Rel8

Release 1.0.0

unknown

Jan 20, 2023

GETTING STARTED

Welcome to Rel8! Rel8 is a Haskell library for interacting with PostgreSQL databases, built on top of the fantastic Opaleye library.

The main objectives of Rel8 are:

- *Conciseness*: Users using Rel8 should not need to write boiler-plate code. By using expressive types, we can provide sufficient information for the compiler to infer code whenever possible.
- *Inferrable*: Despite using a lot of type level magic, Rel8 aims to have excellent and predictable type inference.
- *Familiar*: writing Rel8 queries should feel like normal Haskell programming.

CHAPTER

ONE

GETTING STARTED

In this section, we'll take a look at using Rel8 to work with a small database for Haskell packages. We'll take a look at idiomatic usage of Rel8, mapping tables to Haskell, and then look at writing some simple queries.

Before we get started, we'll be using the following language extensions and imports throughout this guide:

```
{-# language BlockArguments #-}
{-# language DeriveAnyClass #-}
{-# language DeriveGeneric #-}
{-# language DerivingStrategies #-}
{-# language DerivingVia #-}
{-# language DuplicateRecordFields #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language OverloadedStrings #-}
{-# language StandaloneDeriving #-}
{-# language TypeApplications #-}
{-# language TypeFamilies #-}

import Prelude
import Rel8
```

1.1 The Example Schema

Before we start writing any Haskell, let's take a look at the schema we'll work with. The *author* table has three columns:

Column Name	Type	Nullable
`author_id`	`integer`	not null
`name`	`text`	not null
`url`	`text`	

and the *project* table has two:

Column Name	Type	Nullable
`author_id`	`integer`	not null
`name`	`text`	not null

A `project` always has an `author`, but not all `author`s have projects. Each `author` has a name and (maybe) an associated website, and each project has a name.

1.2 Mapping Schemas to Haskell

Now that we've seen our schema, we can begin writing a mapping in Rel8. The idiomatic way to map a table is to use a record that is parameterised what Rel8 calls an *interpretation functor*, and to define each field with `Column`. For this type to be usable with Rel8 we need it to be an instance of `Rel8able`, which can be derived with a combination of `DeriveAnyClass` and `DeriveGeneric` language extensions.

Following these steps for `author`, we have:

```
data Author f = Author
  { authorId :: Column f Int64
  , name     :: Column f Text
  , url      :: Column f (Maybe Text)
  }
  deriving stock (Generic)
  deriving anyclass (Rel8able)
```

This is a perfectly reasonable definition, but cautious readers might notice a problem - in particular, with the type of the `authorId` field. While `Int64` is correct, it's not the best type. If we had other identifier types in our project, it would be too easy to accidentally mix them up and create nonsensical joins. As Haskell programmers, we often solve this problem by creating `newtype` wrappers, and we can also use this technique with Rel8:

```
newtype AuthorId = AuthorId { toInt64 :: Int64 }
  deriving newtype (DBEq, DBType, Eq, Show)
```

Now we can write our final schema mapping. First, the `author` table:

```
data Author f = Author
  { authorId   :: Column f AuthorId
  , authorName :: Column f Text
  , authorUrl  :: Column f (Maybe Text)
  }
  deriving stock (Generic)
  deriving anyclass (Rel8able)
```

And similarly, the `project` table:

```
data Project f = Project
  { projectAuthorId :: Column f AuthorId
  , projectName     :: Column f Text
  }
  deriving stock (Generic)
  deriving anyclass (Rel8able)
```

To show query results in this documentation, we'll also need `Show` instances: Unfortunately these definitions look a bit scary, but they are essentially just `deriving (Show)`:

```
deriving stock instance f ~ Result => Show (Author f)
deriving stock instance f ~ Result => Show (Project f)
```

These data types describe the structural mapping of the tables, but we also need to specify a `TableSchema` for each table. A `TableSchema` contains the name of the table and the name of all columns in the table, which will ultimately allow us to `SELECT` and `INSERT` rows for these tables.

To define a `TableSchema`, we just need to fill construct appropriate `TableSchema` values. When it comes to the `tableColumns` field, we construct values of our data types above, and set each field to the name of the column that it maps to.

First, `authorSchema` describes the column names of the `author` table when associated with the `Author` type:

```
authorSchema :: TableSchema (Author Name)
authorSchema = TableSchema
  { name = "author"
  , schema = Nothing
  , columns = Author
      { authorId = "author_id"
      , authorName = "name"
      , authorUrl = "url"
      }
  }
```

And likewise for `project` and `Project`:

```
projectSchema :: TableSchema (Project Name)
projectSchema = TableSchema
  { name = "project"
  , schema = Nothing
  , columns = Project
      { projectAuthorId = "author_id"
      , projectName = "name"
      }
  }
```

There is also some generics machinery available if you want to grab the field information from your `Rel8able` type:

```
projectSchema :: TableSchema (Project Name)
projectSchema = TableSchema
  { name = "project"
  , schema = Nothing
  , columns = namesFromLabels @(Project Name)
  }
```

This will create a `TableSchema` for `Project` where every column name corresponds exactly to the name of the field. If you need more flexibility, you can use `namesFromLabelsWith`, which takes a transformation function.

