Декларативное программирование клиентов Postgres в Haskell с помощью Hasql

Hasql

Высокопроизводительный драйвер PostgreSQL для Haskell.

Source: https://github.com/nikita-volkov/hasql

API: http://hackage.haskell.org/package/hasql

Экскурс

Haskell

Компилируемый чисто функциональный язык с сильной типизацией, статической проверкой и автоматическим выведением типов.

Чистая функция

Удовлетворяет двум требованиям:

1. При повторном её исполнении с теми же параметрами она всегда возвращает тот же результат.

Чистая функция

Удовлетворяет двум требованиям:

1. При повторном её исполнении с теми же параметрами она всегда возращает тот же результат. (Референциальная прозрачность).

Чистая функция

Удовлетворяет двум требованиям:

1. При повторном её исполнении с теми же параметрами она всегда возвращает тот же результат. (Референциальная прозрачность).

2. Исполнение функции не оказывает какого-либо воздействия на состояние программы и не осуществляет взаимодействия с внешним миром.

Чистая функция

Удовлетворяет двум требованиям:

1. При повторном её исполнении с теми же параметрами она всегда возвращает тот же результат. (Референциальная прозрачность).

2. Исполнение функции не оказывает какого-либо воздействия на состояние программы и не осуществляет взаимодействия с внешним миром. (Отсутствие сторонних эффектов).

Примеры чистых функций

Java:

```
public static String exclamation ( String phrase ) {
  return phrase + "!";
}
```

Haskell:

```
exclamation :: String -> String
exclamation phrase = phrase <> "!"
```

Примеры грязных функций

Референциально непрозрачная функция:

```
public static String exclamation ( String phrase ) {
  return phrase + generateRandomString() + "!";
}
```

Функция со сторонними эффектами:

```
public static String exclamation ( String phrase ) {
  launchRockets();
  return phrase + "!";
}
```

Преимущества чистых функций

- Предсказуемость
- Возможность использования и тестирования в полной изоляции

Чистые функции в языках программирования

В большинстве языков любая функция может быть грязной и это даже никаким образом не отображается в типах.

В Haskell все функции чистые!

Полиморфизм

```
public static <A> Integer listLength ( List<A> as )
```

Метод `listLength` и тип `List` можно называть полиморфными, потому что они параметризованы абстрактным типом `A`.

Полиморфизм

```
public static <A> Integer listLength ( List<A> as )
```

Метод `listLength` и тип `List` можно называть полиморфными, потому что они параметризованы абстрактным типом `A`.

Для сведения, в Haskell аналогичное объявление выглядело бы так:

```
listLength :: List a -> Int
```

Кортежи (полиморфные типы-множества)

Java:

```
Tuple2<A, B>
```

Haskell:

```
(a, b)
```

Кортежи (полиморфные типы-множества)

Для сведения, объявить эти типы можно было бы как-то так:

Java:

```
public class Tuple2<A, B> {
  public final A _1;
  public final B _2;
  // Ridiculous Java boilerplate excluded
}
```

Haskell:

```
data (,) a b = (,) a b
```

Кортежи (полиморфные типы-множества)

Java:

```
Tuple2<Integer, String>
```

Haskell:

```
(Int, String)
```

Кортежи (полиморфные типы-множества)

Java:

```
Tuple3<Integer, String, Integer>
```

Haskell:

```
(Int, String, Int)
```

Кортежи (полиморфные типы-множества)

- `(Int, String)` - кортеж, объединяющий 2 элемента: `Int` и `String`.
- `(Int, Int, Double)` - кортеж, объединяющий три элемента.
- ...

Unit

`()` - тип данных, имеющий единственное значение `()`. На словах называется "Unit".

Думать о нём можно как о кортеже из нуля элементов.

Используется для указания отсутствия значения.

Как взаимодействовать с внешним миром?

Ведь все функции в Haskell оперируют значениями, референциально прозрачны и не производят сторонних эффектов...

Как взаимодействовать с внешним миром?

Решение - завернуть взаимодействие с внешним миром в отдельный тип данных:

```
IO a
```

IO

`IO a` - описание действия с внешним миром, возвращающего результат типа `a`.

