Лукьянов Георгий

ФУНКЦИОНАЛЬНЫЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ ФАЙЛОВ В ФОРМАТЕ MARKDOWN

georgiylukjanov@gmail.com

1. Введение

Markdown – облегчённый язык разметки, позволяющий с лёгкостью верстать документы, которые в последствии могут быть преобразованы в более серьёзные форматы, такие как HTML, LaTex, MS Document и т. п. [1]

В данной работе рассматривается задача синтаксического анализа файлов в формате markdown с применением технологий функционального программирования. В качестве языка реализации используется язык Haskell [2] .

Синтаксический анализ текстов, таких как, например, исходные коды на языках программирования – задача, вставшая перед программистами на самой заре развития информатики. Традиционные методы решения этой задачи, описанные, например, в книге [3], используют императивный подход. В этой работе используются методы функционального подхода к программированию, одними из основных достоинств которого является высокий уровень абстракции и большая информативность кода.

2. Комбинаторы парсеров

Разрабатываемый парсер (синтаксический анализатор) основан на концепции, описанной в статье [4] – монадических комбинаторов парсеров (Monadic Parser Combinators). Эти три слова ссылаются на две крайне важные вещи из мира функционального программирования: комбинаторы и монады. Под комбинатором подразумевается функция высшего порядка (функция, аргументами и возвращаемым значением которой могут являться другие функции) – в данном случаем мы будем иметь дело с построением рабочих парсеров на основе некоторого набора примитивных. Монада же – это сущность, пришедшая в программирования из абстрактной математической теории категорий, для нужд программиста достаточно думать о монаде как о вычислении, происходящем в некотором контексте. Более подробно о монадах и о функциональном программировании вообще можно узнать из книги [5].

3. Существующие решения

На данный момент одной из самых популярных библиотек функционального парсинга является библиотека Parsec [6] являющаяся библиотекой парсеров общего назначения. Существует критика этой библиотеке, основанная на её неэффективности, связанной с использованием типа String, который имеет линейную сложности доступа к элементу, так как основан на односвязном списке. Кроме того, широко используется библиотека attoparsec, в которой тип String уступил место более эффективному типу ByteString, но и эта библиотека имеет ряд недостатков, в числе которых – типовый мономорфизм (возможна

обработка строк, представленных типом ByteString и только им).

Также существует opensource-проект Pandoc, включающий в себя широкий спектр парсеров и кодогенераторов для преобразования любых текстовых форматов в любые другие. В Pandoc есть в наличии средства, позволяющие преобразовывать markdown в LaTeX, но, в виду величины и сложности проекта, разработчики оставляют без внимания некоторые недоработки, которые доставляют конечным пользователям существенные неудобства.

Исходя из приведённых выше соображений, было решено разработать проект, избавленный от недостатков вышеперечисленных решений и попутно исследовать возможность применения свежих техник функционального программирования, о которых речь пойдёт ниже (в разделе 5), к построению парсеров.

4. Абстрактное синтаксическое дерево для Markdown-файлов

Имея инструменты, описанные в предыдущем разделе, можно реализовать программу, получающую на вход файл в формате markdown и генерирующую абстрактное синтаксическое дерево, представляющее документ в этом формате.

Приведём упрощённое (ради краткости) описание сконструированного типа языка Haskell, представляющее markdown-документ, являющегося аналогом контекстно-свободной грамматики:

```
type Document = [Block]

data Block = Blank
           | Header (Int,Line)
           | Paragraph [Line]
  deriving (Show,Eq)

data Line = Empty | NonEmpty [Inline]
  deriving (Show,Eq)

data Inline = Plain String
            | Bold String
            | Italic String
  deriving (Show,Eq)
```

Документ является списком блоков, блок может быть пустым блоком, заголовком, или параграфом. Параграф, согласно базовому синтаксису формата markdown [8], является списком строк, строка же может включать в себя как простой, так и стилизованный текст.

Получив дерево документа, можно выполнить генерацию кода для популярных форматов разметки. На данном этапе развития проекта

выполнен кодогенератор в формат HTML.

5. Перспективы дальнейшего развития проекта

Намечено два вектора дальшейшего развития: расширение грамматики документа и усложнение архитектуры парсера.

Перспективной задачей является создание системы видения электронного конспекта, который может содержать математические формулы. Для этого предлагается расширить синтаксис Markdown вставками из Latex. Имея парсер для того гибридного языка разметки, можно будет вести быстрое и эффективное конспектирование лекций и докладов, предметная область которых насыщена математикой. Применение такого парсера не будет ограничиваться конспектированием, возможно также использование гибридного языка разметки для верстки презентаций и статей.

