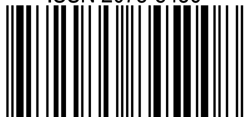


Практика функционального программирования

Выпуск 6 • Ноябрь 2010

undev

КТО ПИШЕТ СОФТ ДЛЯ ТАКИХ ПРОЕКТОВ?

МЫ.

Компания Андев постоянно ищет талантливых программистов и тимлидеров на C, C++, C#, Objective C, Ruby on Rails, Erlang и Haskell для реализации огромных проектов.

Заполните анкету на
http://undev.ru
и назначьте встречу
с нашим представителем.

Офис на Красном Октябре, бесплатные обеды, спортивный зал и солярий, хорошая зарплата, быстрые повышения.
Мы помогаем талантливым разработчикам переехать в Москву и не слишком ценим корпоративный булшит.

«Привет, bopm!»

Последняя ревизия этого выпуска журнала, а также другие выпуски могут быть загружены с сайта fprog.ru.

Журнал «Практика функционального программирования»

Авторы статей:	Джо Армстронг, Дмитрий Демещук, Саймон Пейтон Джонс, Евгений Кирпичёв, Влад Патрышев, Дэн Пипони
Выпускающий редактор:	Евгений Кирпичёв
Редактор:	Лев Валкин
Корректор:	Сергей Дымченко
Иллюстрации:	**Обложка** © iStockPhoto/Maxym Boner
Шрифты:	**Текст** PT Sans © ООО НПП «ПараТайп» **Обложка** Days © Александр Калачёв, Алексей Маслов Cuprum © Jovanny Lemonad
Ревизия:	3110 (2010-12-07)
Сайт журнала:	http://fprog.ru/

Свидетельство о регистрации СМИ
Эл № ФС77–37373 от 03 сентября 2009 г.

Журнал «Практика функционального программирования» распространяется в соответствии с условиями Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License.
Копирование и распространение приветствуется.

© 2010 «Практика функционального программирования»

Пожертвования

Продолжаем традицию публикации записок, приложенных к пожертвованиям.

lisp не мёртв, он пахнет гвоздиками!	Yandex.Money
Спасибо за хороший журнал!	Yandex.Money
Интересный журнал, спасибо	Yandex.Money
На пиво :)	Yandex.Money
спасибо	WebMoney
За хороший журнал	WebMoney
Спасибо за отличный журнал	WebMoney
More Haskell!	WebMoney
re: Поддержите журнал	WebMoney
от Василия	PayPal
E39F4504–7D6A–4a8e–A0E9–50F51C29E5E5 EZ	PayPal
Спасибо за проделанную работу, успехов в дальнейшем развитии журнала. Меньше теории и больше практики!	PayPal
спасибо	SMS
c_/ c_/ и спасибо!	SMS
l-t@, всем чмоки // мутантег	SMS
Спасибо за интересные и полезные статьи.	SMS
Долгих лет журналу!	SMS
Ждём продолжений! Удачи!	SMS
Ребята, вы — молодцы. Так держать.	SMS
Текст сообщения	SMS
Давайте же уже шестой номер выпускайте!	SMS
<STX> MSG TXT <ETX>	SMS
С благодарностью.	SMS
Где уже шестой номер-то?	SMS
Молодцi хлопцi, добру справу робите!	SMS
Не хотите выпускать журнал, ну и не надо.	SMS
Итого	**10036.59**

Спасибо за поддержку!

fprog.ru/donate

Оглавление

От редактора **7**

1. Интервью с Джо Армстронгом. **9**

2. Интервью с Саймоном Пейтоном Джонсом. **14**

3. Почему Скала. Влад Патрышев **26**
3.1. Почему Скала 27
3.2. Особенности языка 27
3.3. С точки зрения Джавы 28
3.4. С точки зрения функциональных языков 31
3.5. Сопоставление с образцом: подробности 36
3.6. Встроенный XML 38
3.7. Примеры кода 38
3.8. Тестирование 39
3.9. Готовые продукты на Скале 41
3.10. Где что на нём делают 42
3.11. Рекомендуемая литература 42

4. Сравнение Erlang и Node.js. Дмитрий Демещук **45**
4.1. Введение 46
4.2. Язык твой — друг твой 47
4.3. Многопользовательские и многозадачные 50
4.4. Сборка надежной конструкции из ненадежных компонентов . . 55
4.5. Слон в посудной лавке 58
4.6. Немного о распределенности 60
4.7. Тесты производительности 61
4.8. Аудитория 64

4.9. Итоги 65

5. Быстрое инкрементальное сопоставление с регулярными выражениями при помощи моноидов. Dan Piponi 67

5.1. Задача 68
5.2. Конечные автоматы 68
5.3. Подвешенные деревья 71
5.4. Цикл взаимодействия 72
5.5. Обсуждение 73

6. Инкрементальные регулярные выражения. Евгений Кирпичёв 75

6.1. Введение 76
6.2. Автоматный подход 79
6.3. Верёвки 82
6.4. Моноиды 85
6.5. Схема программы 88
6.6. Design challenges 93
6.7. Примеры и тесты 98
6.8. Что дальше? 104
6.9. Заключение 105

От редактора

И вот, после долгого летне-осеннего перерыва, мы снова с вами, дражайшие читатели!

Гвоздь сегодняшней программы — эксклюзивные интервью с титанами функционального мира, создателями языков Haskell и Erlang: Саймон Пейтон Джонс и Джо Армстронг поделятся мыслями и ответят на вопросы о своих детищах и обо всем остальном, от работы в Microsoft Research до творчества Тарковского.

Также приглашаем:

- тех, для кого «web scale» — не пустой звук — на просмотр смертельной битвы между Erlang и node.js, организованной Дмитрием Демещуком;
- тех, кто испытывает противоречивые чувства по отношению к Java — на обзор языка Scala от Влада Патрышева;
- тех, кто неравнодушен к растительному миру и абстрактной алгебре — на ознакомление с алгоритмическим творчеством Дэна Пипони (создателя спецэффектов ко всем трем «Матрицам» и бога Haskell) и на его развитие в статье вашего покорного слуги.

Кроме того, время, не потраченное на написание и редактирование статей, не прошло даром, и мы приготовили для вас несколько приятных новостей.

Печатный номер

Уже с первого номера нашего журнала многие читатели интересовались, когда же появится его печатный вариант, и можно будет не портить зрение, щурясь в монитор, или с довольным видом перелистывать журнал в транспорте, ловя неодобрительные взгляды едущих рядом императивных программистов.

Наконец, эта мечта осуществилась: мы начали сотрудничество с издательством «Самиздал», и шестой номер уже можно приобрести. Первым двумста счастливчикам номер достанется всего за 99 рублей, остальным — всего за 200. Получить журнал можно самовывозом, курьерской доставкой по Москве или почтовой по России — подробности на сайте издателя.

Реклама

Позволим себе вновь напомнить, что аудитория журнала составляет более 10,000 человек — и каких! Рекламодатели! Мы всегда готовы обсуждать варианты размещения рекламы, интересной нашей аудитории, в электронном и печатном варианте журнала. Первопроходцем стала компания Undev.ru — кто последует за ними?

Библиотечка ПФП

И, наконец, мы запустили проект «Библиотечка ПФП». Его цель — заполнить пробелы в существовании русскоязычных переводов классических статей-«жемчужин» из области функционального программирования и других областей.

В первом выпуске — перевод статьи Ерона Фоккера «Систематическое конструирование однокомбинаторного базиса для λ-термов», сделанный Романом Душкиным.

Ну а для тех, кому журнала мало, напоминаем: бесценный источник функциональных новостей — само русскоязычное сообщество функциональных программистов. Следите за блогами в твиттере и Russian Lambda Planet!

Приятного чтения!

С самыми чистыми пожеланиями,
Евгений Кирпичёв, jkff@fprog.ru

Интервью с Джо Армстронгом

Аннотация

Джо Армстронг — один из создателей языка Erlang. Работая в лаборатории Ericsson в 1986 году, он входил в состав команды, которая разработала и реализовала первую версию языка. С тех пор, помимо работы над самим языком, он написал несколько книг об Erlang, провел первый учебный курс по Erlang и обучил языку сотни программистов. Невзирая на свой напряженный график (Джо регулярно читает лекции и выступает на конференциях), он любезно согласился дать интервью нашему журналу.

Joe Armstrong is one of the inventors of Erlang. While at the Ericsson computer science lab in 1986, he was part of the team that designed and implemented the first version of Erlang. He wrote several Erlang books including “Programming Erlang: Software for a Concurrent World”. Joe held the first ever Erlang course and has taught Erlang to hundreds of programmers and held many lectures and keynotes describing the technology.

— Есть ли конкуренция между Erlang и MPI? Имеет ли смысл людям, которые занимаются высокопроизводительными вычислениями, искать способы задействовать Erlang, несмотря на его низкую производительность в вычислительных задачах?

Джо: Они не конкурируют — это довольно разные вещи. Erlang — язык программирования. MPI — коммуникационный протокол. На Erlang можно реализовывать полноценные приложения (например, СУБД, веб-серверы и т. п.). На MPI этого сделать нельзя, поскольку это не язык программирования. СУБД и т. п. придется писать на хост-языке, использующем библиотеки MPI.

Однако Erlang прекрасно подошел бы в качестве языка координирования высокопроизводительных вычислений. Можно реализовать «числодробилку» на Си или в FPGA, а координировать вычисления с помощью Erlang. Именно так мы и делаем телекоммуникационные системы. Мы реализуем низкоуровневые вещи на Си или в железе, а высокоуровневые — на Erlang.

— Существуют ли планы сделать Erlang более подходящим для EDSL, например, разрешив пользовательские операторы?

Джо: Нет — но на Erlang нетрудно писать парсеры для DSL. Уже существует несколько таких парсеров, и можно ожидать, что появятся и еще. Надо двигаться в направлении PEG-грамматик и некоторых идей из OMeta.

— Есть ли у каких-нибудь из современных молодых языков существенные шансы догнать или даже перегнать Erlang в его нише?

Джо: Возможно, Scala, но Scala и Erlang создавались для разных задач, так что в Scala будут трудности со многим, в чем хорош Erlang. Поведение сборщика мусора Erlang и его взаимодействие с динамической подменой кода невозможно реализовать поверх JVM, так как её семантика кардинально отличается от семантики Erlang VM.

— Планируется ли разделить спецификацию и реализацию языка Erlang, чтобы можно было реализовать свой собственный Erlang?

Джо: Это уже было сделано много лет назад. Спецификация существует. Проблема в том, что она не обновляется. Однако отсутствие актуальной спецификации — не помеха тому, чтобы реализовать новую виртуальную машину для Erlang.

Трудности реализации движка Erlang в основном обусловлены размером встроенных библиотек. Например, реализация неблокирующего ввода-вывода требует большого количества работы.

— Планируется ли расширять механизм статической типизации Erlang?

Джо: Нет.

— Сейчас в мире ПО наблюдается «сдвиг парадигмы» в сторону массивного параллелизма и распределенности, и Erlang прекрасно с ним справляется. Как Вы думаете, каким будет следующий сдвиг парадигмы? Переживет ли Erlang и его?

Джо: Полная виртуализация всех ресурсов. Мы до сих пор размышляем в терминах «мой компьютер», «мой жесткий диск». Сегодня «мои данные» — это что-то живущее на «моем диске». Ситуация меняется. Скоро все данные и вычислительные ресурсы переместятся в облако. Erlang хорошо подходит, чтобы всем этим управлять.

— Первый прототип Erlang был написан на Prolog. Сильно ли это повлияло на Erlang, если не считать синтаксиса? Как Вы считаете, стоит ли изучать Prolog?

Джо: Думаю, Erlang родился из смеси Prolog и Smalltalk. Идеи сопоставления с образцом пришли из Prolog. Идеи обмена сообщениями — из Smalltalk. Стоит ли изучать Prolog? Безусловно: Prolog — очень красивый язык программирования. Возможно, самый красивый из существующих языков.

— Многие современные языки (например, Scala и F#) обзаводятся библиотеками для эмуляции многозадачности в стиле Erlang. Есть ли у них шансы стать столь же мощными инструментами распределенного программирования лишь за счет использования основных идей, или все-таки для этого необходима и среда выполнения, сравнимая по мощности с Erlang?

Джо: Эти библиотеки реализуют лишь малую часть многозадачности в стиле Erlang. Они реализуют самые простые части, и опускают сложные (и полезные), например, динамическую замену кода и особую семантику распространения ошибок.

— В 2008 году Вы упоминали CouchDB и Scalaris как первые «killer applications» на Erlang. Появились ли с тех пор другие примеры?

Джо: RabbitMQ, MochiWeb, ejabberd, …

— Как, на Ваш взгляд, соотносятся предложение и спрос на программистов на Erlang?

Джо: Спрос на опытных программистов на Erlang намного превышает предложение.

— Вы бы рекомендовали кому-то, желающему стать ученым в области информатики, поехать в Швецию?

Джо: Да — но есть и множество других отличных стран. Важно не само место, а люди, обитающие в этом месте.

— Насколько важную роль сыграла Ваша докторская[1] диссертация в развитии Erlang? Вы бы рекомендовали людям писать докторские диссертации?

Джо: Диссертация была совершенно не важна — я написал её в 2003 г., а самая творческая часть работы над Erlang проходила в 1986–87 гг.

Цель диссертации по информатике — получить работу в университете; если вы не собираетесь преподавать в университете, я бы на вашем месте оставил эту идею. Множество отличных исследований в области информатики ведутся людьми без докторской степени, которые работают в индустрии. Думаю, в классических науках, таких как физика или химия, ситуация иная — однако в информатике, судя по всему, нет большой разницы между качеством академических и промышленных исследований.

— Чем отличается сообщество Erlang от сообществ других языков?

Джо: Оно меньше...

— Что бы Вы изменили в индустрии разработки ПО, если бы у Вас была такая возможность?

Джо: Windows.

— Как Вы распределяете время между оплачиваемой работой, хобби-проектами и отдыхом?

Джо: Мои хобби-проекты и есть мой отдых; кроме того, они довольно сильно пересекаются и с оплачиваемой работой. Особенность программирования (или любой другой работы, где платят за то, что вы работаете мозгами) состоит в том, что нельзя отключить мозги, когда заканчивается оплачиваемое время.

---

[1] Имеется в виду Ph. D., аналог кандидатской степени в российской системе образования. — *Ред.*

Мой мозг все время занимается решением программистских задач, нравится мне это или нет. Так уж получается. Когда я сплю, мой мозг решает программистские задачи. Отдых — это время, когда я работаю, но мне за это не платят.

Работа программиста — странная штука. Я постоянно напоминаю своему начальнику, что мне платят за то, чтобы я думал, а не за то, чтобы я сидел в определенном месте в определенное время и нажимал пластиковые клавиши на ящике.

— Вы устали от интервью про Erlang? О чём бы Вы предпочли рассказать?

Джо: Устал от интервью про Erlang? Нет. Программирование — просто одна из тем обсуждения. Есть множество других тем — зачем эта чертова война, почему у нас такая безумная власть, как замечательна эта книга или этот фильм...

(Вопрос: В прошлую субботу я смотрел «Солярис» Тарковского — прекрасный фильм — жаль, что я не понимаю русский: субтитры были некачественными. У меня вопрос по поводу концовки: я смотрел этот фильм около 35 лет назад и мне кажется, что концовка изменилась — та, что я помню, и та, что я видел в прошлую субботу, отличались.