Note: You might be wondering why this information isn't in the definitions of `Author` and `Project` above. Rel8 decouples `TableSchema` from the data types themselves, as not all tables you define will necessarily have a schema. For example, Rel8 allows you to define helper types to simplify the types of queries - these tables only exist at query time, but there is no corresponding base table. We'll see more on this idea later!

With these table definitions, we can now start writing some queries!

1.3 Writing Queries

1.3.1 Simple Queries

First, we'll take a look at `SELECT` statements - usually the bulk of most database heavy applications.

In Rel8, `SELECT` statements are built using the `Query` monad. You can think of this monad like the ordinary `[]` (List) monad - but this isn't required knowledge.

To start, we'll look at one of the simplest queries possible - a basic `SELECT * FROM` statement. To select all rows from a table, we use `each`, and supply a `TableSchema`. So to select all `project` rows, we can write:

```
>>> :t each projectSchema
each projectSchema :: Query (Project Expr)
```

Notice that `each` gives us a `Query` that yields `Project Expr` rows. To see what this means, let's have a look at a single field of a `Project Expr`:

```
>>> let aProjectExpr = undefined :: Project Expr
>>> :t projectAuthorId aProjectExpr
projectAuthorId aProjectExpr :: Expr AuthorId
```

Recall we defined `projectAuthorId` as `Column f AuthorId`. Now we have `f` is `Expr`, and `Column Expr AuthorId` reduces to `Expr AuthorId`. We'll see more about `Expr` soon, but you can think of `Expr a` as "SQL expressions of type `a`".

To execute this `Query`, we pass it to `select`. This function takes both a database connection (which can be obtained using hasql's `acquire` function), and the `Query` to run:

```
>>> Right conn <- acquire "user=postgres"
>>> :t select conn (each projectSchema)
select conn (each projectSchema) :: MonadIO m => m [Project Result]
```

When we `select` things containing `Expr`s, Rel8 builds a new response table with the `Result` interpretation. This means you'll get back plain Haskell values. Studying `projectAuthorId` again, we have:

```
>>> let aProjectResult = undefined :: Project Result
>>> :t projectAuthorId aProjectResult
projectAuthorId aProjectResult :: AuthorId
```

Here `Column Result AuthorId` reduces to just `AuthorId`, with no wrappping type at all.

Putting this all together, we can run our first query:

```
>>> select conn (each projectSchema) >>= mapM_ print
Project {projectAuthorId = 1, projectName = "rel8"}
Project {projectAuthorId = 2, projectName = "aeson"}
Project {projectAuthorId = 2, projectName = "text"}
```

We now know that `each` is the equivalent of a `SELECT * ` query, but sometimes we're only interested in a subset of the columns of a table. To restrict the returned columns, we can specify a projection by using `Querys Functor` instance:

```
>>> select conn $ projectName <$> each projectSchema
["rel8","aeson","text"]
```

1.3.2 Joins

Another common operation in relational databases is to take the `JOIN` of multiple tables. Rel8 doesn't have a specific join operation, but we can recover the functionality of a join by selecting all rows of two tables, and then using `where_` to filter them.

To see how this works, first let's look at taking the product of two tables. We can do this by simply calling `each` twice, and then returning a tuple of their results:

```
>>> :{
mapM_ print =<< select conn do
  author  <- each authorSchema
```

(continues on next page)

(continued from previous page)

```
  project <- each projectSchema
  return (projectName project, authorName author)
:}
("rel8","Ollie")
("rel8","Bryan O'Sullivan")
("rel8","Emily Pillmore")
("aeson","Ollie")
("aeson","Bryan O'Sullivan")
("aeson","Emily Pillmore")
("text","Ollie")
("text","Bryan O'Sullivan")
("text","Emily Pillmore")
```

This isn't quite right, though, as we have ended up pairing up the wrong projects and authors. To fix this, we can use `where_` to restrict the returned rows. We could write:

```
select conn $ do
  author  <- each authorSchema
  project <- each projectSchema
  where_ $ projectAuthorId project ==. authorId author
  return (project, author)
```

but doing this every time you need a join can obscure the meaning of the query you're writing. A good practice is to introduce specialised functions for the particular joins in your database. In our case, this would be:

```
projectsForAuthor :: Author Expr -> Query (Project Expr)
projectsForAuthor a = each projectSchema >>= filter \p ->
  projectAuthorId p ==. authorId a
```

Our final query is then:

```
>>> :{
mapM_ print =<< select conn do
  author  <- each authorSchema
  project <- projectsForAuthor author
  return (projectName project, authorName author)
:}
("rel8","Ollie")
("aeson","Bryan O'Sullivan")
("text","Bryan O'Sullivan")
```

1.3.3 Left Joins

Rel8 is also capable of performing `LEFT JOIN`s. To perform `LEFT JOIN`s, we follow the same approach as before, but use the `optional` query transformer to allow for the possibility of the join to fail.

In our test database, we can see that there's another author who we haven't seen yet:

```
>>> select conn $ authorName <$> each authorSchema
["Ollie","Bryan O'Sullivan","Emily Pillmore"]
```

Emily wasn't returned in our earlier query because - in our database - she doesn't have any registered projects. We can account for this partiality in our original query by wrapping the `projectsForAuthor` call with `optional`:

```
>>> :{
mapM_ print =<< select conn do
  author   <- each authorSchema
  mproject <- optional $ projectsForAuthor author
  return (authorName author, projectName <$> mproject)
:}
("Ollie",Just "rel8")
("Bryan O'Sullivan",Just "aeson")
("Bryan O'Sullivan",Just "text")
("Emily Pillmore",Nothing)
```

1.3.4 Aggregation

Aggregations are operations like `sum` and `count` - operations that reduce multiple rows to single values. To perform aggregations in Rel8, we can use the `aggregate` function, which takes a `Query` of aggregated expressions, runs the aggregation, and returns aggregated rows.

