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

Haskell

- WELCOME TO ALL THOSE LEARNING HASKELL
  - Содержание
  - Ужасно простой веб стек на Haskell
  - Какую типобезопасную библиотеку базы данных вы должны использовать?
- Функциональщина
  - Отчет по stack скриптованию: как и почему...
- IaaC
  - Haskell-based Infrastructure
- Скрипты на Хаскеле (пробую писать)

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.

WELCOME TO ALL THOSE LEARNING HASKELL

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....

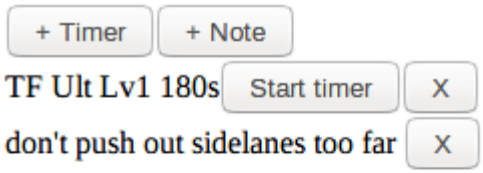
Возможность показать что вам не нужно проводить часы насколько ваш стек крут, - это возможность научиться самому. Я написал пример веб-приложения использующий самые простые библиотеки, которые я смог найти. Если вы не уверенны, как строить реальное приложение на Haskell, почему бы не начать с этого?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

WELCOME TO ALL THOSE LEARNING HASKELL

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?

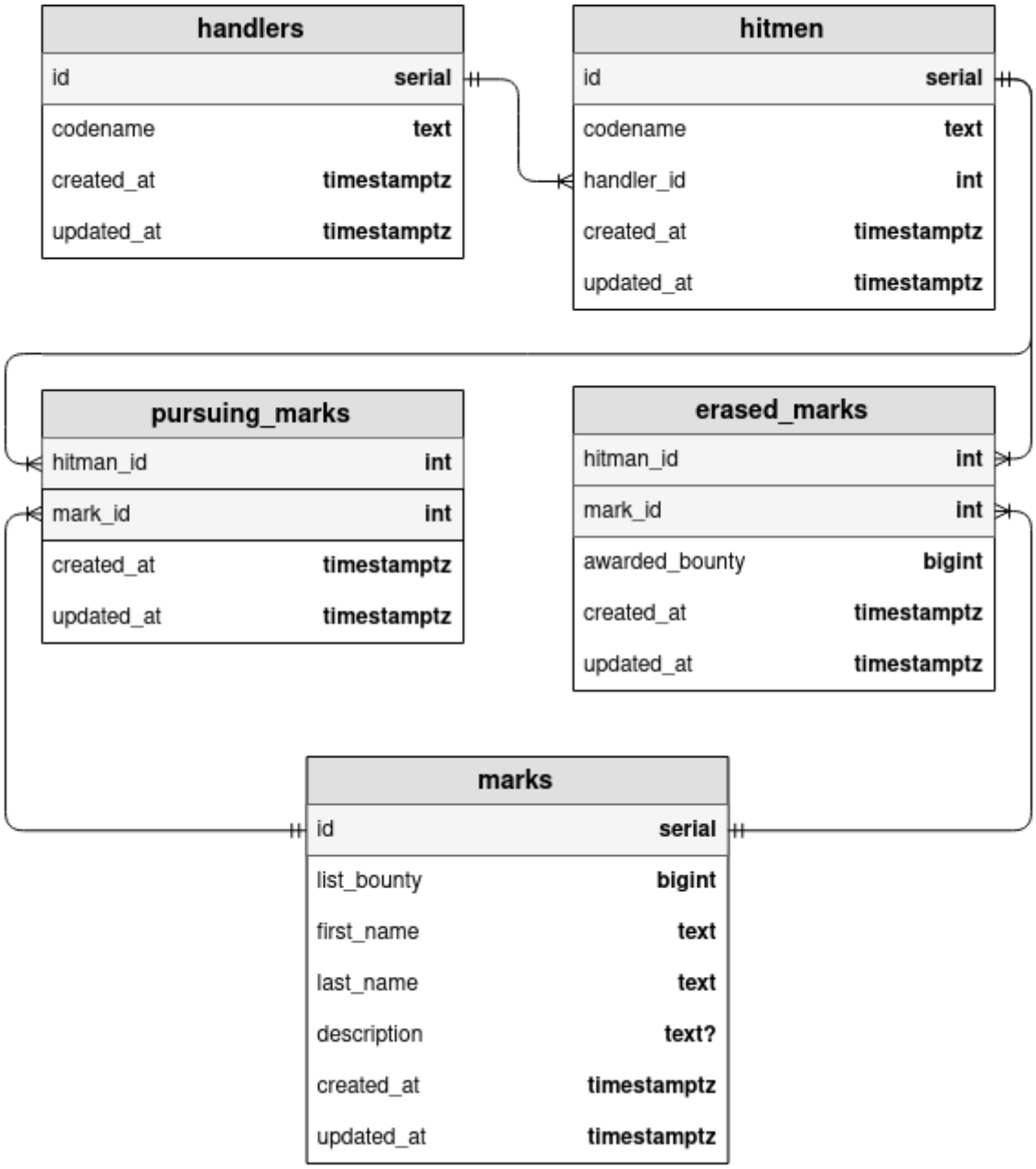
Вполне возможно, что вы как и я сагитировались на преимущества строгой типизации чтобы знать, чтобы писать приложения лучше.(Если нет, то на данный момент будем считать так)

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 улучшенную 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)
```

Функциональщина

Отчет по stack скриптованию: как и почему...

Введение

Почему stack скрипт?

Если вы делитесь маленьким, одиночным модулчем, самостоятельным примером haskell, то stack script дает нам простой способ получить воспроизводимую сборку, просто зафиксировав зависимости с помощью Stackage внутри комментариев в начале кода на Haskell.

Есть как минимум две дополнительные причины, кроме воспроизведения сборки приложений, возможно вам захочится использовать Stack скриптовый функционал:

- Низкий конфигурационный уровень: написание независимого комплириуемого файла с Haskell кодом, с зависимостями без необходимости настроек нового stack или cabal проекта.
- Использование Hashell как я зыка скриптования, изи замена для Shell/Bash/Zsh. Этот способ использования объединяется с использованием Turtle библиотеки, так же у этого подхода есть недостатки.

О статье

Stack это инструмент построения, главным образом, разработанный для воспроизвдения сборки приложений, выполняемое с помощью специального `resolver`(разрешателя зависимостей) в конфигурацийонном файле, обычно ваш проект `stack.yaml` и `package.yaml`. С помощью возможностей скриптования Stack, мы можем воспроизвести сборку приложения указывая `resolver`, но перенося эту спецификацию в файле который мы собираем или как аргумент командной строки. Отсюда, с целью упростить, мы предположим что эти скрипты запускаются вне stack проекта, и stack вызывается в той же директории что и скрипт.

> Заметка: Когда мы запускаем stack скрипт внутри проекта stack, важно принять во внимание, что stack прочитает настройки из project.yaml и stack.yaml, что можно привести к проблемам.

Примеры кода

Содераxние

Эта статья содержить следующие примеры использования скриптов и stack:

- Базовый пример интепретатора скриптов
- Просто Servant сервер который раздает статические данные вашей текущей папки.
- Пример stack как замена bash
- Использование скрипта за для запуска ghci

Базовый пример stack скрипта

Для нашего первого примера, мы будем использовать stack для запуска одного файла написаного на Haskell в виде скрипта.

Вот исходный код, который мы хотим запустить, в файле под названием `simple.hs`:

```
main :: IO ()
main = putStrLn "compiled & run"
```

Для его запуска с stack интерпретатора, мы можем выполнить следующее:

```
$ stack script simple.hs --resolver lts-14.18
```

Аргументы для `resolver` обязательны, и stack скомпилирует и запустить простой `simple.hs` файл сразу после того как будет вызван `lts-14.18` снимок.

Как альтернатива, мы можем сложить всю конфигурационную информацию в сам script, как показано ниже:

```
{- stack script
 --resolver lts-14.18
-}
main :: IO ()
main = putStrLn "compiled & run"
```

что может быть скомпилено и запущенно с помощью:

```
stack simple.hs
```

Простой Servant сервер.

Можно использовать haskell, и скриптовые возможности Stack, вместе с Turtle блиблотекой как замену shell скриптов. Для этого нам нужно добавить следующие строки в начало haskell файла:

```
#!/usr/bin/env stack
{- stack script
 --compile
 --copy-bins
 --resolver lts-14.17
 --install-ghc
 --package "turtle text foldl async"
 --ghc-options=-Wall
```

```
-}
```

Это stack скрипт делает пару вещуй:

- --compile и --copy-bins создает бинарник основываясь на имени файла
- устанавливает ghc, если необходимо, с помощью `install-gch`
- собирает скрипт с набором пакетов из lts-14.17

С помощью turle, мы получаем переносимый способ для запуска shell команд, мне удалсоь создать отличную haskell программу для замены shell скрипта, который я использовал для автоматизации задачь для развертывания этого блога.

Основа моего скрипта развертывания - `turtle`, как видно дальше, ниже представлен полный пример:

```
import qualified Turtle as Tu
import qualified Control.Foldl as L
import qualified Data.Text as T
import Control.Concurrent.Async
import System.IO

argParser :: Tu.Parser Tu.FilePath
argParser = Tu.argPath "html" "html destination directory"

main :: IO ()
main = do
  -- 53 files copied over into destinationDir
  hSetBuffering stdout NoBuffering
  destinationDir <- Tu.options "Build blog and copy to directory" argParser
  Tu.with (Tu.mktempdir "/tmp" "deploy") (mainLoop destinationDir)
```

Одна отличная штука про `turtle` это `Tu.with` функция, которая позволяет запускать нашу `main` логику во временной директории, которая в дальнейшем ощичаещтся после заврешения `mainLoop`.

Использование stack скрипта для запуска ghci.

Мы уже видели примеры stack скриптов, но есть еще, что должно быть в наборе разработчика Haskell. Stack скрипты можно использовать для запуска ghci repl. Представим мы работаем над новой ADT, и мы хотим написать новый объект `QuickCheck`, как нам может помочь script?

Следующий заголовок загрузит список приведенный ниже в ghci repl:

```
{- stack
 --resolver nightly
 --install-ghc
 exec ghci
 --package "QuickCheck checkers"
-}
module XTest where
```

Отметим еще пару вещей о порядке аргументов:

- Файл скомпилируется, и затем откроет консоль с загруженным модулем XTest.
- Если `exec ghci` сразу же не стоит за `stack`, тогда `--packages` должен быть перед `exec ghci`

ghcid

Теперь можно запустить скрипт выше с помощью ghcid, для получения практически постоянной обратной связи компилятора используя следующую команду:

```
bash$ ghcid -c "stack XTest.hs"
```

Заключение

Я часто нахожу себя за написанием маленьких haskell обрывков, однако но обычно это связано с изучением новых типов данных, использованием библиотке, или воспроизвдения примеров из статей или книг. В этом случае, `Stack` скриптовая возможность позволяет мне указать зависимости с помощью снимка в заголовке фала, и не беспокоится о ломающих изменениях, или настройках проекта со всеми верными зависимостями. Я должен обратиться к вам товарищи хаскеллята, использовать возможность stack скриптов когда кто-то делится своим кодом в сети, чтобы помочь остальным запустить их код сегодня, и в любое другое время в будущем.

IaaC

IaaC

Haskell-based Infrastructure

In my previous post I focused on the build and development tools. This post will conclude my series on Capital Match by focusing on the last stage of the rocket: How we build and manage our development and production infrastructure. As already emphasized in the previous post, I am not a systems engineer by trade, I simply needed to get up and running something while building our startup. Comments and feedback most welcomed!

Continuous Integration

Continuous Integration is a cornerstone of Agile Development practices and something I couldn't live without. CI is a prerequisite for Continuous Deployment or Continuous Delivery: It should ensure each and every change in code of our system is actually working and did not break anything. CI is traditionally implemented using servers like Jenkins or online services like Travis that trigger a build each time code is pushed to a source control repository. But people like David Gageot, among others, have shown us that doing CI without a server was perfectly possible. The key point is that it should not be possible to deploy something which has not been verified and validated by CI.

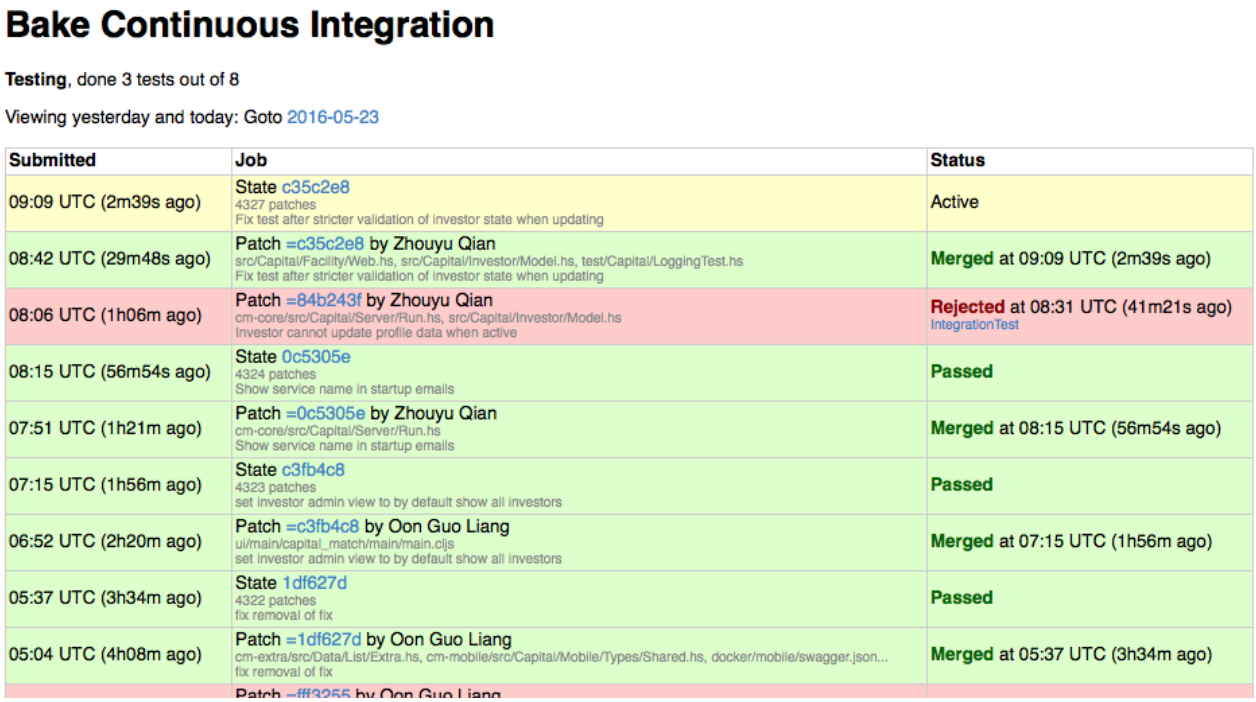
CI Server

We settled on using a central git repository and CI server, hosted on a dedicated build machine:

- Git repository's master branch is "morally" locked: Although technically it is still possible to push to it, we never do that and instead push to a review branch which is merged to the master only when build passes,
- The git repository is configured with a git deploy hook](https://www.digitalocean.com/community/tutorials/how-to-use-git-hooks-to-automate-development-and-deployment-tasks) that triggers a call to the CI server when

we push on the review branch,

- Our CI server is implemented with bake, a robust and simple CI engine built - guess what? - in Haskell. Bake has a client/server architecture where the server is responsible for orchestrating builds that are run by registered clients, which are supposed to represent different build environments or configurations. Bake has a very simple web interface that looks like



Bake Continuous Integration

Testing, done 3 tests out of 8

Viewing yesterday and today: Goto 2016-05-23

Submitted	Job	Status
09:09 UTC (2m39s ago)	State c35c2e8 4327 patches Fix test after stricter validation of investor state when updating	Active
08:42 UTC (29m48s ago)	Patch =c35c2e8 by Zhouyu Qian src/Capital/Facility/Web.hs, src/Capital/Investor/Model.hs, test/Capital/LoggingTest.hs Fix test after stricter validation of investor state when updating	**Merged** at 09:09 UTC (2m39s ago)
08:06 UTC (1h06m ago)	Patch =84b243f by Zhouyu Qian cm-core/src/Capital/Server/Run.hs, src/Capital/Investor/Model.hs Investor cannot update profile data when active	**Rejected** at 08:31 UTC (41m21s ago) IntegrationTest
08:15 UTC (56m54s ago)	State 0c5305e 4324 patches Show service name in startup emails	**Passed**
07:51 UTC (1h21m ago)	Patch =0c5305e by Zhouyu Qian cm-core/src/Capital/Server/Run.hs Show service name in startup emails	**Merged** at 08:15 UTC (56m54s ago)
07:15 UTC (1h56m ago)	State c3fb4c8 4323 patches set investor admin view to by default show all investors	**Passed**
06:52 UTC (2h20m ago)	Patch =c3fb4c8 by Oon Guo Liang ui/main/capital_match/main/main.cljs set investor admin view to by default show all investors	**Merged** at 07:15 UTC (1h56m ago)
05:37 UTC (3h34m ago)	State 1df627d 4322 patches fix removal of fix	**Passed**
05:04 UTC (4h08m ago)	Patch =1df627d by Oon Guo Liang cm-extra/src/Data/List/Extra.hs, cm-mobile/src/Capital/Mobile/Types/Shared.hs, docker/mobile/swagger.json... fix removal of fix	**Merged** at 05:37 UTC (3h34m ago)
	Patch =fff3255 by Oon Guo Liang	

- Bake provides the framework for executing “tests”, reporting their results and merging changes to master branch upon successful build, but does not tell you how your software is built: This is something we describe in Haskell as a set of steps (bake calls them all tests) that are linked through dependencies and possibly dependent on the capabilities of the client. Here is a fragment of the code for building Capital Match:

