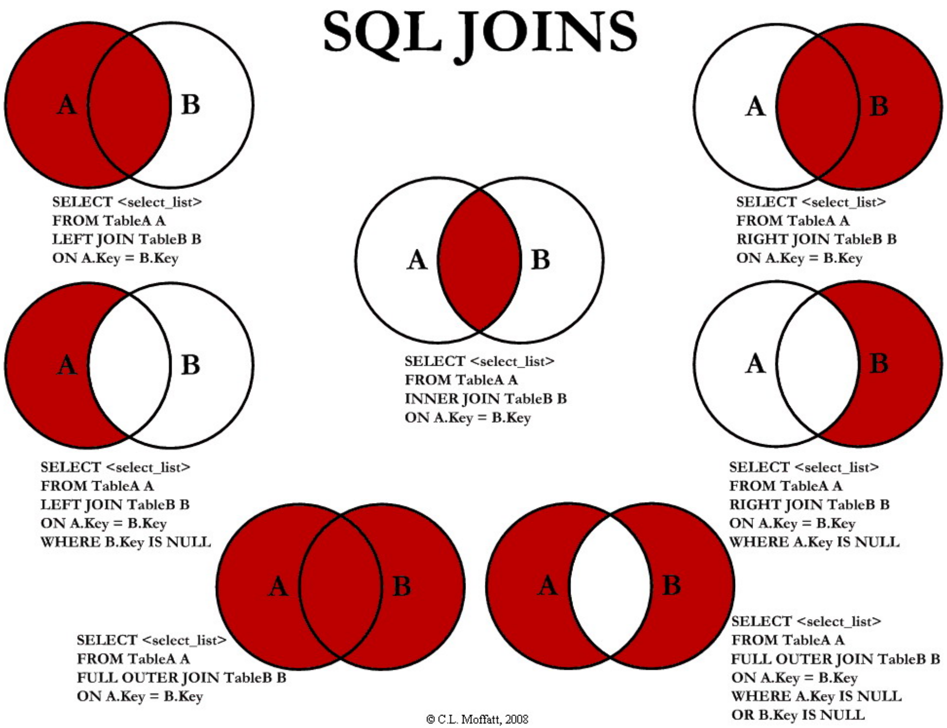



```sql
create table todos(
  id serial primary key,
  title varchar(50) not null,
  due_date date not null,
  prio int
)

create table hashtags(
  hashtag varchar(50) not null,
  todo_id int not null,
  primary key (hashtag, todo_id)
)
```

Sample application and design

Quick refresher on JOINS



Sample application and design

(Some of the) queries we'll need

To see all our TODOs

```
select * from todos
```

To find a specific TODO

```
select * from todos where id = ?
```

To complete a TODO

```
delete from todos where id = ?
```

To add a TODO

```
insert into todos (title, due_date, prority) values (?, ?, ?)
```

Sample application and design

(Some of the) queries we'll need

List TODOs that are due by a specific date

```
select * from todos where due_date = ?
```

List TODOs ordered by priority

```
select * from todos order by prio
```

List late TODOs

```
select * from todos where due_date < current_date
```

List TODOs due on a specific date and belonging to a certain hashtag

```
select t.* from hashtags h
join todos t on h.todo_id = t.id
where h.hashtag  = ?
and   t.due_date = ?
```

Sample application and design

In short, we'll need plenty of queries to match all the possible combinations of options that I can accept.

- I can compose options
- Can I compose my queries?

Current state of things

Bryan O'Sullivan and Leon Smith's `postgresql-simple`

I can design my types to my liking

```
-- file simple/src/Simple/Todo.hs
data Todo = Todo { getId       :: !(Maybe Int) -- Can be null
                 , getTitle    :: !String      -- Title of the todo
                 , getDueDate  :: !Date        -- Date of the todo
                 , getPrio     :: !(Maybe Int) -- Priority of the todo
                 } deriving (Show)

-- file simple/src/Simple/Hashtag.hs
data Hashtag = Hashtag { getTodoId  :: !Int    -- Cannot be null
                       , getHashtag :: !String -- Hashtag string val
                       } deriving (Show, Eq)
```

Current state of things

Define how each row maps to the declared data

```
-- file simple/src/Simple/Todo.hs
instance FromRow Todo where
    fromRow = Todo <$> field -- id
                   <*> field -- title
                   <*> field -- due date
                   <*> field -- prio

instance ToRow Todo where
    toRow t = [ toField . getTitle $ t
              , toField . getDueDate $ t
              , toField . getPrio $ t
              ]

-- file simple/src/Simple/Hashtag.hs
instance FromRow Hashtag where
    fromRow = Hashtag <$> field -- the todo id
                      <*> field -- the hashtag string

instance ToRow Hashtag where
    toRow h = [ toField . getTodoId $ h
              , toField . getHashtag $ h
              ]
```