To start, let's look at a simple aggregation that tells us how many projects exist:

Todo:

```
>>> error "TODO"
```

Rel8 is also capable of aggregating multiple rows into a single row by concatenating all rows as a list. This aggregation allows us to break free of the row-orientated nature of SQL and write queries that return tree-like structures. Earlier we saw an example of returning authors with their projects, but the query didn't do a great job of describing the one-to-many relationship between authors and their projects.

Let's look again at a query that returns authors and their projects, and focus on the /type/ of that query:

```
projectsForAuthor a = each projectSchema >>= filter \p ->
  projectAuthorId p ==. authorId a

let authorsAndProjects = do
      author  <- each authorSchema
      project <- projectsForAuthor author
      return (author, project)
      where

>>> :t select conn authorsAndProjects
select conn authorsAndProjects
  :: MonadIO m => m [(Author Result, Project Result)]
```

Our query gives us a single list of pairs of authors and projects. However, with our domain knowledge of the schema, this isn't a great type - what we'd rather have is a list of pairs of authors and /lists/ of projects. That is, what we'd like is:

```
[(Author Result, [Project Result])]
```

This would be a much better type! Rel8 can produce a query with this type by simply wrapping the call to `projectsForAuthor` with either `some` or `many`. Here we'll use `many`, which allows for the possibility of an author to have no projects:

```
>>> :{
mapM_ print =<< select conn do
  author       <- each authorSchema
  projectNames <- many $ projectName <$> projectsForAuthor author
  return (authorName author, projectNames)
:}
("Ollie",["rel8"])
("Bryan O'Sullivan",["aeson","text"])
("Emily Pillmore",[])
```

CHAPTER

TWO

DBTYPE

The `DBType` type class provides a bridge between database expression values and Haskell values. Rel8 comes with stock instances for most types that come predefined with PostgreSQL, such as *int4* (which is represented as by `Data.Int.Int32`) and timestamptz (which is `UTCTime` from `Data.Time`). This means that you need at least one `DBType` instance per database type, though most of these primitive `DBTypes` should already be available.

2.1 Combining `newtype` and `DBType`

You can define new instances of `DBType` by using Haskell "generalized newtype deriving" strategy. This is useful when you have a common database type, but need to interpret this type differently in different contexts. A very common example here is with auto-incrementing `id` counters. In PostgreSQL, it's common for a table to have a primary key that uses the `serial` type, which means the key is an `int8` with a default value for `INSERT`. In Rel8, we could use `Int64` (as `Int64` is the `DBType` instance for `int8`), but we can be even clearer if we make a `newtype` for each *type* of id.

If our database has users and orders, these tables might both have ids, but they are clearly not meant to be treated as a common type. Instead, we can make these types clearly different by writing:

```
newtype UserId = UserId { getUserId :: Int64 }
  deriving newtype DBType

newtype OrderId = OrderId { getOrderId :: Int64 }
  deriving newtype DBType
```

Now we can use `UserId` and `OrderId` in our Rel8 queries and definitions, and Haskell will make sure we don't accidentally use an `OrderId` when we're looking up a user.

If you would like to use this approach but can't use generalized newtype deriving, the same can be achived by using `mapTypeInformation`:

```
instance DBType UserId where
  typeInformation = mapTypeInformation UserId getUserId typeInformation
```

2.2 Parsing with DBType

DBTypes can also *refine* database types with parsing, which allows you to map more structured Haskell types to a PostgreSQL database. This allows you to use the capabalities of Haskell's rich type system to make it harder to write incorrect queries. For example, we might have a database where we need to store the status of an order. In Haskell, we might write:

```
data OrderStatus = PaymentPending | Paid | Packed | Shipped
```

In our PostgreSQL we have a few choices, but for now we'll assume that they are stored as `text` values.

In order to use this type in Rel8 queries, we need to write an instance of `DBType` for `OrderStatus`. One approach is to use `parseTypeInformation`, which allows you to refine an existing `DBType`:

```
instance DBType OrderStatus where
  typeInformation = parseTypeInformation parser printer typeInformation
    where
      parser :: Text -> Either String OrderStatus
      parser "PaymentPending" = Right PaymentPending
      parser "Paid" = Right Paid
      parser "Packed" = Right Packed
      parser "Shipped" = Right Shipped
      parser other = Left $ "Unknown OrderStatus: " <> unpack other

      printer :: OrderStatus -> Text
      printer PaymentPending = "PaymentPending"
      printer Paid = "Paid"
      printer Packed = "Packed"
      printer Shipped = "Shipped"
```

2.3 Deriving DBType via ReadShow

The `DBType` definition for `OrderStatus` above is a perfectly reasonable definition, though it is quite verbose and tedious. Rel8 makes it easy to map Haskell types that are encoded using `Read`/`Show` via the `ReadShow` wrapper. An equivalent `DBType` definition using `ReadShow` is:

```
data OrderStatus = PaymentPending | Paid | Packed | Shipped
  deriving stock (Read, Show)
  deriving DBType via ReadShow OrderStatus
```

2.4 Storing structured data with JSONEncoded

It can occasionally be useful to treat PostgreSQL more like a document store, storing structured documents as JSON objects. Rel8 comes with support for serializing values into structured representations through the `JSONEncoded` and `JSONBEncoded` deriving-via helpers.