```
data Action = Cleanup
            | Compile
            | Dependencies
            | RunDocker
            | Deploy ImageId
            | IntegrationTest
            | UITest
            | EndToEndTest
            deriving (Show,Read)

allTests :: [Action]
allTests = [ Compile
```

```
            , Dependencies
            , IntegrationTest
            , UITest
            , EndToEndTest
            , Deploy appImage
            , RunDocker
            ]

execute :: Action -> TestInfo Action
execute Compile = depend [Dependencies] $ run $ do
  opt <- addPath ["."] []
  () <- cmd opt "./build.sh --report=buildreport.json"
  Exit _ <- cmd opt "cat buildreport.json"
  sleep 1
  incrementalDone

execute IntegrationTest = depend [Compile] $ run $ do
  opt <- addPath ["."] []
  () <- cmd opt "./build.sh test"
  incrementalDone
```

The code is pretty straightforward and relies on the toplevel build script build.sh which is actually a simple wrapper for running our Shake build with various targets.

- The output of the CI process, when it succeeds, is made of a bunch of docker containers deployed to Dockerhub, each tagged with the SHA1 of the commit that succeeded,
- We extended bake to use git notes to identify successful builds: We attach a simple note saying Build successful to those commits which actually pass all the tests. We also notify outcome of the build in our main Slack channel,
- Bake server and client are packaged and deployed as docker containers, which means we can pull and use those containers from any docker-enabled machine in order to reproduce a CI environment or trigger builds through bake’s command-line interface,
- As the last stage of a successful build we deploy a test environment, using anonymized and redacted sample of production data.

Testing

An significant time slice of our build is dedicated to running tests. Unit and server-side integration tests are pretty straightforward as they consist in a single executable built from Haskell source code which is run at IntegrationTest stage of the CI build process. Running UI-side tests is a little bit more involved as it requires an environment with PhantomJS and full ClojureScript stack to run leiningen. But the most interesting tests are the end-to-end ones which run Selenium tests against the full system.

- The complete ETE tests infrastructure is packaged as - guess what? - a set of containers orchestrated with docker-compose and mimicking production setup:
  - One container per service,
  - One container for the nginx front-end,
  - One container for the SeleniumHub,
  - One container for a Firefox node in debug mode (this allows us to use VNC to log into the container and see the Firefox instance executing the tests),
  - and one container for the test driver itself,
- Tests are written in Haskell using hs-webdriver, and we try to write them in a high-level yielding something like:

```
it "Investor can withdraw cash given enough balance on account" $ runWithImplicitWait $ do

  liftIO $ invokeApp appServer $ do
    iid <- adminRegistersAndActivateInvestor arnaudEMail
    adjustCashBalance_ (CashAdjustment iid 100001 (TxInvestorCash iid))

  userLogsInSuccesfully appServer arnaud userPassword
  goToAccountSummary
  cashBalanceIs "S$\n1,000.01"

  investorSuccessfullyWithdraws "500.00"

