Type-safe database queries via row-generic programming: a monadic EDSL for PureScript

(Bezpieczne zapytania bazodanowe
z wykorzystaniem polimorficznych rzędów:
monadyczny EDSL w języku PureScript)

Kamil Listopad

Praca inżynierska

Promotor: dr Filip Sieczkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

29 stycznia 2019

Abstract

We present a method of writing SQL queries that are correct by construction, in a functional language. We provide an EDSL that supports simple and secure ways of describing queries that can be arbitrarily nested with joins, aggregates, etc. We utilise PureScript's row types to represent rows from the database tables as records, which gives us greater flexibility by being able to refer to columns by name rather than position in a tuple. In this thesis we describe how to define queries using our EDSL, explain its operations and data types used. We also discuss some interesting implementation techniques that include row-generic programming, existential types and types equality for GADT substitution.

Przedstawiamy metodę pisania zapytań SQL w języku funkcyjnym, których poprawność jest gwarantowana przez system typów. Dostarczamy wygodnych i ekspresywnych sposobów opisywania kwerend, które pozwalają na łączenie tablic, używanie agregacji oraz mogą być dowolnie w sobie zagnieżdżane. Wykorzystujemy typy rzędowe języka PureScript przy reprezentacji wierszy bazodanowych jako rekordy, które dają nam większą elastyczność, ponieważ możemy odwoływać się do poszczególnych kolumn po ich nazwie, zamiast pozycji w krotce. W tej pracy opisujemy jak definiować zapytania używając naszego EDSLa, wyjaśniamy jego operacje i typy danych. Omawiamy również kilka interesujących technik implementacyjnych, tj. programowanie generyczne na typach rzędowych, typy egzystencjalne i zastąpienie GADT wykorzystując równość typów.

Contents

Chapter 1

Introduction

Querying databases for information is one of the most common tasks in software development. SQL-based relational databeses are especially popular. Programming languages need to support these operations in some way. We can distinguish several general ways to implement this feature.

The simplest of them acts as a thin layer between the user code and the database engine. It assumes that a programmer will write SQL as string and submit it for execution. Then the programmer can ask the engine to return results, where each database row is usually represented as an array of values. This approach is commonly extended with the technique called *prepared statements*[11]. This allows a programmer to prepare an SQL statement template and send it to the database. Certain values are left unspecified, with placeholders used instead. The database can then prepare the statement for fast and secure executions multiple times when the application finally binds values to parameters.

Another solution that is often used in object-oriented programming languages is *object-relational mapping*[9] (*ORM*). In these languages, objects are the primary data type. Therefore, it is useful to create a mapping between classes and database tables. This raises some problems because object models and relational models do not work very well together[16]. A purely object-oriented approach to data access would be impractical. Naturally, we would navigate from one association to another walking the object network. This is not an efficient way of retrieving data from a relational database. Therefore, *ORM*s usually support some filtering capabilities to load only the necessary entities, before walking the object network.

And finally some solutions implement a way to write SQL-like code inside the host language. This takes a form of a *domain specific language*[3] (*DSL*) which is *embedded* in the general purpose programming language. As a example we can consider *LINQ*[8], which is a Microsoft .NET Framework component that adds native data querying capabilities. This allows the compiler to type-check queries together with the host program.

```
var results = from c in SomeCollection
              where c.SomeProperty < 10
              select new {c.SomeProperty, c.OtherProperty};
```

Unfortunately, both solutions do not provide full SQL capabilities. ORMs impose restrictive limitations on how queries are created to match OOP world with classes and inheritance. On the other hand LINQ allows us to write only simple queries, but its idea can be expanded futher to implement EDSLs that can handle more complicated SQL constructions in a convenient and safe way.

Inspired by LINQ Anton Ekblad developed *Selda* [5], DSL embedded in Haskell, which supports writing type-safe queries. The SQL code that Selda generates is guaranteed to be correct. In particular, it supports arbitrarily nested queries with capabilities of aggregation. On top of that he achieved this with easy to use methods, with syntax similar to the SQL.

1.1 About PureScript version of *Selda*

The aim of this work was to create a SQL library inspired by *Selda* for PureScript. Following in the footsteps of our predecessor, we provided a monadic interface for our EDSL, because this is the standard abstraction in PureScript just like in Haskell. It supports a convenient way of writing SQL-like queries. The generated SQL code is guaranteed to be correct and type-safe. PureScript Selda allows us to load data from tables, filter results, combine them with `JOIN`s, use aggregation capabilities and also supports arbitrarily nested queries.

However, there are several improvements we wanted to make in comparison with Haskell version. PureScript provides polymorphic records which we used to define database tables and to model database rows. It significantly improves usability compared to the less elegant syntax of the heterogeneous lists used in the original Haskell implementation.

In our implementation, we focused on *PostgreSQL* to provide practically critical features such as `RETURNING` in the **insert** operation. Our implementation depends on a low-level PostgreSQL client, which we use to execute generated queries. Nevertheless, the main part of the code used to describe queries is independent of the database engine. Our solution focuses exclusively on data management in a relational database. In particular we do not validate defined tables against the actual schema. Most of the similar solutions generate SQL that is unreadable to humans, which causes problems when debugging. With little effort, we managed to provide a generation method that outputs readable SQL, which has similar structure to the actual query code written in Selda.