В версии, которую я, кажется, видел много лет назад, когда в финальной сцене камера взлетает вверх, Кельвин выходит из дома (где в комнате течет вода), садится в машину и уезжает. Пока он едет, из океана на Солярисе появляется дорога и исчезает, когда он проезжает мимо — я точно помню этот момент — однако на DVD, который я смотрел в прошлую субботу, камера взлетела вверх, показала дом посреди океана Соляриса, и сцена закончилась; поездки на машине не было. Я ничего не путаю? Кто-нибудь смотрел оригинал фильма до того, как он попал на DVD? Напишите мне на erlang@gmail.com.)

— Планируете ли Вы посетить Россию?

Джо: Я не планирую *не* посещать Россию.

Интервью с Саймоном Пейтоном Джонсом

Аннотация

Саймон Пейтон Джонс известен как один из основателей языка Haskell, а также как архитектор и главный разработчик компилятора GHC. Он также является редактором Haskell Report — документа, формально определяющего язык. Кроме того, Саймон придумал неофициальный девиз языка: «избегать популярности любой ценой».

В этом интервью Саймон расскажет о себе, о работе в Microsoft Research, о своем взгляде на тенденции развития языков программирования вообще и Haskell в частности, о пользе езды на велосипеде и о своем отношении к Erlang.

Simon Peyton Jones is known as one of the key people in the creation of the Haskell language, the architect and main developer of the GHC compiler. He is also an editor of the Haskell Report, the language's formal definition. Besides, Simon gave the language its unofficial motto: "avoid success at all costs".

In this interview Simon will speak about himself, about his work at Microsoft Research, about his views on the trends in the development of programming languages and Haskell in particular, about the usefulness of biking and about his attitude to Erlang.

— При просмотре страницы на сайте Microsoft, где представлена Ваша деятельность, создаётся впечатление, что Вы в первую очередь занимаетесь Haskell, проектом с открытым исходным кодом. Зачем это нужно Microsoft? Не похоже, чтобы эти исследования соответствовали общему направлению деятельности компании.

Саймон: Отношение компании Microsoft к исследовательской деятельности достаточно зрелое. Если вы хотите организовать исследовательскую лабораторию, можно, конечно, попытаться предугадать, какие направления исследований окажутся прибыльными, и вкладываться в них. Но выбрать таким образом удачное направление очень сложно. Как знать, что окажется полезным, а что нет? Поэтому Microsoft использует более обширный подход. Отношение компании к исследованиям можно примерно описать так: мы наймём лучших исследователей и фактически предоставим им свободу делать то, что получается у них лучше всего. Прямо скажем, Microsoft демонстрирует определённую смелость, беря на себя риск и финансируя группу людей, чья работа не имеет прямого отношения к продукции компании, но может оказаться крайне важной в дальнейшем. На мой взгляд, так было в компании Bell Labs. Они финансировали работу, не ориентированную непосредственно на развитие продуктов компании и при этом получившую многочисленные Нобелевские премии, провели много успешных фундаментальных исследований, результаты которых находят применение много лет спустя. Я считаю, это замечательно, что компания Microsoft готова финансировать не только исследования, направленные на развитие продуктов, но и такие, которые расширяют границы наших знаний. И кстати, это даже закреплено в формулировке миссии Microsoft Research. По-моему, она состоит из трёх частей. Первая — расширение границ знаний; буквально так. Вторая — внедрение технологий в продукты компании и третья — формирование своего рода резерва экспертных знаний, которые могут понадобиться компании в коммерческих целях, если возникнет необходимость совершать резкие скачки в направлении деятельности. Деятельность Microsoft Research в целом соответствует этой миссии на протяжении уже двадцати лет своего существования. Microsoft делает очень полезное дело. Меня это, конечно, весьма радует.

— Говоря о практическом применении исследований, связанных с Haskell, как Вы можете прокомментировать неудачную, в конечном итоге, попытку внедрения Software Transactional Memory в C#?

Саймон: Я не участвовал в принятии решения о *не*внедрении. В моём понимании, рантайм .NET — большой и сложный зверь. Чтобы добавить транзакционную память, они пытались решить задачи гораздо сложнее тех, с которыми столкнулись мы в Haskell STM, где побочные эффекты являются не правилом, а исключением. Они работали с рантаймом, для которого побочные эффекты не являются исключением. И поскольку побочные эффекты должны отслеживаться системой транзакционной памяти и такое отслеживание является дорогой операцией, их исходная позиция была менее выгодна. Они собирались внедрить и многие другие сложные сущности, такие как открытые вложенные транзакции, а также модификацию адресов памяти как внутри транзакции, так и снаружи. В Haskell STM переменные делятся на два класса. Переменные класса `TVar` могут быть изменены внутри транзакции, но не снаружи. Переменные класса `IORef` могут изменяться снаружи, но не внутри. Подобное разделение значительно облегчает управление состояниями. Разработчики, которые делали CLR STM, довольно глубоко об этом задумывались, на мой взгляд. Они решили, что, мол, вы знаете, наверное хочется создать структуру, часто в единственном потоке, а затем «раздать» её другим. Пока эта структура создаётся, вы не всегда хотите, чтобы она строилась исключительно внутри какой-то транзакции. Разработчики хотели иметь возможность модифицировать состояние внутри и снаружи транзакций. Но это приводит к большому количеству трудностей, гораздо большему, чем я ожидал. Почитайте статью Тима Харриса (Tim Harris) об этом. Короче говоря, они столкнулись с гораздо более широкой задачей, чем мы, разработчики Haskell STM. Остаётся только гадать, получили бы они куда более дешёвое решение, как в плане времени исполнения, так и в плане сложности реализации, возьмись они за более узкую задачу. Тогда, будь они менее амбициозны, возможно, всё это закончилось бы работоспособным прототипом, а не таким, который, похоже, оказался слишком тяжёл.

— Ограничение спектра возможностей некоторым безопасным документированным набором могло бы обеспечить STM безопасное отступление. Это могло бы...

Саймон: Могло. Но это всё гипотезы. Я встречался с их командой один или пару раз. Команда была большая, и многие работали по два-три года. Они очень умные люди. То, что они сделали, не было глупой ошибкой. Я не хочу гадать, что они могли бы сделать. Я просто предполагаю, что возьмись они за что-то менее грандиозное, они могли бы получить куда более быструю и менее сложную реализацию. Но я не знаю, устроило бы это их клиентов. Как

я уже говорил, я не хочу сказать, что они приняли неправильное решение, потому что меня там не было. Просто из мира Haskell я знаю, что у всего есть достаточно серьёзные ограничения. Это одна из причин, почему я люблю Haskell. (*смеётся*)

— Да, какими видами спорта Вы занимаетесь и чем увлекаетесь?

Саймон: Я изредка катаюсь на горных лыжах, может быть, раз в год. Кататься я люблю, но большой роли в моей жизни это не играет.

— Что в Вашем образе жизни Вы считаете особенно полезным?

Саймон: Из приближённого к спорту — я много езжу на велосипеде. Я езжу по всему Кембриджу и провожу на велосипеде много времени. Я не склонен к продолжительным двадцатимильным загородным поездкам, вместо этого я просто езжу из пункта А в пункт Б. Это мой способ перемещения.

Что ещё помогает мне в моих исследованиях?.. Я придаю большое значение сотрудничеству. Мне непросто заниматься исследованиями в одиночку, поэтому я всё время сотрудничаю с людьми. Практически каждая моя статья написана в соавторстве. Я не знаю, можно ли это считать образом жизни, но для меня исследования — это социальная, а не индивидуальная деятельность.

— Следующая тема будет близка сердцам многих российских исследователей, испытывающих трудности с финансами. Один из первых вопросов, которые они задают: каково финансовое положение исследователей в Microsoft?

Саймон: В отличие от представителей академической среды, исследователи получают хорошее индивидуальное финансирование. Исследователи в университетах вынуждены подавать заявки на гранты. У них есть с полдюжины студентов-исследователей и несколько ассистентов, таким образом, они фактически получают внешнее финансирование на построение команды. В Microsoft этого меньше, если, конечно, вы не делаете что-то, непосредственно связанное с продуктами компании (в этом случае компания оплачивает работу тех, кто будет помогать с разработкой программного обеспечения). В каком-то смысле моя ситуация в проекте GHC (Glasgow Haskell Compiler) исключительна — Microsoft выдаёт мне некоторую сумму денег на поддержку проекта. Поддержка первого уровня обеспечивается консультантами на контрактах, что довольно нетипично. Что я хочу сказать: команды, как правило,

малочисленны, если, конечно, исследователи не соберутся вместе большой компанией и не решат, что они хотят делать что-то большое. Компании нравятся амбициозные внутренние проекты. Если вы придёте и скажете: «нам очень хочется сделать вот эту сказочную штуку в средний срок; нас несколько человек и для выполнения задачи нам нужны вот такие дополнительные ресурсы», то для обсуждения подобных вещей существуют свои механизмы.

— Как Вы думаете, существуют ли декларативный и императивный способы мышления? Считаете ли Вы, что естественное мышление человека принадлежит к той или иной категории?

Саймон: Я всегда с подозрением отношусь к фразе «это естественно». Обычно люди говорят, что что-то является естественным, потому что оно естественно для них. Совсем необязательно это будет естественным для кого-то ещё. Я вполне могу сказать, что декларативное мышление и мышление чисто функциональное, или даже преимущественно функциональное, заставляет вас применять иной подход к задачам. Чтобы писать поистине функциональные, а не императивные программы, действительно необходимо в некоторой степени перестроить мозги. И не слегка. Это очень существенная перестройка. Я не хочу сказать этим, что какой-то из способов мышления более естественен, чем другой. Главным двигателем моей исследовательской программы и причиной, по которой я считаю, что компании Microsoft полезно финансировать мои исследования и в частности Haskell, является то, что цена, которую приходится платить за неограниченные побочные эффекты, присутствующие в широко распространенной императивной модели, становится всё выше по мере того, как мы создаём всё более грандиозные программные продукты, от которых мы хотим корректности, а также параллелизма. Наблюдаемая повсюду ничем не ограниченная масса побочных эффектов на деле обходится нам всё дороже. Функциональный подход всё более оправдывает себя. Мне не важно, естественно это или нет. Суть в том, что его ценность непрерывно растёт. Я не знаю, все ли начнут использовать Haskell через 10 или 20 лет. Но я уверен, что те языки, которые будут популярны через 10 или 20 лет, так или иначе будут иметь механизмы контроля, управления и отделения побочных эффектов. Как бы то ни было, если вы хотите контролировать побочные эффекты, за идеями следует обращаться к сообществу чистых функциональщиков. Я отношусь к Haskell как к своеобразной лаборатории для развития подобных идей. Вне зависимости от того, используется ли именно Haskell, идеи остаются прежними и могут

применяться в другом обличье. Не только в рамках того, что носит название H-A-S-K-E и двойное L.

— Каково Ваше отношение к модели параллелизма, реализованной в языке Erlang? Другими словами, к модели работы, при которой задача декомпозируется на множество взаимодействующих частей. Вполне вероятно, что в более императивном языке подобная модель управления также возможна.

Саймон: Erlang мне по душе своей чёткой позицией. Мне нравится Haskell своей чёткой позицией по отношению к побочным эффектам. То же самое происходит в Erlang с его жёстким разделением между процессами: «не делиться ничем». Все данные в каждом сообщении сериализуются.[1] Таким образом, прерывание любого процесса не влияет на другие. Для этого требуется очень своеобразная и чёткая позиция, и мне это нравится.

Не всякая многопоточная программа может быть эффективно написана подобным образом. Я не думаю, что проблема параллелизма в программировании может быть решена единственной парадигмой. Парадигма языка Erlang предоставляет один из способов написания многопоточных программ. Другим способом может быть применение транзакционной памяти. Или параллелизма по данным. Я думаю, что ни одна из них не будет одинаково хороша для всех задач. Параллелизм — сложная штука. Для разных ситуаций нам понадобится масса различных парадигм программирования.

— В чем слабые места Haskell?

Саймон: Он сложный. Пожалуй, это его самое слабое место. Думаю, в какомто смысле он таким и задуман. В GHC я хотел реализовать множество фич и посмотреть, как они будут сочетаться и какие понравятся людям. Но в результате, несомненно, получился сложный язык. В некоторых отношениях Haskell прост. Он остался верен своим принципам — чистому функциональному программированию и ленивой стратегии вычислений. Но нынешняя система типов очень сложна. Думаю, самое сложное в Haskell — это то, что это очень большой язык. Надеюсь, однажды кто-нибудь выделит из Haskell лучшие части и соберет из них что-то поменьше. Не знаю, что из этого может выйти. Мы понемногу пытаемся его упрощать, убираем некоторые возможности. В стандарте Haskell 2010 отсутствуют n+k-шаблоны, потому что они в целом довольно избыточны. Мне нравится GHC как платформа для

[1] *Прим. перев.*: Имеется в виду, что данные в сообщениях передаются только по значению, и никогда — по ссылке.

исследований — у него открытый исходный код. Кто угодно может внести изменения. Если у вас есть идея для системы типов или оптимизатора, можете реализовать ее и посмотреть, что получится. Но все же, несмотря на мои попытки переписывания и упрощения, это по-прежнему огромный программный продукт. Многих людей это отталкивает. Мне нравится, например, что есть Jhc — намного более компактная и существенно отличающаяся от GHC реализация, над которой работает Джон Мичэм (John Meacham). Думаю, одна из проблем Haskell с точки зрения практиков состоит в том, что для его использования приходится мыслить иначе. У Haskell особый подход к программированию. Если он вам незнаком, то изучить его с нуля довольно трудно. Если посмотреть на использование Haskell в индустрии, то выяснится: многие используют его едва ли не тайком. Это не начальник сказал им — «мы будем использовать Haskell, изучите его». Мы изучили Haskell. Мы считаем, что это круто. Мы будем использовать его как бы втихаря. Их начальник не хочет использовать его для чего-то существенного, так как задается вопросами, удастся ли нанять программистов на Haskell и как обучить ему остальных программистов. Здесь проблема курицы и яйца. Думаю, это общая проблема всех новых языков. Какие еще слабости? Что скажете?

— Вы же упомянули кривую обучения — тут есть слабость, не так ли? От нее происходит и еще одна. Нет никакого руководства по особенностям языка и каким-то разработанным и оттестированным абстракциям Haskell. Нет никакой «большой книги» о том, что такое Haskell. Приходится читать научные статьи. Приходится читать списки рассылки и черпать идеи оттуда.

Саймон: Да, это так. Думаю, правда, одна из сильных сторон Haskell — это его дружелюбное и сильное сообщество. Это встречается не во всех научных сообществах. Программисты на Haskell обычно более готовы помогать людям, изучающим этот язык, и отвечают на письма конструктивно. Очень приятно быть членом такого сообщества.

— Да, действительно.

Саймон: Вообще это проблема роста, не так ли? В некоторых отношениях ситуация гораздо лучше, чем раньше. Если бы вы спросили меня о главных слабостях языка пять лет назад, я бы ответил — недостаток библиотек. Тогда не было ничего похожего на большие библиотеки классов для Java или CPAN для Perl. Сейчас есть Hackage, где уже есть тысячи библиотек и каждую неделю появляются новые. В последние пять лет ситуация кардинально из-

менилась. Возможно, если язык продолжит находиться в фокусе внимания, люди будут писать больше книг и заполнят пустоты, о которых Вы говорите.