Примеры использования IO

```
putStrLn :: String -> IO ()
```

```
readFile :: FilePath -> IO String
```

Комбинирование действий IO

Имея следующие функции:

```
putStrLn :: String -> IO ()
```

```
readFile :: FilePath -> IO String
```

```
(>>=) :: IO a -> (a -> IO b) -> IO b -- "flatMap" anybody?
```

Можем их скомбинировать так:

```
outputFile :: FilePath -> IO ()
outputFile filePath =
  readFile filePath >>= putStrLn
```

Нотация "do"

```
outputFile :: FilePath -> IO ()
outputFile filePath =
  readFile filePath >>= putStrLn
```

Идентично следующему объявлению с помощью нотации "do":

```
outputFile :: FilePath -> IO ()
outputFile filePath =
  do
    contents <- readFile filePath
    putStrLn contents
```

Нотация "do" для более сложных комбинаций

```haskell
outputTwoFiles :: FilePath -> FilePath -> IO ()
outputTwoFiles filePath1 filePath2 =
  do
    contents1 <- readFile filePath1
    contents2 <- readFile filePath2
    putStrLn (contents1 <> "\n" <> contents2)
```

Напоминает императивный код, не так ли?

Однако это, по-прежнему, чистая функция из двух значений `FilePath` в значение `IO ()`. Напоминаю, что, в отличие от императивных языков, функция сама никаких обращений к файловой системе не делает, а лишь создаёт спецификацию того, что нужно сделать.

Как сделать качественную библиотеку?

Простота

Простота

Вот такая функция простая?

```
execParams :: Connection                       -- connection
           -> ByteString                       -- statement
           -> [Maybe (Oid, ByteString, Format)] -- parameters
           -> Format                           -- result format
           -> IO (Maybe Result)                -- result
```

Простота

Вот такая функция простая?

```
execParams :: Connection                       -- connection
           -> ByteString                       -- statement
           -> [Maybe (Oid, ByteString, Format)] -- parameters
           -> Format                           -- result format
           -> IO (Maybe Result)                -- result
```

Hell no!

В чём проблема той функции?

В чём проблема той функции?

Количество вещей, о которых нам надо заботиться.

Задача 1

Сокращать количество проблем, которые нам надо учитывать в каждом случае.

Взаимодействие с внешним миром

Взаимодействие с внешним миром

- "Боже, а что если что-то случится между данными выражениями?"
- "А что, если программа находится в неожидаемом состоянии?"
- "А что, если произойдёт сбой или где-нибудь выкинется исключение?"

В чистоте спасение!

- "Боже, а что если что-то случится между данными выражениями?"
- "А что, если программа находится в неожидаемом состоянии?"
- "А что, если произойдёт сбой или где-нибудь выкинется исключение?"

Haskell позволяет чётко ограничивать код, взаимодействующий с внешним миром.

Код, не взаимодействующий с внешним миром, вообще не сталкивается с упомянутой категорией проблем, и это прекрасно!

Задача 2

Минимизировать и изолировать IO.

Абстракция

Абстракция может кусаться, если перестараться!

Чрезмерные абстракции или страдают от недостатка возможностей, или "протекают".

Протекающие абстракции

Автор абстрагировался над какими-то действиями, а потом окольными путями пытается вернуть потерянный функционал.

Например, Hook в типичном ORM.

Протекающие абстракции

Результат - концептуальная каша.

Задача 3

Абстрагироваться насколько возможно, но дисциплинированно.

Подходы

Поиск общего между реляционной и функциональной моделями

Поиск общего

Задача - найти общие черты, не теряя в возможностях.

Пойдём от общего к частному.

Поиск общего

- Нет непосредственного соответствия для базы данных (схемы)

Поиск общего

- Нет непосредственного соответствия для базы данных (схемы)
- Нет непосредственного соответствия для отношений

Поиск общего

- Нет непосредственного соответствия для базы данных (схемы)
- Нет непосредственного соответствия для отношений
- Нет непосредственного соответствия для таблиц

Поиск общего

- Нет непосредственного соответствия для базы данных (схемы)
- Нет непосредственного соответствия для отношений
- Нет непосредственного соответствия для таблиц
- Нет непосредственного соответствия даже для строк

Поиск общего

- Нет непосредственного соответствия для базы данных (схемы)
- Нет непосредственного соответствия для отношений
- Нет непосредственного соответствия для таблиц
- Нет непосредственного соответствия даже для строк

Всё то же касается и любого ОО-языка. Именно поэтому ORM заведомо обречены либо быть серьёзно урезанными в возможностях, либо быть текущей абстракцией!

