Если вы видите что-то необычное, просто сообщите мне.

WELCOME TO ALL THOSE LEARNING HASKELL

Содержание

Так как вы только начинаете изучать Haskell или борьба выяснить как совместить то что вы изучили с практикой написания программой реального мира, или даже закопаться еще дальше в функциональное программирование во все её углы экосистемы, мы все знает боль отчаяния, когда гуглишь кучи эзотерических идей о которых мы не слышали, отчаянные попытки соединить отдельные кучоски и понять Haskell из маленьких кусочков информации разбросанных по всему интернету.

Пробираясь через бумаги непостижимых изучений, черезмерно педатичные вопросы на StackOverlow и обширные блог посты, борясь за эту искру, за момент когда "ага" и всё встаёт на место. Звучит знакомо?

Ничего из этого тут вы не встретите. Тут, в этом блоке, где вы найдете не вздорных объяснений идей Haskell, написаных простым английским, связанным с реальным миром программирования для которых вы будете их использовать. Ни сумашедшей математики, ни пронизанной формализмом плавителей мозгов, которые только ученые способны понять. Haskell объясняется для простых работников.

Звучит не плохо? Отлично, погружаемся. Ниже несколько статей для начала.

Применение Haskell к реальным проблемам

- Ужасно простой веб стек на Haskell
- Какую типобезопасную библиотеку базы данных вы должны использовать?
- Вещи которые должен пройти инженер когда изучает Haskell
- Задачи для понимания линзы

Базовые идеи

- Получение монады состояния из изходных принципов
- Получение монады чтения из изходных принципов
- Получение монады записи из изходных принципов
- Как делать базовый отлов ошибок и логирование в Haskell

Начинающий уровень Haskell

- Вы уже умны чтобы писать на Haskell
- Путь опыта Haskell

Философия высокоуровнего дизайна

- Как Haskell делает вашу жизнь проще?
- Разрешить нельзя запретить: как спроектировать программу Haskell
- Попробуем расширенные штуки-дрюки
- Список статей Haskell о хорошем дизайне, хорошем тестировании.

Ужасно простой веб стек на Haskell

В Haskell есть большой выбор распространненных библиотек для всех простых нужд, от логирования доступа в базу данных до маршрутизации и подъема веб-сервера. Всегда хорошо иметь свободу выбора, но если вы просто начинаете, количество решений может сильно мешашть. Может быть вы даже еще не уверены, что вы способны понять важное различние между выборами. Вам нужно сделать запрос в бд. Вам нужны гарантирование строгие имена колонки и глубокое SQL встраивание которое Squeal дает вам, или вы возможно предпочтете относительную простоту или Opaleye с типобезопасностью? Или, может. лучше использовать postgresql-simple и оставить всё проще-простого? Или что насчет использования Selda? Или что на счет....

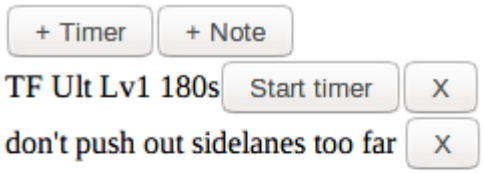
Возможность показать что вам не нужно проводить часы насколько ваш стек крут, - это возможность научиться самому. Я написал пример веб-приложения использующий самые простые библиотеки, которые я смог найти. Если вы не уверенны, как строить реальное приложение на Haskell, почему бы не начать с этого? Я умышленно постарался составить кодову базу максимально простой.

Пройдёмся по библиотекам которые я выбрал и что вы должны ожидать от них, ну и понять что это за приложене.

Хорошо, что же такое это ваше веб-приложеение?

Это сайт, где пользователи могут создать таймеры и записки.