Current state of things

Sample methods

```haskell
-- file simple/src/Simple/Todo.hs
addTodo :: Connection -> Todo -> IO (Only Int)
addTodo conn t
    = head <$> query conn q t
      where
        q = [sql| insert into todos (title, due_date, prio) values (?, ?, ?)
                  returning id |]

-- file simple/src/Simple/Commands.hs
runAddCommand :: Connection -> String -> [Flag] -> IO ()
runAddCommand c desc flags = do
    let priority = prioFromFlags flags
    let dueDate  = either error id (dueDateFromFlags flags)
    let todo = T.Todo { T.getId      = Nothing
                      , T.getTitle   = desc
                      , T.getDueDate = dueDate
                      , T.getPrio    = priority
                      }

    Only tid <- T.addTodo c todo
    putStrLn (unwords ["Added todo", show tid])
```

Current state of things

Sample methods

```
-- file src/Simple/Todo.hs
allTodos :: Connection -> IO [Todo]
allTodos conn = query_ conn q
                where
                  q = [sql| select id, title, due_date, prio
                            from todos |]

-- Same query, with 1 restriction
findTodo :: Connection -> Int -> IO (Maybe Todo)
findTodo conn tid = do
               result <- query conn q (Only tid)
               return $ if length result == 1 then
                    Just (head result)
               else
                    Nothing
               where
                 q = [sql| select id, title, due_date, prio from todos
                           where id = ? |]
```

Current state of things

Sample methods

```haskell
-- file simple/src/Simple/Todo.hs
deleteTodo :: Connection -> Int -> IO Int64
deleteTodo conn tid = execute conn q (Only tid)
                      where
                        q = [sql| delete from todos where id = ? |]

-- file simple/src/Simple/Commands.hs
runCompleteCommand :: Connection -> Int -> IO ()
runCompleteCommand c tid = do
    maybeTodo <- T.findTodo c tid
    affected  <- T.deleteTodo c tid

    if isNothing maybeTodo then
        hPutStrLn stderr "Todo not found!" >> exitWith (ExitFailure 1)
    else if affected > 0 then
        putStrLn $ unwords
            [ "   Completed"
            , T.getTitle (fromJust maybeTodo)
            ]
    else
        hPutStrLn stderr "Error: Something went wrong!"
     >> exitWith (ExitFailure 1)
```

Current state of things

More queries

```
-- Same query again, this time sorted in a specific way
allTodosByPrio :: Connection -> IO [Todo]
allTodosByPrio conn = query_ conn q
                      where
                        q = [sql| select id, title, due_date, prio from todos
                                  order by prio desc
                                  nulls last |]

-- Same query as before, this time with another restriction
allLateTodos :: Connection -> IO [Todo]
allLateTodos conn = query_ conn q
                    where
                      q = [sql| select id, title, due_date, prio from todos
                                where due_date < current_date |]
```

What's the problem?

Current state of things

Although `postgresql-simple` is amazing, let's look at a few issues we wish we could address:

Problems

- I'm repeating myself constantly.
- My queries are almost always the same, varying in by restriction, or order specification.
- "QuasiQuoted" strings are hard to concatenate... but do we really want ease of concatenation? **NO!** `sql` -QuasiQuotes are hard to concatenate on purpose!
- My `FromRow` and `ToRow` instances will be constantly in need of updates as the requirements change.

Current state of things

Problems (cont.)

- `field` functions provide a convenient row parser, however the result is not very semantically-meaningful.
- I could write a query in a quasiquote and not know that anything is wrong until I run it.
- I have no guarantee that I'm typing a type-safe query (i.e. The `getDate` from `Todo` could be a `String`, and my program would compile, but it would not crash until I run it, since the field in the table is defined as a `date`. This is also a problem when refactoring my database).
- It's very easy to have a compiling program that can have innocent and/or trivial mistakes that make it crash.

Current state of things

Case in point:

```
-- file simple/src/Simple/BadTodo.hs
data BadTodo = BadTodo !(Maybe Int) -- Can be null
                       !String      -- Title of the todo
                       !String      -- Date of the todo, this time as a String
                       !(Maybe Int) -- Priority of the todo
                       deriving (Show)

let getBadTodos = do {
    c <- connect defaultConnectionInfo
    badTodos <- allBadTodos -- Todos with a date type as String
}

getBadTodos

*** Exception: Incompatible {
    errSQLType = "date",
    errSQLTableOid = Just (Oid 16401),
    errSQLField = "due_date",
    errHaskellType = "Text",
    errMessage = "types incompatible"
}
```

Haskell Relational Record (HRR)