— Кстати о внимании, Вы действительно это заметили? Я заметил, что в районе 2005 и 2006 годов произошел огромный скачок интереса к функциональным языкам вообще и к Haskell в частности.

Саймон: Я обычно рисую график популярности Haskell медленно вырастающим до нескольких тысяч пользователей в течение первых 15 лет его существования. А затем в последние пять лет, примерно с 2005 года, интерес очень резко повысился. Трудно сказать, почему именно. Цикл жизни этого языка необычен. Обычно язык либо приобретает популярность довольно быстро, как Perl, Clojure, Ruby, Python или C++, либо не приобретает популярность вообще. Haskell далек от этой тенденции: резкий рост интереса через 15 лет после создания языка очень необычен.

— Что Вы думаете о JIT-компиляции; Вы занимались какими-нибудь исследованиями о компромиссе между статическими оптимизациями в GHC и динамическими, в стиле JIT?

Саймон: В общем-то нет — я вообще не занимался такими исследованиями. Когда появилась идея JIT-компиляции, она преподносилась как механизм дистрибуции. Можно было передавать JIT-код через всю планету, и он был бы переносимым. Можно было компилировать его на лету. Умные JIT-компиляторы (а они действительно очень умные) используют, скажем так, информацию о поведении программы в динамике. Часто вызываемые методы компилируются более тщательно или специализируются в конкретном контексте вызова. Это намного труднее в случае статической компиляции, когда неизвестно, какие методы будут вызываться часто. Во время выполнения можно выяснить эти методы. Честно говоря, не знаю, занимался ли кто-либо такими оптимизациями для функциональных программ. Было бы довольно интересно попробовать. Думаю, в каком-то смысле это сделано в языках вроде Scala или F#, поскольку они компилируются в виртуальных машинах с помощью их мощных JIT-компиляторов и динамических оптимизаторов. Эти языки могут выиграть от такого подхода. Мои же исследования посвящены оптимизации кода на той стадии, пока он еще представляет собой функциональную программу и не превращен в последовательность машинных инструкций. Думаю, когда в вашем распоряжении есть больше информации, чем просто тело метода, можно сделать намного больше.

— Если уж на то пошло, как Вам кажется, трудно ли будет применить этот подход к оптимизации Haskell, или большой разницы нет? У нас есть пример — F#. Haskell — особый случай?

Саймон: Будут определенные затруднения. Haskell очень динамичен из-за функций высшего порядка и ленивых вычислений. Очень много диспетчеризации по указателям на код. В динамическом оптимизаторе можно сказать, «в этом косвенном переходе чаще всего адресом было вон то место». Так работает динамическая диспетчеризация в объектно-ориентированных языках. Не думаю, что кто-то детально исследовал, насколько это применимо в функциональных языках, и как можно было бы этим воспользоваться. Наверное, что-то такое происходит в F#. Думаю, это открытое поле для исследований. Я могу только строить предположения. У меня нет никаких данных, я не работал над этой областью. Я не могу сказать ничего такого, чего не мог бы сказать любой аспирант.

— Насколько я понимаю, две активных области исследований сейчас — Data Parallel Haskell и суперкомпиляция.

Саймон: Я только что вернулся из России, с конференции Meta 2010, которая была во многом посвящена суперкомпиляции. Сейчас эта тема меня особенно интересует. Эта технология компиляции существует уже давно. Ее создал русский ученый, Валентин Турчин. Он выразил ее в терминах своего языка РЕФАЛ, о котором больше никому не было известно. Лишь позднее Роберт Глюк (Robert Glück) и его студент Мортен Соренсен (Morten Sørensen) объяснили суперкомпиляцию более понятно для остальных. Но и это было достаточно давно. Я недоумевал, почему никто не применяет суперкомпиляцию для построения настоящих оптимизирующих компиляторов? Теперь я кое-что понимаю! На первый взгляд кажется, что эту технологию можно просто реализовать по статье, но на самом деле там много «ручек». Нужно принять много архитектурных решений. И неочевидно, какие именно решения надо принять.

Вместе с Максом Болингброком (Max Bolingbroke) — аспирантом, с которым я работаю — мы построили довольно интересный модульный суперкомпилятор. Если посмотреть, как устроен суперкомпилятор, то видно, что он состоит из нескольких частей, слепленных в один запутанный клубок. Когда я попытался по-настоящему понять, как работают суперкомпиляторы, это оказалось очень трудно. Было очень трудно понять статьи. Я читал их одну за другой. Я говорил себе, «Кажется, теперь понятно», но потом осознавал,

что на самом деле ничего не понятно. Кажется, теперь мы придумали, как разбить эти задачи на несколько задач поменьше. И как в том же духе разбить суперкомпилятор, сделав его более модульным. Теперь исследование различных архитектурных решений выглядит более реалистичным. Я очень рад этому. Кажется, мы начинаем понимать, как можно применить суперкомпиляцию на практике в настоящем компиляторе. Я очень надеюсь сделать оптимизатор GHC суперкомпилятором. Посмотрим, что из этого выйдет. Макс работает над этим в рамках своей докторской.

Для меня было большой честью быть приглашенным на Meta 2010 и встретиться с такими людьми, как Сергей Романенко, сыгравшими важную роль в развитии суперкомпиляции.

— Еще один вопрос. У Вас есть какая-то безумная научная идея, в осуществимость которой никто не верит, а Вы — верите и пытаетесь осуществить?

Саймон: В общем, можно просто сказать «чистое функциональное программирование». Это не только моя идея. Эту безумную идею разделяют еще сотни людей. Но сообщество программистов в целом пока в нее не верит. Если же говорить более конкретно — я думаю, идея вложенного параллелизма (nested data parallelism) выглядит интересно и многообещающе. Я до сих пор не знаю, можно ли по-настоящему заставить это работать. Вот почему это интереснейшая область для исследований. На самом деле я имею в виду не только параллелизм по данным. Это идея Гая Блеллоха (Guy Blelloch), над которой работали Мануэль Чакраварти (Manuel Chakravarty), Роман Лещинский (Roman Leshchinskiy), Габриэль Келлер (Gabriele Keller) и я. Получить выигрыш в производительности оказалось труднее, чем мы думали. Это очень гибкая парадигма программирования. Если у нас действительно получится реализовать преобразование из программ с «вложенным» параллелизмом в «плоский» без сильного замедления, это было бы просто потрясающе. Не уверен, что можно сказать, что никто не верит в осуществимость этого. Скорее всего, для 99.99% мейнстрима это так — они просто никогда об этом не слышали. Это для них за горизонтом. Но мне все равно кажется, что это достойная цель.

— Вы видите сейчас какие-нибудь тренды в развитии языков программирования, за которыми стоит следить, чтобы не упустить что-нибудь важное?

Саймон: Сейчас в области функционального программирования я усиленно работаю над повышением выразительности системы типов. Типизация — са-

мый успешный, притом с большим отрывом, из формальных методов в программировании. Тип — это нечто вроде слабой спецификации функции. Программисты все время пишут типы, а компиляторы при каждой компиляции их проверяют. Этот формальный метод очень активно используется. По мере того, как системы типов становятся более выразительными, выясняется, что солидная часть работы на начальном этапе проектирования программы — придумывание и записывание типов. Люди иногда спрашивают, «Что служит аналогом UML для Haskell?» Когда меня впервые спросили об этом 10 лет назад, я подумал, «Ума не приложу. Может быть, нам стоит придумать свой UML». Сейчас я думаю, «Это просто типы!» Люди рисуют UML-диаграммы, чтобы понять общую схему программы. Именно этим занимаются программисты на функциональных языках и на Haskell, когда придумывают сигнатуры типов для модулей и функций в этих модулях.

Одно из интересных направлений развития этого явно успешного метода — инструмента, которым программисты пользуются каждый день — повышение его выразительности, чтобы типы были более «говорящими», а спецификации — более богатыми. Тогда станет можно доказывать более интересные свойства программ. Эта тенденция существует, и, я думаю, будет продолжаться, по крайней мере в мире функционального программирования. В области типизации функциональное программирование далеко ушло от императивного. В мире функциональных языков с типизацией происходит множество интересных вещей, вовсе не заметных, к примеру, в Java. Scala, например, впитала в себя многое из функционального программирования. Вероятно, этот язык лидирует по сложности системы типов среди императивных языков. Мне кажется, стоит следить за системами типов.

— Что из Ваших творений вызывает у Вас наибольшую гордость, если оставить в стороне Haskell, или, возможно, даже исследовательскую деятельность вообще?

Саймон: Сейчас я много времени уделяю так называемой рабочей группе «Computing at School» В британских школах студентам преподают предмет под названием ICT, «Информационные и коммуникационные технологии». Он ужасно скучный. Мы берем самую интересную в мире тему, информатику, и заставляем детей поверить, что она очень скучная. Что они узнают из ICT? Они узнают об электронных таблицах, базах данных, PowerPoint и Word. Они изучают Microsoft Office, и не один раз. Это все делается с добрыми намерениями, поскольку это все-таки полезные умения. Но они вообще не занимаются настоящими вычислениями. Они вообще не имеют дела с

настоящей информатикой. Я принимаю активное участие в рабочей группе, которая пытается исправить ситуацию, по крайней мере в Великобритании. В США есть так называемая «Ассоциация Преподавателей Информатики», CSTA. В США она играет похожую роль. Я горд, что рабочая группа по преподаванию информатики выросла от идеи до 300–400 человек за пару лет. Кажется, к ней начинают всерьез прислушиваться.

Почему Скала

Влад Патрышев
vpatryshev@fprog.ru

Аннотация

Статья знакомит читателя с языком программирования Ска́ла, который сочетает возможности функционального и объектно-ориентированного программирования.

This article introduces the Scala programming language, which combines functional and object-oriented programming features.

3.1. Почему Скала

Языки приходят и уходят; некоторые остаются. Ска́ла — не первый язык, комбинирующий объектное ориентирование с функциональным программированием, но он появился в нужный момент, когда Джава уже всем откровенно поднадоела. Теперь Скала вполне готова постепенно заменить Джаву, работая на той же виртуальной машине и предоставляя доступ к тем же джавным библиотекам.

Мартин Одерски в начале 2000-х разрабатывал параллельно Скалу и дженерики для Джавы. Джавная версия дженериков оказалась несколько недоделанной, если не сказать — ошибочной (невозможно указать вариантность параметров [11]). В Скале этот недостаток устранён: вариантность присутствует, и недоуменные вопросы «насчёт дженериков», так часто возникающие при программировании на Джаве, сами собой отпадают.

Вариантность параметра типа — это указание на то, что происходит при замене типа на подтип: получим ли мы подтип или нет. Например, немодифицируемый список строк можно использовать как подтип немодифицируемого списка объектов (ковариантность); для модифицируемых такое явление не наблюдается (инвариантность); бывают и случаи, когда отношение подтипов меняет направление (контравариантность).

Так как Скала работает на привычной JVM, и даже компилируется в Джаву, использовать её можно практически везде, где есть Джава версии не ниже 5 (т. е., не на Blackberry). Переход с Джавы на Скалу можно начать с небольшого ознакомления с языком; нет необходимости всё глубоко изучать и всё радикально переписывать. Инфраструктура может оставаться той же самой; например, можно использовать тот же Tomcat, но часть сервлетов будет на Скале.

3.2. Особенности языка

Скала — язык, на котором можно писать в объектно-ориентированном стиле и в функциональном — по вкусу. Если вы привыкли программировать на Джаве, то при переходе на Скалу поменяется не так уж много: в-основном, выкинете лишние операции и будете наслаждаться новыми удобствами. Если вы любите программировать на Хаскеле, то обнаружите, что, по крайней мере внешне, многое из того, к чему вы привыкли, можно вполне разборчиво выразить на Скале. Система типов в Скале мощнее, чем в Хаскеле-98

(благодаря наличию higher-ranking types, т. е. дженериков высшего порядка).

3.3. С точки зрения Джавы

Итак, на Скале можно писать как на обычной Джаве со слегка изменённым синтаксисом. Всё есть объект: нет примитивных типов. (Читатель, конечно, тут же возмутится — а как же производительность? Стандартный ответ на это таков: JIT всё перемелет в эффективный код. К тому же, начиная с версии 2.8.0, используется специализация с помощью примитивных типов; за подробностями отсылаю к источникам.)

Все джавные классы доступны из Скалы, и наоборот. Единственная сложность состоит в том, что коллекции в Джаве и в Скале очень сильно отличаются, и чтобы работать с джавной коллекцией на Скале так, как это принято на Скале, эту коллекцию надо конвертировать, обычно неявным способом. Но если вы будете писать код «как на Джаве», то можно и не конвертировать.

Например, в компании «KaChing» Скала используется вперемешку с Джавой, и неявные преобразования из скальных коллекций в джавные и обратно — обычное дело. Добавим ещё, что в Скале имеются такие же аннотации, что и в Джаве, что даёт возможность использовать и Guice, и Hibernate.

В Скале нет джавных массивов, а есть вместо него класс `Array`; поэтому квадратные скобки можно использовать для других целей. В Скале квадратные скобки используются для параметров типа, вместо угловых.

Аналогом джавного понятия интерфейса в Скале является типаж (`trait`); типаж может содержать частичную реализацию, и работать в качестве абстрактного класса — хотя класс может наследовать сразу несколько типажей. Это не ведёт к проблемам, типичным для множественного наследования, т. к. речь не идёт о классах, а о типажах.

Метод может принадлежать классу или объекту; функции также определяются внутри класса или объекта. Если вам нужна функция, не привязанная к экземпляру класса, то её можно определить в статическом объекте. По традиции Скалы, к классу добавляется так называемый объект-компаньон (companion object) с тем же именем, что и класс:

```
class A {...}

object A {
    def myStaticMethod(s:String) : Integer {...}
```

```
}
```

Эти же объекты-компаньоны можно использовать в качестве фабрик.

Оператор «==» в Скале соответствует, буквально, джавному `.equals()` для ненулевых ссылок и равенству для `null`; для буквального (ссылочного) равенства в Скале нужно использовать метод eq, определённый для класса `AnyRef`.

Тип переменной определяется через двоеточие: `var s : String`. Функции и методы определяются через `def`:

```
def square(n : Int) = n * n
```

Вот как в Скале выглядят шаблоны (generics):

```
var m : Map[String, List[Timestamp]]
...
val s = m(”today”)(5)
```

(Здесь мы извлекаем список по ключу `”today”`, затем берём пятый элемент списка.)

В отличие от Джавы, шаблоны могут быть высших порядков (см. [9]):

```
trait Functor[F[_]] {
  def fmap[A, B](fa: F[A], f: A => B): F[B]
}
```

Такая конструкция невозможна в Джаве (см., например, [8]). Невозможно написать вот такой интерфейс:

```
interface Functor<F<?> extends Collection<>> {
  public <A, B> F<B> fmap(A a, Function<A, B> f);
}
```

Циклы можно писать практически как в Джаве; цикл `while` ничем не отличается, а цикл, известный под названием «foreach», выглядит так:

```
def main(args: Array[String]) {
    for (arg <- args)
        println(arg + ”: ” + calculate(arg))
}
```

Можно писать и более вычурные циклы:

```
for (x <- expr1; if expr2; y <- expr3) ...
```

Это, конечно, эквивалентно следующему:

```
for (x <- expr1) {
  if (expr2) {
    for (y <- expr3) ...
  }
}
```

Долгожданная новинка — замыкания (closures). Конечно, в Джаве тоже можно написать что-то вроде замыканий, используя интерфейсы и анонимные классы, но на практике всё это выливается в совершенно нечитаемый код. На Скале замыкания выглядят гораздо более естественно:

```
List("abc.scala", "xyz.py", "file.sh", "myclass.java").
    filter (name => name.endsWith(".java"))
```

Разумеется, джавный стиль программирования — это только верхушка айсберга; им удобно пользоваться, если вы только переходите с Джавы на Скалу. По мере освоения вы обнаружите, что Скала способна на очень многое, практически невыразимое на Джаве.