There usage is very similar to `ReadShow` - simply derive `DBType via JSONEncoded`, and Rel8 will use `ToJSON` and `FromJSON` instances (from `aeson`) to serialize the given type.

For example, a project might use event sourcing with a table of events. Each row in this table is an event, but this event is stored as a JSON object. We can use this type with Rel8 by writing:

```
data Event = UserCreated UserCreatedData | OrderCreated OrderCreatedData
  deriving stock Generic
  deriving anyclass (ToJSON, FromJSON)
  deriving DBType via JSONBEncoded Event
```

Here we used `JSONBEncoded`, which will serialize to PostgreSQL `jsonb` (binary JSON) type, which is generally the best choice.

2.5 The `DBType` subtypes

2.5.1 `DBEq`

The `DBEq` type class represents the subclass of database types that support equality. By supporting equality, we mean that a type supports the = operator, and also has a suitable notion of equality for operations like `GROUP BY` and `DISTINCT`. On the one hand, this class is like Haskell's `Eq` type class. The main difference is that this class has no methods.

2.5.2 `DBOrd`

The `DBOrd` type class represents the subclass of database types that support the normal comparison operators - <, <=, >= and >.

2.5.3 `DBMax` and `DBMin`

The type classes indicate that a database type supports the `min()` and `max()` aggregation functions.

2.5.4 `DBSemigroup` and `DBMonoid`

These type classes exist to give Rel8's API a similar feel to Haskell programming. Many database types have a sensible monoid structure, with the presence of a `mempty`-like expression, and an associative operation to combine `Exprs`.

2.5.5 `DBNum`, `DBIntegral` and `DBFractional`

These type classes are used to present a familiar numeric type hierarchy for Haskell programmers.

`DBNum` This class indicates that a type supports the +, −, and * operators, along with the `abs()`, `negate()` and `sign()` functions. Database types that are instances of `DBNum` allow `Num (Expr a)` to be used (allowing you to combine expressions with Haskell's normal + function).

`DBIntegral` If a type is an instance of `DBIntegral`, it means that the type stores integral (whole) numbers. The `Rel8.Expr.Num` module provides familiar `Expr` functions like `fromIntegral` to convert between types.

`DBFractional` If a type is an instance of `DBFraction`, it means that the type supports the / operator, and literals can be created via Haskell's `Rational` type class. This type class provides the `Fracitonal (Expr a)` instance.

2.5.6 DBString

This type class indicates that a database type supports the `string_agg()` aggregation function.

CHAPTER

THREE

EXPR AND SQL

3.1 Expr

Now that we've seen how types are bridged between a database and Haskell via `DBType`, we can start looking at how expressions are modelled. In Rel8, any expression will be a value of the form `Expr a`, where `a` is the type of the expression. For example, the SQL expression `user.id = 42` is an `Expr Bool`, and the subexpressions `user.id` and `42` might be `Expr UserId`s.

`Expr`s are usually created by combining other `Expr`s together, like using numeric operations like + and *, or by quoting Haskell values into queries with `lit`. Continuing with the example of `user.id = 42`, we can write this in Rel8 as:

```
userId ==. lit (UserId 42)
```

Here it's assumed that a `userId :: Expr UserId` expression is already in scope, and we can compare that against another `Expr UserId` using the `==.` operator (which is like Haskell's normal == operator, but lifted to work on `Expr`). The expression `lit (UserId 42)` quotes the Haskell term `UserId 42` literally as the SQL expression `42`. This particular encoding is chosen because `UserId` is simply a `newtype` around `Int64`.

3.2 null

So far we've only talked about `DBType`, which represents database types *excluding* `null`. Rel8 chooses this encoding because `null` is rather special in SQL, and doesn't really constitute a distinct type. For example, there is no SQL notion of "stacking" nulls - which is to say a type like `Maybe (Maybe UserId))` has no real analogous type in SQL.

Of course, as `null` is pervasive in SQL queries, Rel8 does support `null` - simply wrap up your type in `Maybe`. `Nothing` will be translated as `null`, and `Just` is used to represent non-null values.

Rel8 comes with a set of functions to work with `null` values thash should feel familiar to Haskell programmers:

- `null` creates null values (you can also use `lit Nothing`).
- `nullify` turns an `Expr a` into a `Expr (Maybe a)` (like `Just`).
- `nullable` allows you to eliminate null values, like the `maybe` function.
- `isNull` and `isNonNull` work like `isNothing` and `isJust`, respectively.
- `mapNullable` is like `fmap` for `Maybe`, and allows you to map over non-null values.
- `liftOpNullable` is like `liftA2` for `Maybe`, and allows you to combine expressions together, given a way to combine non-null expressions.

3.3 `SQL` and null-polymorphic expressions

Through the API reference documentation for Rel8, you might encounter the `Sql` type class. For example, if we look at the type of `litExpr`, we have:

```
litExpr :: Sql DBType a => a -> Expr a
```

Here `Sql DBType a` means that `a` can either be literally a type that has an instance of `DBType` (like `UserId` or `Bool`), *or* that same type wrapped in `Maybe` (so `Maybe UserId` or `Maybe Bool`). `Maybe` here encodes the SQL concept of `null`.