Например, один из примеров использования может быть готовка: Кому-то нужно настройки различные таймеры для отслеживания прогресса различных реагентов, или составить заметки о вещах о которых необходимо беспокоиться, вещи которые могут быть улучшены в следующий раз повторя рецепт. Другой вариант может быть игра MOBA, как LoL или Dota2, где можно открыть страничку во втором мониторе для ослеживания кулдаунов, а так же записи о том как противостоять противникам и их кулдаунам во время битвы.



Давайте я покажу:

- Сессии, пользователи должны иметь возможность обновить страницу, или уйти и вернуться, а элементы должна всё еще оставаться.
- Постоянство и доступ к базе данных, нам нужно хранить таймеры и записки для каждого пользователя. Еще тонкость, это таймеры должны зранить оставшеся время. (Что если это 30 минутный таймер, а пользователь случайно закрыл вкладку?)
- Настройка во время работы, так как мы не можем хардкодить информацию о подключении к бд.
- Логирование. Само-собой разумеющееся для веб-приложения.

Исходный код приложения.

Что это за библиотеки?

Маршрутизация веб-сверера: Spock

Spock - из-за простоты использования. Если вы когда-либо пользовали Sinatra Ruby, Spock должен быть очень похож. Он так же идеть с обработкой сессии из коробки, что очень здорово.

Для примера, определим сервер с несколькими маршрутами для обратной отправки HTML и Json может выглядить следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}

import Web.Spock as Spock
import Web.Spock.Config as Spock
import Data.Aeson as A

main :: IO ()
main = do
  spockCfg <- defaultSpockCfg () PCNoDatabase ()
  runSpock 3000 $ spock spockCfg $ do
    get root $ do
      Spock.html "<div>Hello world!</div>"
    get "users" $ do
      Spock.json (A.object [ "users" .= users ])
    get ("users" <//> var <//> "friends") $ \userID -> do
      Spock.json (A.object [ "userID" .= (userID :: Int), "friends" .= A.Null ])

  where users :: [String]
        users = ["bob", "alice"]
```

Доступ к базе данных: postgresql-simple

postgresql-simple - просто позволяет вам запустить SQL запрос к вашей базе данных, с минимумом дополнительных излишеств, таких как защита против injection-аттак. Он просто делает, что вам нужно, не больше.

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

userLoginsQuery :: Query
userLoginsQuery =
  "SELECT l.user_id, COUNT(1) FROM logins l GROUP BY l.user_id;"

getUserLogins :: Connection -> IO [(Int, Int)]
getUserLogins conn = query_ conn userLoginsQuery
```

Настройки: configurator

configurator читает конфигурационные файлы и парсит их в типы данных Haskell. Немного больше чем просто обычная читалка файлом конфига. имеет несколько трюков в рукаве. Конфигурационные атрибуты могут быть вложенными для группировки, так же конфигуратор предоставляет быструю перезагрузку при изменении настроек конфига, если это нужно.

Пример конфиг файла.

```
app_name = "The Whispering Fog"

db {
  pool {
    stripes = 4
    resource_ttl = 300
  }

  username = "pallas"
  password = "thefalloflatinium"
  dbname = "italy"
```

```
}
```

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Configurator as Cfg
import Database.PostgreSQL.Simple

data MyAppConfig = MyAppConfig
  { appName :: String
  , appDBConnection :: Connection
  }

getAppConfig :: IO MyAppConfig
getAppConfig = do
  cfgFile <- Cfg.load ["app-configuration.cfg"]
  name <- Cfg.require cfgFile "app_name"
  conn <- do
    username <- Cfg.require cfgFile "db.username"
    password <- Cfg.require cfgFile "db.password"
    dbname <- Cfg.require cfgFile "db.dbname"
    connect $ defaultConnectInfo
      { connectUser = username
      , connectPassword = password
      , connectDatabase = dbname
      }
  pure $ MyAppConfig
    { appName = name
    , appDBConnection = conn
    }
```

Логирование: fast-logger

fast-logger - предоставляет разумно простое в использовании средство логирования. В примере веб приложения я просто использую для вывода `stderr`, но у библиотеки есть возможность для логирования в файлы в том числе. Так как есть множество типов, в большинстве вы захотите определить функции-помошники которые просто принимают `LoggerSet` и сообщение которое нужно записать.