Две новых особенности языка (относительно Джавы) удивят и кого-то порадуют, а кого-то и нет: неявные преобразования и миксины.

В Джаве мы привыкли, что `int` приводится к `long` и что в определённом контексте любой объект приводится к `String` — а в Скале этими преобразованиями можно управлять. В качестве примера возьмём строку, и превратим её, с помощью неявного преобразования, в объект, очень похожий на массив:

```
implicit def randomAccess (s:String) =
    new RandomAccessSeq[Char] {
      def length = s.length
      def apply(i: Int) = s.charAt(i)
}
```

Теперь можно писать `val c = "This is a string"(2)` — строка будет неявно преобразована в `RandomAccessSeq[Char]`, а значение c будет вычислено и равно `'i'`. Ниже приводятся другие примеры неявных преобразований.

Теперь мы можем обращаться со строками, как если бы это были массивы:

```
// returns true if string contains a digit
"abc123" exists (_.isDigit)
```

Миксины в Скале реализованы с помощью типажей. Миксины позволяют добавлять к классу функциональность, определённую не в непосредственном суперклассе, а где-то ещё; в Скале типаж может содержать определения методов. Вот простой пример:

```
class Human(name: String, sex: Sex) {
  def ping() { println(”?!”) }
}

class Russian(firstName: String,
              patronymic: String,
              lastName: String,
              sex: Sex)
    extends Human(firstName + ” ” + patronymic + ” ” +
      lastName, sex) {
     ...
    }

trait Programmer {
  override def ping() { println(”I’m busy!”) }
  def debug(byte[] binary) { println(”omg...”) }
}

class SovietProgrammer(firstName: String,
                       patronymic: String,
                       lastName: String,
                       birthDate: Timestamp)
    extends Russian(firstName, patronymic, lastName,
      null)
    with Programmer {
 ...
}
```

Здесь методы debug() и ping наследуются из типажа Programmer.

3.4. С точки зрения функциональных языков

Несмотря на то, что Скала бегает на JVM и предоставляет все возможности «объектного программирования», это вполне функциональный язык. Функция может передаваться в качестве параметра, может быть значением переменной, может быть анонимным литералом, может возвращаться:

```
def myFun(x:Integer) = ”Look, ” + x + ” * ” + x + ” = ”
    + (x * x)
val sameThing = (x: Integer) => ”Look, ” + x + ” * ” +
    x + ” = ” + (x * x)
List(1,2,3) map {x => x * x * x}
List(0., 3.14159265358) map { sin(_) }
def const[X, Y](y: Y) = (x: X) => y
def c123 = const(123)
val x = c123(”abc”) // it is 123
```

В последних двух примерах мы взяли списки и применили к каждому элементу функцию, получая новый список. В Скале имеется масса традиционных для ФП операций над списками, например:

```
val list = ”Clatto” :: ”Verata” :: ”Nicto” :: Nil //
    Same as List(”Clatto”, ”Verata”, ”Nicto”)
val list = List(”double”, ”double”) ::: List(”toil”,
    ”and”, ”trouble”) // concatenation
list.exists(x => x.toString.length == 4) // wtf method
    of looking for TRUE
list.drop(2).dropRight(4).filter(s => (s endsWith ”y”))
list.map(s => ”\”” + s + ”\””) // list comprehension
List(1,2).flatMap(List(123,_,456)) // lists built by
    List(123,_,456) are concatenated
list.foreach(print)
list.reverse
list.head
list.tail
list.last
```

К спискам, в частности, применимы свёртки:

```
val list = List(1,2,3,4,5)
// folds the list, adding up the elements:
list.foldLeft(0)(_+_)
```

Есть кортежи (ака n-ки); есть аналог data types, есть карринг — многое из того, ради чего люди переходят на Хаскель. Но так как язык этот в принципе-то императивный, то при вводе-выводе можно формально обойтись без монад. Пример кортежа:

```
val x = (123, ”abc”, new Date)
```

Пример алгебраического типа:

```
abstract class Option[T]
case class None[T]() extends Option[T]
case class Some[T](value: T) extends Option[T]
```

`case class` — особый вид класса, в частности, пригодный для использования в конструкции switch. В таком классе много чего хорошего; казалось бы, почему не сделать все классы case-классами? Проблема в том, что сравнение с образцом становится проблематичным при наличии наследования; поэтому у таких классов наследование ограничено.

Пример карринга:

```
def sum(x:Int)(y:Int) = x + y
sum(1)(2)
```

Здесь sum определяется как функция, которая принимает целочисленный аргумент (x) и возвращает функцию, которая принимает целочисленный аргумент (y). Если мы напишем `sum(5)`, то это и будет называться «карринг».

Классы типов (type classes) реализованы с помощью типажей и полиморфизма, например:

```
trait Functor[F[_]] {
  def fmap[A, B](fa: F[A], f: A => B): F[B]
}
```

Выше, в разделе «С точки зрения Джавы», этот же пример приведён в качестве образца шаблона высшего порядка... на одну и ту же вещь можно по-разному смотреть.

Вместо переменных в Скале используется `val` — такие значения определяются один раз и не меняются. Можно использовать и `var` — тогда это будет обычная переменная, как в Джаве. `val` можно объявить ленивой:

```
lazy val x = buildBigThingThatTakesLong(parameters)
```

В этом случае значение вычисляется только в момент, когда оно требуется, в выражении или в качестве неленивого параметра.
В качестве альтернативы ленивой переменной можно использовать функцию без параметров:

```
def x = buildBigThingThatTakesLong(parameters)
```

Между ленивой переменной и функцией без параметров имеется существенное различие: ленивая переменная вычисляется один раз, а функция — каждый раз.

Ещё Скала замечательна сопоставлением с образцом (*pattern matching*; подробнее на с. 36) и частичными функциями.

Как задаётся частичная функция? Через сопоставление с образцом:

```
val second : List[Int] => Int = {
  case x::y::_ => y
}
```

Это эквивалентно следующему коду:

```
val second = new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }

  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
```

(Выражение `x::y::z::Nil` эквивалентно выражению `List(x, y, z)`).
Частичная функция может использоваться по-разному:

```
val pf : PartialFunction[X,Y] = ...
val y = pf(x) // exception happens if pf is not defined
    on the value of x
val yOpt : Option[Y] = pf.lift(x) // returns either
    Some(y) or None
val b : Boolean = pf.isDefinedAt(x)
val pfAlt : PartialFunction[X,Y] = ...
val y = pf.orElse(pf1) // if (pf.isDefinedAt(x)) pf(x)
    else pf1(x)
```

Естественно, возникает вопрос: как же так, язык объектно-ориентированный; есть методы у объектов — как это всё сочетается с наличием обычных функций, не методов? Ну вот взять, например,

```
class A(i:Integer) {
  def f1(j: Integer, k: Integer) = i+j+k
```

```
    val f2 = (j:Integer, k: Integer) => i+j+k
  }
```

Здесь очевидно, что f1 — метод, а f2 — функция. Какая разница между f1 и f2? Можно ли написать val f3 = f1? Нет, нельзя. Чтобы превратить метод класса в полноправную функцию (т. е., экземпляр типа Function), нужно, формально говоря, добавить подчёркивание после пробела:

```
  val f3 = f1 _
```

Эта запись буквально означает, что мы определяем новое значение, имеющее тип «функция», и эта функция имеет те же параметры, что и метод f1, и тот же тип результата; и для вычисления f3 нужно подставить её параметры в f1. Так как f1 вполне может использовать члены класса, в котором она определена, то мы получаем замыкание (closure). f3 можно передавать в качестве параметра или возвращать; это теперь самостоятельная сущность.

Обычные списки в Скале неленивы: если функция возвращает список, то все его элементы должны быть вычислены. Но есть ленивые потоки (Stream). Вот, например, направляем ленивый поток целых чисел в решето Эратосфена и получаем ленивый поток простых чисел:

```
def sieve(s: Stream[Int]): Stream[Int] =
  Stream.cons(s.head, sieve(s.tail filter {_ % s.head
     != 0}))

def primes = sieve(Stream from 2)

primes take 100 foreach println
```

Очень легко создавать встроенные предметно-ориентированные языки (DSL), см. например, [5]:

```
val orders = List[Order](

  // use premium pricing strategy
  new Order to buy(100 sharesOf "IBM")
            maxUnitPrice 300
            using premiumPricing,

  // use default pricing strategy
  new Order to buy(200 sharesOf "GOOGLE")
            maxUnitPrice 300
```

```
              using defaultPricing,

  // use custom pricing strategy
  new Order to sell(200 bondsOf ”Sun”)
              maxUnitPrice 300
              using {
                (qty, unit) => qty * unit - 500
              }
)
```

Как ни странно это выглядит, но всё это — Скала, а не макросы и не текст для парсера. Как и в предыдущих примерах, здесь мы пользуемся удобствами языка.

3.5. Сопоставление с образцом: подробности

Одна из приятнейших особенностей Скалы — сопоставление с образцом. После Джавы — сплошное удовольствие.

Сначала простой пример. Сопоставляем значения, как в обычном switch/case:

```
def matchMe (s: String): Int = s match {
  case ”ichi” => 1
  case ”ni”   => 2
  case ”san”  => 3
  case _      => -1
}
```

На самом деле Джава с 2007 года обещает конструкцию switch и для строк… Скоро будет.

В Скале с помощью образцов можно сопоставлять не только значения, но и алгебраические типы данных, особенно учитывая, что они допускают деконструкцию:

```
trait Expr {
  case class Num(value : int) extends Expr
  case class Var(name : String) extends Expr
  case class Mul(lft : Expr, rgt : Expr) extends Expr
}
...
// Simplification rule:
```

```
expr match {
  case Mul(x, Num(1)) => x
  case _ => e
}
```

Сопоставление с образцом, вместе с регулярными выражениями и с экстракторами, даёт удобную возможность разбирать текст:

```
scala> val Entry = """(\w+)\s*=\s*(\d+)""".r
Entry: scala.util.matching.Regex = (\w+)\s*=\s*(\w+)

scala> def parse(s: String) = s match { case Entry(x,y)
    => (x, y) }
parse: (s: String)(String, Int)

scala> parse("salary=120")
res50: (String, Int) = (salary,120)
```

Это, конечно, если язык описывается регулярным выражением. Если же грамматика посложнее, то нужно писать парсер-комбинатор, как в нижеследующем примере.

```
// a regular tree, not binary, Tree[X] = X + Tree[X]^n
abstract class Tree[X] // main class for a tree
case class Leaf[X](v:X) extends Tree[X]
case class Branch[X](kids:List[Tree[X]]) extends Tree[X]

class TreeParser extends JavaTokenParsers {
  // A tree is either a leaf or a branch
  private def node : Parser[Tree[String]] =
      leaf | branch
  // a branch is a sequence of trees in parentheses;
  // postprocessing consists of calling a constructor
  private def branch : Parser[Branch[String]] =
    "("~repsep(node, ",")~")" ^^ { case "("~nodes~")"
       => Branch(List() ++ nodes) }
  // a leaf is a string wrapped in a constructor
  private def leaf : Parser[Leaf[String]] =
      string ^^ {case s => Leaf(s)}
  // a string is a string is a string
  private def string : Parser[String] =
     regex("""\w+""".r)
  // this is all that's exposed
```

```
    def read(input: CharSequence) = parseAll(node,
        input).get
}

def parseTree(input: CharSequence) = (new
    TreeParser).read(input)

scala> parse("(a,(b,(c,de,(),f)),g)")
res58: Tree[String] = Branch(List(Leaf(a),
    Branch(List(Leaf(b), Branch(List(Leaf(c), Leaf(de),
    Branch(List()), Leaf(f))))), Leaf(g)))
```

3.6. Встроенный XML

Для определённого рода приложений (ну, скажем, для веба) тот факт, что XML является в Скале частью языка, освобождает от необходимости писать массу рутинного ненужного кода. Вот пример:

```
class Person(val id: Int,
             val firstName: String,
             val lastName: String,
             val title: String) {

    def toXML = {
        <person>
            <id>{id}</id>
            <firstName>{firstName}</firstName>
            <lastName>{lastName}</lastName>
            <title>{title}</title>
        </person>
    }
}
```

3.7. Примеры кода

Любимый хаскельцами квиксорт, правда, в неленивом варианте (т. е., не имеющий практического смысла), т. к. списки не ленивы:

```
def qsort(list: List[Int]): List[Int] =
    list match {
```

```
        case Nil => Nil
        case x::xs =>
             qsort(xs.filter(_ < x)) :::
             x ::
             qsort(xs.filter(_ >= x))
    }
```

Здесь мы берём список, и, если он пуст, возвращаем его же, а иначе берём первый элемент; сортируем элементы, которые меньше этого первого; сортируем элементы, которые больше этого первого; наконец, соединяем всё в нужном порядке.

Хитрый трюк с неявным преобразованием, позволяющий определять операторы на существующих классах:

```
>  class Fact (n: Int) {
>    def fact(n: Int): BigInt = if (n == 0) 1 else
   fact(n-1) * n
>    def ! = fact(n)
>  }
>  implicit def int2fact(n: Int) = new Fact(n)
>
> println(42!)

1405006117752879898543142606244511569936384000000000
```

Что тут у нас получилось? Мы вычисляем `42!`; т. к. оператор `!` определён на значениях типа `Fact`, то мы пробуем неявное преобразование `Integer` в `Fact` — а так как такое преобразование определено (`int2fact`), то его и используем. После чего к результату применяем оператор `!` — и получаем в результате `BigInt`; его и печатаем в `println()`.

3.8. Тестирование

Для юниттестов имеется два пакета: `scalatest` ([12]) и `scalacheck`. Вот типичный пример теста для `scalatest`:

```
import org.scalatest.FlatSpec
import org.scalatest.matchers.ShouldMatchers

class StackSpec extends FlatSpec with ShouldMatchers {
```

```
    "A Stack" should "pop values in LIFO order" in {
      val stack = new Stack[Int]
      stack.push(1)
      stack.push(2)
      stack.pop() should equal (2)
      stack.pop() should equal (1)
    }

    it should "throw Exception on popping empty stack" in
       {
      val emptyStack = new Stack[String]
      evaluating { emptyStack.pop() }
          should produce [NoSuchElementException]
    }
  }
```

В версии для scalacheck это может выглядеть примерно так:

```
  val stackIsLIFO = forall {
    (stack: Stack[Int], x: Int, y: Int) =>
     stack.push(x)
     stack.push(y)
     (y != stack.pop) |: "stack top should pop" &&
     (x != stack.pop) |: "stack bottom lost"
  }
```

scalacheck для тестирования генерирует порядочное количество различных стеков, проверяя указанные свойства. Какие именно стеки сгенерируются, зависит от интеллекта (искусственного) — необходимый набор тестовых примеров выводится из условий и из определения.
Но можно писать и по-простому, как в JUnit:

```
  def testParse_positive_5 {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    assert(2 == stack.pop())
    assert(1 == stack.pop())
    assert(stack.isEmpty)
  }
```

3.9. Готовые продукты на Скале

Скалу можно скачать с http://www.scala-lang.org/downloads; этот дистрибутив включает в себя компилятор, библиотеки, repl-интерпретатор, и документацию. Очень рекомендую прекрасный плагин для Скалы в IntelliJ: этот плагин делает всё, что надо — рефакторинг, юниттесты, есть и отладчик. Версия 2.8 плагина для Eclipse тоже вполне работоспособна; существует также плагин для Netbeans.