Some functions work regardless of whether or not a value is null, and in these cases you'll see `Sql DBType a`. `Sql` can be used with any `DBType` subtype. For example, the type of `div` is:

```
div :: Sql DBIntegral a => Expr a -> Expr a -> Expr a
```

Which means `div` works on any `DBIntegral a`, including `Maybe a`.

CHAPTER

FOUR

TABLES

So far we've seen that `DBType` bridges Haskell types to database types, and `Expr` lets us write SQL expressions using these types. The next concept is to understand how tables are mapped into Rel8.

In Rel8, we understand a table to be a list of columns. Each column has some associated metadata - a column has a type, but it also has information as to whether it can store `null`, the name of the column, and whether this column has a default value. All of this metadata can be configured when you define custom table types, but first we'll look at some built in tables.

4.1 `Expr` is a single-column table

Any time you have an `Expr`, you also have a table. All `Exprs` are tables that consist of exactly one column. This means that whenever a table is expected, you can usually use an `Expr` as well.

4.2 Tuples combine tables

You can use Haskell's normal tuple syntax to combine tables into larger tables. Now that we know that an `Expr` is a table with one column, we can use tuples to build larger tables. For example, if we have `userId :: Expr UserId` and a `name :: Expr Text`, we can pair these together as `(userId, name) :: (Expr UserId, Expr Text)`.

Rel8 supports tuples with two, three, four and five elements. Beyond that, we suggest writing a custom table type (though you can also nest tuples).

4.3 Custom table types

The primary way to define a table in Rel8 is to use the higher-kinded data pattern. Rel8 advocates this system because it means you can define your data type once, but use it in the context of Haskell expressions (for example, to serialize it as JSON to use as a REST API call response type), and also within Rel8 queries. This allows developers to share their understanding of a type in multiple domains, while also reducing the amount of code that has to be written and maintained.

To define a custom table using this pattern, you define a Haskell data type that has a single parameter (conventionally called `f`). Next, we suggest using record syntax to define the fields of your data type, and for each field use the `Column` type family to define the type of the column. Once all fields have been defined, you can bridge this type with Rel8 by deriving `Generic` and `Rel8able` instance.

A typical table definition might look like this:

```haskell
data User f = User
  { userId :: Column f UserId
  , userName :: Column f Text
  , userCreatedAt :: Column f UTCTime
  , userEmail :: Column f (Maybe EmailAddress)
  }
  deriving stock (Generic)
  deriving anyclass (Rel8able)
```

CHAPTER

FIVE

WRITING QUERIES WITH `QUERY`

To fetch data from a database, Rel8 allows you to write `SELECT` queries using the `Query` monad. While this monad might look a little different from ordinary SQL, it is equal in expressivity, but due to Haskell's `do` notation we get to benefit from all the means of abstraction available.

5.1 Understanding the `Query` monad

Before we look at special functions for working with `Query`, we'll first take a moment to understand how the `Query` monad works. First, what does the type `Query a` *mean*? To be a `Query a` means to be a SQL `SELECT` query that selects rows of type `a`. Usually `a` will be an instance of `Table Expr`, such as `Expr Text`, or maybe `BlogPostComment Expr`.

As `Query` is an instance of `Monad` means that we already have three familiar APIs to work with: `Functor`, `Applicative`, and `Monad`.

5.1.1 `Functor Query`

The `Functor` instance gives us access to `fmap`, and its type is:

```
fmap :: (a -> b) -> Query a -> Query b
```

`fmap` uniformly transforms rows of one type into rows of another type. In SQL, this is corresponds to a *projection*. For example, if we have a `Query (User Expr)`, we might do `fmap userId` to transform this into a `Query (Expr UserId)`.

5.1.2 `Applicative Query`

The `Applicative` instance for `Query` gives us:

```
pure :: a -> Query a
(<*>) :: Query (a -> b) -> Query a -> Query b
```

`pure` constructs a `Query` that returns exactly one row - a row containing the `a` that was given. This might seem fairly pointless, but it's an important `Query` when compared with `(<*>)`. The `<*>` combines two `Query`s by taking their *cartesian product*, followed by a projection that combines each row into a new row.

One example of using the `Applicative` operators is to combine two `Query`s into a tuple:

```
pure (,) <*> queryA <*> queryB
```

5.1.3 `Monad Query`

The final type class to discuss is `Monad Query`. `Monad Query` has two methods:

```
return :: a -> Query a
(>>=) :: Query a -> (a -> Query b) -> Query b
```

`return` is the same `pure`, so we won't discuss this further. The much more interesting operation is `>>=` - commonly referred to as "bind". This operator allows you to *extend* a `Query` with a new query. In SQL this is also similar to a cartesian product, but uses the `LATERAL` modifier to allow the second query to refer to columns from the first.

This extension operator allows you to expand each row in the first query into zero, one, or many rows, according to the given function. For example, if we have a database of orders, we might write:

```
getAllOrders >>= \order -> getUserById (orderUserId order)
```

This `Query` will return, for each `Order`, the `User` who placed that order. In this case, this is a one-to-one relationship, so we get back exactly as many rows as there are orders.