```
import System.Log.FastLogger as Log

logMsg :: Log.LoggerSet -> String -> IO ()
logMsg logSet msg =
  Log.pushLogStrLn logSet (Log.toLogStr msg)

doSomething :: IO ()
doSomething = do
  logSet <- Log.newStderrLoggerSet Log.defaultBufSize
  logMsg logSet "message 1"
  logMsg logSet "message 2"
```

Генерация HTML: blaze-html

Так как у нас не будет большого количества HTML, которое нужно будет генерировать в этом проекте, стоит упомянуть blaze-html for the parts that I did need.

Это естественно просто мелкое встраивание HTML в Haskell DSL. Если вы можете написать HTML, вы уже знаете как использовать библиотеку.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.ByteString.Lazy

import Text.Blaze.Html5 as HTML
import Text.Blaze.Html5.Attributes as HTML hiding ( title )
import Text.Blaze.Html.Renderer.Utf8 as HTML

dashboardHTML :: HTML.Html
dashboardHTML = HTML.html $
  HTML.docTypeHtml $ do
    HTML.head $ do
      HTML.title "Timers and Notes"
      HTML.meta ! HTML.charset "utf-8"
      HTML.script ! HTML.src "/js/bundle.js" $ ""
    HTML.body $ do
      HTML.div ! HTML.id "content" $ ""
```

```
dashboardBytes :: ByteString
dashboardBytes = HTML.renderHtml dashboardHTML
```

Сборка и фротентд: make + npm

Да да, это не библиотеки. Но всё же, нам нужно что-то похожее на JavaScript фронтенд, так как таймеры должны обновляться в реальном времени. Webpack создает JS пакет, в то время как Make собирает результат приложения.

Я не хочу об этом много говорить. Есть множество источников про использование и того и другого инструмента.

Мне нужно это использовать?

Нет, конечно же нет. Если вы исследуете Haskell изначально, то вам возможно интересно. Не позволяйте мне вас удержать или диктовать что вы должны делать. Пока это приложение работает, многие части его могут считаться неидиоматическими для производства Haskell. Для примера, множество хаскеллят, скорей всего, будут использовать `Servant` вместо `Spock` для создания API точек доступа. Если вы заинтересовались еще чем-то, то должны следовать дальше.

Считайте эти библиотеки и это приложение как точку отсчёта. Я прошу вас использовать этот код как возможность изучить и понять как и что работает, затем начать мастерить. Одна из прекраснейших вещей про Haskell это то, насколько просто перерабатывать или обновлятся без проблем что-то сломать. Как только вы сделаете это приложение, почему бы не заменить части на более интересные библиотеки которые дают вам больше гарантий. как пусть постеменного изучения Haskell?

- Upgrade the DB access to use a type-safe query library instead of postgresql-simple. I recommend Opaleye!

- Upgrade the API definition to use Servant instead of Spock.
- Add automated testing using QuickCheck or hedgehog. For instance, you could test the property that every error response from the server also sends back a JSON error message. And you could even try replacing the frontend and build system.
- Upgrade the frontend code to use PureScript or Elm instead of vanilla JavaScript.
- Upgrade the build system to use Shake instead of Make to make things more robust.

Какую типобезопасную библиотеку базы данных вы должны использовать?

Beam или Squeal: что лучше? Или может быть вы слышали про отличную штуку Selda или Opaleye. Множество мнений, редкие руководства.

Чтобы ответить на вопрос, я взял 7 популярных библиотек для базы данных и реализовал один и тот же проект, на каждой из них.

Участники:

- Beam
- Opaleye
- Squeal
- Persistent + Esqueleto
- Hasql
- Groundhog
- Selda

Почемы мы используем эти библиотеки?