Ну и наконец, Lift, ради которого уже стоит срочно браться за Скалу, так как этот веб-фреймворк лет на десять обогнал большинство остальных.

Lift, фреймворк для создания веб-приложений, разумеется, опирается на всё, что было достигнуто в вебе за последние десять лет, но он идёт дальше. Прежде всего, это строгое разделение компонент MVC; затем, каждая форма уникальна, и невозможно передать одну и ту же информацию дважды или вернуться на ту же страницу: каждый экземпляр формы содержит уникальную метку. Так как Скала может содержать XML, не нужны отдельные JSP, это всё пишется прямо в коде — но презентация отделена от логики структурой приложения.

Вот что сказал Мартин Одерски, дизайнер Скалы: «Lift — единственный фреймворк из появившихся за последние годы, предлагающий свежие и оригинальные решения для разработки веб-приложений. Это не какое-нибудь незначительное улучшение предыдущих решений; он задаёт новые стандарты. Если вы разрабатываете веб-приложения, вы должны изучить Lift. Если вы даже не используете его ежедневно, он изменит ваш подход к веб-приложениям.»

Образец HTML и соответствующего кода на Скале:

```
<table>
 <lift:Show.users>
   <tr>
    <td><f:first_name>David</f:first_name></td>
    <td><f:last_name>Pollak</f:last_name></td>
   </tr>
 </lift:Show.users>
</table>

class Show {
  def users(xhtml: NodeSeq) =
   Users.findAll.flatMap(user => bind("f",
```

```
            xhtml, ”first_name” -> user.firstName,
                  ”last_name” -> user.nameName))
   }
```

3.10. Где что на нём делают

Самый сейчас передовой пример — это foursquare.com, небольшой нью-йоркский стартап, чья социальная сеть следит за вашим местоположением и даёт возможность пригласить всех в понравившийся вам ресторан и отследить, кто где находится. За полгода весь код переписан на ScalaLift.

Yammer: Artie, служба рассылки сообщений, написана на Скале.

Twitter: порядочное количество серверного кода переписано на Скалу, а остальное пока на Ruby on Rails.

LinkedIn: там было порядочное количество скальщиков, но, пока писалась эта статья, они все, похоже, уже разбежались — как и из Гугла до того.

KaChing: желающие пишут на Скале, хотя основная масса кода на Джаве; проблем совместимости нет.

Есть ещё пара стартапов в Сан-Франциско, пишущих исключительно на Скале; но если вы пойдёте на Dice, то обнаружите с десяток контор, где требуется Скала, даже в таких экзотических местах, как штат Теннесси.

3.11. Рекомендуемая литература

David Pollack «Beginning Scala» [10] — пособие для начинающих. Если вы серьёзно хотите изучить и использовать Скалу, вряд ли вам нужна эта книга. Для серьёзного изучения лучше всего подойдёт

Martin Oderski, Lexi Spoon, Bill Venners «Programming in Scala» [13]. Большая книга, но её не обязательно читать вдоль, можно и поперёк. [14] — перевод главы из этой книги.

Dean Wampler, Alex Payne «Programming Scala» [1]. В этой книге меньше страниц, но больше Скалы, различных пикантных деталей. Хороша в качестве «второй книги» по этому языку.

[18], [16], [17] — три статьи Антона Панасенко, дельно описывающие те или иные аспекты языка.

[15] — русский перевод статьи «Обзор языка Scala» Мартина Одерски и множества других.

[9] — проект ScalaZ, Тони Морриса (Tony Morris). Читается как стихи Лорки в оригинале.

Ну и, наконец, для настоящих героев — книга Gregory Meredith «Pro Scala: Monadic Design Patterns for the Web» [7]. Мередит — большой теоретик, и читать его непросто. Но стоит того: если вы научитесь рассуждать на его уровне, то уже ничего не страшно.

Derek Chen-Becker, Tyler Weir, Marius Danciu «The Definitive Guide to Lift» [6]. Небольшая книга по ScalaLift. Так как Lift развивается быстрее, чем печатаются книги, то лучше информацию черпать из следующих источников: [4], [3], [2].

Литература

[1] *Alex Payne D. W.* Programming Scala: Scalability = Functional Programming + Objects, URL: http://programming-scala.labs.oreilly.com/ (дата обращения: 29 ноября 2010 г.). — O'Reilly, 2009.

[2] *Chen-Becker D.* Проект на Lift — «Мелочь в Кармане». — Проект, URL: http://github.com/tjweir/pocketchangeapp/tree/master/PocketChange (дата обращения: 29 ноября 2010 г.).

[3] *David Pollak Derek Chen-Becker M. D., Weir. T.* Starting with Lift. — Веб-страница, URL: http://old.liftweb.net/docs/getting_started/mod_master.html (дата обращения: 29 ноября 2010 г.).

[4] *et al. D. P.* Getting Started With Lift. — Веб-сайт, URL: http://liftweb.net/getting_started (дата обращения: 29 ноября 2010 г.).

[5] *Ghosh D.* Designing Internal DSLs in Scala. — Блог, URL: http://debasishg.blogspot.com/2008/05/designing-internal-dsls-in-scala.html (дата обращения: 29 ноября 2010 г.).

[6] *Marius Danciu Derek Chen-Becker T. W.* The Definitive Guide to Lift: A Scala-based Web Framework, URL: http://www.amazon.com/Definitive-Guide-Lift-Scala-based-Framework/dp/1430224215 (дата обращения: 29 ноября 2010 г.). — Apress, 2007.

[7] *Meredith G.* Pro Scala: Monadic Design Patterns for the Web, URL: http://www.amazon.com/Pro-Scala-Monadic-Design-Patterns/dp/143022844X (дата обращения: 29 ноября 2010 г.). — Apress, 2010?

[8] *Moresmaugh J.* Java and higher order generics. — Блог, URL: http://jpmoresmau.blogspot.com/2007/12/java-and-higher-order-generics.html (дата обращения: 29 ноября 2010 г.).

[9] *Morris T.* Проект scalaz. — Проект в Google Code, URL: http://code.google.com/p/scalaz/ (дата обращения: 29 ноября 2010 г.).

[10] *Pollack D.* Beginning Scala, URL: http://www.amazon.com/Beginning-Scala-David-Pollak/dp/1430219890 (дата обращения: 29 ноября 2010 г.). — Apress, 2009.

[11] *Rupp D.* Java generics broken? we report, you decide. — Блог, URL: http://davidrupp.blogspot.com/2008/01/java-generics-broken-we-report-you.html (дата обращения: 29 ноября 2010 г.).

[12] *Venners B.* Scalatest. — Проект, URL: http://www.scalatest.org/ (дата обращения: 29 ноября 2010 г.).

[13] *Venners B., Odersky M., Spoon L.* Programming in Scala: A Comprehensive Step-by-step Guide, URL: http://www.amazon.com/Programming-Scala-Comprehensive-Step---step/dp/0981531601/ (дата обращения: 29 ноября 2010 г.). — Artima, 2008.

[14] *Лекси Спун, Бил Веннерс, Мартин Одерски.* Первые шаги в Scala. — RSDN, URL: http://www.rsdn.ru/article/scala/scala.xml (дата обращения: 29 ноября 2010 г.).

[15] *Мартин Одерски и другие.* Обзор языка программирования Scala. — RSDN, URL: http://www.rsdn.ru/article/philosophy/Scala.xml (дата обращения: 29 ноября 2010 г.).

[16] *Панасенко ☒.* Scala: Actors (part 2). — Блог, URL: http://blog.apanasenko.me/2009/12/scala-actors-part-2/ (дата обращения: 29 ноября 2010 г.).

[17] *Панасенко ☒.* Scala: Functional Language (part 3). — Блог, URL: http://blog.apanasenko.me/2009/12/scala-functional-language-part-3/ (дата обращения: 29 ноября 2010 г.).

[18] *Панасенко ☒.* Scala: введение в мир FL JVM (part 1). — Блог, URL: http://blog.apanasenko.me/2009/12/scala-fl-jvm-part-1/ (дата обращения: 29 ноября 2010 г.).

Сравнение Erlang и Node.js

Дмитрий Демещук
dem@fprog.ru

Аннотация

Многие программисты, пишущие на JavaScript, сокрушаются: «Ну почему на JS можно писать только клиентскую часть? Так хочется написать полноценный клиент-серверный продукт на одном языке».

С недавних пор такое желание стало осуществимым. На базе созданного датской командой Google движка V8, компилирующего JavaScript в машинный код, создана платформа Node.js (далее — Node) — технология, позволяющая писать полноценные асинхронные серверные приложения на JavaScript. Технология быстро получила популярность у веб-разработчиков и уже применяется в серьезных проектах с большими нагрузками для асинхронной обработки данных.

В данной статье представлен краткий обзор возможностей и архитектуры Node, а также сравнение ее с платформой Erlang, ранее использовавшейся в основном в телеком-проектах, но теперь все чаще применяемой и в веб-сфере.

The V8 engine, created by the danish Google team, which compiles JavaScript to machine code, has given rise to the Node.js technology. It allows one to write full-fledged asynchronous server applications in JavaScript.

This article gives a short overview of the features and architecture of Node.js and compares it to the Erlang platform, which was mostly used in telecom projects before, but lately has found more and more use in the context of web development.

4.1. Введение

Erlang создавался компанией Ericsson для использования в телекоммуникационных системах. Сперва разработчики в течение трех лет экспериментировали с различными языками, выявляя их достоинства и недостатки для поставленной задачи. Для экономии времени разработчиков требовался язык высокого уровня, освобождавший их от рутинной работы вроде ручного выделения и освобождения памяти, и позволяющий программисту сфокусироваться на оперировании абстракциями предметной области. Из первоначального списка из примерно 30 языков победителями вышли Lisp, Parlog и Prolog; от последнего Erlang и унаследовал большую часть синтаксиса.

Основными требованиями к новому языку были: параллелизм (нужна была возможность обслуживать тысячи клиентов одновременно), отказоустойчивость (телеком-системы слишком масштабны, чтобы самому программисту имело смысл даже пытаться предусмотреть все возможные ошибки) и возможность обновления кода на лету, без останова выполнения программы.

И вот, в 1987 году появился первый прототип Erlang, а через пять лет он уже стал использоваться в двух production-системах. С тех пор в Erlang было внесено множество изменений: добавлены инструменты для создания распределенных систем, оптимизированы многие механизмы, написано множество вспомогательных библиотек, и т. д.

Сейчас Erlang используется в production многими компаниями. Яндекс и Facebook используют ejabberd — многофункциональный сервер мгновенного обмена сообщениями, написанный на Erlang. Сервисы многих телеком-операторов (T-Mobile, Telia, Mobilearts, Cellpoint, Ericsson и др.) используют платформу Erlang, годами не останавливая выполняющийся на своих серверах код. Применяется Erlang и в системах мониторинга (Corelatus), платежных системах (Kreditor). Наконец, получивший популярность RabbitMQ, использующийся тысячами компаний для связи между различными компонентами своих систем, тоже написан на Erlang.

Основная причина появления Node.js — популярность языка JavaScript. По сути, JS был и остается единственным универсальным языком для создания фронт-энда в вебе. Его нишу неоднократно пытались занять другие технологии (Flash, Silverlight), но в конце-концов они проигрывали и отходили на второй план.

Развитие веб-технологий стало еще больше способствовать популярно-

сти JavaScript. Чтобы понять масштабность этого развития, можно взглянуть на тренды вакансий. Например, в начале 2005 года по данным indeed.com спрос на JS-программистов был вдвое меньше, чем спрос на разработчиков на C++. Сейчас, по тем же данным, JS уже обогнал C++, и темпы роста спроса на него и не думают снижаться.

Но дело не только в своего рода монополии JavaScript. Другая причина его популярности — удобство. Вобрав в себя немного от объектной и немного от функциональной парадигм, JS завоевал любовь множества программистов своей простотой и удобством.

Второй основной предпосылкой для создания Node.js явилось сильное возрастание производительности браузерных JS-движков. Постоянная гонка веб-браузеров заставила разработчиков оптимизировать JavaScript-движки (Rhino и SpiderMonkey от Mozilla, V8 от Google, Chakra — движок Internet Explorer 9.0 и др.), делая их максимально быстрыми, легкими и производительными. Многие программы, написанные на чистом C, будут работать медленнее, чем код, сгенерированный таким движком из JavaScript. Node, будучи основанным на гугловском движке V8, стал, пожалуй, одной из самых производительных серверных технологий.

Эти два фактора быстро сделали Node.js очень популярным в вебе. У него очень большое и активное сообщество[1], постоянно появляются новые библиотеки и фреймворки различного назначения.

Несмотря на то, что технология совсем молодая, многие уже с успехом используют ее в работающих проектах. Среди самых известных — Yammer, сервис корпоративного микроблоггинга, Plurk — крупнейший азиатский микроблоггинговый сервис, Adcloud — рекламный сервис, базирующийся на облаках Amazon EC2, и многие другие.

4.2. Язык твой — друг твой

JavaScript, как очень известный и популярный язык, в особом представлении не нуждается. Главный его инструмент — first-class functions, то есть возможность использовать функции как полноценный тип данных (так же как числа, массивы или объекты). Функции можно объявлять анонимно, замыкать в них переменные, объявленные вне тела функции, передавать в ка-

[1] Так, на Github насчитывается больше тысячи репозиториев, так или иначе использующих эту технологию.

честве аргументов, сохранять в структурах данных (списках и словарях), а потом вызывать в любой нужный момент.

Для тех, кто пришел на него с объектных языков, подобных C++ или Java, JS покажется гораздо более привычным, чем Erlang. Имея почти полноценные объекты и подобие классов (сочетание примитивов и наследования через прототипы), дающие возможность наследования и полиморфизма во вполне привычном виде, и позволяя компенсировать отсутствие некоторых возможностей из ООП своей функциональной природой (например, инкапсуляцию можно сделать через замыкания), он позволяет разработчику писать в знакомом ему объектном стиле.

Кроме того, JS поддерживает хорошо зарекомендовавший себя формат данных JSON, который почти повсеместно используется для хранения и передачи данных, причем даже не только в Web. А в версии 1.8 стандарта появилась еще и нативная поддержка XML, позволяющая эффективно работать с XML — E4X.

Начиная с версии стандарта 1.7, JavaScript начал поддерживать destructuring assignment — присваивание, являющееся почти полноценным аналогом сопоставления с образцом, присущего высокоуровневым функциональным языкам:

```
var bar = 'foo',
    foo = 'bar';

[foo, bar] = [bar, foo];
```

С приходом пятой версии стандарта ECMAScript, фундамента языка JavaScript, в JS появился весьма полезный метод `Object.freeze()`, позволяющий делать объект неизменяемым (immutable) — впрочем, речь идет о «поверхностной» заморозке, а не о «глубокой». О пользе immutability будет рассказано далее. Интересно то, что в JS immutability может быть задействована по желанию программиста, позволяя использовать достоинства обоих методов там, где это нужно. `Object.freeze()` поддерживается и в Node.