Chapter 2

Usage

2.1 Schema declaration

Our EDSL only manages data in the database and does not alter nor validate schema. It requires the user to provide necessary information about the database. Before performing any data manipulation the user have to declare tables he wants to use.

2.1.1 Table representation

One way to model database tables is to use heterogeneous lists, where we would specify each column with their name and type in the signature. Then the order of types in the list would have to correspond to the one in the definition of the SQL table. This is how it is implemented in the Haskell Selda. However, we believe that it is safer and more convenient not to rely on the order of columns in the SQL table definition. We used PureScript records and row of types for this job.

Consider the following SQL table definition:

```
CREATE TABLE people (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  age INTEGER
);
```

To represent this table definition in PureScript, we use a **Table** type which is parameterized by a row of types that describes the columns and their types from the database.

```
people :: Table (id :: Int, name :: String, age :: Maybe Int)
people = Table { name: "people" }
```

Notice that nullable values are represented with **Maybe**. Also, we have not mentioned that **id** is a primary key, because for the purpose of this EDSL we do not

need this information.

```
newtype Table ( r :: # Type ) = Table { name :: String }
```

The **Table** constructor takes the name of the table as a string value. Columns are specified in the **Table** type using row type (denoted by the kind **# Type**) as shown in the previous example. It is important to note that the specification of the columns lives only at the typelevel.

2.2 Simple queries

Queries are created with the **selectFrom** function. Let's start with an example of obtaining identifiers paired with the names from the **people** table that we declared before.

```
selectFrom people \{ id, name, age } → do
  pure { id, name }
```

To understand how the **selectFrom** works, we must first see its type.

```
selectFrom :: ∀ r s cols res
            . FromTable s r cols
           ⇒ Table r
           → ({ | cols } → Query s { | res })
           → FullQuery s { | res }
```

From the type signature we can see that its first argument is a table definition. The **selectFrom** uses the table definition **Table r** to construct a record of values representing the columns of the table. These values are of abstract type **Col s a**, which represents a column of type **a** (e.g. **Col s String** may represent a column **name :: String** from the **people** table definition), type parameter **s** will be explained later. The record with values of type **Col s a** is created using the **FromTable** type class, which takes the row type **r** from the table definition **Table r** and creates this record of column representations.

In our example **FromTable** maps the row type:
(**id** :: **Int** , name :: **String** , age :: **Maybe Int**)
from the table definition **people** to the record
{ **id** :: Col s **Int** , name :: Col s **String** , age :: Col s (**Maybe Int**)} .

The second argument of the **selectFrom** specifies the query. It is a function that is applied to the record of type **{ | cols }**, which represents a single row of the table, to get the rest of the query details, namely **Query**. The **selectFrom** function combines the table definition **Table r** with the **Query** and returns a fully specified query — **FullQuery**.

2.2.1 The Query monad

The **Query** is a monad that represents the state of the query description. Operations that act on this type modify the state by specyfing additional parts of the query (adding more components to it). To understand this better we can imagine that **Query** starts as a description with just a `FROM` clause filled with one table that was passed to the **selectFrom** function. Then, the **pure** call in the example adds a `SELECT` clause and finishes the query. Before that we can write other operations that add components that correspond to a `WHERE`, `GROUP BY`, `JOIN`, etc.

One important design choice in our implementation is the distinction between finished descriptions and those still susceptible to change. This difference is reflected in the return type of the **selectFrom** function. **FullQuery** is a frozen version of the **Query** data type. It is a fully specified query that is valid on its own and can be executed (Section 3.4) or used as a nested query (Section 2.4).

2.2.2 Joins and restricts

Now we can discuss operations of the **Query** monad. To limit the returned rows only to those that meet the given condition, we use a **restrict** operation, which corresponds to `WHERE` in SQL. In order to be able to create reasonable logical conditions, we need several binary operators. To avoid name conflicts with normal operators, we define them with a dot preceding their names.

Another desired SQL operation is the ability to `JOIN` tables. This is done using **leftJoin** function which combines tables with `LEFT JOIN`.

Types for those operations are given in the listing below. First of them is simple, it takes a condition of type **Col s Boolean**, which will usually be an expression built using relational operators such as `(.>)`.

```
restrict :: ∀ s. Col s Boolean → Query s Unit

leftJoin :: ∀ r s cols mcols
          . FromTable s r cols
         ⇒ WrapWithMaybe { | cols } { | mcols }
         ⇒ Table r
         → ({ | cols } → Col s Boolean)
         → Query s { | mcols }

(.>), (.==), ... :: ∀ s a. Col s a → Col s a → Col s Boolean
```

Let us now consider the **leftJoin** function. Its first argument is a table definition **Table r**, which is used to create a record **{ | cols }**. The second argument encodes a condition that determines when to join the rows. This function takes the representation of columns from the table and returns a boolean expression. When returning a left-table row for which there is no match in the right table, empty (null)

values are inserted into right-table columns. Thus, each value of type **Col s a** in the record **{ | cols }** is mapped to **Col s (Maybe a)** to reflect that the value is nullable. This is done using **WrapWithMaybe** type class.