- Developed Kei Hibino and other good people at Asahi Net, Inc
- Developed as a response to issues found using HaskellDB
- Runs on top of `HDBC`

```
...
executable haskellerz-sqlgen-hrr
  ghc-options:          -Wall -Werror -fsimpl-tick-factor=500
  main-is:              Main.hs
  build-depends:        base                   == 4.8.*
                      , relational-record      == 0.1.3.*
                      , relational-query       >= 0.3
                      , relational-query-HDBC  == 0.6.*
                      , persistable-record     == 0.4.0.*
                      , HDBC                   == 2.4.0.1
                      , HDBC-postgresql        == 2.3.2.*
                      , HDBC-session           == 0.1.0.*
                      , template-haskell       == 2.10.*
                      , time                   == 1.5.*
                      , bytestring             == 0.10.6.*
```

Haskell Relational Record (HRR)

My new definition for `Todo`

```
-- file hrr/src/HRR/DataSource.hs
-- ... imports ...
connect' :: IO Connection
connect' = connectPostgreSQL "dbname=postgres"

defineTable :: String -> -- ^ Schema name
               String -> -- ^ Table name
               [Name] -> -- ^ Derives
               Q [Dec]   -- ^ Quoted result
defineTable = defineTableFromDB connect' driverPostgreSQL
```

```
-- file hrr/src/HRR/Todo.hs
{-# LANGUAGE TemplateHaskell #-}

import HRR.DataSource

$(defineTable "public" "todo" [''Show])
```

Haskell Relational Record (HRR)

What happened?

- HRR will use TH to generate our record type
- A database connection will be needed on compilation time
- Our record will have the fields of the table in CamelCase, with the database types mapped to Haskell types by default according to Database.Relational.Schema.PostgreSQL.
- I did have to define the `defineTable` method, see file `hrr/src/HRR/DataSource.hs`, with PostgreSQL-specific functions.
- Some "queries" will be generated for me already, namely: `todo`, `insertTodo`, `selectTodo`, `updateTodo`, ...
- I also have my `Pi`s! Or the indexes of my tables

Haskell Relational Record (HRR)

I now "have"

```
ghci> :t Todo
Todo :: Int32          -- ^ The Todo id    | id      :: Todo -> Int32
     -> Day            -- ^ The date       | dueDate :: Todo -> Day
     -> String         -- ^ The title      | title   :: Todo -> String
     -> Maybe Int32    -- ^ The priority   | prio    :: Todo -> Maybe Int32
     -> Todo

ghci> :t todo
todo :: Relation () Todo

ghci> show todo
"SELECT id, due_date, title, prio FROM PUBLIC.todo"

ghci> :t selectTodo
selectTodo :: Query Int32 Todo

ghci> :t updateTodo
updateTodo :: KeyUpdate Int32 Todo

ghci> :t insertTodo
insertTodo :: Insert Todo
```

Haskell Relational Record (HRR)

About those `Pi` s

```
ghci> :t id'
id' :: Pi Todo Int32

ghci> :t dueDate'
dueDate' :: Pi Todo Day

ghci> :t prio'
prio' :: Pi Todo (Maybe Int32)

ghci> :t title'
title' :: Pi Todo String
```

`Pi` s serve to describe that the type of they key is `r1` for record type `r0`.

In this context, they are simply an index of an array with *phantom types*.

```
id' :: Pi Todo Int32
id' = Database.Relational.Query.Pi.Unsafe.definePi
        (columnOffsetsTodo array-0.5.1.0:Data.Array.Base.! 0)
```

Haskell Relational Record (HRR)

Sample methods

```
-- file hrr/src/HRR/Commands.hs
import qualified HRR.Todo as T

-- | Important part
runFindCommand :: (IConnection conn) => conn -> Int32 -> [Flag] -> IO ()
runFindCommand conn x flags =
        runQPrint conn x T.selectTodo (printTodo conn)
```

```
-- | Helper functions to print out records
printTodo :: (IConnection conn) => conn -> T.Todo -> IO ()
printTodo = ...

runQPrint :: (Show a, IConnection conn, FromSql SqlValue a, ToSql SqlValue p)
          => conn -> p -> Query p a -> (a -> IO ()) ->IO ()
runQPrint = ..
```

Adding and deleting can get a bit more complex...

Haskell Relational Record (HRR)

Sample methods

Problem: My new `Todo`'s `id` is an `Int32`, not a `Maybe Int` like we had before. And `insertTodo` doesn't care that my `id` is `serial`.