Going in the other direction, we have:

```
getAllUsers >>= \user -> getOrdersForUser (userId user)
```

This is a different query, as we start by fetching all `Users`, and for each user find all `Orders` they have placed. This `Query` has a different cardinality, as we're following a one-to-many relationship: any `User` may have zero, one, or many orders.

Haskell has special syntax for working with monads - `do` notation. `do` notation allows you to write these queries in a simpler form, where we don't have to introduce functions. Expanding on the latter query, we could write:

```
do user  <- getAllUsers
   order <- getOrdersForUser user
   return (user, order)
```

Now we have a query that, for each `User`, fetches all orders for that user. The final `return` means that for each `User` and `Order`, we'll return a single row.

5.2 Selecting rows from tables

With the more theoretical side of `Query` considered, we can start looking at the more pragmatic side, and how `Query` can express some common SQL idioms.

First, one of the most common operations is to select all rows from a *base table*. In SQL, this is a `SELECT * FROM x` query, and in Rel8 we use `each` with a `TableSchema`.

5.3 Limit and offset

The SQL `LIMIT` and `OFFSET` keywords are available in Rel8 as `limit` and `offset`. Note that, like SQL, the order of these operations matters. Usually, the correct thing to do is to first apply an offset with `offset`, and then use `limit` to limit the number of rows returned:

```
limit n . offset m . orderBy anOrdering
```

These operations are similar to Haskell's `take` and `drop` operations.

5.4 Filtering queries

Rel8 offers a few different ways to filter the rows selected by a query. Perhaps the most familiar operation is to apply a `WHERE` clause to a query. In Rel8, this is done using `where_`, which takes any `Expr Bool`, and returns rows where that `Expr` is true. For example, to select all public blog posts, we could write:

```
blogPost <- each blogPostSchema
where_ $ blogPostIsPublic blogPost
```

An alternative way to write `WHERE` clauses is to use `filter`. This operator is similar to the `guard` function in `Control.Monad`, but also returns the tested row. This allows us to easily chain a filtering operation on a query. The above query could thus be written as:

```
blogPost <- filter blogPostIsPublic =<< each blogPostSchema
```

`where_` and `filter` allow you to filter rows based on an expression, but sometimes we want to filter based on another query. For this, Rel8 offers `present` and `absent`. For example, if all blog posts have a list of tags, we could use `present` to find all blog posts that have been tagged as "Haskell":

```
blogPost <- each blogPostSchema
present do
  filter (("Haskell" ==.) . tagName) =<< tagFromBlogPost blogPost
```

Notice that this example uses `present` with a query that itself uses `filter`. For each blog post, `present` causes that row to be selected only if the associated query finds a tag for that blog post with the `tagName` "Haskell".

Like `filter` there is also a chaining variant of `present` - `with`. We could rewrite the above query using `with` as:

```
haskellBlogPost <-
  each blogPostSchema >>=
  with (filter (("Haskell" ==.) . tagName) <=< tagFromBlogPost)
```

5.5 Inner joins

Inner joins are SQL queries of the form `SELECT .. FROM x JOIN y ON ...` Rel8 doesn't offer a special function for these queries, as the same query can be expressed by selecting from two tables (this is called taking the *cartesian product* of two queries) and then filtering the result.

If we wanted to join each blog post with the author of the blog post, we would write the SQL:

```
SELECT * FROM blog_post JOIN author ON author.id = blog_post.id
```

The alternative way to write this query with `WHERE` is:

```
SELECT * FROM blog_post, author WHERE author.id = blog_post.id
```

and this query can be written in Rel8 as:

```
blogPost <- each blogPostSchema
author <- each authorSchema
where_ $ blogPostAuthorId blogPost ==. authorId author
```

Hint: A good pattern to adopt is to abstract out these joins as functions. A suggested way to write the above would be to extract out an "author for blog post" function:

```
blogPost <- each blogPostSchema
author <- authorForBlogPost blogPost
```

where:

```
authorForBlogPost :: BlogPost Expr -> Query (Author Expr)
authorForBlogPost blogPost =
  filter ((blogPostAuthorId blogPost ==.) . authorId) =<<
  each authorSchema
```

While this is a little more code over all, in our experience this style dramatically increases the readability of queries using joins.

5.6 Left (outer) joins with `optional`

A left join is like an inner join, but allows for the possibility of the join to "fail". You use left joins when you want to join optional information against a row.

In Rel8, a `LEFT JOIN` is introduced by converting an inner join with `optional`. While this approach might seem a little foreign at first, it has a strong similarity with the `Control.Applicative.optional` function, and allows you to reuse previous code.

To see an example of this, let's assume that we want to get the latest comment for each blog post. Not all blog posts are popular though, so some blog posts might have no comment at all. To write this in Rel8, we could write:

```
blogPost <- each blogPostSchema

latestComment <-
  optional $ limit 1 $
    orderBy (commentCreatedAt >$< desc) $
      commentsForBlogPost blogPost
```

`optional` will transform a `Query a` into a `Query (MaybeTable a)`. `MaybeTable` is similar to the normal `Maybe` data type in Haskell, and represents the choice between a `justTable x` and a `nothingTable` (like `Just x` and `Nothing`, respectively). When you execute a query containing `MaybeTable x` with `select`, Rel8 will return `Maybe x`. `MaybeTable` comes with a library of routines, similar to the functions that can be used to operate on `Maybe`. For more details, see the API documentation.