Let us consider an example that shows how to use **leftJoin** in practice. We want to get people's IDs along with their account balance. However, we don't want to omit people who don't have an account. We use previously defined **people** table to get IDs and a table called **bankAccounts** that is defined below, which keeps an account balance for some people.

```
bankAccounts :: Table ( personId :: Int, id :: Int, balance :: Int )
bankAccounts = Table { name: "bank_accounts" }

q :: ∀ s. FullQuery s {id :: Col s Int, balance :: Col s (Maybe Int)}
q = selectFrom people \{ id } → do
  { balance } ← leftJoin bankAccounts \b → id .== b.personId
  pure { id, balance }
```

An SQL equivalent to the above query `q` can be found below, where we also provide the state of the database and the results of running the query **q**.

```
SELECT id, balance FROM people
LEFT JOIN bank_accounts ON (people.id = bank_accounts.personId)
```

people		
id	**name**	**age**
1	'name1'	11
2	'name2'	22
3	'name3'	33

bank_accounts		
id	**personId**	**balance**
1	1	100
2	1	150
3	3	300

```
[ { id: 1, balance: Just 100 }
, { id: 1, balance: Just 150 }
, { id: 2, balance: Nothing }
, { id: 3, balance: Just 300 }
]
```

We can see that the person without any account was included in the results. Also one person was included twice due to having more than one bank account.

2.3 Aggregate

Another feature we want to support are aggregation functions, such as **count**, **max**, etc. Simple implementation of **max** shows a problem with such queries. It would allow us to write queries that are incorrect.

```
max_ :: ∀ s a. Col s a → Col s a
```

```
selectFrom people \{ name, age } → do
  pure { name, maxAge: max_ age }

SELECT name, MAX(age) as maxAge
FROM people

> ERROR: column "people.name" must appear in the GROUP BY
         clause or be used in an aggregate function
```

SQL forces us to specify the column **name** in the `GROUP BY` clause if we want to return it together with the **max(age)**. Queries that use aggregate functions and `GROUP BY` require special treatment. Such queries must return values that come from aggregate functions or columns that are in the `GROUP BY` clause.

To resolve this problem we first need to introduce a **groupBy** function and enforce that every column in the result was *grouped by* or is a result of an aggregate function. To satisfy this requirement we introduce a distinct type **Aggr s a** for columns that can be returned as a result of aggregate queries. Values of this type are introduced using **groupBy** or aggregate functions. Queries that return a record with `Aggr` values mixed with `Col`s are considered incorrect and result in a type error.

The example below uses the **groupBy** on the column **name** to get **Aggr s String** and uses it in the result record. Function **aggregate** satisfies the requirement and transforms values of type **Aggr s a** in the result record to **Col s a**.

```
groupBy :: ∀ s a. Col s a → Query s (Aggr s a)

max_ :: ∀ s a. Col s a → Aggr s a

aggregate :: ∀ s aggr res. UnAggr { | aggr } { | res }
          ⇒ FullQuery s { | aggr }
          → FullQuery s { | res }

q :: ∀ s. FullQuery s
          { maxAge :: Col s (Maybe Int), name :: Col s String }
q = aggregate $ selectFrom people \{ id, name, age } → do
  name' ← groupBy name
  pure { name: name', maxAge: max_ age }
```

2.4 Sub queries

One final feature we would like to provide is the support for arbitrary nested queries. We want to allow the **selectFrom** and **leftJoin** to take a sub query instead of a **Table**. Consider the following example which uses a join on an inner query, introduced by the **leftJoin_** function.

```
selectFrom people \{ id, name, age } → do
  { balance } ← leftJoin_ (\b → id .== b.personId) $
    aggregate $ selectFrom bankAccounts \b → do
      personId ← groupBy b.personId
      pure { personId, balance: max_ b.balance }
  pure { id, balance }
```

While the above query is not problematic in itself, the fact that **id**, **name** and **age** are visible inside the sub query would allow us to write an ill-scoped join.

```
selectFrom people \{ id, name, age } → do
  { balance } ← leftJoin_ (\b → id .== b.personId) $
    aggregate $ selectFrom bankAccounts \b → do
      personId ← groupBy b.personId
      -- `id` should not be in scope of this inner query
      restrict $ id .> lit 1
      pure { personId, balance: max_ b.balance }
  pure { id, balance }
```

The problem with inner queries is that declared names in an outer query descriptions are accessible in the nested ones. This is forbidden in the SQL language. Names from the outside cannot leak into sub queries.