Solution: Create a "partial" `Todo` and my own `insert` for it.

```
-- file hrr/src/HRR/Todo.hs
data PiTodo = PiTodo { piTodoTitle :: String
                     , piTodoDate  :: Day
                     , piTodoPrio  :: Maybe Int32
                     }

$(makeRecordPersistableDefault ''PiTodo) -- Create the `Pi`s

-- Creates a "bindable" placeholder for insertions
piTodo' :: Pi Todo PiTodo
piTodo' = PiTodo |$| title'
                 |*| dueDate'
                 |*| prio'
```

Haskell Relational Record (HRR)

Sample methods

```haskell
-- file hrr/src/HRR/Commands.hs
insertTodo :: InsertQuery T.PiTodo -- An insert query to be bound by a partial
insertTodo = derivedInsertQuery T.piTodo' . relation' $
    placeholder $ \ph ->
        return $ T.PiTodo |$| ph ! T.piTodoTitle'
                          |*| ph ! T.piTodoDate'
                          |*| ph ! T.piTodoPrio'

runAddCommand :: (IConnection conn) => conn -> String -> [Flag] -> IO ()
runAddCommand conn title flags = do
    let prio    = prioFromFlags flags
    let dueDate = dueDateFromFlags flags

    -- I can build now my partial record
    let piToInsert = T.PiTodo { T.piTodoTitle = title
                              , T.piTodoDate  = dueDate
                              , T.piTodoPrio  = prio
                              }

    runInsertQuery conn insertTodo piToInsert
```

This is arguably a bit cumbersome to do...

Haskell Relational Record (HRR)

Sample methods

Alternative...

```
-- file hrr/src/HRR/Commands.hs
insertTodoAlt :: Insert ((String, Day), Maybe Int32)
insertTodoAlt = derivedInsertValue $ do -- uses "Assignings" Monad
    (phTitle, ())   <- placeholder (\ph -> T.title'   <-# ph)
    (phDueDate, ()) <- placeholder (\ph -> T.dueDate' <-# ph)
    (phPrio, ())    <- placeholder (\ph -> T.prio'    <-# ph)
    return (phTitle >< phDueDate >< phPrio)

runAlternativeAddCommand :: (IConnection conn)
                         => conn -> String -> [Flag] -> IO ()
runAlternativeAddCommand conn title flags = do
    let dueDate  = dueDateFromFlags flags
    let priority = prioFromFlags flags
    runInsert conn insertTodoAlt ((title, dueDate), priority)
```

Haskell Relational Record (HRR)

Sample methods

```
--- file hrr/src/HRR/Commands.hs
deleteTodo :: Delete Int32
deleteTodo = derivedDelete $ \projection ->
                fst <$> placeholder
                    (\ph -> wheres $ projection ! T.id' .=. ph)

runCompleteCommand :: (IConnection conn) => conn -> Int32 -> IO ()
runCompleteCommand conn x = do
    todos <- runQuery conn T.selectTodo x
    i     <- runDelete conn deleteTodo x

    if i > 0 then
        putStrLn ("Compeleted todo " ++ T.title (head todos))
    else
        putStrLn ("Could not complete todo " ++ show x)

    commit conn
```

Haskell Relational Record (HRR)

Simple queries

```
import qualified HRR.Todo as T

todosByPriority :: Relation () T.Todo
todosByPriority = relation $ do -- QuerySimple Monad
    t <- query T.todo
    desc $ t ! T.prio'
    return t
```

Produces

```
SELECT ALL T0.id AS f0, T0.due_date AS f1, T0.title AS f2, T0.prio AS f3
FROM PUBLIC.todo T0
ORDER BY T0.prio DESC
```

```
gchi> :t query
query :: MonadQuery m => Relation () r -> m (Projection Flat r)
ghci> :t relation
relation :: QuerySimple (Projection Flat r) -> Relation () r
```

Haskell Relational Record (HRR)

Simple queries

```
import qualified H.Hashtag as H

hashtagsForTodo :: Relation Int32 H.Hashtag
hashtagsForTodo = relation' . placeholder $ \ph -> do
    hashtags <- query H.hashtag
    wheres $ hashtags ! H.todoId' .=. ph
    return hashtags
```

Produces

```
SELECT ALL T0.todo_id AS f0, T0.hashtag_str AS f1
FROM PUBLIC.hashtag T0
WHERE (T0.todo_id = ?)
```

Haskell Relational Record (HRR)

Simple queries

```
import Data.Time.Calendar      (Day)
import qualified HRR.Todo as T

todosByPriorityAndBeforeDate :: Relation Day T.Todo
todosByPriorityAndBeforeDate = relation' . placeholder $ \ph -> do
    t <- query todosByPriority
    wheres $ t ! T.dueDate' .<=. ph
    return t
```

Produces

```
SELECT ALL T1.f0 AS f0, T1.f1 AS f1, T1.f2 AS f2, T1.f3 AS f3
FROM (SELECT ALL
      T0.id AS f0, T0.due_date AS f1, T0.title AS f2, T0.prio AS f3
      FROM PUBLIC.todo T0 ORDER BY T0.prio DESC) T1
WHERE (T1.f1 <= ?)
```

HRR defines interfaces between Haskell pure values and query projection values in Database.Relational.Query.Pure

Haskell Relational Record (HRR)

Simple queries

```
import Data.Time.Calendar      (Day)
import qualified HRR.Todo as T

todoIdAndTitleByPriorityAndBeforeDate :: Relation Day (Int32, String)
todoIdAndTitleByPriorityAndBeforeDate = relation' . placeholder $ \ph $ do
    (ph, t) <- query' todosByPriorityAndBeforeDate
    return (ph, t ! T.id' >< t ! T.title')
```

`><` Operator constructs pair results. Same as `(,) |$| x |*| y`. Provided functions `fst'` and `snd'`.

