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

Часть 4: Обучение управляемое компиляцией

В части 2 и 3, мы обсудили некоторые общие цели идей обучения, и увидели пару их применений к Haskell. Но в какой-то момент этим необходимо воспользоваться. Как мы на самом деле изучаем что-то с нуля?

В этой части я поделюсь своим подходом для решения проблем обучения. Я буду называть это "Обучение управляемое компиляцией".

Дилемма

Представим. Вы сделали отличный прогресс в личном проекте. Вам нужн добавить еще один компонент чтобы всё собрать во едино. Вы используете внешнюю библиотеку в качестве помошника и вы застряли. ВЫ гадаете с чего начать. Вы подглядываете в документацию библиотеки. Но это не помогает.

Так как документация блиблиотек Haskell не всегда хороша, но есть спасительная благодать. Haskell жестко типизирован. В общем, когда он собирается, то это работает как мы ожидаем. На крайняк это более распространнено в Haskell, чем в других языках. Это может быть обоюдоострым мечом, при желании изучить новую библиотеку.

С другой стороны, если вы можете сколотить правильные типы для функций, вы на правильном пути. Однако, если вы не знаете достаточно о типах в библиотеке, то трудно понять откуда начинать. Что делать, если вы не знаете как собирать что-то правильно? Вы можете написать много кода с догадками, но в результате вы получите множество сообщений с ошибками. Так как вы не знакомы с типами, это будет трудно дешифровать.

Чтобы изучить новую библиотеку или систему, вам нужно начать с написания маленького кода, такого, чтобы он мог скомпилироваться. Идея зваимодейстовать с обечением управляемое компиляцией очень просто для TDD. Для начала давайте взглянем в общем на неё.

TDD

TDD это пример разработки ПО где вы пишете сначала тесты потом сам код. Вы предполагаете эффект, который должен делать код, и какой результат функции должен быть. Затем вы пишете тесты указывая ожидания от функции. Вы пишете только исходный код для свойства как только вы удовлетворенны набором ваших тестов.

Как только вы это выполните, результаты теста ведут разработку. Вам не нужно проводить много времени выясняя, какой кусок кода вы должны реализовать. Вы находите первый падающий тест, исправляете его и повторяете. Задача написать как можно меньше кода для прохождения теста. Очевидно, вы не должны просто хардкодить функцию, для прохождения тестов. Ваш тест должен быть достаточно крепким, насколько это возмоно.

Если вы пытаетесь написать код по возможности проходящий тестирование, вы можете закончить с неорганизованным кодом. Это не есть хорошо. Главная цель в TDD сражаться с циклом Red-Green-Refactor. Сначала вы пишете тесты, которые падают(Red). Делаете так, чтобы все тесты стали(green). Затем реорганизовать ваш код таким образом, чтобы он подчинялся общему стилю который вы используете. После того как это завершено, вы двигаетесь к следующему функциональному коду.

Обучение управляемое компиляцией

TDD это великолепно, но мы не можем его применять к изучению новой библиотеки. Если вы не знаете типы, вы не можете написать хорошие тесты. Поэтому нужно использовать другой процесс. В некотором роде, мы используем систему типов и компилятор в качестве теста

понимаем ли мы наш код. Мы можем использовать знания чтобы сделать код, который удовлетворяет двум параметрам:

1. Провести нашу разработку и знать, что именно мы хотим реализовать.
2. Избежать малодушия при виде "горы ошибок".

Подход выглядит сделующим образом:

1. Определим функцию которую реализуем, и затем сделаем заглушку как `undefined`. (Код должен компилироваться)
2. Сделаем небольшие изменения определении функции, так что бы проходила компиляция.
3. Определим следующий кусок кода, для написания, будь-то это `undefined` значение, которое нужно заполнить, или заглушка для конструктора объектов.
4. Повторяем шаги 2-3.

Отметим, что в конце каждого шага этого процесса, вы должны иметь компилируемый код. Здсь значение `undefined` это отличный инструмент. Это значение в Haskell которое может принимать любое значение, таким образом, что вы можете сделать заглушку для любой функции или значения в ней. Ключ в том, чтобы видеть следующий уровень реализации.

CDL на практике

Вот пример, запуска кода через этот процесс от "One Week Apps". Сначала я определяю функцию которую я хочу написать.

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView = undefined
```

Эта функция говорит, что мы хотим иметь возможность принимать `App Info` объект нашего Swift приложения, так же как и `View` объект, и создать Swift файл для вида. Теперь мы должны определить следующий шаг. Мы хотим, чтобы наш код собирался всё время пока мы решаем проблему. Тип `SwiftFile` это обертка вокрук списка типа `FileSection`. Поэтому мы можем сделать так:

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView _ _ = SwiftFile []
```

И он всё еще компилируется! Предположительно, он совершенно не завершен! Но мы сделали маленький шак в правильном направлении.

Для следующего шага, нам нужно определить какие `FileSection` объекты помещаются в список. В этом случае мы хотим три различных разделов. Первая - у нас есть раздел комментариев в верху. Второе - есть раздел "важное". И есть главный раздел реализации. Мы можем поместить выражение в эти три списка, и затем использовать заглушку ниже:

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView _ _ = SwiftFile [commentSection, importsSection, classSection]
  where
    commentSection = undefined
    importsSection = undefined
    classSection = undefined
```

Этот код всё еще компилируется. Теперь мы можем заполнить раздел по очереди, вместо того, чтобы напрягаться написанием кода целиком. Каждый из разделов имеет свою компонентную часть, которую мы разобъем дальше.

Используя наши знания о типе `FileSection`, мы можем использовать конструктор `BlockCommenSection`. Он просто принимает список строк. Так же, мы воспользуемся конструктором `ImportSection` для импорта раздела. Он так же принимает список. Продолжим следующим образом:

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView _ _ = SwiftFile [commentSection, importsSection, classSection]
  where
    commentSection = BlockCommentSection []
    importsSection = ImportsSection []
    classSection = undefined
```

И снова наш код компилируется, а мы в свою очередь сделали небольшой прогресс. Теперь определим какая строка нам нужна для раздела комментарив, и добавим её. Теперь можно добавить `Import` объектов для раздела `imports`. Если что-то пойдет не по плану мы увидим только одну ошибку и мы будем знать где она происходит. Это делает процесс разработки

гораздо быстрее.

Заключение

Мы поговорили о подходе изучения новых библиотек, но это подходит и к обычной разработке. Избегайте желания уходить с головой и писать сразу сотни строк кода! Вы пожалеете об этом когда увидите кучу сообщений об ошибке! Неспешность и твердость позволит выйграть гонку. Вы выполните гораздо больше если разобъете на маленькие детали, и воспользуетесь компилятором в качестве проверки вашего кода.

Revision #3
Created 2022-03-11 05:29:19 UTC by gasick
Updated 2022-09-04 16:15:22 UTC by gasick