Learn to Rel8

Teo Camarasu (CircuitHub)

Sep 17 2026

London Haskell Meetup

https://informal.codes/talks/london1

Rel8 generates SQL

- PostgreSQL
- Natural language and SHOUTING.
- Difficult to factor code, limited datatypes.
- We want a strong type system, eg, migrations.

Rel8 generates SQL

- PostgreSQL
- Natural language and SHOUTING.
- Difficult to factor code, limited datatypes.
- We want a strong type system, eg, migrations.
- We'd rather be writing Haskell

Rel8

- Write SQL by writing (almost) idiomatic elementary Haskell.
- Friendly type errors, and protected from magic.
- Talk by Ollie Charlies from ZuriHac 2021.

Rel8

- `rel8` written by my colleagues: Ollie and Shane
- `rel8` builds on `opaleye` (Tom Ellis, et al.)
- `opaleye` builds on the HaskellDB project (Daan Leijen, et al.)
- Similar in spirit to Acadia and to Beam

Elementary Haskell

- Tuples, Records, Sum types
- Basic types like `Double`, `Int64`, `Text`, `UTCTime`, etc
- `List`, `Maybe`, `Either`
- And functions between them!
- Folds, and scans

Introducing Lists

```
pure 1 :: [Int64]

[1, 2, 3, 4] :: [Int64]

[1,2] ++ [3,4]


(,) <$> [1, 2]
    <*> ['A', 'B']
```

```
SELECT 1;

VALUES (1), (2), (3), (4);

VALUES (1), (2) UNION ALL VALUES (3), (4);

SELECT x, y
  FROM (VALUES (1),(2)) x
  JOIN (VALUES ('A'), ('B')) y
  ON true;
```

Lateral join

- `Applicative` = `JOIN`
- `Monad` (`concatMap`) = `LATERAL JOIN`
- Postgres can often optimize this away like `ApplicativeDo`.

Records

```haskell
{-# LANGUAGE DuplicateRecordFields #-}
newtype ProductId = ProductId Int64
  deriving newtype (Eq, Ord)

-- | Something sold in the London Haskell store.
data ProductResult = ProductResult
  { id :: ProductId
  , name :: Text
  , sellable :: Bool
  , cost :: Int64
  }
```

Simple query (WHERE)

```haskell
products :: [ProductResult]
products =
  [ ProductResult (ProductId 0) "t-shirt" True 34
  , ProductResult (ProductId 1) "mascot" True 49
  , ProductResult (ProductId 2) "Monad tutorial" False 0
  ]
sellableProducts :: [ProductResult]
sellableProducts = do
  product <- products
  guard $ product.sellable
  pure product
```

```sql
SELECT * FROM products WHERE sellable;
```

OrderLine

```
newtype OrderId = OrderId Int64
  deriving newtype (Eq)

data OrderLineResult = OrderLineResult
  { id :: OrderId
  , productId :: ProductId
  , quantity :: Int64
  }
```

Simple JOIN

```
orderLines :: [OrderLineResult]
orderLines =
  [ OrderLineResult (OrderId 0) (ProductId 1) 10
  , OrderLineResult (OrderId 0) (ProductId 2) 2
  , OrderLineResult (OrderId 1) (ProductId 2) 1
  ]

orderLinesFromProductId :: ProductId -> [OrderLineResult]
orderLinesFromProductId productId =
  filter (\o -> o.productId == productId) orderLines
```

Simple JOIN

```haskell
orderLinesWithProducts :: [(ProductResult, OrderLineResult)]
orderLinesWithProducts = do
  product <- products
  orderLine <- orderLinesFromProductId product.id
  pure (product, orderLine)
```

```sql
SELECT * FROM products p JOIN order_lines o ON p.id = o.product_id;
```

Maybes in SQL

```haskell
productMaybeOrder :: [(ProductResult, Maybe OrderLineResult)]
productMaybeOrder = do
  product <- products
  let mOrderLine = listToMaybe $ orderLinesFromProductId product.id
  pure (product, mOrderLine)
```

```sql
SELECT p.*, x.*
  FROM products p
  LEFT JOIN LATERAL (SELECT * FROM orders o
                     WHERE o.product_id = p.id
                     LIMIT 1) x
  ON true;
```

Calculating earnings from product

```haskell
ordersGroupedByProduct :: [[(ProductResult, OrderLineResult)]]
ordersGroupedByProduct =
  groupOn ((.id) . fst) $ sortOn ((.id) . fst) $
  orderLinesWithProducts
earningsPerProduct :: [(First ProductId, Sum Int64)]
earningsPerProduct = map (foldMap f) ordersGroupedByProduct
  where
    f (p, o) = (First (Just p.productId), Sum (p.cost * o.quantity) )
productsByEarnings :: [(First ProductId, Sum Int53)]
productsByEarnings = sortOn snd $ earningsPerProduct
```

```sql
SELECT p.id, SUM(p.cost * o.quantity) AS sum
FROM products p JOIN order_lines o
ON p.id = o.product_id GROUP BY p.id ORDER BY sum;
```