Produces

```
SELECT ALL T2.f0 AS f0, T2.f2 AS f1
FROM (SELECT ALL
      T1.f0 AS f0, T1.f1 AS f1, T1.f2 AS f2, T1.f3 AS f3
      FROM (SELECT ALL
            T0.id AS f0, T0.due_date AS f1, T0.title AS f2, T0.prio AS f3
            FROM PUBLIC.todo T0 ORDER BY T0.prio DESC) T1
WHERE (T1.f1 <= ?)) T2
```

Fun fact: In college I had a hard time with my naming skills

Haskell Relational Record (HRR)

Simple queries

We're now not only repeating less code. We're also rationalizing about our queries by the types that we're binding and the types we're returning.

Haskell Relational Record (HRR)

Composing queries

```
-- file hrr/src/HRR/Reports.hs

-- | The product of all todos with their hashtags
todosAndHashtags :: Relation () (T.Todo, Maybe H.Hashtag)
todosAndHashtags = relation $ do
    t <- query T.todo
    h <- queryMaybe H.hashtag
    on $ just (t ! T.id') .=. h ?! H.todoId'
    return $ t >< h

-- | Filter out the todos with hashtags and keep those only without
todosWithoutHashtags :: Relation () T.Todo
todosWithoutHashtags = relation $ do
    t <- query todosAndHashtags
    let todo = t ! fst'
    let maybeHashtag = t ! snd'
    wheres $ isNothing (maybeHashtag ?! H.todoId')
    return $ T.Todo |$| todo ! T.id'
                    |*| todo ! T.title'
                    |*| todo ! T.dueDate'
                    |*| todo ! T.prio'
```

Haskell Relational Record (HRR)

Composing queries

```
ghci> import HRR.Reports as R
ghci> show R.todosAndHashtags

SELECT ALL
    T0.id AS f0, T0.title AS f1, T0.due_date AS f2, T0.prio AS f3,
    T1.hashtag_str AS f4, T1.todo_id AS f5
FROM PUBLIC.todo T0
LEFT JOIN PUBLIC.hashtag T1
ON (T0.id = T1.todo_id)

ghci> show R.todosWithoutHashtags

SELECT ALL T2.f0 AS f0, T2.f1 AS f1, T2.f2 AS f2, T2.f3 AS f3
FROM (SELECT ALL
          T0.id AS f0, T0.title AS f1, T0.due_date AS f2,
          T0.prio AS f3, T1.hashtag_str AS f4, T1.todo_id AS f5
      FROM PUBLIC.todo T0
      LEFT JOIN PUBLIC.hashtag T1
      ON (T0.id = T1.todo_id)) T2
WHERE (T2.f5 IS NULL)
```

Haskell Relational Record (HRR)

Aggregation

```
-- file hrr/src/HRR/Reports.hs
countLateTodos :: Relation Day Int
countLateTodos = aggregateRelation' . placeholder $ \ph -> do -- QueryAggregate
    t <- query T.todo
    wheres $ t ! T.dueDate' .<=. ph
    return $ count (t ! T.id')

countFutureTodos :: Relation Day Int
countFutureTodos = aggregateRelation' . placeholder $ \ph -> do
    t <- query T.todo
    wheres $ t ! T.dueDate' .>. ph
    return $ count (t ! T.id')

-- Produces
--
-- Future:                            | Late:
-- SELECT ALL COUNT(T0.id) AS f0      | SELECT ALL COUNT(T0.id) AS f0
-- FROM PUBLIC.todo T0                | FROM PUBLIC.todo T0
-- WHERE (T0.due_date > ?)            | WHERE (T0.due_date <= ?)
```

Haskell Relational Record (HRR)

Aggregation

```
-- file hrr/src/HRR/Reports.hs
todosMultipleHashtags :: Relation () Int32
todosMultipleHashtags = aggregateRelation $ do
    t <- query T.todo
    h <- query H.hashtag
    on $ t ! T.id' .=. h ! H.todoId'
    g <- groupBy $ t ! T.id'
    having $ count (h ! H.hashtagStr') .>. value (1 :: Int)
    return g
```