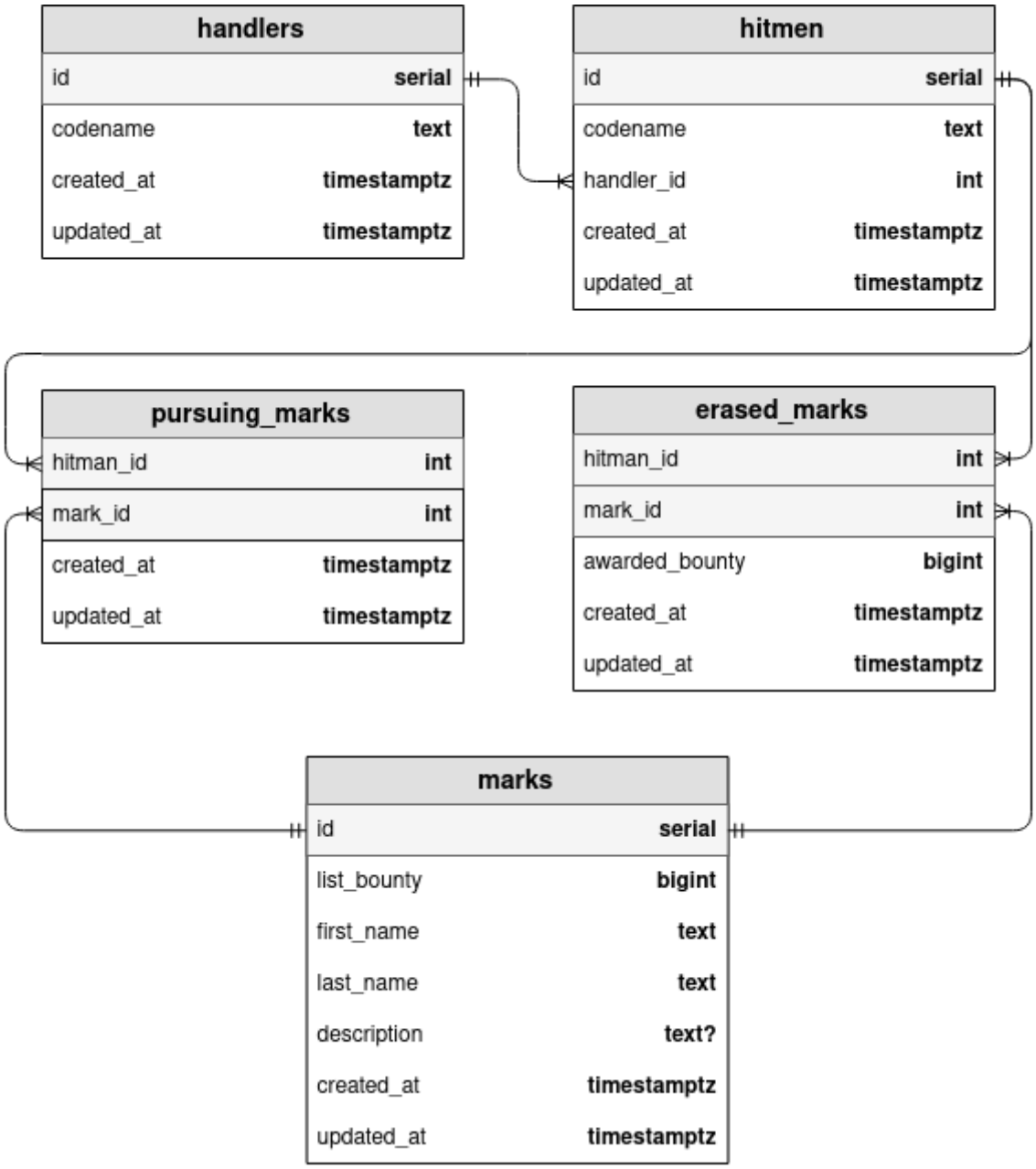
Вполне возможно, что вы как и я сагитировались на преимущества строгой типизации чтобы знать, чтобы писать приложения лучше.(Если нет, то на данный момент будем считать так) Ваше приложение, допустим, требует возможность хранить данные постоянно. Вы можете использовать `postgresql-simple` для чего угодно, но есть небольшое смущение в том, что

придется писать чистые SQL запросы, и надеятся что они работают в языке который хочет делать больше.