  cashBalanceIs "S$\n500.01"
  userLogsOut
```

- Those tests only need a single URL pointing at an arbitrary instance of the system, which makes it “easy” to run them during development outside of docker containers. It’s even possible to run from the REPL which greatly simplifies their development,

- Getting the docker-based infrastructure right and reliable in CI was a bit challenging: There are quite a few moving parts and feedback cycle when working with containers is slow. We ran into subtle issues with things like:
  - Differing versions of Firefox between local environment and container leading to different behaviours, like how visibility of DOM elements is handled which may or may not prevent click actions to complete
  - Timezone differences between various containers yielding different interpretations of the same timestamp (official Selenium docker images are configured to use PST whereas test driver container uses SGT,
  - Connections and timeouts issues between all the containers depending on open ports and network state,
  - …
- However, once in place and executing reliably, those tests really payoff in terms of how much confidence we have in our system. We don't aim to provide 100% feature coverage of course and try to keep ETE tests small: The goal is to ensure our system's main features are still usable after each change.

Deployment

Provisioning & Infrastructure

We are using DigitalOcean's cloud as our infrastructure provided: DO provides a much simpler deployment and billing model than what provides AWS at the expense of some loss of flexibility. They also provide a simple and consistent RESTful API which makes it very easy to automate provisioning and manage VMs.

- I wrote a Haskell client for DO called hdo which covers the basics of DO API: CRUD operations on VMs and listing keys,
- Provisioning is not automated as we do not need capacity adjustments on the go: When we need a machine we simply run the script with appropriate credentials. Having a simple way to provision VMs however has a nice side-effect: It makes it a no-brainer to fire copy

of any environment we use (Dev, Ci or Production) and configure it. This was particularly useful for pairing sessions and staging deployment of sensitive features,
- We also use AWS for a couple of services: S3 to backup data and host our static web site and CloudFront to provide HTTPS endpoint to website.

Configuration Management

Configuration of provisioned hosts is managed by propellor, a nice and very actively developed Haskell tool. Configuration in propellor are written as Haskell code using a specialized “declarative” embedded DSL describing properties of the target machine. Propellor’s model is the following:

- Configuration code is tied to a git repository, which may be only local or shared,
- When running ./propellor some.host, it automatically builds then commits local changes, pushing them to remote repository if one is defined. All commits are expected to be signed,
- Then propellor connects through SSH to some.host and tries to clone itself there, either by plain cloning from local code if some.host has never been configured, or by merging missing commits if host has already been configured (this implies there is a copy of git repository containing configuration code on each machine),
- In case architectures are different, propellor needs to compile itself on the target host, which might imply installing additional software (e.g. a Haskell compiler and needed libraries…),
- Finally, it runs remote binary which triggers verification and enforcement of the various “properties” defined for this host. Propellor manages security, e.g. storing and deploying authentication tokens, passwords, ssh keys…, in a way that seems quite clever to me: It maintains a “store” containing sensitive data inside its git repository, encrypted with the public keys of accredited “users”, alongside a keyring containing those keys. This store can thus be hosted in a public repository, it is decrypted only upon deployment and decryption requires the deployer to provide her key’s password.

Here is an example configuration fragment. Each statement separated by & is a property that propellor will try to validate. In practice this means that some system-level code is run to check if the property is set and if not, to set it.