Produces

```
SELECT ALL T0.id AS f0 FROM PUBLIC.todo T0
INNER JOIN PUBLIC.hashtag T1
ON (T0.id = T1.todo_id)
GROUP BY T0.id
HAVING (COUNT(T1.hashtag_str) > 1)
```

Haskell Relational Record (HRR)

Aggregation

```
mostPopularHashtags :: Relation () (String, Int32)
mostPopularHashtags = aggregateRelation $ do
    h <- query H.hashtag
    g <- groupBy $ h ! H.hashtagStr'
    having $ count (h ! H.hashtagStr') .>. value (1 :: Int)
    return $ g >< count (h ! H.hashtagStr')
```

Produces

```
SELECT ALL T0.hashtag_str AS f0, COUNT(T0.hashtag_str) AS f1
FROM PUBLIC.hashtag T0
GROUP BY T0.hashtag_str
HAVING (COUNT(T0.hashtag_str) > 1)
```

```
ghci> :t aggregateRelation
aggregateRelation
  :: QueryAggregate (Projection Aggregated r) -> Relation () r
```

Haskell Relational Record (HRR)

Accumulates various context in a State Monad context (like join product, group keys and ordering.)

Important data structures

- `Pi p a`
- `Relation p a`
- `Projection c a`
- `Query`, `InsertQuery`, `Update`...

Important Operators

- `!`, `?!`
- `><`
- `|$|`, `|*|`
- `.<.`, `.>.`, `.>=.`, `.<=.`, `.=.`

Important Functions

- `query`
- `wheres`
- `on`
- `in`
- `or`
- `asc`, `desc`
- `groupBy`
- `having`

Opaleye

Developed by

- Tom Ellis - *Cambridge, UK*

Significant contributions by folks using it in production.

Runs on top of `postgresql-simple`

```
executable haskellerz-sqlgen-opaleye
  main-is:              Main.hs
  ghc-options:          -Wall -Werror
  build-depends:        base                 == 4.8.*
                      , mtl                  >= 2.2
                      , opaleye              >= 0.4.1
                      , product-profunctors
                      , semigroups
                      , text
                      , time
                      , postgresql-simple
```

Opaleye

My new definition for `Todo`

```haskell
-- file opaleye/src/OpaleyeDemo/Todo.hs
{-# LANGUAGE Arrows #-}

import Data.Profunctor.Product.TH (makeAdaptorAndInstance)

data Todo' i t d p = Todo { _id      :: i
                          , _title   :: t
                          , _dueDate :: d
                          , _prio    :: p
                          } deriving Show

makeAdaptorAndInstance "pTodo" ''Todo'

type TodoColumns
  = Todo' TodoIdColumn (Column PGText) (Column PGDate) PrioColumn
type TodoInsertColumns
  = Todo' TodoIdColumnMaybe (Column PGText) (Column PGDate) PrioColumn
type Todo
  = Todo' TodoId String Day Prio
```

Polymorphic records are important!

See all column types: Opaleye.PGTypes

Opaleye

My new definition for `Todo`

```haskell
-- file opaleye/src/OpaleyeDemo/Todo.hs

todoTable :: Table TodoInsertColumns TodoColumns
todoTable = Table "todos" $ pTodo Todo
    { _id      = pTodoId . TodoId $ optional "id"
    , _title   = required "title"
    , _dueDate = required "due_date"
    , _prio    = pPrio . Prio $ required "prio"
    }

todoQuery :: Query TodoColumns
todoQuery = queryTable todoTable
```

`Query TodoColumns` is an alias of `QueryArr () TodoColumns`

Opaleye

Easier to use with specific types...

... In order to make sure that we're never joining incoherently

```
-- file opaleye/src/OpaleyeDemo/Ids.hs

data TodoId' a = TodoId { todoId :: a } deriving Show

makeAdaptorAndInstance "pTodoId" ''TodoId'

type TodoId = TodoId' Int
type TodoIdColumn = TodoId' (Column PGInt4)
type TodoIdColumnMaybe = TodoId' (Maybe (Column PGInt4))
type TodoIdColumnNullable = TodoId' (Column (Nullable PGInt4))
```

Product profunctors allow Opaleye to make transformations when running the query against the database.