Но к счатсью, есть множество возможностей Haskell экосистемы для типобезопасных SQL запросов. Вы можете убедиться, что вы не забыли вылкючить столбцы в ваш вывод, или получить их в неправильном порядке. Вы можете даже переисползовать запросы легко, сочиная их напрямую в Haskell, затем создавая простые запросы для отправки на бэкенд дб. Всё с проверкой типов помогает вам, убеждаясь что вы не создали неправильный запрос и вызвали ошибку выполнения.

Что педставялет из себя проект в примере?

Мы создаем бэкенд для веб-сайта для профессионального киллера. Представим Fiverr или Upwork, только где платят за убийство. У каждого клиллера есть обработчик(один обработчик может обрабатывать несколько клиллеров), и киллер преследует "отметки". Как только работа выполнена, киллер отмечает цели как "удаленные". Мы смоделируем, как это будет в базе данных. Добавленная сущность `erased_marks` вовсе не удаляет `pursuing_marks`.



Для нашего занятия, мы используем Postgres как бэкенд нашей бд. В тоже время библиотеки которые мы рассматриваем(к примеру Beam) доволно агностични относительно самой бд, и может использоватьсья для любой базыданных, другие (как Opaleye и Squeal) работают только с Postgres.

Чтобы обслуживать данные нашего бэкенда, нам нужны запросы. Уточним, это запросы изменяются от простых до запросов которые содеражт объединения, подзапросы, и агрегаторы.

- Получить всех киллеров
- Получить всех киллеров которые предследуют целей(то есть имет неудаленные цели)

- Получить все цели которые были уничтожены за данное время.
- Получить все отметки которые были уничтожены за время определенным киллером.
- Получить общее вознаграждение для всех клиллеров
- Получить общее вознаграждение для опредленного киллера
- Получить для всех киллеров последнее убийство
- Получить последнее убийство для определенного клиллера
- Получить цели, которые имеют только одного преследователя.
- Получить все "возможные отметки"(то есть отметки которые киллер удаляет без дополнительного преследования цели)

Мы так же хотим написать обновления и вставки каждый библиотеки чтобы посмотреть как они обрабатывают их. Это должно напрячь все возможности запросов каждой библиотеки чтобы найти грубые точки.

Ну чтож со всем этим прыгаем в сравнение.

Beam

Beam - это попытка решить проблемы типо-безопасности SQL совместим с абсолютным игнорированием бэкенда. Способ с которым это получается решается добавлением типа параметра в каждый запрос для бэкенда и имеет множество типов классов что определения функциональности. К сожалению, он устарел, очень быстро, особенно когда вам нужно использовать типы классов с именами вроде `HasSqlEqualityCheck backend` - `Int64` and `BeamSqlT071Backend backend`, так как бог запрещает вам использовать `BIGINT`.(А вам нужно обе в одном запросе, между прочим)

Это можно обойти простым игнорированем бекенда целиком и указать определенный бэкенд в каждом запросе, но даже тогда тип вашего запроса будет кучей непостижимых `QExpr`, `QAgg`, и что там еще есть из параметров для запроса определеня объема.

Когда вы прошли это всё, собрка и создание запросов в `Beam` довольно приятны, подзапросы могут быть переиспользованны достаточно легко используя сущности `Beam` монад для этих запросов, объединений в одну строку. Легко определить запросы которые возвращают к этому еще и кортоежи, без надобности определения новых типов. Это потому что типы `Beam`

противны. Вам нужно обхдить все эти псевдонимы и семьи умных типов но в тоже время... к чему Intepeживания, когда другие библиотеки делают всё тоже самое без костылей?