Mapping Elementary Haskell to SQL

Elementary Haskell	SQL
`List`	`SELECT`
`Applicative`	`(CROSS) JOIN`
`Monad`	`LATERAL (CROSS) JOIN`
`listToMaybe`	`LEFT JOIN`
`guard/filter`	`WHERE`
`take`	`LIMIT`
`foldMap`	`Aggregate functions`

Learn to Rel8

Basics

```
SELECT 100;
```

```
lit :: Serializable exprs a => a -> exprs
lit @Int64 :: Int64 -> Expr Int64
```

Basics

```
SELECT 100;
```

```
lit :: Serializable exprs a => a -> exprs
lit @Int64 :: Int64 -> Expr Int64

pure (lit @Int64 100) :: Query (Expr Int64)
select $ pure $ lit @Int64 100 :: Rel8.Statement (Query (Expr Int64))
```

Basics

```
SELECT 100;
```

```
lit :: Serializable exprs a => a -> exprs
lit @Int64 :: Int64 -> Expr Int64

pure (lit @Int64 100) :: Query (Expr Int64)
select $ pure $ lit @Int64 100 :: Rel8.Statement (Query (Expr Int64))

run $ select $ pure $ lit @Int64 100 :: Hasql.Statement () [Int64]
run :: Rel8.Statement (Query (Expr Int64))  -> Hasql.Statement () [Int64]
```

Basics

```
SELECT 100;
```

```
lit :: Serializable exprs a => a -> exprs
lit @Int64 :: Int64 -> Expr Int64

pure (lit @Int64 100) :: Query (Expr Int64)
select $ pure $ lit @Int64 100 :: Rel8.Statement (Query (Expr Int64))

run $ select $ pure $ lit @Int64 100 :: Hasql.Statement () [Int64]
run :: Rel8.Statement (Query (Expr Int64))  -> Hasql.Statement () [Int64]

showQuery $ pure $ lit @Int64 100 :: Text
```

Meaning of Query

A `Query`:

- can be evaluated via a SELECT to give a list of row results.
- `Functor`, `Applicative`, `Monad`

Cards

A card is a tuple of a suit and a rank.

```
deck :: Query (Expr Char, Expr Int64)
deck = do
  suit <- values $ map lit ['D', 'H', 'C', 'S']
  rank <- values $ map lit [1..13]
  pure (suit, rank)
```

Face Cards

```
(>.) :: Expr Int64 -> Expr Int64 -> Expr Bool

faceCard :: (Expr Char, Expr Int64) -> Expr Bool
faceCard (_suit, rank) = rank >. lit 10

guard :: Bool -> [()]
where_ :: Expr Bool -> Query ()

faceCards :: Query (Expr Char, Expr Int64)
faceCards = do
  card <- deck
  where_ $ faceCard card
  pure card
```

Red Cards

```
List.filter :: (a -> Bool) -> [a] -> [a]
Rel8.filter :: (a -> Expr Bool) -> a -> Query a

redCards :: Query (Expr Char, Expr Int64)
redCards = do
  card <- deck
  Rel8.filter (\(suit, _) -> lit 'D' ==. suit ||. lit 'H' ==. suit) card
```

Expr

- An `Expr a` is a SQL expression with Haskell type `a`.
- `Expr` does **not** have a `Functor` instance!
- `DBType a` typeclass maps basic Haskell types to Postgres types.
- `DBNum a` is a "lifted" `Num`
- `DBOrd`, `DBEq`, etc.

Id types

```
newtype ProductId = ProductId Int64
  deriving newtype (Eq, DBType, DBEq, DBOrd)
newtype OrderId = OrderId Int64
  deriving newtype (Eq, DBType, DBEq, DBOrd)
data Suit = D | H | S | C
  deriving (Read, Show)
  deriving DBType via ReadShow Suit

lit (ProductId 10) ==. lit (ProductId 20) :: Expr Bool
```

Data types

```haskell
data ProductResult = ProductResult
  { id :: ProductId
  , name :: Text
  , sellable :: Bool
  , cost :: Int64
  }
```

```haskell
data ProductExpr = ProductExpr
  { id :: Expr ProductId
  , name :: Expr Text
  , sellable :: Expr Bool
  , cost :: Expr Int64
  }
```

Naive approach using Higher-kinded Data

```haskell
data Product f = Product
  { id :: f ProductId
  , name :: f Text
  , sellable :: f Bool
  , cost :: f Int64
  }
```

```haskell
data Product Result = Product
  { id :: Result ProductId
  , name :: Result Text
  , sellable :: Result Bool
  , cost :: Result Int64
  }
```

```haskell
data Product Expr = Product
  { id :: Expr ProductId
  , name :: Expr Text
  , sellable :: Expr Bool
  , cost :: Expr Int64
  }
```