2.4.1 Scoping inner queries

This problem was already solved by the author of the Haskell Selda[5] and described in his paper[4]. The solution adds an additional type parameter **s** to our EDSL monad, a phantom type that we call **s**. Every value of type **Col s a** also tracks this type, where **s** comes from the monad where the value was created. The idea is that we associate expressions `Col s a` with a scope level encoded in **s**. The **Query s** monad operations operate only on these values of type **Col s a** with the same scope **s**. When selecting from a nested query we increase its scope with **Inner s**. As a consequence we can no longer use values from the outer queries in the inner ones, since the compiler will not be able to unify the type **s** with **Inner s**. This idea is used in **leftJoin_** definition below.

```
data Inner s

leftJoin_ :: ∀ s cols mcols inner
           . FromSubQuery s inner cols
          ⇒ WrapWithMaybe { | cols } { | mcols }
          ⇒ ({ | cols } → Col s Boolean)
          → FullQuery (Inner s) { | inner }
          → Query s { | mcols }
```

For readability purposes the arguments of **leftJoin_** are flipped. Now, we first provide the condition function, which determines when to join and then an

inner query — a product of the **selectFrom** function (optionally preceded by the **aggregate** if the nested query uses the aggregation functionality). This nested query is denoted by the type **FullQuery (Inner s) { | inner }** this means its scope level is raised to **Inner s** and the record **{ | inner }** has values of type **Col (Inner s) a**. In order to use these results we provide a **FromSubQuery** type class, which works similarly to the **FromTable** type class, but here it maps the record **{ | inner }** to a record **{ | cols }** with unwrapped values **Col s a**, now usable in the outer context. Then this record is mapped again to get optional values, as we discussed previously in the description of the **leftJoin** function.

Chapter 3

The Backend

In this chapter we focus on how data types that are not visible to the end-user are implemented. Our goal is to present challenges we have encountered in the course of development and our approach to overcome them.

We begin with the most crucial data type, the **Query** monad, which represents the state of the abstract syntax tree of the query at a particular point of the program execution. As its purpose suggests, it should be implemented using the **State** monad, but in addition we need to track a type parameter **s**, which represents the scoping level described in the Section 2.4.1.

```
newtype Query s a = Query (State GenState a)
```

3.1 Query AST

Looking from high level perspective how can a query AST look like?

First, it needs information about **sources** from which we load data. They consist of tables and perhaps other nested queries, which the query uses. Additionally it needs to know how they are combined.

Second part of the state describes the resultant columns, elements that will appear after `SELECT` in the generated SQL. These values we call **cols** and they are created using the result record from the **selectFrom** function.

Other parts specify the `WHERE` and `GROUP BY` clauses at the resulting SQL and are represented as lists of proper expressions. We have already encountered the **Col** abstract type, which represents these expressions on the client-side. Now we will cover how it is implemented and used in the AST.

We distinguish several types of these expressions. One of them is **Literal**, which denotes simple values like numbers, strings, booleans and others. **Column** is used as a representation of an actual column from a database table. Other forms of

expressions consists of binary operations and aggregation functions.

3.2 Col

In the following, we cover two major types that **Col** depends upon: the representations of literals and columns. Binary operations and aggregation functions are defined similarly to the **Literal**.

3.2.1 Column

Type **Column a** is a record with a column **name** and its **namespace**. For columns that comes directly from tables, the namespace is just the table name or its alias.

```
newtype Column a = Column { namespace :: String, name :: String }
```

It turns out that this type can be used to represent results from a nested query as they are accessed outside the query using its alias and their name. Consider following example of a inner query with an alias **sub_q1**.

```
(SELECT max(age) as maxAge ...) as sub_q1
```

Its result **max(age)** is visible outside the query as **sub_q1.maxAge**. So it is sufficient to represent it using the **Column** data type with namespace **sub_q1** and name **maxAge**.

3.2.2 Literal

For literals we aim for a data type **Literal a**, but we want to limit the possible types of parameter **a** only to the supported ones (like integers, strings and booleans). These restrictions may be imposed using *generalized algebraic data type* (*GADT*).

```
data Literal a where
  LBoolean :: Boolean → Literal Boolean
  LString  :: String  → Literal String
  LInt     :: Int     → Literal Int
```

This allows us to have well-typed literal values with a specific type **a** for each case. Unfortunately PureScript does not support *GADT*, so the definition presented above needs to be encoded using just *ADT* with explicit type restrictions. We cover the technical aspects of such encodings in 4.1.

3.2.3 Expr

The **Col s a** represents expressions of type **a**, but it also has type parameter **s**, which is not useful on backend-side, because we know that only well-scoped expressions are

allowed by frontend-side functions like **selectFrom**. That is why **Col** is implemented as a newtype for type **Expr**, which represents expressions mentioned above.

```
data Expr a where
  EColumn :: Column  a → Expr a
  ELit    :: Literal a → Expr a
  EFn     :: Fn a      → Expr a
  EBinOp  :: BinOp i o → Expr i → Expr i → Expr o

newtype Col s a = Col (Expr a)

data BinOp i o where
  Or :: BinOp Boolean Boolean
  Gt :: BinOp i Boolean
  Eq :: BinOp i Boolean

data Fn o where
  FnMax :: Expr o → Fn o
  ...
```

Aggregate functions with return type **a** are contained in **Fn a** type. Binary expressions are built using **BinOp i o**, which uses two type parameters: *input* and *output*. These are used to combine elements of some type (e.g. **Int**) and return result of potentially different type (e.g. **Boolean** for integer equality). The following example shows how to write a expression that compares a column with a literal.

```
pid :: Expr Int
pid = EColumn $ Column { namespace: "people", name: "id" }

l5 :: Expr Int
l5 = ELit $ LInt 5

pid_eq_l5 :: Expr Boolean
pid_eq_l5 = EBinOp Eq pid l5
```