Наконец, JS в ходе своего развития приобрел много вкусного синтаксического сахара, который позволяет писать более элегантный и лаконичный код.

Следует заметить, что, несмотря на очень низкий порог вхождения (буквально несколько дней), в JS есть некоторые моменты, которые обычно плохо понятны новичкам и требуют несколько больше времени, нежели постижение «обязательных» азов. Сюда входят замыкания, наследование через

прототипы, особенности контекстов вызова функций, и некоторые другие тонкости языка. Конечно, инкапсуляция и наследование не являются обязательными для понимания, и без них тоже можно писать работающий код, но с ними выходит гораздо короче и элегантнее, да и ошибок меньше.

Что очень важно, язык все еще не стоит на месте и продолжает развиваться. Жаль только, что браузеры пока поддерживают совсем разные стандарты, да еще и с разной степенью добросовестности. Но старые браузеры уходят, а новые поддерживают уже более современные стандарты, поэтому рост продолжается, несмотря на эти сложности.

Erlang на фоне JS выглядит гораздо более непривычно. Первое, что сбивает с толку — это immutability (отсутствие присваивания). Переменные — на самом деле не переменные, а имена, раз и навсегда связанные со значениями. Как позже будет показано, это качество языка, несмотря на свою непривычность, избавляет программиста от множества возможных ошибок. Кроме того, неизменяемые переменные очень сильно упрощают процесс рецензирования кода: не нужно запоминать или искать, какое последнее значение мы присвоили переменной.

Ещё одна особенность — в Erlang весьма неуклюжие механизмы работы с глобальными переменными.[2] Платформа, таким образом, словно бы дополнительно поощряет стремление избегать их использования.

Массивов и объектов в привычном понимании здесь нет. Зато есть списки и кортежи, из которых уже рождаются другие, присущие в основном функциональным языкам, типы данных: словари, очереди, множества и т. д. Есть, впрочем, и массивы, реализованные в виде отдельного модуля (реализация всё же основана на деревьях с высокой степенью ветвления), но на практике они редко используются, гораздо реже обычных списков.

В Erlang есть first-class functions, основа функциональной парадигмы. Здесь также есть замыкания и лямбды (анонимные функции). В отличие от Node.js, присутствует поддержка хвостовых вызовов. Они не только дают преимущество хвостовой рекурсии, не требующей стека вызовов, но и позволяют эффективно реализовывать интересные и полезные механизмы, например, конечные автоматы. Через них же в Erlang и ему подобных языках

[2] По сути, в Erlang нет привычных глобальных переменных, а есть process dictionary и настройки приложения. Первое сами создатели Erlang называют злом, которое следует использовать только в очень редких случаях, по необходимости, а второе используется в основном при инициализации приложения, но не при его работе.

реализуются циклы. От своих функциональных предков Erlang получил и сопоставление с образцом. Оно позволяет писать лаконичный код и помогает удобно обрабатывать ошибки.

Отдельного упоминания заслуживает OTP (Open Telecom Platform), набор готовых Erlang-библиотек. OTP содержит, например, такие «модели поведения (behaviors)» (термин из OTP), как «обработчик событий» (`gen_event`) и «конечный автомат» (`gen_fsm`).

Ещё стоит отметить возможности Erlang при работе с бинарными данными. Сочетая в себе удобный формат двоичных данных и сопоставление с образцом, Erlang позволяет с легкостью оперировать сложными двоичными протоколами, выполнять кодирование/декодирование данных и т. п.:

```
<< ProtocolVersion:4/integer-unit:8,
   HandshakeMessage:20/binary, SomeBitFlag:1/integer,
   Rest/binary >> = Packet.
```

Несмотря на некоторую необычность и непохожесть на большинство популярных языков программирования, Erlang оказывается довольно быстрым в изучении. Вероятно, этому способствует небольшое количество языковых конструкций. По опыту различных команд разработчиков, порог вхождения для новичков составляет всего около двух недель.

4.3. Многопользовательские и многозадачные

Большинство современных систем необходимо проектировать с расчетом на высокую нагрузку: сотня тысяч пользователей потребует сотню тысяч сетевых соединений, несколько сотен тысяч запросов к базе данных, — и все это надо обслуживать одновременно.

Обе рассматриваемые технологии созданы как раз с расчетом на такие условия — Erlang и Node.js способны обрабатывать сотни тысяч одновременных подключений. Но решается эта задача в обеих платформах по-разному.

Работающая на базе Erlang система изнутри представляет собой множество мельчайших процессов, очень похожих на реализацию Green Threads в некоторых языках, но по поведению более близких к системным процессам, чем к потокам. Они существуют исключительно на уровне виртуальной машины, которая и занимается их планировкой, а снаружи, на уровне ОС, виден только основной процесс — сама виртуальная машина. Эти процессы

легковесны — простейший процесс может занимать в памяти около нескольких сотен байт. На 64-битной архитектуре Erlang позволяет создать их более двухсот миллионов (до 268435456, если быть дотошным). На практике, однако, число одновременно запущенных процессов обычно куда более скромно: десятки или сотни тысяч. Что также очень важно, запуск и остановка таких процессов происходит очень быстро — всего несколько микросекунд.

Процесс в Erlang порождается функцией `spawn()`. В качестве аргумента в нее передается функция, которая будет асинхронно выполняться в порожденном процессе. Все функции, вызываемые в ней, будут выполняться в рамках этого же процесса. Процесс может завершиться в трех случаях: когда при выполнении возникла какая-либо ошибка, когда исходная функция вернула какое-то значение, или же когда процесс просто завершили командой `exit()` (на самом деле есть еще один случай, относящийся к механизму «связывания» процессов). Если, например, функция будет рекурсивно вызывать саму себя, то процесс не завершится, пока не будет «убит», пока не возникнет ошибка, или пока не будет остановлена сама виртуальная машина.

Таким образом, область видимости процесса ограничивается областью видимости его функции, что обеспечивает инкапсуляцию данных. Так как в Erlang нету понятия глобальных переменных, то единственный способ хранить какие-либо данные внутри процесса, представленного рекурсивно вызывающей себя функцией, — это передавать эти данные в качестве аргумента этой функции. Получается своего рода состояние, которое «протягивается» через весь процесс и, возможно, как-то изменяется с течением времени:

```
...
spawn(fun init/0),
...

init() ->
    ZeroState = 1,
    loop(ZeroState).

loop(State) ->
    NewState = do_something(State),
    io:format(”Okay, this is my new internal state:
       ~p~n”, [NewState]),
    loop(NewState).
```

При этом, процессы в Erlang могут общаться между собой посредством

сообщений. Любое сообщение, отправленное процессу, попадает в очередь, а затем может быть считано им из этой очереди. Выходит, данные внутри процесса невозможно менять извне напрямую — можно только послать ему сообщение, реагируя на которое, он как-то изменит свое внутреннее состояние.

С учетом вышенаписанного, процессы можно сравнить с объектами. У объекта тоже есть внутреннее состояние (в нашем случае — все данные в private-доступе) и есть методы (послали процессу сообщение — вызвали его метод).

В свою очередь, Node построен на событиях. Client-side программистам на JavaScript должна быть очень хорошо знакома эти схема: создаем событие, назначаем ему функцию-callback, и продолжаем выполнение программы. В момент возникновения события функция будет выполнена автоматически:

```
var events = require('events');

var server = new events.EventEmitter();

server.on('test', function() {
    console.log('All right. Message received.');
});

server.emit('test');
```

В первую очередь, такой подход удобен для любых IO-операций, которые отнимают больше всего времени: работа с диском, сетью, базой данных. Можно не ожидать ответа от удаленного сервера (он ведь может и вообще не придти), а вместо этого, отправив запрос, продолжить выполнение программы. Когда сервер ответит — выполнится привязанный к соответствующему событию callback.

Вообще говоря, код, выполняющийся таким образом, выполняется строго последовательно. Любая асинхронно вызываемая функция в нужный момент вклинивается в императивно выполняемую последовательность команд. Распараллелить такую схему работы программы непросто, и Node до сих пор не поддерживает SMP, выполняясь на одном ядре процессора. Впрочем, этот недостаток компенсируется прекрасной производительностью и возможностью использования нескольких процессов Node одновременно.

Таким образом, базовой единицей асинхронности в Node является функция. Казалось бы — тот же процесс в Erlang. Но есть существенные отличия.

Во-первых, эта функция выполняется не параллельно остальным асинхронно вызванным функциям, а встраивается в общую очередь. Когда эта функция начинает выполняться, управление остальному коду не передастся, пока функция не завершит работу. Именно поэтому важно после начала какой-либо длинной операции завершать работу функции, давая возможность выполняться остальному коду, и продолжить работу с изначальной операцией внутри callback'а, повешенного на событие окончания этой длительной операции.

Во-вторых, любая функция видит не только контекст, в котором была объявлена, но и контекст глобальных переменных. И это выход в случаях, когда асинхронно выполняющимся функциям надо как-то обмениваться данными. Если две функции были вызваны в одном контексте, они могут использовать локальную переменную, (желательно через замыкание, для хотя бы частичной защиты данных), либо, если контексты вызова у них совсем разные, есть еще выход в виде глобального пространства имен.

Как будет выглядеть, например, веб-сервер, написанный на обеих платформах?

Классическое решение на Erlang — создать процесс, который будет непрерывно слушать на заданном порту, а, приняв новое подключение, создавать новый процесс, который и будет это подключение обрабатывать (пример простейшего веб-сервера на Erlang можно посмотреть здесь). Что мы получаем при такой схеме?

1) Разделение труда. Основной процесс занимается только тем, что принимает новые подключения. Каждый порождаемый процесс обрабатывает только его собственного клиента. С точки зрения ООП, мы здесь имеем один объект-диспетчер, принимающий вызовы от клиентов, и множество объектов-обработчиков, каждый из которых выполняется одним и тем же кодом, но работает с разными данными.
2) Что если в одном из клиентских процессов произойдет непредвиденная ошибка, и он завершится? Так как процессы изолированы друг от друга, то память всех остальных запущенных процессов останется в неизмененном состоянии, и вся остальная система не пострадает. Более того, завершаясь, процесс может послать своему родительскому процессу сообщение с деталями о возникшей ошибке, если связать их функцией `link()` или вместо `spawn()` вызвать `spawn_link()`.

Этот момент — первая опора отказоустойчивости Erlang. Если какой-то процесс завершился, остальная система не прекращает свою работу. Поэтому главная мантра при написании кода на Erlang — «Let it fail» («Пусть падает»). Если нужно, ошибку можно вывести в консоль или записать в лог-файл, а часто можно просто закрыть на нее глаза.

Отдельный вопрос — что делать, если завершился главный процесс, который принимает новые подключения. Для этого у Erlang есть еще один мощный инструмент — supervisor, — который будет освещен в следующей главе.