Еще одни красный флаг - нет простого путит получить определенных строк, поэтому нужно явно собирать и группировать все толбцы, что вам нужны. `Beam` очень похож на не правиольный тип `SUM`, по крайней мере в Postgres, он не правильно меняет типа колонок. Для примера, в Postgres обход по `BIGINIT` колонке выдаст резульат `NUMERIC`, но в стране `Beam` если у вашего типа данных атрибут `Int64`, то он всё равно постарается выдать вам `Int64`, что становится ошибкой выполнения.

В любом случае я не могу рекомендовать `Beam`, он слишком привередлив.

```
latestHits :: Q Postgres HitmanDB s
  ( HitmanT (QExpr Postgres s)
  , MarkT (QExpr Postgres s)
  )
latestHits = do
  hitman <- allHitmen
  mark <- allMarks

  (minHID, minCreated, minMarkID) <- minID
  (latestHID, latestCreated) <- latest

  guard_ (minHID ==. latestHID)
  guard_ (just_ minCreated ==. latestCreated)
  guard_ (minHID ==. HitmanID (_hitmanID hitman))
  guard_ (minMarkID ==. just_ (_markID mark))

  pure (hitman, mark)

  where minID = aggregate_ (\em -> ( group_ (_erasedMarkHitman em)
                                   , group_ (_erasedMarkCreatedAt em)
                                   , min_ (getMarkID $ _erasedMarkMark em)
                                   ))
                           allErasedMarks

        latest = aggregate_ (\em -> ( group_ (_erasedMarkHitman em)
                                    , max_ (_erasedMarkCreatedAt em)
                                    ))
```

```
                          allErasedMarks


        getMarkID (MarkID id) = id
```

Opaleye

Opaleye это SQL DSL разработанная для Postgres. Из коробки, это решение «просто работает» без всяких настроек или полного игнорирования частей ядра библиотеки.

Написание запросов работает. Создание запросов работает. Типы (относительно) простые и с ними легко работать.

Главное отличие между Opaleye и другими библиотеками это способ определения схемы таблицы. Все определения схем выполняются на уровне определений, таким образом, чаще всего не зависят от самого типа домена. Это связано со способом настройки(используется product-profunctros), вы можете легко абстрагировать общие столбцы. На пример: я использовал это чтобы абстрагировать определения `created_at` и `updated_at` временные отметки. Дальше, Opaleye, показывает отличия между временем записи и временем чтения данных, ну что ж, это просто, скажем, для определенных колонок нельзя писать с помощь `insert/updates`(как было сказано выше с отметками времени).

Так как прошлые версии Opaleye имеет проблемы с правильным типом агрегатора к примеру `sums`, что касается Opaleye версии 0.6.7006.1, библиотека имеет улучшеную функцию для обработки. Вдобавок, теперь возможно использовать библиотеку целиком с помощь интерфейса монады вместо ссылок, избегая перегрузки познания, который в прошлом был необходим. Одно из препятствий которое необходимо будет изучить это `product-profunctors` используются везде. Однако можно легко обойтись без глубокого знания этой темы. Нужно просто добавить `p2` и `p3` везде где вам говорит документация.