3.3 GenState

Having dispensed with the preliminaries, we now turn to the **GenState** definition.

```
data SQL
  = FromTable { name :: String, alias :: String }
  | SubQuery String GenState

data Source
  = Product SQL
  | LeftJoin SQL (Expr Boolean)
```

```
type GenState =
  { sources   :: Array Source
  , cols      :: Array (Tuple String (Exists Expr))
  , restricts :: Array (Expr Boolean)
  , aggr      :: Array (Exists Expr)
  , nextId    :: Int
  }
```

Sources are modelled with a list of left-joined or cross-joined **SQL** components, which are either aliased tables or nested queries with an alias. The `WHERE` clause of the resulting SQL is represented as an array of boolean expressions, which we call **restricts**.

Missing part is the **Exists** type used in **cols** and **aggr**. We want to create a heterogeneous list - a collection of values with potentially different types. **Exists** allows us to hide the type parameter of **Expr**. We cover the **Exists** type in 4.2.

This way we can define **aggr** as a list of expressions of some types and similarly **cols**, but remembering the name (or column alias) for each result. The linal component, **nextId**, provides unique identifiers used for table/query aliasing.

3.4 Running the Query

In the following example we show how to execute a query. We first define a query, which returns pairs of ID and maximum account balance for each person.

```
q :: ∀ s. FullQuery s
        { id :: Col s Int, balance :: Col s (Maybe Int) }
q = selectFrom people \{ id, name, age } → do
      { balance } ← leftJoin_ (\b → id .== b.personId) $
        aggregate $ selectFrom bankAccounts \b → do
          personId ← groupBy b.personId
          pure { personId, balance: max_ b.balance }
      pure { id, balance }

main :: Effect Unit
main = launchAff_ do
  rows ← withPG dbconfig $ query q
  liftEffect $ for_ rows \{ id, balance } →
    log $ "user: " <> show id <> " has " <> show balance
```

In order to execute a query we need to provide configuration record for the database. Since we support PostgreSQL there is a function called **withPG**, which takes the database configuration and executes the given query in the PostgreSQL database. When executing multiple queries, it is convenient to do it in a separate computation, which will be performed using **withPG** just once. The computation takes place in the monad called **MonadSelda**. The function **query** takes a

previously defined **FullQuery q**, generates the SQL and runs it in **MonadSelda** returning array of result records, which we called **rows** in the example. Then we iterate over **rows** to output some information about each record. Haskell's **IO** monad is called **Effect** in PureScript. The monad on which **withPG** operates is called **Aff**, to use **log** function we lift computation from **Effect** to **Aff** using **liftEffect**. Then finally **launchAff_** is called to execute **Aff** and get desired result of type **Effect Unit**.

Generated SQL for the query **q** and the program output is given below. The state of the database is given in the Section 2.2.2.

```
SELECT people_0.id AS id, sub_q1.balance AS balance
FROM people people_0
LEFT JOIN (
  SELECT bank_accounts_0.personId AS personId,
         max(bank_accounts_0.balance) AS balance
  FROM bank_accounts bank_accounts_0
  GROUP BY bank_accounts_0.personId
) sub_q1 ON ((people_0.id = sub_q1.personId))
```

```
user: 1 has (Just 150)
user: 3 has (Just 300)
user: 2 has Nothing
```

Below we provide a shortened signatures of **query** and **withPG** functions.

```
query :: ...
     ⇒ FullQuery s (Record i)
     → MonadSelda (Array (Record o))
```

```
withPG :: ∀ a. PoolConfiguration → MonadSelda a → Aff a
```

Chapter 4

Pure concepts

This chapter is devoted to general functional techniques used in the project, which are worth describing.

4.1 GADT without GADT

In the Section 3.2 we defined **Literal** as a *GADT* (*generalized algebraic data type*) [6], but this is not supported by PureScript. Let us encode the **Literal** using *ADT* for now.

```
data Literal
  = LBoolean Boolean
  | LString String
  | LInt Int
```

However, in this form we do not have any information about the type of a literal we represent. Previously we knew that **Literal a** represented a value of type **a**. We need to add type parameter **a** to the **Literal** definition and assure the compiler that, for example, **LInt 5** constructs a value of type **Literal Int**.

4.1.1 Encoding GADTs with type equality [14]

To this end, assume we have a type **Equal a b** that represents the proposition that **a** and **b** are equal types. Furthermore, assume that this type supports the following operations.

```
refl   :: ∀ a.       Equal a a
trans  :: ∀ a b c.   Equal a b → Equal b c → Equal a c
symm   :: ∀ a b.     Equal a b → Equal b a
coerce :: ∀ a b.     Equal a b → a → b
lift   :: ∀ a b f.   Equal a b → Equal (f a) (f b)
```

Listing 4.1: **Equal** operations

Types of **refl**, **symm**, **trans** say that **Equal** is an equivalence relation — i.e., it is a congruence. While operation **lift** assures that the relation is preserved by every type constructor. And the **coerce** function allows us to safely coerce values between types if we have proof for their equality. Since **a** and **b** are equal, we expect that **coerce** behaves like an identity function, because this is the only function of type `a → a`.