Описанные в статье [4] парсеры имеют довольно бедные средства для отслеживания ошибок, возникающих в процессе выполнения анализа, а также невысока их эффективность в плане расхода памяти и скорости выполнения. Необходимо модифицировать архитектуру парсера, добавив нужные возможности. Это можно сделать при помощи механизма преобразователей монад (Monad Transformers), позволяющего добавить к заданной монаде (в данном случае – к монаде Parser) свойства других монад. К сожалению, известной проблемой трансформеров монад является падения производительности с ростом количества монад в стеке.

Альтернативным способом комбинирования монад является библиотека так называемых «Расширяемых Эффектов» (Extensible Effects), описанных в статье [7], планируется использование именно этого механизма для комбинирования монад. Ставится также цель сравнения эффективности парсеров, основанных на этих альтернативных технологиях по расходу памяти и времени выполнения.

Для повышения эффективности планируется отказаться от использования типа String языка Haskell, который представляется односвязным списком символов и перейти к более общему классу типов, подразумевающему наличие более эффективных реализаций (таких, например, как класс Text). Известно, что множество строк в совокупности с операцией конкатенации образуют мноноид – в статье [6] рассматриваются некоторые специфические подвиды полугрупп и моноидов и строится класс типов, естественными экземплярами которого являются строковые типы языка Haskell, планируется использовать результаты этой статьи для абстракции от какого-либо конкретного строкового типа.

6. Заключение

Функциональные парсеры являются одним из красивейших примеров применения методов функционального программирования. Такие концепции как функции высших порядков, каррированые функции, монады и многие другие вещи из мира функционального программирования

позволяют резко сократить объём кода, а система типов языка Haskell помогает программисту обнаруживать большинство ошибок на этапе компиляции. К счастью, идеи функциональных языков всё шире распространяются за пределы академической среды и находят применения в ряде промышленных языков программирования.

7. Приложение

Исходные тексты доступны на популярном git-хостинге open-source проектов GitHub [9], для тестирования кода используется онлайн-сервис непрерывной интеграции Travis CI.

Для того, чтобы получить исполняемые файлы проекта для вашей платформы необходимо провести их компиляцию из исходных кодов. Для этого потребуется компилятор языка Haskell, самым развитым и популярным из которых является компилятор GHC [13]. Для большинства изучающих язык Haskell рекомендуется устанавливать не сам GHC, а интегрированную систему под названием Haskell Platform [14], включающую в себя не только компилятор, но и обширный набор библиотек и инструментов разработки, в том числе интерпретатор GHCi, часто используемый в роли отладчика при разработке нового кода, а также менеджер пакетов cabal, который служит для поиска и установки библиотек.

Именно cabal нам и поможет получить исполняемые файлы. Для этого необходимо ввести в терминал команду: `cabal configure && cabal build`, в случае обнаружения неустановленных зависимостей следует выполнить команду `cabal install --only-dependencies`. Кроме того, в последних версиях cabal появился механизм песочниц (sandbox), позволяющий оградить каждый проект в свою виртуальную область, где установлены нужные только этому проекту библиотеки, это очень полезно, так как позволяет бороться с явлением, называемым cabal hell [16]. Для того, чтобы воспользоваться механизмом песочниц следует выполнить команду cabal sandbox initi (перед выполнением остальных команд).

Список использованных источников

1) Описание формата Markdown // http://daringfireball.net/projects/markdown/
2) Официальный сайт языка программирования Haskell // https://www.haskell.org/haskellwiki/Haskell
3) Компиляторы: принципы, технологии и инструменты // A. V. Aho, M. S. Lam, R. Sethi, J. D. Ulman
4) Monadic Parser Combinators // Graham Hutton, Erik Meijer – Department of Computer Science, University of Nottingham, 1996
5) Learn you a Haskell for hreat good!: a beginner's guide// Miran Lipovaca, 2011

6) Parsec // https://www.haskell.org/haskellwiki/Parsec
7) Attoparsec // Bryan O'Sullivan – https://github.com/bos/attoparsec
8) Pandoc // http://johnmacfarlane.net/pandoc/
9) Adding Structure to Monoids // Mario Blaževi´c – Stilo International plc
10) Extensible Effects An Alternative to Monad Transformers // Oleg Kiselyov, Amr Sabry, Cameron Swords – Indiana University, USA
11) Документация на синтаксис формата Markdown // http://daringfireball.net/projects/markdown/syntax
12) Git-репозиторий с кодом проекта // https://github.com/geo2a/markdown_monparsing
13) GHC – The Glasgow Haskell Compiler // https://www.haskell.org/ghc/download
14) The Haskell Platform // https://www.haskell.org/platform/
15) Cabal user guide // https://www.haskell.org/cabal/users-guide/
16) Cabal/Survival // https://www.haskell.org/haskellwiki/Cabal/Survival