Hint: `optional` converts an inner join into a `LEFT JOIN`, but you can also go the other way - and turn a `LEFT JOIN` back into an inner join! To do this, you can use `catMaybeTable`, which will select only the rows when the left join was successful.

5.7 Ordering results

Rel8 supports ordering the results returned by a `Query`, using SQL's `ORDER BY` syntax. To specify an ordering, you use `orderBy` and supply an appropriate `Order` value.

An `Order` is built by combining the order of individual columns, each of which can be either ascending or descending. To order a single column, you combine `asc` or `desc` with `Orders` *contravariant* interface. For example, if we have a table with a `orderId` column, we can order a `Query (Order Expr)` by `orderId` with:

```
orderBy (orderId >$< asc)
```

To order by multiple columns, combine the individual orders with `Orders` `Monoid` instance. We could extend the above example to order by the order date first (with the most recent orders first) with:

```
orderBy (mconcat [orderDate >$< desc, orderId >$< asc])
```

5.8 Aggregating queries

Todo: Write this

5.9 Set operations

Todo: Write this

CHAPTER

SIX

INSERT, UPDATE AND DELETE

While the majority of Rel8 is about building and executing `SELECT` statement, Rel8 also has support for `INSERT`, `UPDATE` and `DELETE`. These statements are all executed using the `insert`, `update` and `delete` functions, all of which take a record of parameters.

Note: This part of Rel8's API uses the `DuplicateRecordFields` language extension. In code that needs to use this API, you should also enable this language extension, or you may get errors about ambiguous field names.

6.1 DELETE

To perform a `DELETE` statement, construct a `Delete` value and execute it using `delete`. `Delete` takes:

`from` The `TableSchema` for the table to delete rows from.

`using` This is a simple `Query` that forms the `USING` clause of the `DELETE` statement. This can be used to join against other tables, and the results can be referenced in the `deleteWhere` parameter. For simple `DELETE`s where you don't need to do this, you can set `using = pure ()`.

`deleteWhere` The `WHERE` clause of the `DELETE` statement. This is a function that takes two inputs: the result of the `using` query, and the current value of the row.

`returning` What to return - see *RETURNING*.

6.2 UPDATE

To perform a `UPDATE` statement, construct a `Update` value and execute it using `update`. `Update` takes:

`target` The `TableSchema` for the table to update rows in.

`from` This is a simple `Query` that forms the `FROM` clause of the `UPDATE` statement. This can be used to join against other tables, and the results can be referenced in the `set` and `updateWhere` parameters. For simple `UPDATE`s where you don't need to do this, you can set `from = pure ()`.

`set` A row to row transformation function, indicating how to update selected rows. This function takes rows of the same shape as `target` but in the `Expr` context. One way to write this function is to use record update syntax:

```
set = \from row -> row { rowName = "new name" }
```

`updateWhere` The `WHERE` clause of the `UPDATE` statement. This is a function that takes two inputs: the result of the `from` query, and the current value of the row.

returning What to return - see *RETURNING*.

6.3 INSERT

To perform a `INSERT` statement, construct a `Insert` value and execute it using `insert`. `Insert` takes:

into The `TableSchema` for the table to insert rows into.

rows The rows to insert. These are the same as `into`, but in the `Expr` context. You can construct rows from their individual fields:

```
rows = values [ MyTable { myTableA = lit "A", myTableB = lit 42 }
```

or you can use `lit` on a table value in the `Result` context:

```
rows = values [ lit MyTable { myTableA = "A", myTableB = 42 }
```

onConflict What should happen if an insert clashes with rows that already exist. This corresponds to PostgreSQL's `ON CONFLICT` clause. You can specify:

Abort PostgreSQL should abort the `INSERT` with an exception

DoNothing PostgreSQL should not insert the duplicate rows.

DoUpdate PostgreSQL should instead try to update any existing rows that conflict with rows proposed for insertion.

returning What to return - see *RETURNING*.

6.4 RETURNING

PostgreSQL has the ability to return extra information after a `DELETE`, `INSERT` or `UPDATE` statement by attaching a `RETURNING` clause. A common use of this clause is to return any automatically generated sequence values for primary key columns. Rel8 supports `RETURNING` clauses by filling in the `returning` field and specifying a `Projection`. A `Projection` is a row to row transformation, allowing you to project out a subset of fields.

For example, if we are inserting orders, we might want the order ids returned:

```
insert Insert
  { into = orderSchema
  , rows = values [ order ]
  , onConflict = Abort
  , returning = Projection orderId
  }
```

If we don't want to return anything, we can use `pure ()`:

```
insert Insert
  { into = orderSchema
  , rows = values [ order ]
  , onConflict = Abort
  , returning = pure ()
  }
```

6.5 Default values

It is fairly common to define tables with default values. While Rel8 does not have specific functionality for `DEFAULT`, there are a few options:

6.5.1 `unsafeDefault`

Rel8 does not have any special support for `DEFAULT`. If you need to use default column values in inserts, you can use `unsafeDefault` to construct the `DEFAULT` expression:

```
insert Insert
  { into = orderSchema
  , rows = values [ Order { orderId = unsafeDefault, ... } ]
  , onConflict = Abort
  , returning = Projection orderId
  }
```

Warning: As the name suggests, this is an unsafe operation. In particular, Rel8 is not able to prove that this column does have a default value, so it may be possible to introduce a runtime error. Furthermore, `DEFAULT` is fairly special in SQL, and cannot be combined like other expressions. For example, the innocuous expression:

```
unsafeDefault + 1
```

will lead to a runtime crash.