Node поставляет удобный HTTP-сервер прямо в коробке. Создали объект `http`, повесили обработчик на событие нового подключения, вызвали метод `listen()`, указав нужный порт — и простейший веб-сервер готов:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8124/');
```

Что же произойдет, если в функции-callback'е в процессе выполнения возникает ошибка? Если выброшенное исключение не будет отловлено через `catch`, HTTP-сервер прекратит работу, и программа просто завершится.

Для программы с парой-тройкой обработчиков событий это не причиняет неудобств: список возможных ошибок будет невелик, и обрабатывать их будет просто. Но, если система усложняется и начинает представлять собой целое дерево обработчиков, которые создают и контролируют друг друга, количество обработчиков ошибок все возрастает, и контроль над распространением ошибок становится все сложнее.

К счастью, для легкого приложения вышеописанные ситуации вряд ли будут проблемой. Во многих случаях таковых может даже не возникнуть. Однако, с разрастанием кода приходятся все больше следить за возможными ошибками, и внимательно обрабатывать исключения везде, где они могут возникнуть.

Кроме того, памятуя о том, что для обмена сообщениями функциям придется использовать общие куски памяти, придется очень аккуратно следить за их общей частью: изменение этой области памяти повлияет на выполнение всех частей кода, использующих ее. Если функция неожиданно прервала выполнение с ошибкой, эти общие данные могут остаться в промежуточном

противоречивом состоянии, будучи не до конца обработанными. Пусть даже нам не важно успешное завершение данной функции, но все остальные, обращаясь к этим некорректным данным, могут или давать неверный результат, или тоже завершаться с ошибкой.

Сохранять и восстанавливать глобальное состояние после возникших ошибок — задача не из приятных. Поэтому, чем больше будут изолированы друг от друга все функции в программе — тем лучше.

4.4. Сборка надежной конструкции из ненадежных компонентов

В предыдущей главе уже мельком упоминалась еще одна задача, связанная с нашим HTTP-сервером: закрытие слушающего сокета. Если по какой-то причине сокет закрылся, его нужно пересоздать, при этом не завершая программу, которая все еще обрабатывает успевших подключиться до закрытия сокета клиентов.

В Node объект `http` для этого имеет специальное событие `close` — на него можно повесить, например, перезапуск сервера, или сообщение об ошибке. Очевидно, список возможных системных ошибок (закрытие сокета, отсутствие файла на диске, недоступный IP-адрес, и т. д.) довольно ограничен, и по большей части уже предусмотрен в реализованных библиотеках. А значит, на системные события подобного рода всегда есть возможность отреагировать.

Однако, в процессе работы сервера может возникнуть ошибка, не являющаяся системной, но требующая переинициализации объекта. Для того, чтобы понять, как обрабатывать ошибки такого рода, следует рассмотреть принцип работы объекта `net`, на котором и базируется библиотека `http`.

`Net` подписывается на получение от операционной системы уведомлений о новых подключениях и при получении такого уведомления вызывает функцию `accept`, аналогичную по поведению `accept`'у, используемому в C. Вызов `accept`'а завернут в `try`-`catch`, причем дважды: первый, более глубокий `catch` в случае, когда открытых подключений слишком много, временно останавливает сервер, так что тот перестает принимать новые подключения и ждет, пока освободятся файловые дескрипторы. Затем этот `catch` в любом случае вызывает `throw`, пропуская ошибку на уровень выше. Второй `catch` закрывает сокет, если он еще открыт, отписывает объект

от уведомлений от операционной системы, а затем сам посылает объекту сообщение `close`, которое программист может обработать.

Впрочем, создание слушающего сокета тоже завернуто в `try-catch`, и поэтому в случае, например, недостаточных прав или занятого порта, при инициализации объекта будет выброшено исключение.

Наконец, весь цикл работы TCP-сервера охраняет еще один `try-catch`, который, если до него дойдет ошибка, выбросит нас на событие `"error"`. Своего рода перестраховка от редких и совсем уж специфических ошибок.

Итого, объект имеет уже как минимум три различных типа ошибок, различающихся по области распространения: ошибки при создании объекта, ошибки при выполнении внутренних механизмов объекта и, наконец, ошибки, возникающие внутри callback'ов, связанных с определенными событиями этого объекта. Всего же в реализации объекта целых девять блоков `try-catch`, и каждый, само собой, обрабатывает свои специфические случаи. И еще один `try-catch`, скорее всего, придется вставить в функцию-обработчик событий, чтобы как-то отлавливать ошибки при обработке клиентских подключений.

Таким образом, чтобы адекватно разделять и контролировать возникающие в программе исключения, необходимо все блоки программы, в которых возникают одинакового рода ошибки, оборачивать каждый в свой `try-catch`, в котором уже будет происходить обработка ошибки. Для сравнения, в Erlang такая гранулярность `try-catch` автоматически обеспечивается процессами: каждый процесс уже является `try-catch`-блоком.

Впрочем, на практике ситуация с отловом ошибок в Node оказывается не настолько страшной. Так, весь код нативных JS-модулей Node содержит всего лишь два десятка блоков `try-catch` — не так уж и много для реализации полноценного серверного API. Для большой системы, вероятно, потребуется вставить множество своих собственных, а для легкого веб-приложения эти два десятка предусматривают большинство возможных ошибок.

В Erlang задача переинициализации «корневых» механизмов решается немного более общо. Платформа OTP (Open Telecom Platform), неотъемлемая часть любой крупной Erlang-системы, поставляет для таких случаев специальный вид процессов — supervisor.

Задача supervisor'а — контролировать другие процессы. Если какой-то из подконтрольных ему процессов завершается, supervisor автоматически перезапускает его (а может, и все остальные процессы, если выставлена соответствующая опция). Можно задать максимальную частоту перезапусков

(если она будет превышена — supervisor завершится с ошибкой) и максимальное число перезапусков для каждого подконтрольного процесса.

Если наш HTTP-сервер запустить с помощью supervisor'а, то в случае любой ошибки тот просто перезапустит корневой процесс, в котором заново создастся слушающий сокет. При этом, ввиду полной изоляции процессов в Erlang, уже запущенные клиентские процессы затронуты не будут и спокойно доработают до конца.

В предыдущей главе было упомянуто, что первая особенность Erlang, дающая платформе высокую отказоустойчивость — это изолированность процессов друг от друга. Supervisor, таким образом, является вторым столпом отказоустойчивости Erlang. Если процесс для нас важен (например, когда он запускает другие процессы или даже целые компоненты системы), и в случае ошибки его нужно запустить заново — его можно контролировать через supervisor, который его автоматически перезапустит, что бы ни произошло. Удобство здесь заключается в том, что можно абстрагироваться от природы ошибок, просто выполняя корректирующие действия, когда это нужно. Но, как и в случае с клиентскими процессами, любую ошибку корневого процесса тоже можно отследить и, если надо, обработать.

Таким образом, парадигма «Let it fail» работает и здесь: если процесс может упасть — пусть падает, мы его автоматически перезапустим.

Замечательно то, что такое «наплевательское» отношение к процессам в большинстве случаев избавляет программиста от необходимости делить ошибки на какие-либо классы и категории: по фатальности, по области распространения, и т. д. Ведь если сбойный процесс можно завершить или перезапустить, сама ошибка для программы уже не имеет никакого значения. Детали этой ошибки можно записать в логи, чтобы потом программист посмотрел на них и, если конкретно такого рода ошибки нужно предупредить в будущем, внес в код нужные изменения.

Большие системы в Erlang представляют собой целые деревья supervisor'ов, где более мелкие supervisor'ы отвечают за некие однотипные группы процессов или за крупные компоненты системы, и, в свою очередь, эти supervisor'ы контролируются и при необходимости перезапускаются другими, более высокоуровневыми supervisor'ами. При этом, каждый компонент системы, вырванный из контекста этого дерева, может быть ненадежен и падать при каждой неучтенной ошибке. Но, за счет supervisor'ов, система в целом становится стабильной и надежной.

Бывают случаи, когда процессу нельзя позволить завершиться (напри-

мер, если нельзя терять его состояние, даже в случае ошибки). Тогда на помощь придут сопоставление с образцом и старые добрые `try-catch`, которые помогут обработать ошибку и не дать процессу умереть.

Ещё сложнее становится, если некоторым ошибкам нужно дать пройти на уровень выше, к родительскому событийному объекту. Приходится в соответствующем блоке try-catch вычленять такие ошибки и снова делать throw, чтобы обработать эти специфические ошибки уровнем выше.[3] В результате, если ошибке нужно «пропрыгать» таким образом несколько уровней, приходится плодить ненужный код, в сущности делающий одно и то же — пропускающий нужный нам класс ошибок дальше, к вышестоящему try-catch. Ещё хуже, если блоку try-catch придется фильтровать несколько разных классов ошибок: например, одни пропускать дальше, другие обрабатывать, а третьи игнорировать. Каждый такой блок будет превращаться в нагромождение if'ов или case'ов.

4.5. Слон в посудной лавке

Помимо ошибок программных, есть еще и ошибки программистские. Разнообразие и фатальность у вторых порой ничуть не меньше, чем у первых. Интересно посмотреть, насколько Erlang и Node толерантны к программистам-новичкам, часто делающим на первых порах много ошибок, а также какие инструменты предлагают им для улучшения качества своего кода.

Один из плюсов в Node, который, несомненно, понравится и новичкам, и профессионалам — подробный и понятный backtrace. Любая ошибка сопровождается стеком вызовов, классом ошибки, описанием, и т. д. Особенно удобно это качество в сочетании с возможностью Node работать в режиме интерпретатора, превращая консоль Node в полноценный дебаггер. Кроме того, существуют мощные интерактивные дебаггеры для Node, которые даже можно подключать удаленно, например, ndb.

Добавить сюда возможность подсоединять консоль Node к уже запущенному приложению — и получится удобное средство отладки продукта на лету. Учитывая темпы роста Node, вполне возможно, что такая возмож-

[3] Подобные ситуации встречаются, например, в исходном коде tcp-модуля Node.js.

ность скоро будет реализована на уровне самой платформы, либо на уровне фреймворка.

Немного сложнее дело обстоит с терпимостью к ошибкам программиста. Любые глобальные переменные можно запортить, и тогда под угрозу встанет выполнение всей программы. Именно поэтому опытные программисты стараются по максимуму прятать все переменные, замыкая их и делая таким образом недоступными извне. Часто, программа на JS вообще представляет собой анонимную функцию, тут же, на месте вызванную, и таким образом не оставляющую ни одной глобальной переменной.

Наличие вышеупомянутого метода `Object.freeze()` может во многих случаях упростить жизнь, защищая переменные от лишних изменений, но необходимо помнить, «заморожен» ли объект, либо проверять его состояние каждый раз при изменении.

Проблема глобального состояния, общего для нескольких асинхронно выполняющихся функций, выше уже рассматривалась. Ошибки программиста в этом случае тоже могут быть проблемой, так как callback'и оказываются друг от друга не изолированы.

Backtrace в Erlang более скуп и лаконичен и часто больше сбивает новичков с толку, чем помогает им локализовать ошибку. Однако, привыкнув к нему, можно пользоваться им не менее быстро и эффективно, чем более дружественным backtrace'ом из C или того же Node.

Erlang позволяет подключаться к приложению через открытую консоль, что помогает проанализировать состояние системы.

Также, Erlang обладает мощными средствами интроспекции, позволяющими производить отладку системы любой сложности на нужном уровне глубины (например, на уровне только одного типа процессов или даже одного-единственного процесса).[4]

Что касается ошибок программиста, то некоторые неудобные на первый взгляд качества Erlang неожиданным образом защищают систему от вторжений плохого кода. Так, те самые, поначалу столь непривычные, «непеременные переменные» мешают новичку повредить данные, выполнив какие-то неправильные действия в середине кода. И, если, внутри процесса данные защищены иммутабельностью, то межпроцессное взаимодействие даёт дополнительную защиту: при отправке между процессами данные копируются (передаются только по значению, а не по ссылке). Таким образом, данные

[4] Подробнее эта система интроспекции рассмотрена в [1].

труднее затереть или повредить; меньше боязни, что новый написанный модуль повлечет за собой проблемы в работе всей системы.

Изолированность процессов также ограждает приложения от некорректно работающих новых компонентов. Каждое приложение может иметь собственный supervisor, который отвечает за перезапуск только его компонентов, тем самым делая его целиком независимым от всей остальной системы. Понадобилось организовать коммуникацию между компонентами системы — данные внутри другого модуля или приложения не затрагиваются, а вместо этого пересылается сообщение (по сути, вызывается метод объекта-процесса).

Эти свойства тем ценнее, чем больше людей работает над разработкой системы. Изоляция данных, процессов и приложений позволяет не задумываться о коде, который пишут другие, фокусируясь на собственном.

4.6. Немного о распределенности

Распределенные системы, в которых разные компоненты могут находиться на разных машинах, нуждаются в эффективных механизмах удаленного вызова процедур (Remote Procedure Call, сокращенно — RPC). Кроме того, для многоядерных процессоров для повышения производительности необходимо иметь возможность использовать все ядра, что обычно достигается порождением нескольких процессов, каждый из которых выполняется на собственном ядре, а при необходимости эти процессы общаются между собой (этот механизм называется IPC — Inter-Process Communication).

Как уже было упомянуто ранее, процесс работы Node сложно распараллелить. Но все же возможность работы на нескольких ядрах там есть. Node умеет работать с системными процессами и обмениваться с ними данными через stdin/stdout. Метод, конечно, небыстрый, но для многоядерных процессоров должен давать ощутимый выигрыш. Работа с процессами на уровне функционала чистого Node.js очень напоминает процессы в низкоуровневых языках вроде C — неудобно, но не смертельно.

Впрочем, для Node уже сейчас написано очень много библиотек и framework'ов, среди которых есть и интерфейсы для более или менее удобной работы с процессами. И это направление развивается с большой скоростью, так как однопроцессорность Node многим мешает.

C RPC, как и с IPC, тоже не все гладко. Единственное, что предлагает нативный Node для этих целей — старые добрые сокеты. Framework'и упроща-

ют жизнь, но все равно напоминают скорее высокоуровневые client-server-библиотеки, чем полноценные RPC-интерфейсы.

Что касается Erlang, эти строки кода скажут больше, чем любые рассуждения:

```
{Node, ProcessID} ! Message.

receive
   Value  -> io:format(”Process replied: ~p~n”,
      [Value]);
after 1000 -> io:format(”Timeout 1s.~n”)
end.