To use this in our **Literal** definition, we will add for every constructor additional parameter. We will use **Equal** to bind type **a** to appropriate type.

```
data Literal a
  = LBoolean Boolean (Equal Boolean a)
  | LString String (Equal String a)
  | LInt Int (Equal Int a)
```

To find out that this is useful, imagine that we have a value of type **Literal a**. Let's say we pattern-matched it and we are considering case with **LInt i eq**. Thus, **eq** allows us to coerce between type **a** and **Int**. Consider the following example of an eval function, which would be impossible to write with basic Literal definition 4.1, because we could not return **Int** value where value of type **a** is expected.

```
eval :: Literal a → a
eval (LInt i eq) = coerce eq i
...
```

4.1.2 User-defined type equality

Now, we define **Equal** and its operations that we mentioned before (4.1).

```
data Equal a b = Coerce (∀ f. f a → f b)
```

As a logical formula, this definition says that **a** and **b** are equal if for every property **f** that **f a** holds then **f b** also holds. The type **f** ranges over type constructors.

Starting with the easiest, we will prove reflexivity and transitivity. Note that >>> is a PureScript flipped composition operator.

```
refl :: ∀ a. Equal a a
refl = Coerce \x → x
```

```
trans :: ∀ a b c. Equal a b → Equal b c → Equal a c
trans (Coerce f) (Coerce g) = Coerce (f >>> g)
```

Now things get trickier with every defined operation. To implement **coerce** we need identity type constructor.

```
coerce :: ∀ a b. Equal a b → a → b
coerce (Coerce f) a = unId $ f $ Id a

newtype Id a = Id a
unId (Id a) = a
```

Id a gets transformed into **Id b** using function **f**, so we need to wrap with **Id**, use **f** and unwrap. The **lift** uses variation of this technique but utilising **Compose** type constructor.

```
lift :: ∀ f a b. Equal a b → Equal (f a) (f b)
lift (Coerce f) = Coerce (unCompose <<< f <<< Compose)

newtype Compose f g a = Compose (g (f a))
unCompose (Compose x) = x
```

On the other hand **symm** implementation uses **refl** to start with **Equal a a**, map it using f to get **Equal a b** and follow with **FlipEqual** to get **Equal b a**.

```
symm :: ∀ a b. Equal a b → Equal b a
symm (Coerce f) = unFlipEqual $ f $ FlipEqual $ refl
  where
  (_ :: FlipEqual a a) = FlipEqual refl
  (_ :: FlipEqual a b) = f $ FlipEqual refl
  (_ :: Equal b a) = unFlipEqual $ f $ FlipEqual refl

newtype FlipEqual a b = FlipEqual (Equal b a)
unFlipEqual (FlipEqual x) = x
```

4.2 Existential types

Existential types [10] express type abstraction. Their typical application is typing abstract data types, objects, and implicitly heterogeneous lists. We can use existentials to define abstract data types — structures with operations whose implementation is hidden [13].

For example, let us implement a purely functional counter, which allows us to create a **new** counter, **get** an integer value from a counter and increment (**inc**) its value. We want to hide its implementation and allow only operations that the interface exposes. We can specify the type signature for our counter as follows.

```
newtype Counter = exists a. Counter
  { new :: a, get :: a → Int, inc :: a → a }
```

Unfortunately, this syntax with **exists** is not allowed by PureScript. Let us rewrite the **Counter** type and provide an implementation for it. Then we discuss how to substitute **exists**.

```
newtype Counter a = Counter
  { new :: a, get :: a → Int, inc :: a → a }

myCounter :: Counter Int
myCounter = Counter { new: 0, get: \i → i, inc: \i → i + 1 }
```

Now the **myCounter** can be used to manipulate **Int**s. This is because the type parameter **a** is not abstract, but concrete — Int. To hide the implementation and make **a** abstract assume that we have following operations.

```
data Exists f

mkExists :: ∀ f a. f a → Exists f
runExists :: ∀ f r. (∀ a. f a → r) → Exists f → r
```

The **mkExists** function lets us hide the implementation. We provide a concrete value of type **f a** and we get **Exists f**, where **a** is abstract. We can use **Exists** type constructor to replace **exists a. Counter a** with **Exists Counter**. Now to use the implementation we need to provide a function that is polymorphic over **a** (so it needs to work for any **Counter**) and pass it to the **runExists**. This ensures us that we cannot inspect the concrete implementation under the **mkExists**.

4.2.1 Heterogeneous lists

In the Section 3.3 we used the type constructor **Exists** to implement heterogeneous lists with **Expr a** values in it.