Opaleye

Simple reads, inserts, deletes

```
-- file opaleye/src/Todo.hs

insertTodo :: Connection -> TodoInsertColumns -> IO TodoId
insertTodo conn t = fmap head (runInsertReturning conn todoTable t _id)

deleteTodo :: Connection -> TodoId -> IO Int64
deleteTodo conn tid = runDelete conn todoTable
                      (\t -> todoId (_id t) .=== (pgInt4 . todoId) tid)
```

Opaleye

Simple reads, inserts, deletes

```
-- file opaleye/src/Todo.hs
selectTodo :: TodoId -> Query TodoColumns
selectTodo tid = proc () -> do
    todos    <- todoQuery -< ()
    restrict -< todoId (_id todos) .== pgInt4 (todoId tid)
    returnA  -< todos
```

```
SELECT "id0_1" as "result1_2",
       "title1_1" as "result2_2",
       "due_date2_1" as "result3_2",
       "prio3_1" as "result4_2"
FROM (SELECT *
      FROM (SELECT "id" as "id0_1",
                   "title" as "title1_1",
                   "due_date" as "due_date2_1",
                   "prio" as "prio3_1"
            FROM "todos" as "T1") as "T1"
WHERE (("id0_1") = 1)) as "T1"
```

Arrows instead of Monads!

Opaleye

Of course, you don't "have" to

```
selectTodo' :: TodoId -> Query TodoColumns
selectTodo' tid = todoQuery >>> keepWhen
                  (\todos -> todoId (_id todos) .== pgInt4 (todoId tid))
```

```
ghci> printSql (selectTodo' $ TodoId { todoId = 1 })
```

```
SELECT "id0_1" as "result1_2",
       "title1_1" as "result2_2",
       "due_date2_1" as "result3_2",
       "prio3_1" as "result4_2"
FROM (SELECT *
      FROM (SELECT "id" as "id0_1",
                   "title" as "title1_1",
                   "due_date" as "due_date2_1",
                   "prio" as "prio3_1"
            FROM "todos" as "T1") as "T1"
WHERE (("id0_1") = 1)) as "T1"
```

Same output, no arrow syntax.

Opaleye

Profunctors and Product Profunctors

First-class representations of transformations

```
ghci> import Data.Profunctor.Product
ghci> :t p1
p1 :: ProductProfunctor p => p a1 b1 -> p a1 b1
ghci> p1 (+2) (2 :: Int)
4
ghci> :t p2
p2 :: ProductProfunctor p => (p a1 b1, p a2 b2) -> p (a1, a2) (b1, b2)
ghci> p2 ((+1), (*2)) (1 :: Int, 2 :: Int)
(2,4)
```

We derived `pTodo` before. Same principle applies.

Opaleye uses Product Profunctors extensively. Namely:

- `TableDefinition`: Constructs table definitions.
- `QueryRunner`: Turn a `Query` into `haskells`.
- `Aggregator`: Applies an aggregator to the result of a query.
- etc.

Opaleye

Composing queries

```
-- file opaleye/src/Todo.hs
todoQuery :: Query TodoColumns
todoQuery = queryTable todoTable

todosByPriority :: Query TodoColumns
todosByPriority = orderBy (descNullsLast (prio . _prio)) todoQuery
```

```
SELECT "id0_1" as "result1_2",
       "title1_1" as "result2_2",
       "due_date2_1" as "result3_2",
       "prio3_1" as "result4_2"
FROM (SELECT *
      FROM (SELECT *
            FROM (SELECT *
                  FROM (SELECT "id" as "id0_1",
                               "title" as "title1_1",
                               "due_date" as "due_date2_1,
                               "prio" as "prio3_1"
                        FROM "todos" as "T1") as "T1") as "T1"
            ORDER BY "prio3_1" DESC NULLS LAST) as "T1") as "T1"
```

Opaleye

Composing queries

```
-- file opaleye/src/OpaleyeDemo/Reports.hs
todosAndHashtags :: Query (T.TodoColumns, H.HashtagNullableColumns)
todosAndHashtags = leftJoin T.todoQuery H.hashtagQuery eqTodoId
    where eqTodoId (todos, hashtags) = T._id todos .=== H._todoId hashtags

todosWithoutHashtags :: Query T.TodoColumns
todosWithoutHashtags = proc () -> do
    (todos, hashtags) <- todosAndHashtags -< ()
    restrict -< isNull ((I.todoId . H._todoId) hashtags)
    returnA -< todos
```

```
--- file opaleye/src/OpaleyeDemo/Hashtag.hs
type HashtagNullableColumns
  = Hashtag' TodoIdColumnNullable HashtagStrColumnNullable
```

```
ghci> :t leftJoin
:: (..) => Opaleye.Query columnsA
        -> Opaleye.Query columnsB
        -> ((columnsA, columnsB) -> Column PGBool)
        -> Opaleye.Query (columnsA, nullableColumnsB)
```

Opaleye

Composing queries

Compound columns

```
-- file opaleye/src/Reports.hs
todosOpDate
    :: (Column PGDate -> Column PGDate -> Column PGBool)
    -> QueryArr Day T.TodoColumns
todosOpDate op = proc day -> do
    todos <- T.todoQuery -< ()
    restrict -< T._dueDate todos `op` pgDay day
    returnA -< todos

lateTodos :: Day -> Query T.TodoColumns
lateTodos day = proc () -> do
    todos <- todosOpDate (.<=) -< day
    returnA -< todos

futureTodos :: Day -> Query T.TodoColumns
futureTodos day = proc () -> do
    todos <- todosOpDate (.>) -< day
    returnA -< todos
```