6.5.2 Reimplement default values in Rel8

If you only need to access default values in Rel8, another option is to specify them in Rel8, rather than in your database schema.

Hint: A common default value for primary keys is to use *nextval* to obtain the next value from a sequence. This can be done in Rel8 by using the `nextval` function:

```
insert Insert
  { into = orderSchema
  , rows = values [ Order { orderId = nextval "order_id_seq", ... } ]
  , onConflict = Abort
  , returning = Projection orderId
  }
```

CHAPTER

SEVEN

COOKBOOK

This cookbook exists to help you easily form Rel8 queries. It's main purpose is to help those familiar with SQL to translate their queries into Rel8 `Query`s.

7.1 `SELECT * FROM table`

To select from a table, use `each`.

7.2 Inner joins

To perform an inner join against two queries, use `where_` with a join condition. For example, the following SQL:

```
SELECT * FROM table_a JOIN table_b ON table_a.x = table_b.y
```

can be written as:

```
myQuery = do
  a <- each tableA
  b <- each tableB
  where_ $ tableAX a ==. tableBY b
```

7.3 Left (outer) joins

A `LEFT JOIN` query is performed by using `optional`.

For example, the query:

```
SELECT * FROM table_a LEFT JOIN table_b ON table_a.x = table_b.y
```

can be written as:

```
myQuery = do
  a <- each tableA

  maybeB <- optional do
    b <- each tableB
    where_ $ tableAX a ==. tableBY b

  return (a, maybeB)
```

Note that `maybeB` here will be a `MaybeTable`, which is the Rel8 `Query`-equivalent of `Maybe`. This allows you to observe if a left join has succeeded or failed.

7.4 Ordering results

A `Query` by default has no ordering - just like in SQL. If you rows back in a certain order, you can use `orderBy` with an `Order`.

For example, the query:

```
SELECT * FROM my_table ORDER BY my_table.id ASC, my_table.x DESC NULLS FIRST
```

can be written as:

```
myQuery =
  orderBy (mconcat [ myTableId >$< asc, myTableX >$< nullsFirst desc ]) $
  each myTableSchema
```

Note that we use `>$<` (from `Data.Functor.Contravariant`) to select table columns, and we can use `mconcat` to combine orderings.

If all columns of a table have an ordering, you can also use `ascTable` and `descTable`. For example:

```
myQuery = orderBy ascTable $ each myTableSchema
```

7.5 Aggregations

Aggregations in Rel8 work by using `aggregate`, which takes a `Query (Aggregate a)` and gives you back a `Query a`.

The following query:

```
SELECT sum(foo), count(distinct bar) FROM table_a
```

can be written as:

```
myQuery = aggregate do
  a <- each tableA
  return $ liftF2 (,) (sum (foo a)) (countDistinct (bar a))
```

where `liftF2` comes from `Data.Functor.Apply` from the `semigroupoids` library.

7.6 Combining aggregations

As `Aggregate` is an instance of `Apply` (which is very similar to `Applicative`), individual aggregations can be combined. For example, one way to take the average rating would be to write the query:

```
SELECT sum(rating.score) / count(rating.score) FROM rating
```

In Rel8, we can write this as:

```
myQuery = aggregate do
  rating <- each ratingSchema
  return $ liftF2 (/) (sum (score rating)) (count (score rating))
```

You can also use `RebindableSyntax` and `ApplicativeDo`:

```
{-# language ApplicativeDo, RebindableSyntax #-}

import Data.Functor.Apply ((<.>))

myQuery = aggregate do
  rating <- each ratingSchema
  return do
    scoreSum        <- sum (score rating)
    numberOfRatings <- count (score rating)
    return (scoreSum / numberOfRatings)
  where (<*>) = (<.>)
```

For large aggregations, this can often make queries easier to read.

7.7 Tree-like queries

Rel8 has a fairly unique feature in that it's able to return not just lists of rows, but can also return *tree*s.

To understand what this means, we'll consider a small example query for blog posts. We want our query to return:

1. The latest 5 blog posts that have at least one tag each.
2. For each blog post, all tags.
3. For each blog post, the latest 3 comments if they exist.

In Rel8, we can write this query as:

```
latestBlogPosts = do
  post <- each postSchema

  -- Returns a `NonEmptyTable a` which ends up as a `Data.List.NonEmpty a` after the␣
↪query is run
  tags <- some $ do
    tag <- each tagSchema
    where_ (tagPostId tag ==. postId post)
    return (tagName tag)

  -- Returns a `ListTable a` which ends up as a `[a]` after the query is run
  latestComments <-
    many $
    limit 3 $
    orderBy (commentCreatedAt >$< desc) do
      comment <- each commentSchema
      where_ (commentPostId comment ==. postId post)

  return (post, tags, latestComments)
```

CHAPTER

EIGHT

MORE RESOURCES

- The Haskell API documentation describes how individual functions are types are to be used.
- If you have a question about how to use Rel8, feel free to open a GitHub discussion.
- If you think you've found a bug, confusing behavior, or have a feature request, please raise an issue at Rel8's issue tracker.