The following example shows how we can use the existential abstraction to create a list with expressions of different types. Afterwards we can use it to create a **String** representation for each value using **runExists**.

```
literals :: Array (Exists Literal)
literals = [ mkExists $ LBoolean true identity
           , mkExists $ LInt 1 identity
           ]

strLits :: Array String
strLits = map (runExists showLiteral) literals

showLiteral :: ∀ a. Literal a → String
showLiteral = case _ of
```

```
LBoolean b _ → show b
LString s _ → "'" <> s <> "'"
LInt i _ → show i
```

To finish things off we provide an implementation for **Exists**. There is no built-in **exists** in PureScript that could enable us to write existential types like **forall** does for universal quantification. But we can manage it with just **forall**.

```
data Exists f = Exists (∀ r. (∀ a. f a → r) → r)

mkExists :: ∀ f a. f a → Exists f
mkExists fa = Exists \g → g fa

runExists :: ∀ f r. (∀ a. f a → r) → Exists f → r
runExists g (Exists un) = un g
```

We hide the concrete value of type **f a** under a function that applies the function called **g** to it, which treats the type parameter **a** as abstract. In order to unpack the value we need to provide such function **g** in the **runExists**.

4.3 Row-generic programming

Recall the **selectFrom** function defined in the Section 2.2. Its second argument is a function where we specify the query using a record with the same labels as in the type of the given **Table**, which represents columns from the database table. This record is a representation of the row type in the **Table** definition.

4.3.1 PureScript records

Row type in the **Table** definition is present only in the type signature. In order to create its value representation we need to get acquainted with primitives [12] that PureScript provides to work with row types. The **kind** for row types is denoted as **# Type**. **Symbol** is the kind for labels in a record. The **Lacks** type class asserts that a label does not occur in a given row.

```
class Lacks (label :: Symbol) (row :: # Type)
class Cons (label :: Symbol) (a :: Type)
           (tail :: # Type) (row :: # Type)
           | label a tail → row, label row → a tail
```

The **Cons** type class is a 4-way relation, which says that **row** is obtained from **tail** by inserting a new **label** with a value of the type **a**. It is defined with the use of *functional dependencies* [7]. This extension lets us specify explicit dependencies between the parameters. The notation "| label a tail → row" indicates that parameters **label**, **a** and **tail** uniquely determine the **row** parameter. Similarly **label** and **row** determine **a** and **tail**.

Now we can try to implement simple fold over records: function that dumps a record to string. Simple function will not work, since we have to do different things depending on the given record. In order to solve this problem we need to write a type class parameterized with a row type.

```
class ShowRecord (r :: # Type) where
  showRecord :: { | r } → String
```

Type constructor for a record is: **data Record :: # Type -¿ Type**, but instead of writing **Record (a :: A, b :: B | tail)** it is recommended to use the syntactis sugar with curly braces **{ a :: A, b :: B | tail }**.

We would like to write two instances of the **ShowRecord** type class. The first would handle the case in which a record is empty. The second instance would consider a row type with at least one element in it, which is denoted in types as follows: **Cons label a tail row**. Problem occurs during implementation for empty row.

```
instance showRecordNil :: ShowRecord () where
  showRecord {} = ""
```

It is forbidden to use the type for empty row **()** in type class instance. To fix this, PureScript provides **RowList**, a type level list that represents an ordered view of a row of types.

4.3.2 RowList

We have to base our induction on something simpler than row types. **RowList** is a typelevel list representation for a row type. For example **(b :: Int, a :: String)** and its RowList counterpart **Cons "a" String (Cons "b" Int Nil)**. It determines the order in which we will visit labels, because they are sorted in the output list.

The idea is that instead of walking inductively over a row type, we will create its typelevel list representation and base our recursion on it while having original record to extract values from it.

```
kind RowList
data Cons :: Symbol → Type → RowList → RowList
data Nil :: RowList
class RowToList (r :: # Type) (rl :: RowList) | r → rl
```

Cons is similar to the earlier one, but this one is not a type class but a type constructor for **RowList**. Type class **RowToList** let's us create **RowList** from given row type.

We have to tweak our type class and how it would work. Instead of folding over a row type, we will fold over a RowList. But we need original record, because

we need values from it. So the general idea described earlier stays almost the same. First lets change our type class and implement first instance.

```
class ShowRecord (rl :: RL.RowList) (r :: # Type) where
  showRecord :: RLProxy rl → { | r } → String

instance showRecordNil :: ShowRecord RL.Nil r where
  showRecord _ _ = ""
```

ShowRecord is extended with **RowList** and **showRecord** function gets additional parameter. This is just a proxy for the **RowList**, since it exists only as a type we cannot avoid using **RLProxy**. Instance for **Nil** is simple enough, it is just returning empty string.

```
else instance showRecordCons
  :: ( Cons label a rowTail row, Show a
     , IsSymbol label, ShowRecord tail row
     ) ⇒ ShowRecord (RL.Cons label a tail) row
where
showRecord _ record = reflectSymbol _sym <> ": "
  <> show head <> " " <> showRecord tail record
    where _sym = (SProxy :: SProxy label)
          tail = (RLProxy :: RLProxy tail)
          head = Record.get _sym record
```

The second instance that handles **RL.Cons** case is somewhat verbose. The most important part is **ShowRecord (RL.Cons label a tail) row**, it says that we are considering case in which **RowList** consists of **label**, type **a** and **tail** for which we are recursively calling showRecord. **Cons label a rowTail row** is required by **Record.get** to get a value under **label**. **reflectSymbol _sym** outputs **label** representation as string.