```
ciHost :: Property HasInfo
ciHost = propertyList "creating Continuous Integration server configuration" $ props
               & setDefaultLocale en_us_UTF_8
               & ntpWithTimezone "Asia/Singapore"
               & Git.installed
               & installLatestDocker
               & dockerComposeInstalled
```

In practice, we did the following:

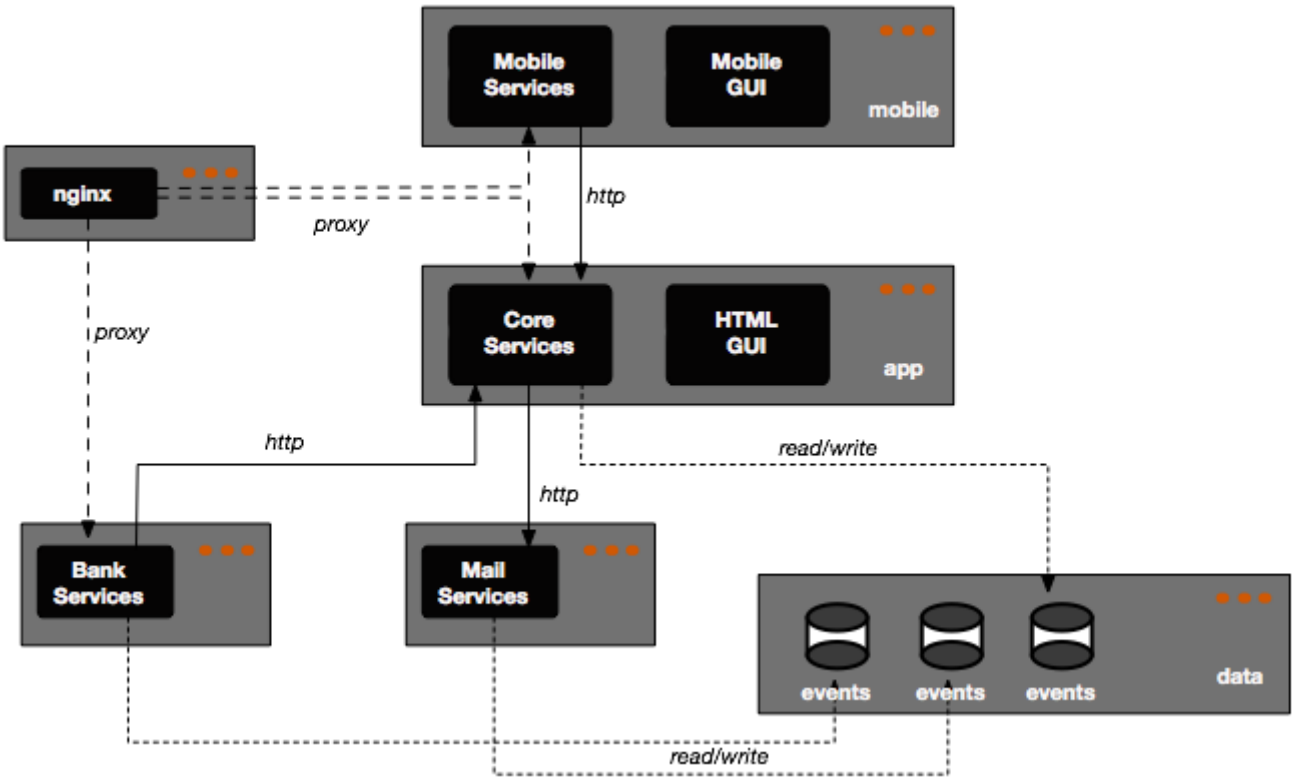
- All known hosts configurations are defined in a configuration file (a simple text file containing a Haskell data structure that can be Read) and tells, for each known IP/hostname, what type of configuration should be deployed there and for production hosts what is the tag for containers to be deployed there. As this information is versioned and committed upon each deployment run, we always know which version of the system is deployed on which machine by looking at this configuration,
- We also defined a special clone configuration which allows us to deploy some version of the system using cloned data from another system,
- We ensure the application is part of the boot of the underlying VM: Early on we had some surprises when our provider decided to reboot the VM and we found our application was not available anymore...

Deployment to Production

Given all the components of the application are containerized the main thing we need to configure on production hosts apart from basic user information and firewall rules is docker itself. Apart from docker, we also configure our nginx frontend: The executable itself is a container but the configuration is more dynamic and is part of the hosts deployment. In retrospect, we could probably make use of pre-canned configurations deployed as data-only containers and set the remaining bits as environment variables.

Doing actual deployment of a new version of the system involves the following steps, all part of propellor configuration:

- We first check or create our data containers: Those are the containers which will be linked with the services containers and will host the persisted event streams (see post on architecture),
- We then do a full backup of the data, just in case something goes wrong…
- And finally rely on docker-compose to start all the containers. The docker-compose.yml configuration file is actually generated by propellor from some high-level description of the system which is stored in our hosts configuration file: We define for each deployable service the needed version (docker repository tag) and use knowledge of the required topology of services dependencies to generate the needed docker links, ports and names. The net result is the something like the following. The dark boxes represent services/processes while the lighter grayed boxes represent containers:



We were lucky enough to be able to start our system with few constraints which means we did not have to go through the complexity of setting up a blue/green or rolling deployment and we can live with deploying everything on a single machine, thus alleviating to use more sophisticated container orchestration tools.

Rollbacks

Remember our data is a simple persistent stream of events? This has some interesting consequences in case we need to rollback a deployment:

- If the version number has not been incremented, rollbacking simply means reverting the containers' tag to previous value and redeploying: Even if some events have been recorded before we are notified of an issue implying rollback is needed, they should be correctly interpreted by the system,
- If the version has changed during deployment, then either we cannot rollback because new events have been generated and stored and we must roll-forward ; or we can rollback at the expense of losing data. This is usually not an option but still is possible if stored events are "harmless" business-wise, like authentication events (logins/logouts): A user will simply have to login again.

Monitoring

Monitoring is one the few areas in Capital Match system where we cheated on Haskell: I fell in love with riemann and chose to use it to centralize log collections and monitoring of our system.

- Riemann is packaged as a couple of containers: One for the server and one for the dashboard, and deployed on a dedicated (small) VM. Both server and dashboard configuration are managed by propellor and versioned,
- As part of the deployment of the various VMs, we setup and configure stunnels containers which allow encrypted traffic between monitored hosts and monitoring server: On the monitoring host there is a stunnel server that redirects inbound connections to running docker containers, whereas on monitored hosts the stunnel server is referenced by clients and encapsulate traffic to remote monitoring host transparently,
- Riemann is fed 2 types of events:
  - System level events which are produced by a collectd installed on each deployed host,
  - Applicative level events which are produced by the deployed services as part of our logging system,
- Applicative events are quite simple at the moment, mostly up/down status and a couple of metrics on HTTP requests and disk storage latency and throughput,

- There is a simple riemann dashboard that presents those collected events in a synoptic way,
- It is very easy to extend riemann with new clients or external connectors: At one point I considered using LogMatic to host some business-level dashboards and it took me a few hours to build a riemann plugin to send events to Logmatic's API,
- Riemann's event model is very simple and flexible hence it is an ideal candidate for being a one-stop sink for all your events: Dump all events to riemann using a single connector in the application and configure riemann server to massage the events and feed specialized clients,
- There a couple of alerts configured in Riemann that notifies slack when disks fill up or hosts are down. We also have set up external web monitoring of both application and web site using Check My Website.

Discussion

Some takeaways

- Docker has its shortcomings, is far from being perfect and is becoming bloated like all enterprise software, but packaging all parts of a system as containers is a good thing. It allowed us to grow a flexible yet consistent system made of a lot of moving parts with diverse technological requirements. Containers are obviously great for development, providing a simple and efficient way of packaging complex tools and environments in an easy to use way. But they are also great for operations: They are more flexible than VMs, they can be as secure if one takes care to trim them down to the bare minimum, and pretty compact, they give you great flexibility in terms of deployment,
- I still don't have much experience, apart from small experiments, on how to deploy docker over multiple machines. However the ecosystem of tools for managing more complex deployments is growing and maturing fast and beside I have a couple ideas on how to do it in a "simple way" using OpenVSwitch,
- Docker containers should do one and only one thing and they should be kept minimal: Don't use default fat images and try to trim them down to the bare minimum (e.g.

executable + support libraries + configuration files),

- I did not pay enough attention to build time, or more precisely I did not pay attention often enough,
- Automating as much as possible of the whole system is an investment: If you are going to throw it away in a few months, don't do it ; but if you are going to live with it for years, do it now because later it will be too late to really payoff,
- Having automated ETE tests is a great thing but they should be kept to a minimum: Always consider the relative size of the layers in the pyramid and do not try to cover bugs or "deviant" behaviour at the level of ETE tests,
- Monitoring must be baked into the system from the onset, even if with simple solutions and basic alerts. It is then easy to extend when business starts to understand they could leverage this information,
- propellor is a great tool for provisioning. I tried things like Chef or Puppet before and the comfort of working in Haskell and not having to delve into the intricacies of complex "recipes" or custom DSL is invaluable. Propellor is simple and suits my requirements pretty well, however there are a couple of pain points I would like to find some time to alleviate:
  - Tying deployment runs to git commits is really a good thing but this should be more customizable: I would like to keep deployment code in the same repository than production code but this currently would yield a lot of identically named commits and pollute the log of the repository,
  - Propellor needs to be built on the target machine as it is an executable: It can upload itself when architecture matches hence it would be better to run deployment inside a dedicated container that match the target OS in order to remove the need to install GHC toolchain,
  - It is hard to write and maintain idempotent properties: It would be simpler to be able to run propellor only once on a machine, forcing immutable infrastructure.

Conclusion

Growing such a system was (and still is) a time-consuming and complex task, especially given our choice of technology which is not exactly mainstream. One might get the feeling we kept reinventing wheels and discovering problems that were already solved: After all, had we chosen to

develop our system using PHP, Rails, Node.js or even Java we could have benefited from a huge ecosystem of tools and services to build, deploy and manage it. Here are some benefits I see from this “full-stack” approach:

- We know how our system works down to the system level, which allows us to take informed decisions on every part of it while understanding the global picture. The knowledge gained in the process of growing this system has a value in and of itself but is also an asset for the future: The better we know how the system works, the faster we can adpat it to changing requirements and constantly evolving environment,
- It has been definitely frustrating at time but immensely fun to experiment, learn, tweak, fail or succeed, with all those moving parts,
- It forces us to really think in terms of a single unified system: Being in charge of the whole lifecycle of your code, from writing the first line to deployment to production to retirement yields a sense of responsibility one does not gain from working in silos and throwing some bunch of code over the wall to ops team. This is truly DevOps in the way Patrick Debois initially coined the term, as a kind of system-thinking process and genuinely drives you to the You build it, you run it culture,
- Managing operations, even at a small scale, is demanding, hence the need to think about automation, monitoring and short deployment cycles as early as possible in order to minimize the need for manual interventions. This completes a series of post I have written over the past few months, describing my experience building Capital Match platform:

Anatomy of a Haskell-based Application described the overall design and architecture of the application, Using agile in a startup detailed our development process, Haskell-based Development Environment focused on the build system and development environment.

Скрипты на Хаскеле (пробую писать)

Я, кажется, созрел, чтобы переходить от чтения книжек и статей про Хаскель к попыткам что-то на нём писать самому. Вначале какую-нибудь мелочь. Скрипты, в общем. Поскольку я уже как-то публиковал здесь bash-скрипт rss2lj (кросспост RSS в ЖЖ), то решил в качестве упражнения его переписать и улучшить. Думаю, получилось. В этой заметке расскажу о том, как писал. Ну и о впечатлениях. Скрипт выложен на BitBucket и на Hackage.

Задача состоит из кучи рутинных операций. Я думаю, именно поэтому, будет полезно и мне на будущее, и другим начинающим и пробующим, увидеть, как они выполняются на Хаскеле. В частности, по ходу дела я разобрался как

- обрабатывать аргументы коммандной строки,
- читать и писать файлы,
- использовать регулярные выражения,
- отсылать HTTP-запросы,
- выполнять ввод-вывод в уникоде (UTF-8),
- получать системное время.

Писать буду как начинающий — начинающим. На словах получается довольно долго, но сам код получился гораздо короче, чем эта статья (около 200 строк, считая комментарии, необязательные декларации типов, пустые строки и декларации импорта внешних модулей).

Хотя Хаскель язык компилируемый и строго типизированный, использовать его для таких дел вполне можно. Код получается примерно такой же, если не более, краткий, как на Python, а компилируется даже на лету достаточно быстро. Есть и особенности. Во-первых, вместо беззаботного duck-typing здесь — строгая типизация. Поэтому писать надо аккуратнее (но и ошибок при исполнении меньше). Однако в Хаскеле эта строгая типизация сделана на основе системы типов Хиндли–Миллнера и, в отличие от C++, под ногами не

путается. Во-вторых, чтобы использовать преимущества функционального подхода (например, отложенные вычисления, частичное применение функций) нужно отделять чисто функциональную часть программы от императивных фрагментов. В простейшем случае, это означает необходимость отделить операции ввода-вывода от вычислений (преобразования информации). Переводя на Хаскель: функции ввода-вывода будут иметь монадный тип IO a, остальные же будут чистыми (без IO в типе).

Предварительное описание задачи и подхода

В моём примере можно выделить следущие операции ввода-вывода:

- получение URL из аргументов командной строки,
- чтение содержимого RSS или Atom фида по заданному URL,
- чтение (и потом запись) файла со списком уже обработанных записей,
- чтение файла с настройками доступа к учётной записи ЖЖ,
- получение системного времени,
- коммуникация с ЖЖ по установленному протоколу.

И соответственно следующие преобразования данных:

- извлечение идентификаторов всех записей в фиде,
- отсев уже обработанных записей,
- извлечение заголовков, ссылок и текста оставшихся записей,
- форматирование записей по заданному шаблону,
- разбор файла с настройками.

Для разбора произвольных фидов я велосипед изобретать не стал, а воспользовался библиотекой feed. А для всех коммуникаций по HTTP протоколу использовал библиотеку curl (мне понравился её интерфейс). Обе библиотечки нашёл на Hoogle, а установил с помощью

cabal. Из остальных зависимостей: нужен модуль Codec.Binary.UTF8.String (в убунту и дебиан он помещён в пакет libghc6-utf8-string-dev), модуль Text.Regex.Posix (соответственно, пакет libghc6-regex-posix-dev). Потом я сейчас заметил, что использовал urlEncode из Network.HTTP (у меня в ~/.cabal), хотя можно было обойтись пакетным escapeURIString (из Network.URI). То есть одна зависимость могла бы быть попроще.

В отдельный модуль я выделил всё, что касается связи связи с ЖЖ и его протокола (файл LjPost.hs). Собственно всю логику скрипта я поместил в другом файле (Feed2Lj.hs). Вспомогательную утилитку для тестирования модуля LjPost я поместил в RunLjPost.hs. Для использования скрипта она не нужна, я её использовал при его написании.

Модуль отправки сообщений в ЖЖ (LjPost)

Использование библиотеки Curl

Как я уже сказал, для работы по HTTP протоколу я использовал библиотечку curl. Соответственно, помещаю в списке импортов