В конце концов `Opaleye` просто работает, и это мой личный совет. В ней есть немного абстрактная кривая изучения но то дает вам общие возможности и комбинирование частей вашей запросов, это моё личный фаворит среди DB библиотек которые мы рассматриваем.

```
latestHits :: Select (HitmanF, MarkF)
latestHits = do
```

```
  (hID, created, mID) <- byDate
  (maxHID, maxCreated) <- maxDates
  h <- selectTable hitmenNoMeta
  m <- selectTable marksNoMeta

  viaLateral restrict $ hID .== maxHID .&& created .== maxCreated
  viaLateral restrict $ hID .== hitmanID h
  viaLateral restrict $ mID .== markID m

  pure (h, m)

  where byDate = aggregate (p3 (groupBy, groupBy, min)) $ do
          ( ErasedMark { erasedMarkHitmanID = hitmanID
                       , erasedMarkMarkID = markID
                       }
            , (createdAt, _)
            ) <- selectTable erasedMarkTable
          pure (hitmanID, createdAt, markID)
        maxDates = aggregate (p2 (groupBy, max)) $ do
          ( ErasedMark { erasedMarkHitmanID = hitmanID }
            , (createdAt, _)
            ) <- selectTable erasedMarkTable
          pure (hitmanID, createdAt)
```

Squeal

Squeal странный ребенок, менее удобный DSL, он предполагает глубокое встраивание SQL в самом Haskell. Поэтому он гораздо ближе к написанию реального SQL запроса, и не пытается обастрагироваться от этого, чтобы ваши SQL ключевые слова были в нуном месте в вашем запросе.

Эта жестокость делает использование Squeal болезненным, так как используется соблюдение типов, например. вы делаете `WHERE` после таблицы которую вы объединяете из принесенного в рамках. Так как Squeal использует чистую комбинаторноние применение, вместо стрелочной или монадной у других библиотек, использование становится упражнением жонглирования вложенных скобок и постоянного прыгания между

различными уровнями вложнеий. Честно говоря, ощущается как куча.

Squeal так же использует `OverLoadedLabels` для выбора колонок и таблиц, и идет даже дальше чем всё что есть в этом списке, он просит не только ввести вашу колонку, но так же просит отслеживать какое название вы использовали для каждой колонки. Какая, восхитительная, но так же очень раздражает когда создание подзапроса в другом и в этом случае необходимо явно перевыбрать результаты подзапроса используя тоже самое имя.

Эта настройчивость в названии столбцов ведет к множеству проблем в том числе. Способ которым вы возвращаете объекты ваших типов доменов это явное имя столбца запроса такого же как свойство вашего типа данных при использовании SQL запроса `AS`. Нет возможности просто определить ответственность один раз и забыть, что значит, что даже если вы просто выбираете все сущности из одной таблицы, вам нужно явно переназвать все колонки. Красота! Хотите получить для конкретного случая кортеж данных из быстрого запроса? Извините, нельзя так, кортеж не имеет называных полей, как вы можете назвать вашу колонку правильно? На деле, каждый раз когда вы хотите вернуть данные из новой формы, вам нужно определить полностью новый тип данных для этогог и перенаследовать специальные типы классов Squeal.

Пока работал с Squeal, чувствовал себя будто я не останавливался споткаться. Тип запросов Squeal имеет тип параметров для обоих запросов входящих\исходящих, но они не похожи на возможность передать их как параметры позапроса? Поэтому вам нужно закончить просто копировать кодов запросов. Иногда использовать подзапросы просто... которые вызывают ошибку выполнения непонятно почему, даже несмотря на проверку типов. Я надеюсь вы не когда не ошибетесь в названии колонки, или Squeal кинет вам в море непостижимых ошибок типов.

Ктоме всего, Squeal успешен в качестве инструмента встраивания SQL в Haskell, но проиграл в возможности описать общие SQL шаблоны без серьезных последствий. Не рекомендую.

```
latestHits :: Query_ Schema () HitInfo
latestHits = select_
  (#minid ! #hitman_id `as` #hiHitmanID :* #minid ! #mark_id `as` #hiMarkID)
  ( from ((subquery ((select_
           ( #em ! #hitman_id
          :* #em ! #created_at
```

```
        :* (fromNull (literal @Int32 (-1)) (min_ (All (#em ! #mark_id)))) `as` #mark_id
         )
         ( from (table (#erased_marks `as` #em))
           & groupBy (#em ! #hitman_id :* #em ! #created_at ))) `as` #minid ))
    & innerJoin (subquery ((select_
                 ( #em ! #hitman_id
                :* (max_ (All (#em ! #created_at))) `as` #created_at
                 )
                 ( from (table (#erased_marks `as` #em))
                   & groupBy (#em ! #hitman_id) )) `as` #latest))
        (#minid ! #created_at .== #latest ! #created_at)) )
```