For comfortable use it is recommended to write helper function that creates **RowList** and calls **showRecord**.

```
showRec :: ∀ r rl. RL.RowToList r rl
        ⇒ ShowRecord rl r ⇒ { | r } → String
showRec r = showRecord (RLProxy :: RLProxy rl) r

> showRec {a: 1, b: "aux", c: 1.2}
"a: 1 b: \"aux\" c: 1.2 "
```

Chapter 5

Conclusions and future work

We presented a monadic interface that allows the user to write queries for relational databases. We improved the safety of using SQL queries by providing a convenient way to write SQL-like queries that can be type checked with the host program. We managed to ensure that only correct queries go through the compilation phase.

But there are still things to improve. More SQL operations, other types for joins and such functions needs to be implemented. Extended application for nested queries would be very welcome such as **IN** and **NOT IN** functionality.

One of the main problems is the extensibility of the data types used and lack of support for user-defined data types. We would like to allow users to define their own **Literal** constructors. A known *expression problem* [15] appears here. Future work will focus on this aspect. We will try to *open* this **Literal** data type without sacrificing type safety (namely without abandoning type parameter **a** in **Literal** definition).

Type classes allow us to encode complex constraints to ensure safety. But if an error occurs and the type class is not resolved, error message rarely helps to resolve the problem. One potential solution might be to include a message for each type class that describes the potential problem. Ideally, this message should appear together with a compiler error. We need more time to investigate the issue and whether it is a satisfactory solution for the user.

Unfortunately there are still ways to write well-typed code that generates a query, which fails during execution. Recall the **groupBy** function introduced in the Section 2.3. It takes an argument of type **Col s a** and adds it to the `GROUP BY` clause in the generated SQL. So we can call **groupBy** with an actual column from a database. But it is also possible to use any supported literal value e.g. string or boolean. What does it mean to group results by a boolean literal? It turns out that PostgreSQL does not allow it. The documentation [1] says that the expression used in the `GROUP BY` clause can be a column name, ordinal number of an output column or an arbitrary expression formed from input-column values.

There is yet another issue with the **groupBy** function. When we want to use an aggregate function and return a column, we need to use **groupBy**, so that the column in the result has appropriate type for the aggregated query. But when we use **groupBy** it is not enforced by the type system that we must return only values of type **Aggr s a**, that comes from aggregate functions and the **groupBy**. Let us consider a simple query that returns columns **id** and **age** from the table **people**, but it also calls **groupBy** with **age** and discards its result.

```
selectFrom people \{ id, age } → do
  _ ← groupBy age
  pure { x: id, y: age }
```

This results in a runtime error with the following message.

```
error: column "people_0.id" must appear in the GROUP BY
       clause or be used in an aggregate function
```

The function **groupBy** implicitly changes the type of the query to an aggregation query. Such a query should require that every *selected* value appears in the `GROUP BY` clause or comes from an aggregate function.

We can think of two possible solutions to resolve this issue. One requires *indexed monads* [2]. We would add additional type parameter to the **Query** monad that would indicate: *is it an aggregate query?*. This additional type parameter would be explicitly changed by the **groupBy** function. But this approach makes it much harder to use for an end-user. Another way is to implement additional monad **AggrQuery**, that works similarly to the **Query** monad. It would be used in a new function **aggrFrom** (similar to the **selectFrom**, but it requires the **AggrQuery** to return **Aggr** values), which would replace the **aggregate** function. Obviously the **groupBy** would work only with **AggrQuery** and other operations would be available for both **Query** and **AggrQuery** monads. This solution is not desired either, it would greatly complicate the code. Clearly we need a better way that is still convenient to use and does not generate boilerplate code.

Bibliography

[1] URL: https://www.postgresql.org/docs/9.5/sql-select.html.

[2] URL: https://kseo.github.io/posts/2017-01-12-indexed-monads.html.

[3] *Domain-specific language*. URL: https://en.wikipedia.org/wiki/Domain-specific_language.

[4] Anton Ekblad. *Scoping Monadic Relational Database Queries*. URL: https://ekblad.cc/pubs/selda.pdf.

[5] Anton Ekblad. *Selda: A Haskell SQL Library*. URL: https://selda.link/.

[6] *Generalized algebraic data type*. URL: https://en.wikipedia.org/wiki/Generalized_algebraic_data_type.

[7] Mark P. Jones. *Type Classes with Functional Dependencies*. URL: https://web.cecs.pdx.edu/~mpj/pubs/fundeps.html.

[8] *LINQ*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

[9] *Object-relational mapping*. URL: https://en.wikipedia.org/wiki/Object-relational_mapping.

[10] Benjamin C. Pierce. *Types and Programming Languages*. Chap. 24: Existential Types.

[11] *Prepared statement*. URL: https://en.wikipedia.org/wiki/Prepared_statement.

[12] *PureScript type classes for working with row types*. URL: https://pursuit.purescript.org/builtins/docs/Prim.

[13] *Representing existential data types with isomorphic simple types*. URL: http://okmij.org/ftp/Computation/Existentials.html.

[14] *User-defined type equality*. URL: https://blog.janestreet.com/more-expressive-gadt-encodings-via-first-class-modules/#user-defined-type-equality.

[15] Philip Wadler. *The Expression Problem*. URL: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[16] *What is an orm?* URL: http://hibernate.org/orm/what-is-an-orm/.