Поиск общего

- ЕСТЬ соответствие на уровне значений!

Поиск общего. Примитивы

- `BIGINT` от Postgres непосредственно соответствует `Int64` в Haskell
- `NUMERIC` - `Scientific`
- `TIMESTAMPTZ` - `UTCTime`

Вообще, это распространяется на все примитивные значения.

Поиск общего. Массивы

Не проблема! Соответсвуют `Vector`, `[]` или любым другим данным, которые можно свёртывать и развёртывать (fold).

Многоуровневые массивы не составляют исключения.

Поиск общего. Композитные типы

Тоже не проблема. У Haskell есть кортежи и пользовательские типы.

Поиск общего. HStore, JSON, JSONB

Ну, вы поняли. Всё есть.

Минимизация эффектов

Всё, что нужно клиенту базы данных:

Минимизация эффектов

Всё, что нужно клиенту базы данных:

- Поддерживать соединение с БД

Минимизация эффектов

Всё, что нужно клиенту базы данных:

- Поддерживать соединение с БД
- Исполнять SQL

Минимизация эффектов

Всё, что нужно клиенту базы данных:

- Поддерживать соединение с БД
- Исполнять SQL

Всё остальное может быть декларативным.

Hasql, собственно

Взаимодействие с внешним миром

Взаимодействие с внешним миром в Hasql осуществляется следующими функциями:

Взаимодействие с внешним миром

Взаимодействие с внешним миром в Hasql осуществляется следующими функциями:

- `Hasql.Connection.acquire` - установить соединение

Взаимодействие с внешним миром

Взаимодействие с внешним миром в Hasql осуществляется следующими функциями:

- `Hasql.Connection.acquire` - установить соединение
- `Hasql.Connection.release` - закрыть соединение

Взаимодействие с внешним миром

Взаимодействие с внешним миром в Hasql осуществляется следующими функциями:

- `Hasql.Connection.acquire` - установить соединение
- `Hasql.Connection.release` - закрыть соединение
- `Hasql.Session.run` - взаимодействовать с базой данных

Взаимодействие с внешним миром

Взаимодействие с внешним миром в Hasql осуществляется следующими функциями:

- `Hasql.Connection.acquire` - установить соединение
- `Hasql.Connection.release` - закрыть соединение
- `Hasql.Session.run` - взаимодействовать с базой данных

Всё.

Абстракция над запросом

Чего несёт с собой следующая информация?

```
SELECT name, birthday
  FROM person
  WHERE birthday >= $1 AND gender = $2
```

Абстракция над запросом

Чего несёт с собой следующая информация?

```
SELECT name, birthday
  FROM person
  WHERE birthday >= $1 AND gender = $2
```

Что это строчное значение, напоминающее SQL-выражение, содержащее параметры `$1` и `$2`.

Абстракция над запросом

Чего несёт с собой следующая информация?

```
SELECT name, birthday
  FROM person
  WHERE birthday >= $1 AND gender = $2
```

Если типизировать, то это `String`, например.

Абстракция над запросом

Чего несёт с собой следующая информация?

```
SELECT name, birthday
  FROM person
  WHERE birthday >= $1 AND gender = $2
```

Если типизировать, то это `String`, например.

Маловато будет!

Абстракция над запросом

Чего несёт с собой следующая информация?

```
SELECT name, birthday
  FROM person
  WHERE birthday >= $1 AND gender = $2
```

А что, если тип был бы таким:

```
Query  (Day, Gender)  (Vector (Text, Day))
 --     ^ Параметры     ^ Результат
```

Абстракция над запросом

Haskell:

```
Query (Day, Gender) (Vector (Text, Day))
```

Java:

```
Query<Tuple2<Date, Gender>, ArrayList<Tuple2<String, Date>>>
```

Абстракция над запросом

`Query` - полиморфный тип следующего вида:

Haskell:

```
Query parameters result
```

Java:

```
Query<Parameters, Result>
```

Абстракция над запросом

Объявить `Query` можно при помощи этой функции:

```
statement :: ByteString          -- The SQL
          -> Encoders.Params a   -- The parameters encoder
          -> Decoders.Result b   -- The result-set decoder
          -> Bool                -- The "prepared" flag
          -> Query a b
```

Абстракция над запросом

Пример объявления `Query`:

```
selectOfPersonsBornAfter :: Query Day (Vector (Text, Day))
selectOfPersonsBornAfter = statement sql encoder decoder True
  where
    sql = "SELECT name, birthday FROM person WHERE birthday >= $1"
    encoder = Encoders.value Encoders.date
    decoder = Decoders.rowsVector row
      where
        row = liftA2 (,) name birthday
          where
            name = Decoders.value Decoders.text
            birthday = Decoders.value Decoders.date
```

Абстракция над запросом

Итак, `Query` - это полностью энкапсулированная, не протекающая и не ограничивающая нас ни в чём абстракция над рядом связанных проблем:

- SQL-выражение,
- Подставление и кодирование параметров,
- Декодирование и разбор результата,
- Prepared Statements.

Абстракция над запросом

Итак, `Query` - это полностью энкапсулированная, не протекающая и не ограничивающая нас ни в чём абстракция над рядом связанных проблем:

- SQL-выражение,
- Подставление и кодирование параметров,
- Декодирование и разбор результата,
- Prepared Statements.

Спросите: "И чего такого?" Да то, что после того, как `Query` написан, ни одна из этих проблем Вас больше не волнует!

Абстракция над запросом

А ещё!..

`Query` является Профунктором, что, несомненно, доставляет!

DSL кодеков

- Полностью декларативный и изолированный от сторонних эффектов!
- Комбинируемый и гибкий! Поддерживает данные любой сложности.
- Максимально производительный благодаря непосредственному использованию ресурсов и бинарного формата передачи данных.

Транзакции

Типичные проблемы:

- Транзакции не комбинируемы. Невозможно взять две транзакции и объединить в одну.
- Обработка конфликтов и откаты не дружат со сторонними эффектами.
- Длительные операции на стороне клиента в контексте транзакции удерживают замки на БД, что никогда не есть хорошо.

Решение

Решение - сделать транзакции чище, ограничив допустимые действия в контексте транзакции только на общение с БД, на которой выполняется транзакция.

Иными словами, никаких side-effects, вроде обращения по сети куда-то или записи в файл, мутации состояния чего-либо в программе и тп.

Transaction

В итоге, получаем следующую абстракцию:

```
Transaction a
```

Которая умеет выполнять только следующие две операции:

```
sql :: ByteString -> Transaction ()
    -- Исполняет непараметризованный SQL,
    -- который может содержать множество стейтментов.
    -- Ничего при этом не возвращает.
```

```
query :: a -> Query a b -> Transaction b
      -- Исполняет Query, передавая ему параметры.
      -- Возвращает результат Query.
```

Преимущества абстракции Transaction

Transaction комбинируема

Эту абстракцию можно комбинировать тем же способом, что и IO:

```
transferMoneyTransaction :: Int -> Int -> Scientific -> Transaction ()
transferMoneyTransaction accountID1 accountID2 amount =
  do
    query (accountID1, amount) takeMoneyQuery
    query (accountID2, amount) putMoneyQuery

transferMoneyTwiceTransaction :: Int -> Int -> Scientific -> Transaction ()
transferMoneyTwiceTransaction accountID1 accountID2 amount =
  do
    transferMoneyTransaction accountID1 accountID2 amount
    transferMoneyTransaction accountID1 accountID2 amount
```

Transaction абстрагируется над конфликтами

Заявленные свойства позволяют автоматически откатывать и перезапускать транзакции в случае конфликтов. Так это и происходит.

Библиотека транзакции

Данная абстракция представлена в библиотеке "hasql-transaction".

Source: https://github.com/nikita-volkov/hasql-transaction

API: http://hackage.haskell.org/package/hasql-transaction

Производительность Hasql

Бьёт всех конкурентов в 2 раза и более!

Старые бенчмарки:
https://nikita-volkov.github.io/hasql-benchmarks/

Никита Волков