```
import Network.Curl
```

а основную функцию оформляю так, всё это достаточно «императивно»:

```
postToLj ljuser ljpass subj msg = withCurlDo $ do
  curl <- initialize
  ...
```

Функция withCurlDo должна охватывать все вызовы к curl и отвечает за инициализацию и деинициализацию библиотеки; initialize собственно и позволяет к библиотеке потом обращаться. Собственно HTTP запрос делается так (запрашиваю аутентификационный токен ЖЖ):

```
  r <- do_curl_ curl ljFlatUrl getChallengeOpts :: IO CurlResponse
```

Т.е. используем do_curl_, чтобы получить данные HTTP-ответа; результат (HTTP-ответ) связываю (<-) с переменной r; аргументы do_curl_ были определены мной ранее, URL ЖЖ-API

```
ljFlatUrl = "www.livejournal.com/interface/flat"
```

и собственно параметры запроса:

```
getChallengeOpts = CurlPostFields ["mode=getchallenge"] : postFlags
postFlags = [CurlPost True]
```

Дальнейшие действия определяются логикой протокола ЖЖ.

Разбор ответа ЖЖ

Во flat-протоколе, ответ сервера выглядит так:

```
ключ_1
значение_1
ключ_2
значение_2
...
```

Нужно, во-первых, проверять значение ключа success, во-вторых извлекать значения других ключей, для начала ключа challenge.

Поскольку здесь никакого ввода-вывода уже нет, эту часть кода вполне можно написать «функционально». Самый простой и универсальный сделать это, мне кажется, разбить тело ответа (respBody) на строчки (lines), преобразовать их в ассоциативный список (list2alist) и поискать в нём нужный ключ (lookup), получив, может быть (монада Maybe), значение:

```
lookupLjKey :: String -> CurlResponse -> Maybe String
lookupLjKey k = ( lookup k . list2alist . lines . respBody )
```

При этом функция преобразования списка в ассоциативный список простая двухстрочная рекурсия:

```
list2alist :: [a] -> [(a,a)]
list2alist (k:v:rest) = (k,v) : list2alist rest
list2alist _ = []
```

Всё, мы написали всё необходимое, чтобы разбирать ответы сервера.

Вспомогательная функция, проверяем, успешен ли был запрос (тогда и только тогда, когда в ответе есть ключ success со значением OK):

```
isSuccess :: CurlResponse -> Bool
isSuccess = (=="OK") . fromMaybe "" . lookupLjKey "success"
```

Мы определили isSuccess композицией трёх функций. lookupLjKey возвращает монаду Maybe String. Функция fromMaybe достаёт из неё строковое значение. Функция сравнения (==) записана в префиксной форме и сравнивает значение со строкой «OK».

Прошу заметить, что вытащить из монады Maybe собственно значение всегда можно с помощью fromJust, но если там ничего нет (Nothing), то будет возбуждена ошибка. Здесь функция fromMaybe возвращает в такой ситуации значение по умолчанию (пустую строку), но в других местах скрипта я часто использую fromJust без проверок (т.е. при отсутствии значения скрипт будет прерываться). В программах посерьёзнее, я думаю, лучше всегда использовать функции maybe или fromMaybe, позволяющие использовать Maybe-значения, указав для них значения по-умолчанию.

Отправка сообщения в ЖЖ

Возвращаемся к функции postToLj и пишем, что если аутентификационный токе был успешно получен (isSuccess r), взять текущее время (timeopts <- currentTimeOpts, об этом ниже), подготовить запрос для публикациии сообщения (let opts = postOpts ...) и отправить. Результатом функции будет ответ на последний выполненный запрос:

```
  if (isSuccess r)
    then do
      let challenge = fromJust $ lookupLjKey "challenge" r
      timeopts <- currentTimeOpts
      let opts = postOpts ljuser ljpass challenge subj msg timeopts
```

```
      r <- do_curl_ curl ljFlatUrl opts :: IO CurlResponse
      return r
    else return r
```

Как всегда в Хаскеле, если сказал if — then, говори и else (с тем же типом).

Ещё одно «новичковое» замечание: в блоке do мы связываем переменные с монадным значением с помощью (<-) (это соответствует присваиванию в императивных языках), но определяем переменные чистыми выражениями с помощью (=). Вообще, (=) в Хаскеле почти всегда можно читать как «равно по определению». Как только я это понял — жить стало проще ;-)

Теперь подробности. Чтобы отправить сообщение, нужно сформировать POST-запрос согласно протоколу. В моём примере этим занимается функция

```
postOpts u p c subj msg topts =
  CurlPostFields ("mode=postevent" : (authOpts u p c)
                  ++ ["event=" ++ quoteOpt msg, "subject=" ++ quoteOpt subj,
                     "lineendings=unix", "ver=1"]
                  ++ topts ) : postFlags
```

которая аналогичная getChallengeOpts, только список полей, которые нужно отослать, гораздо больше. И есть некоторые тонкости.

Во-первых, нужно защищать («квотировать») некоторые символы в отсылаемых значениях. Их немного, на помощь приходит определение функции с помощью шаблонов аргумента:

```
quoteOpt ('=':xs) = "%3d" ++ quoteOpt xs
quoteOpt ('&':xs) = "%26" ++ quoteOpt xs
quoteOpt (x:xs) = x : quoteOpt xs
quoteOpt [] = []
```

Одно дело сделано. Во-вторых, нужно по имени пользователя, паролю и аутентификационному токену подготовить все поля запроса, касающиеся аутентификации:

```
authOpts u p c = [ "user=" ++ quoteOpt u, "auth_method=challenge",
  "auth_challenge=" ++ quoteOpt c,
  "auth_response=" ++ quoteOpt (evalResponse c p) ]
```

Собственно ответ на токен рассчитывается в одну строчку: evalResponse c p = smd5 (c ++ (smd5 p)) where smd5 = md5sum . fromString Кроме этого нужно импортировать соответствующие функции преобразования уникодной строки в байт-строку UTF-8 и функцию вычисления MD5-суммы:

```
import Data.ByteString.UTF8 (fromString)
import Data.Digest.OpenSSL.MD5 (md5sum)
```

И в-третьих, нужно заполнить в запросе поля, касающиеся времени публикации (текущего времени). Импортируем:

```
import Data.Time
import System.Locale (defaultTimeLocale)
```

Берём текущее время:

```
currentTime = do
  t <- getCurrentTime
  tz <- getCurrentTimeZone
  return $ utcToLocalTime tz t
```

Заметим, что функция эта связана с вводом-выводом и не является «чистой» (не возвращает одно и то же значение всякий раз). По этой причине я предпочёл не вызывать её из «чистой» postOpts, а передать уже готовый список опций, касающихся времени в postOpts из postToLj. Там, напомню, я писал:

```
timeopts <- currentTimeOpts
а currentTimeOpts определил так:
currentTimeOpts :: IO [String]
currentTimeOpts = do
  t <- currentTime
  let opts = [ "year=%Y", "mon=%m", "day=%d", "hour=%H", "min=%M" ]
  return $ map (flip showTime t) opts
```

Т.е. взял текущее время и подставил его в каждый из списка форматов (ЖЖ хочет в таков виде). Вспомогательная функция преобразования времени в строку по формату выглядит так: showTime = formatTime defaultTimeLocale Эта функция двух (неуказанных) аргументов получена каррированием функции formatTime. В map я меняю местами её аргументы (flip),

чтобы формат передавался последним, и «перчу» ещё раз текущим временем.

Всё, у нас уже есть всё необходимое для отправки любых сообщений в любой ЖЖ. Нужно только знать логин и пароль.

Чтение файла конфигурации

Где-то логин и пароль хранить надо, и самое простое, что приходит в голову, поместить его в файле настроек, написанном в виде username=мойлогин password=мойпароль В коде скрипта указываю путь по-умолчанию к этому файлу:

```
ljPassFile = "~/.ljpass"
```

Читаем этот файл и делаем из него знакомый и удобный ассоциативный список:

```
readPassFile f = do
  ljpass <- readFile f
  return $ map (\(f,s) -> (f,tail s)) $ map (break (== '=')) $ lines ljpass
```

Поскольку файл заведомо небольшой, можно использовать простую в обращении readFile. Далее как обычно: режем на строки (lines), каждую строку разбиваем по первому знаку «равно» (map (break (== '='))), правим получившийся ассоциативный список список, откидывая знаки «равно» (λ-функция во втором map). Результат заворачиваем в IO-монаду (return) как того требует тип функции.

Почти готово. Для пущего удобства сделаем себе раскрытие тильды в пути к файлу: expandhome ('~':'/':p) = do h <- getHomeDirectory ; return (h ++ "/" ++ p) expandhome p = return p и собственно функцию, которая, будет нам давать значение любого ключа из файла конфигурации:

```
readLjSetting key = do
  passfile <- expandhome ljPassFile
  s <- readPassFile passfile
  return (lookup key s)
```

В этот раз нам надо добавить ещё две декларации импорта:

```
import IO
import System.Directory (getHomeDirectory)
```

Последний штрих: в объявлении модуля перечисляем экспортируемые вовне функции, а вспомогательные замалчиваем:

```
module LjPost (readLjSetting, postToLj, isSuccess, lookupLjKey, putLjKey) where
```

Наш модуль готов к использованию. Он позволяет нам задавать настройки доступа в файле конфигурации, понимает ЖЖ-протокол, поддерживает challenge-response аутентификацию и позволяет публиковать в ЖЖ сообщения. Меньше 100 строк кода, если не считать комментарии.

Обработка RSS/Atom фида (Feed2Lj)

Переходим к заключительной части рассказа. Скрипт Feed2Lj.hs берёт URL фида из командной строки, настройки ЖЖ из файла с настройками (для него там добавляем третью настройку, имя файла со списком уже обработанных записей), скачивает фид и отсеивает уже обработанные, необработанные преобразует в plain-text, форматирует по образцу и отсылает в ЖЖ, обновляя список обработанных записей. Теперь подробно.

Получение аргументов командной строки

Получить список аргументов просто, его даёт функция getArgs из System.Environment. У нас аргумент один, адрес фида, поэтому может сразу связать нужную переменную (url) с первым элементом списка, проигнорировав остальные:

```
  url:_ <- getArgs
```

Такое связывание по шаблону мне кажется очень элегантным приёмом.

Скачивание фида

На помощь опять приходит библиотечка curl. И опять связывание по шаблону, чтобы взять только интересующую нас часть результата:

```
   (_,rawfeed) <- curlGetString url []
```

Используем модуль LjPost для чтения настроек

В общем-то вся работа уже сделана, осталось только использовать функцию readLjSetting. У неё тип [Char] -> IO (Maybe [Char]), т.е. по строке она возвращает IO-монаду, внутри которой, может быть строка (значения настройки найдено и считано), а может и не быть (настройка не найдена). Поскольку у нас тут сразу две монады (IO и Maybe), одна в другой, то, чтобы вытащить просто (Just) значение, я поступаю так:

```
ljuser <- return fromJust `ap` readLjSetting "username"
```

т.е. функцию fromJust применяю внутри монады IO (ap из Control.Monad). Аналогично с остальными значениями из файла настроек. Кажется немного громоздно с непривычки, но не так уж сложно потом. Уверен, можно написать короче.

Чтение списка обработанных записей

Мой старый bash-скрипт писал ID записей в файл, одно на строчку, поэтому новый скрипт использует тот же формат (и тот же файл). Читаем файл и преобразуем в список строк:

```
sent_ids <- (return . lines) =<< readFile sentfile
```

Здесь, чтобы не вводить временную переменную, я явно указал функцию связывания вычислений (=<<). return требуется типом (=<<). Результат эквивалентен записи

```
tmp <- readFile sentFile
let sent_ids = lines tmp
```

Отсеиваем обработанные записи

Для начала разберём содержимое фида и подготовим список всех записей. Благодаря библиотечке feed это легко:

```
  let feed = fromJust $ parseFeedString rawfeed
  let items = feedItems feed
```

Ну а отсеять уже обработанные можно с помощью filter:

```
  let newitems = reverse $ filter (isNotSent sent_ids) items
```

Функция-предикат получилась за счёт каррирования isNotSent:

```
isNotSent sent i = ((snd . fromJust . getItemId) i) `notElem` sent
```

Буквально: взять просто ID элемента (возможна ошибка), проверить, что не входит в список sent. Сразу подготовим список ID подлежащих обработке записей:

```
let new_ids = map ( snd . fromJust . getItemId) newitems
```

Отправляем запись в ЖЖ

Тупо используем уже написанный модуль LjPost. Если даны имя пользователя, пароль, шаблон записи для отправки и собственно запись:

```
postItem u p t i = do
  let message = renderItem t i
  let subj = fromJust $ getItemTitle i
  r <- postToLj u p subj message
  if isSuccess r
    then putLjKey "url" r
    else putLjKey "errmsg" r
```

Стоп-стоп-стоп! Какой ещё такой шаблон записи (t) и что делает renderItem? Объясняю: отослать запись нам надо в HTML-е, и хорошо бы можно было менять формат записи, не переделывая весь код. В общем, renderItem — это маленькая template engine, t — её шаблон. Я её опишу в следующих разделах статьи.

Вызываем из main для каждой записи из списка необработанных:

```
  let t = encodeString "<p>%text%</p><p>( <a href=\"%link%\" title=\"%title%\">дальше</a>
)</p>"
  mapM_ (postItem ljuser ljpass t) newitems
```

Здесь мы формируем список IO-действий и их последовательно исполняем (mapM_). То есть последовательно отсылаем все записи из нашего списка. Обратим ещё внимание на encodeString из Codec.Binary.UTF8.String, которая кодирует строку в UTF-8.

Форматирование по шаблону (маленькая template engine)

Напишем нашу маленькую функцию форматирования по шаблону. Пусть, допустим, все параметры шаблона будут представлены как «%параметр%», а спецсимвол «%» будет представлен в шаблоне как «%%». Параметры будет передавать ассоциативным списком, а шаблон — строчкой. На выходе — строчка с подставленными в шаблон параметрами:

```
renderTemplate _ [] = []
renderTemplate alist s =
  let (b,t,a) = s =~ "%[a-z0-9]*%" :: (String,String,String)
      tagval t
```

```
        | t == "%%" = Just "%"
        | otherwise = let inner = take (length t - 2) $ drop 1 t
                      in  lookup inner alist
      val = tagval t
  in if isJust val
       then b ++ (fromJust val) ++ renderTemplate alist a
       else b ++ t ++ renderTemplate alist a
```

Функция форматирования сообщения по шаблону готова. В ней мы последовательно «раскусываем» шаблон с помощью регулярных выражений на «текст-до», «тег» и «текст-после». Подставляем на место «тега» (t) значение соответствующего параметра, если есть, или буквальный «%», если тэг пустой. Продолжаем, пока не кончится шаблон.

О регулярных выражениях. Включаем импортом

```
import Text.Regex.Posix ((=~))
```

После этого можем в любой строчке искать регулярное выражение: строка =~ выражение :: возвращаемый тип Регулярные выражения ведут себя по-разному в зависимости от возвращаемого типа. Мне пока что пригождаются больше всего два из них: Bool для проверки соотвествия строки выражению и тройной кортеж (String,String,String), разрезающий строчку на три части.

Функция форматирования по шаблону готова. Она просто работает со строками (шаблонами) и ассоциативными списками (словарями). А где же обещанная renderItem?

Форматируем запись по шаблону

Итак, renderItem должна получать шаблон и запись из фида, а возвращать строчку. Всё, что делает эта функция — просто достаёт нужные параметры записи, помещает их в ассоциативный список и вызывает функцию форматирования по шаблону renderTemplate. В виде кода это выглядит гораздо понятнее:

```
renderItem :: String -> Item -> String
renderItem t i =
  let title = ( fromJust . getItemTitle ) i
```

```
      link = ( fromJust . getItemLink ) i
      summary = ( takeSentences 5 . eatTags . fromJust . getItemSummary) i
      tags = zip [ "title","link","text" ]
                 [ title, urlEncode link,summary ]
  in renderTemplate tags t
```

Нетривиальна здесь только функция подготовки текста сообщения (summary).

Поскольку я весь текст пересылать не хочу, а хочу только первые несколько предложений, то я вначале преобразую HTML в простой текст (в котором уже нет HTML-тэгов), а затем просто берую первые пять предложений. Таким образом, мне не нужно заботиться о предолжения будут гарантировано законченными.

Функция eatTags использует тот же приём рекурсивного раскусывания строки с помощью регулярных выражений, что и renderTemplate:

```
eatTags [] = []
eatTags s =
  let (b,t,a) = s =~ "</?[^>]*/?>" :: (String,String,String)
  in b ++ eatTags a
```

Все HTML и XHTML теги должны быть этой функцией вырезаны.

Упражнение: изменить функцию так, чтобы тег выразался не бесследно, а заменялся содержимым его аттрибута alt.

Теперь осталось лишь взять первые n предложений. Возьмём вначале одно:

```
takeSentence s =
  let ends = ".?!;"
      (first,rest) = break (`elem` ends) s
  in if not (null rest)
     then (first ++ [head rest],tail rest)
     else (first,[])
```

Тут я обошёлся без регулярных выражений, просто задав список разделителей (ends) и раскусывая строку по символу из их числа (break (`elem` ends)). Напоследок присоединяю разделитель, если он есть, к «откушенному» предложению (break прикрепляет его к «остатку»).

Осталось лишь взять первые n штук:

```
takeSentences n s
  | n > 0     = let (s',r) = takeSentence s
                in s' ++ takeSentences (n-1) r
  | otherwise = ""
```

Теперь любая запись может быть представлена так, как мы захотим. Обновляем список обработанных записей Записи получены, отобраны, отформатированы, отправлены. Осталось только обновить список обработанных. Вначале сохраним предыдущую версию файла (переименованием), а потом запишем на его место новый список:

```
  renameFile sentfile (sentfile ++ "~")
  writeFile sentfile $ unlines (sent_ids ++ new_ids)
```

Здесь использована функция renameFile из System.Directory.

Заключение

Вот вроде и всё. Можно вызывать получившийся скрипт:

```
$ runhaskell Feed2Lj.hs URL-вашего-фида
```

Пробовал пока только с GHC, но, думаю, и с Hugs должно работать. Я, кстати, осознал, что у интерпретатора Hugs есть важное преимущество перед GHC: установка GHC тянет около 100 МБ, а Hugs — всего порядка 10 МБ. Так что как разберусь с Hugs, буду стараться проверять свои скрипты и на нём.

В целом впечатления от опыта «написать на Хаскеле» очень положительные. Во-первых, очень приятно, когда удаётся написать полезную функцию в одну-две строчки. Во-вторых, интересно думать о программе иначе, писать более декларативно. В третьих, очень приятно, когда раз — и работает! (Ну это с любым языком). В четвёртых, мне нравится «математичный» синтаксис Хаскеля, он, по-моему, очень выразителен. Поначалу, пока не знакомо, конечно долго и непривычно, но когда входишь во вкус, получается быстрее и легче.

Кроме, понятно, гугла, большой подмогой является Hoogle. Сообщения GHC довольно подробные и понятные (разбирать ошибки C++-компиляторов про шаблоны гораздо труднее). Радует, что уже сейчас коллекция библиотек весьма богата (кажется, сопоставима с набором библиотек Python в то время, когда я с ним впервые познакомился). С уникодом, опять же, никаких проблем.

Есть и всякие «но»: но в коде других людей мне ещё далеко не всё понятно, но пихать ввод-вывод в любую точку кода в Хаскеле неудобно и не нужно (сделано намеренно, для отладки служит trace из Debug.Trace), но представить порядок ленивых вычислений не всегда легко, но документированы библиотеки в Hackage весьма лаконично (строго, по делу, но не так доходчиво и очевидно для новичков, как, например в Python), но cabal до сих пор нет ни в Debian, ни в Ubuntu.

Но всё равно, мне понравилось. Буду рад замечаниям и вопросам. Уверен, что-то можно было написать лучше (короче, понятнее и выразительнее). Что-то, наверное, забыл объяснить.