Opaleye

Aggregation

```
-- file opaleye/src/Reports.hs
countLateTodos :: Day -> Query (Column PGInt8)
countLateTodos d = aggregate count . fmap (I.todoId . T._id) $ lateTodos d

countFutureTodos :: Day -> Query (Column PGInt8)
countFutureTodos d = aggregate count . fmap (I.todoId . T._id) $ futureTodos d
```

```
SELECT "result0_2" as "result1_3"
FROM (SELECT *
      FROM (SELECT COUNT("id0_1") as "result0_2"
            FROM (SELECT *
                  FROM (SELECT "id" as "id0_1",
                               "title" as "title1_1",
                               "due_date" as "due_date2_1",
                               "prio" as "prio3_1"
                        FROM "todos" as "T1") as "T1"
                  WHERE (("due_date2_1") <= (CAST('2016-05-19' AS date)))) as "T1"
            GROUP BY COALESCE(0)) as "T1") as "T1"
```

Opaleye

`GROUP BY` - `HAVING` as an inner join

```
-- file opaleye/src/Reports.hs
todoIdsWithHashtagAmount :: Query (Column PGInt4, Column PGInt8)
todoIdsWithHashtagAmount
  = aggregate (p2 (groupBy, count))
  $ proc () -> do
        hashtags <- H.hashtagQuery -< ()
        returnA -< ( (I.todoId . H._todoId) hashtags        -- group by
                   , (I.hashtagStr . H._hashtag) hashtags)  -- count


todosMultipleHashtags :: Query T.TodoColumns
todosMultipleHashtags = proc () -> do
    todos         <- T.todoQuery -< ()
    (tid, hcount) <- todoIdsWithHashtagAmount -< ()
    restrict -< (I.todoId . T._id) todos .== tid
    restrict -< hcount .> pgInt8 1                          -- with restriction
    returnA -< todos
```

Opaleye

`GROUP BY` - `HAVING` as an inner join (bis)

```
-- file opaleye/src/Reports.hs

hashtagsCounted :: Query (Column PGText, Column PGInt8)
hashtagsCounted
  = aggregate (p2 (groupBy, count))
  $ proc () -> do
        hashtags <- H.hashtagQuery -< ()
        returnA -< ( (I.hashtagStr . H._hashtag) hashtags
                   , (I.todoId . H._todoId) hashtags)

mostPopularHashtags :: Query (Column PGText)
mostPopularHashtags = proc () -> do
    (hashtag, hcount) <- hashtagsCounted -< ()
    restrict -< hcount .>= pgInt8 2
    returnA -< hashtag
```

Opaleye

Tying the knot

```
import Opaleye

runQuery
  :: Data.Profunctor.Product.Default.Default
       QueryRunner columns haskells =>
     Connection -> Opaleye.Query columns -> IO [haskells]
```

This means that we can run `Query columns`, and so long as there's a `Product Profunctor` adaptor that can transform our `columns` into `haskells`, we'll be able to retrieve (transform) the information.

If there's no such instance, our program won't compile.

More often than not, I was doing `runQuery conn myQuery :: IO [haskells]`

`QueryArr Input Output` is not runnable, as they're in need of input.

Opaleye

Further usable functionality

- Pagination
- Search

Typesafe SQL in Haskell - Ben Kolera - BFPG 2015-09

Closing Notes & Considerations

HRR

- Monadic approach is not mandatory, you can do HRR with Arrows.
- Limited to either records or (nested) pairs.
- Currently there's no 100% way to prove the validity of HRR conversions.

Opaleye

- Schema changes still require code changes.
- No placeholders.
- Varchar lengths are not contemplated.

Closing Notes & Considerations

Note on HaskellDB

As expressed by HRR creators, HaskellDB has a number of drawbacks.

Mainly:

- Limited expression ability of projections.
- Does not provide support for (left, right, or full) outer joins
- Column name conflicts.
- Partial support for placeholders.
- Unclear aggregation semantics.

Example: https://github.com/m4dc4p/haskelldb/issues/22

Closing Notes & Considerations

Reading material

- Oliver Charles on `postgresql-simple` at ZuriHac 2015
- HRR on Haskell Hackathon, December 2012
- Experience Report on HRR
- Extensive examples on HRR
- Reddit thread with opinions on Opaleye and HRR - With creators' comments
- Ben Kolera from Brisbane Functional Programming Group on Opaleye
- Renzo Carbonara on `opaleye-sot`
- Tom Ellis on Arrows
- Trivially generating an invalid query in HRR
- Bugs in Opaleye