Persistent + Esqueleto

Persistent - это легкий уровень для произведения простых CRUD операций. Esqueleto - SQL DSL над Persistent, добавлюящий возможность делать объединения и более сложные запросы.

Много говорить про Persistent не будем, так как библиотека просто предоставляет слой, давайте говорить о Esqueleto. Esqueleto значит быть очень легким языком запроса, в тоже время предоставлять достаточно мощности выполнять ожидаемые вещи от всяких хорошо написанных Haskell библиотек, как некоторые типы безопасности и немного композиционности.

Для меня, однако, я нашел что этот фокус на простых интерфейсах запросов сделал библиотеку такой, что она требует столько же ментальной энергии, сколько написание простого sql заапроса, если не больше. Выглядит это как пол пути применения, где есть некоторая проверка запросов, что вы пишите которая имеет смысл. но библиотеки все еще требует ответственность писать синтаксически верно и правильно сформированные запросы.

На пример: ваш запрос будет щастливо компилироваться без предупреждений но генерировать синтаксически неверные запросы SQL во вермя исполнения если вы забыли `ON` в вашем объединении. Или счастливо падать во время выполнения когда вы пытаетесь выбрать смесь колонок агрегатных и не агрегатных колонок. Запрос DSL сам по себе проктически 1:1 транслируется в чистый SQL включая множество возможных путей

злоупотребения ими. Поэтому я надеюсь что вы уже знакомы с SQL и его кварками, так как Esqueleto делает всего пару попыток скрыть SQL бородавок от вас.

На вершине этого, Esqueleto далеко позади по возможстям поддержки типичной RDBMS функциональности. Заметное отсутствие это можетсов возможностей объединять подзапросы, и вам необходимо писать все ваши запросы используя только один `SELECT` и улучшать условия объединения на существующих таблицах. У меня получилось реализовать все запросы по киллерам, но это потребовало серьезное управление запросами, чтобы все они заработали.

Вывод: даже не смотря на то что у меня получилось реализовать проект на Esqueleto, я чувствовал как будто я не получаю достаточно от простого написания SQL запросов. В других способах выглядело слишком ограниченно, из-за библиотеки каким-то образом открывает базовый набор функций. Вы можете даже познать простоту и гибкость написания SQL, или строгую безопасность типа и сложную компонуемость как в Opaleye. Esqueleto ощущается как неполучившаяся попытка быть маленьким со всех сторон.

```
latestHits :: MonadIO m => SqlPersistT m [(Entity Hitman, Maybe (Entity Mark))]
latestHits = select $
  from $ \(hitman `LeftOuterJoin` emark1
                  `LeftOuterJoin` emark2
                  `LeftOuterJoin` mark) -> do
    on (emark1 ?. ErasedMarkHitmanId ==. emark2 ?. ErasedMarkHitmanId &&.
        emark1 ?. ErasedMarkCreatedAt <. emark2 ?. ErasedMarkCreatedAt &&.
        emark1 ?. ErasedMarkMarkId >. emark2 ?. ErasedMarkMarkId)
    on (emark1 ?. ErasedMarkHitmanId ==. just (hitman ^. HitmanId))
    on (emark1 ?. ErasedMarkMarkId ==. mark ?. MarkId)
    where_ (isNothing $ emark2 ?. ErasedMarkCreatedAt)
    where_ (isNothing $ emark2 ?. ErasedMarkMarkId)
    pure (hitman, mark)
```