Higher-kinded Data and Column

```haskell
data Product f = Product
  { id :: Column f ProductId
  , name :: Column f Text
  , sellable :: Column f Bool
  , cost :: Column f Int64
  }

(x :: Product Expr).name :: Expr Text

(x :: Product Result).name :: Text
```

```haskell
Column Expr a ~ Expr a
Column Result a ~ a

Product Expr ~ ProductExpr
Product Result ~ ProductResult

deriving stock (Generic)
deriving anyclass (Rel8able)
-- or
deriveRel8able ''Product
```

Running a Query

- Running a `Query` of `Expr` gives a `List` of `Result`
- There is non-trivial type family magic here. You don't need to worry!
- `Query (Expr Int, Expr Char)` becomes `[(Int, Char)]`
- `Query (Product Expr)` becomes `[Product Result]`

Defining the schema

```
productSchema :: TableSchema (Product Name)
productSchema = TableSchema
  { name = "products"
  , schema = Nothing
  , columns = Product
     { id = "id"
     , name = "name"
     , sellable = "sellable"
     , cost = "cost"
     }
  }
```

A simple query

```haskell
-- sellableProducts :: [Product Result]
sellableProducts :: Query (Product Expr)
sellableProducts = do
  -- product <- products
  product <- (each productSchema :: Query (Product Expr))
  -- guard $ product.sellable
  where_ $ product.sellable
  pure product
```

```sql
SELECT * FROM products WHERE sellable;
```

Simple JOIN

```haskell
orderLinesSchema :: TableSchema (OrderLine Name)
orderLinesSchema = namesFromLabels

orderLinesFromProductId :: Expr ProductId -> Query (OrderLine Expr)
orderLinesFromProductId productId = do
  orderLine <- each orderLinesSchema
  Rel8.filter (\o -> o.productId ==. productId) orderLine

queryOn :: (DBEq k) =>
  TableSchema (t Name) -> (t Expr -> k) -> k -> Query (t Expr)
queryOn schema f k = each schema >>= Rel8.filter (\x -> f x ==. k)
```

Simple JOIN

```haskell
orderLinesWithProducts :: Query (OrderLine Expr, Product Expr)
orderLinesWithProducts = do
  product <- each productSchema
  orderLine <- orderLinesFromProductId product.id
  pure (orderLine, product)

productMaybeOrder :: Query (Product Expr, MaybeTable Expr (OrderLine Expr))
productMaybeOrder = do
  product <- each productSchema
  mOrderLine <- optional $ orderLinesFromProductId product.id
  pure (product, mOrderLine)
```

`MaybeTable`

- A `MaybeTable` is a `Maybe` lifted into SQL.
- Introduced by `optional :: Query a -> Query (MaybeTable Expr a)`
- Adds an extra tag column.
- Many familiar utility functions, eg, `catMaybe`, `traverseMaybeTable`, etc.
- Others: `NonEmptyTable`, `EitherTable`, `TheseTable`, `NullTable`

Earnings per product

```
data Earnings f = Earnings
  { productId :: Column f ProductId
  , earnings :: Column f Int64
  }
deriveRel8able ''Earnings

earningsPerProduct :: Query (Earnings Expr)
```

Earnings per product

```
orderLinesWithProducts :: Query (OrderLine Expr, Product Expr)
earningsPerProduct :: Query (Earnings Expr)
earningsPerProduct = aggregate1 f orderLinesWithProducts
  where
    f :: Aggregator1 (OrderLine Expr, Product Expr) (Earnings Expr)
    f = Earnings <$> groupByOn ((.id) . snd) <*> sumOn cost

    cost :: (OrderLine Expr, Product Expr) -> Expr Int64
    cost (o, p) = o.quantity * p.cost
```

Aggregator

- Generalizes `foldMap` and `groupBy`
- `Profunctor`, `Applicative`
- Generalies `Fold` from `foldl` package.
- Build up using `Applicative`, access fields using `*On` functions.
- opaleye#631

ListTable

- `ListTable` allows tree shaped data

```
productOrders :: Query (Product Expr, ListTable Expr (OrderLine Expr))
productOrders = do
  product <- each productSchema
  orderLines <- many $ orderLinesFromProductId product.id
  pure (product, orderLines)
```

Not covered

- ADTs and pattern matching
- Window functions

Recap

Elementary Haskell	Rel8
`Int64`	`Expr Int64`
`Ord`, `Eq`, `Num`	`DBOrd`, `DBEq`, `DBNum`
`List`	`Query`, `ListTable`
`Maybe`	`MaybeTable`
`guard`/`filter`	`where_`/`filter`
`sortOn`	`orderOn`
`foldMap(1)`/`groupOn`	`Aggregations`

Future of rel8

- Polymorphic code over pure Haskell and `rel8`
- Compilation times, eg, more TH usage.

Thank you