rpc:call(Node, ModuleName, FunctionName, Arguments).
```

Следует только добавить, что планировщик процессов в Erlang самостоятельно раскидывает их по разным ядрам процессора, избавляя от необходимости запускать на каждое ядро по отдельной виртуальной машине и организовывать связь между ними (что, правда, тоже не составило бы большого труда, учитывая простоту интерфейсов IPC и RPC в Erlang).

4.7. Тесты производительности

Не следует считать нижеприведенные сравнения объективными, так как производительность системы зависит от множества параметров, которые трудно рассмотреть в рамках одной статьи: используемых алгоритмов, настроек системы и платформы, железа, на котором проводились тесты, и самой задачи.

Однако, некоторое представление о производительности платформ отсюда можно получить.

Сравнивать быстродействие арифметических операций на исследуемых платформах — достаточно трудоемкое занятие: различные операции занимают различное время, и трудно поставить Erlang и Node.js в равные условия. Кроме того, в случае с Erlang, возможен ряд оптимизаций, ускоряющих арифметику. В любом случае, одна и та же задача на Erlang и JavaScript, как правило, будет иметь разные оптимальные реализации.

Однако, некоторое представление об общей производительности можно получить, посмотрев на результаты сравнения Erlang и V8 в рамках тестов

shootout. В некоторых тестах видно явное преимущество той или иной платформы, но в целом результаты недалеки друг от друга.

Более того, в общем зачете разница, по сути, совсем исчезает. По графику видно, что обе платформы идут фактически вровень друг с другом, и недалеко от них находится и еще один популярный JavaScript-движок — TraceMonkey.

Также, для сравнения в тот же чарт добавлены некоторые популярные платформы: PHP, Perl, Ruby, Python, C++ и Java. Проигрывая Java и C++, Erlang и Node, тем не менее, идут впереди «традиционных» скриптовых платформ, со значительным перевесом.

Так как для Erlang веб становится самой распространенной областью применения, а для Node — фактически единственной, то интересно сравнить обе технологии в этом ключе.

В качестве объектов исследования были выбраны нативный http-сервер Node и самописный веб-сервер на Erlang.[5]

Чтобы поставить платформы в более-менее равные условия, Erlang запускался без поддержки SMP. Таким образом, веб-сервера работали на одном ядре. В качестве сервера для тестов был использован X-Large High-CPU инстанс облачного сервиса Amazon EC2. Запросы отсылались с такого же инстанса при помощи утилиты httperf.

Также, из настроек виртуальной машины, предоставляемых Erlang, были использованы следующие: `”+K true”` (kernel polling) и `”-smp disable”` (выполнение Erlang на одном процессоре). Исходный код веб-сервера компилировался с флагом `”+native”`.

Node специальных настроек платформы не предоставляет, поэтому запускался стандартным образом.

У httperf регулировались две величины: число запросов в секунду (`--rate`) и общее число запросов за весь тест (`--num-conns`).

При небольшом, до 10000, общем числе запросов, оба веб-сервера показывают хорошую производительность, держа до 6000 запросов в секунду, выдавая до 6000 ответов в секунду. Однако у Node есть незначительный процент ошибок подключений, в то время как Erlang стабильно обрабатывает все запросы без исключения.

Однако после 7000 и более запросов в секунду картина несколько меняется. Скорость ответов от веб-сервера Node резко падает до 1000 и ниже, а

[5] http://fprog.ru/2010/issue6/dmitry-demeshchuk-node.js-vs-erlang/erweb.erl

число ошибок соединения возрастает. После 9000 запросов в секунду то же самое происходит и с Erlang.

При дальнейшем увеличении общего числа запросов за весь тест, до 30000 или 50000, Node начинает резко сдавать свои позиции, и даже при 1000 запросов в секунду в итоге перестает принимать новые подключения.

Главная причина такого явления заключается в особенностях сборщика мусора V8.

Во-первых, собирая мусор по принципу «stop the world», V8 останавливает выполнение JS-кода до окончания освобождения памяти. А, так как весь код в V8 выполняется однопоточно, то каждый раз во время сборки мусора сервер становится недоступным и перестает совершать какие-либо действия.

Во-вторых, сборка мусора в V8 инициируется в следующих случаях: если невозможно выделение памяти, если превышен установленный порог используемой памяти и, наконец, если движку послано сообщение ожидания. Это сообщение ожидания и является на данный момент единственным способом вызова сборщика мусора вручную. Node использует его при нахождении в режиме ожидания, но внешнего интерфейса на уровне JS для сборки мусора не предоставляет.

Ввиду этого неиспользуемые объекты в Node постоянно накапливаются, если сервер постоянно загружен, и освобождение памяти, выделенной под эти объекты, длится довольно продолжительное время, в течение которого сервер становится недоступен. Кроме того, когда объектов скапливается большое количество, открытые сетевые сокеты закрываются все медленнее и медленнее, и число ответов в секунду падает.

В свою очередь, Erlang при тех же 30–50 тысячах запросов продолжает успешно обрабатывать подключения, почти не теряя производительности по сравнению с 10 тысячами запросов. Проблем с освобождением памяти, как у Node, у него нет. Во-первых, код выполняется многопоточно, и сборка мусора не блокирует все потоки разом.[6] А во-вторых, каждый процесс обладает своим сборщиком мусора, работающим независимо от других процессов. В результате по завершении каждого клиентского процесса память, занимаемая им, оперативно подчищается.

Само собой, такой механизм требует больше памяти, чем один глобаль-

[6]Впрочем, в приведенном примере это преимущество не используется, так как Erlang принудительно работает на одном ядре процессора.

ный сборщик мусора, поэтому веб-сервер на Erlang потребовал при прочих равных условиях несколько больше памяти, чем веб-сервер на Node. Тысяча одновременных соединений (без учета HTTP-пакетов) стоила Erlang примерно 2.5 мегабайта памяти, тогда как Node понадобилось всего 2 мегабайта. Конечно же, эта разница в полкилобайта на соединение нивелируется в случае работы с большими объемами пересылаемых данных — пять сотен байт в случае мегабайтного куска данных для отдачи клиенту погоды не делают.

Такие результаты позволяют предположить, что Node больше подходит для comet-серверов, когда открытое HTTP-соединение не закрывается, пока у сервера не появились какие-то данные для отправки клиенту (классический пример — чат). Собственно, очень многие сервисы используют его именно по такому назначению.

Erlang же здесь выглядит более универсальным, но чуть более требовательным к ресурсам.

Наконец, следует привести кое-какие практические результаты. Вышеупомянутый азиатский сервис микроблоггинга Plurk при помощи Node держит более 100 тысяч одновременных подключений. А создатель Last.fm Ричард Джонс (Richard Jones) умудрился создать на Erlang приложение, выдерживающее до миллиона одновременных подключений к одному физическому серверу.

4.8. Аудитория

На сегодняшний день Erlang представляет собой уже вполне стабилизировавшийся продукт. Новые версии выходят по 3–5 раз в год, и в последнее время носят более косметический характер: оптимизировали какой-нибудь механизм, сделали рефакторинг кода в модулях, добавили пару полезных функций, и т. п. Некоторые системы так и продолжают работать на старых версиях, не испытывая больших затруднений.

На Erlang разрабатывается несколько баз данных с открытыми исходниками, несколько веб-серверов, есть крупные фреймворки, предназначенные, в основном, для веба. Реже можно встретить новые поведенческие модели (чаще всего — надстройки над моделями из OTP), средства для сборки, отладки и тестирования производительности.

В этом плане Node.js гораздо живее. Технология молодая, и все еще ак-

тивно развивается, патчи выходят почти каждый месяц, не только исправляя баги, но и добавляя новые возможности.

Сообщество буквально ломится от множества фреймворков и библиотек. Улучшенные IPC и RPC, шаблонизаторы, надстройки над событийной моделью (например, очень удобная библиотека для управления асинхронными вычислениями, напоминающая монаду Cont из языка Haskell), интерфейсы для всех популярных баз данных.

Наконец, можно просто посмотреть на активность в mailing-листах и IRC-каналах обеих платформ: сообщество Node сейчас гораздо активнее и, похоже, уже многочисленнее.

4.9. Итоги

Node, несмотря на свою молодость, несомненно, является очень многообещающей и мощной технологией. Обеспечивая хорошую производительность, платформа подойдет даже для суровых веб-сервисов с миллионной аудиторией, особенно для comet-серверов.

Но идеален Node все-таки для небольших проектов — меньше борьбы с ошибками и меньше головной боли от сборки разрастающейся системы по кусочкам. То же самое относится и к размеру команды — чем меньше людей, тем проще держать продукт в порядке.

И, несомненно, это более простое решение для тех, кто только что пришел с разработки фронт-энда: тут все будет понятно, привычно и удобно.

Erlang же позволит безболезненно строить как небольшие приложения, так и огромные многокомпонентные системы с привлечением большого числа программистов. Иммутабельность и линейность выполнения облегчают чтение и расширение кода, отказоустойчивость помогает новичкам не порушить уже написанную систему. Кроме того, медленные операции вроде обработки больших текстов или крупных вычислений во мноих случаях можно вынести в отдельный «порт» (терминология Erlang — один из способов связи с программами на других языках), написанный на C, OCaml или любом другом подходящем к задаче языке, что скомпенсирует низкую производительность Erlang в этих задачах.

Ввиду своего мощного инструментария для построения распределенных систем Erlang, скорее всего, будет также более удачным решением везде, где требуется RPC.

В общем, вопрос «что учить: Erlang или JavaScript?» имеет однозначный ответ: учите оба. За ними будущее.

Литература

[1] *Трескин М.* Инструменты интроспекции в Erlang/OTP // *Журнал «Практика функционального программирования»*. — 2010. — № 5. http://fprog.ru/2010/issue05/.

Быстрое инкрементальное сопоставление с регулярными выражениями при помощи моноидов

(Fast incremental regular expression matching with monoids)

Dan Piponi

(Перевод Евгения Кирпичёва)

Аннотация

В статье рассматривается применение моноидов к задаче сопоставления строк с регулярными выражениями, а затем строится инкрементальный алгоритм сопоставления, позволяющий при изменениях строки обновлять результат за $O(log\ N)$. Алгоритм основан на структуре данных «подвешенное дерево», позволяющей сделать инкрементальным любой такой алгоритм на основе моноидов.

The article discusses the application of monoids to regular expression matching and then suggests an incremental matching algorithm, which allows to update the result in $O(log\ N)$ after changes of the input string. The algorithm is based on the "finger tree" datastructure, which can be used to incrementalize any such monoid-based algorithm.[1]

[1] Аннотация составлена переводчиком. Оригинал статьи доступен по адресу [5].

5.1. Задача

Пусть заданы регулярное выражение R и строка длины N, и мы хотим выяснить, подходит ли строка под выражение R. Что ж — скорее всего, придётся просмотреть всю строку посимвольно. Как быстро можно выполнить повторную операцию, если строка слегка изменилась? Кажется, в общем случае придётся еще раз просмотреть строку с начала, или, по крайней мере, начиная с изменённого места. Но оказывается, сопоставление с регулярным выражением можно делать инкрементально, и тогда для большинства изменений, производимых над строкой, пересчёт соответствия строки выражению потребует лишь $O(log\ N)$ времени. Это верно даже в случаях, когда для успешного сопоставления требуется учитывать символы на противоположных концах строки. Более того, применение подвешенных деревьев и моноидов делает реализацию этого алгоритма удивительно простой.

Дальнейший материал предполагает некоторый набор знаний, которые можно почерпнуть из ресурсов, доступных в вебе: знакомство с моноидами [1], понимание материалов, представленных Heinrich Apfelmus в статье «Введение в подвешенные деревья» [4] или оригинальной статье о них [3], и уверенное понимание идеи компиляции регулярных выражений в конечные автоматы.

Этот пост, как обычно, написан на Literate Haskell. Нам понадобится пакет fingertree и несколько импортов.

```
{-# LANGUAGE TypeSynonymInstances,FlexibleInstances,
    MultiParamTypeClasses #-}
import qualified Data.Array as B
import Data.Array.Unboxed as U
import Data.Foldable
import Data.Monoid
import Data.FingerTree hiding (fromList)
import qualified Data.List as L
```

5.2. Конечные автоматы

Рассмотрим простое регулярное выражение: `.*(.*007.*).*`. Мы ищем строку `007`, заключенную в скобки, которые могут находиться на расстоянии миллионов символов друг от друга.

Стандартный способ поиска регулярных выражений — компиляция их в конечный автомат. Кода для этого нужно довольно много, но он совершенно

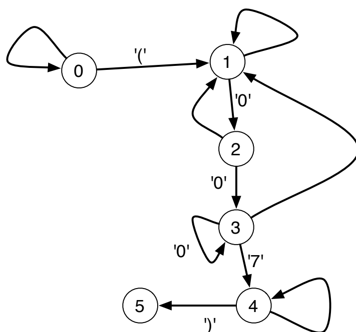


Рис. 5.1. Автомат для выражения `.*(.*007.*).*`

обыкновенный. Поэтому я не буду его приводить, а построю автомат для этого выражения вручную.

На рисунке 5.1 непомеченные ребра означают «если никакое помеченное ребро не подошло». Правила перехода этого автомата можно описать функцией `fsm`.

```
fsm 0 '(' = 1
fsm 0 _ = 0

fsm 1 '0' = 2
fsm 1 _ = 1

fsm 2 '0' = 3
fsm 2 _ = 1

fsm 3 '7' = 4
fsm 3 '0' = 3
fsm 3 _ = 1

fsm 4 ')' = 5
fsm 4 _ = 4

fsm 5 _ = 5
```

Начальное состояние автомата — `0`, а успешное сопоставление соответствует состоянию 5.

Строки можно сопоставлять с помощью следующего кода:

```
matches s = Prelude.foldl fsm 0 s==5
```

Попробуйте вычислить `matches ”(00 7)”` и `matches ”He(007xxxxxxxxxxx)llo”`.

Входные символы можно рассматривать как действия над автоматом. Каждый очередной символ — это функция, переводящая автомат из одного состояния в другое. Мы могли бы воспользоваться `(.)` — функцией композиции языка Haskell — чтобы скомбинировать эти функции. Однако `(.)` ничего особенного не делает, `f . g` это лишь замыкание, означающее «когда придёт время, примени g, а затем `f`».

Впрочем, такие функции переходов можно табулировать следующим образом:

```
tabulate f = array (0,5) [(x,f x) | x ← range (0,5)]
```

Для каждой буквы алфавита у нас есть одна табулированная функция:

```
letters = array (’ ’,’z’) [(i,tabulate (flip fsm i))
                          | i ← range (’ ’,’z’)]
```

Имея две табулированные функции, легко составить таблицу для их композиции. Табулированные функции образуют моноид с композицией mappend. Для большей эффективности воспользуемся распакованными массивами:

```
type Table = UArray Int Int

instance Monoid Table where
   mempty = tabulate id
   f ‘mappend‘ g = tabulate (λstate →
(U.!) g ((U.!) f state))
```

Тут есть одна тонкость: объект типа `Table` в принципе может бы быть любым массивом целочисленных значений с индексами типа `Int`. Но если мы ограничим индексы и значения массива числами от `0` до 5, то действительно получится моноид.

Мы можем проверить соответствие любой строки нашему регулярному выражению: выберем из массива `letters` значения типа `Table`, соответствующие каждому символу строки, объединим таблицы и проверим, отображает ли полученная табулированная функция начальное состояние `0` в конечное состояние 5:

```
matches’ s = table!0==5 where
  table = mconcat (map ((B.!) letters) s)
```

Этот способ медленнее и сложнее, чем наша первоначальная реализация matches, но зато мы выделили в задаче моноидальную структуру. Если хранить строку как последовательность символов, представленную подвешенным деревом, то можно в каждом узле дерева хранить массив Table для подстроки, представляемой этим узлом. Каждый раз при перебалансировке узлов дерева необходимо пересчитывать соответствующие массивы Table. Но это нам подходит, обычно перебалансировка требует лишь $O(log\ N)$ операций, а писать код для этого вовсе не надо — подвешенные деревья сделают всё за нас. В результате представление каждой подстроки обладает тем свойством, что для неё всегда известен соответствующий ей массив Table. Такие деревья можно свободно резать и склеивать, зная что Table всегда будет содержать свежие данные.

Осталось одна небольшая загвоздка: мне хотелось бы иметь возможность произвольного доступа к n-му символу дерева. Как это сделать, разъяснено у apfelmus [4]. Понадобятся одновременно моноиды Size и Table, поэтому я воспользуюсь произведением моноидов [1].

```
data Elem a = Elem { getElem :: a } deriving Show
data Size = Size { getSize :: Int } deriving (Eq,Ord,Show)

instance Monoid Size where
   mempty = Size 0
   Size m ‘mappend‘ Size n = Size (m+n)
```

Остается реализовать measure из статьи о подвешенных деревьях [3]:

```
instance Measured (Size,Table) (Elem Char) where
   measure (Elem a) = (Size 1,(B.!) letters a)
```

5.3. Подвешенные деревья

Теперь мы можем определить строки следующим образом:

```
type FingerString = FingerTree (Size,Table) (Elem Char)
```

Процедура вставки примерно такая же, как в статье:

```
insert :: Int → Char → FingerString → FingerString
insert i c z = l >< (Elem c <| r)
  where (l,r) = split (λ(Size n,_) → n>i) z
```

Обратите внимание, как я достаю размер из моноида-произведения, чтобы выполнить вставку в нужном месте.

Вот пример строки. Подберите её длину в соответствии с имеющейся памятью и мощностью процессора:

```
fromList = L.foldl' (|>) empty
string = fromList (map Elem $ take 1000000 $
    cycle "the quick brown fox jumped over the lazy dog")
```

(Я использую энергичную версию `fromList`, чтобы обеспечить гарантированное построение дерева.)

Функция сопоставления просто вычленяет вторую компоненту моноида и проверяет, отображает ли она начальное состояние в конечное:

```
matches007 string = ((U.!) (snd (measure string)) 0)==5
```

5.4. Цикл взаимодействия

Советую компилировать этот код с оптимизацией, например ghc –make `-O5 -o regexp regexp.lhs`:

```
loop string = do
  print $ "matches? " ++ show (matches007 string)
  print "(Position,Character)"
  r ← getLine
  let (i,c) = read r
  loop $ insert i c string

main = do
  loop string
```

Теперь можно работать с программой интерактивно. На вход принимаются значения вроде `(100, 'f')` (вставить `'f'` в позиции `100`). Исходное дерево может вычисляться несколько секунд, после чего каждое сопоставление будет мгновенным. (Впрочем, вторая проверка тоже может занять несколько секунд, поскольку, несмотря на `foldl'`, дерево ещё не построилось полностью.)

Вот пример входных данных:

```
(100,'(')
(900000,')')
(20105,'0')
(20106,'0')
(20107,'7')
```

5.5. Обсуждение

Обратите внимание, что этот вариант решения содержит значительные издержки. Для каждого символа я храню массив `Table` целиком. Вы вполне можете хранить строку поблочно (как в структуре данных «верёвка» [6]). Тогда при редактировании строки придётся просматривать некоторые блоки повторно — но просмотр, к примеру, блока в 1 килобайт куда дешевле, чем просмотр гигабайтного файла целиком. Кроме того, поблочная обработка, вероятно, ускорит и первый проход, поскольку требует построения дерева значительно меньшего размера.

Когда Хинце и Патерсон писали первую статью про подвешенные деревья, они черпали вдохновение из алгоритмов на основе префиксных сумм. Почти любой алгоритм на их основе можно сделать инкрементальным с помощью подвешенных деревьев. Настоящая статья основана на идее применения подвешенных деревьев к схеме параллельного синтаксического разбора, описанной Хиллисом и Стилом в их классической статье о Connection Machine [2].

Зачем же может понадобиться сопоставлять строки с подобным фиксированным регулярным выражением? Ну, например, на основе этого метода можно построить полноценный инкрементальный лексер. Он будет обновлять результат мгновенно, даже если вставка очередного символа в строку меняет тип лексемы в миллиарде символов от него. С деталями можно ознакомиться в статье Хиллиса и Стила.

Примечательно, что в этом коде нет ничего особенно «Хаскельного» — разве что, на Хаскеле было особенно легко написать прототип. Вы можете сделать то же самое и на C++, скажем, с помощью красно-чёрных деревьев.

Литература

[1] Haskell monoids and their uses. — URL: http://sigfpe.blogspot.com/2009/01/haskell-monoids-and-their-uses.html (дата обращения: 29 ноября 2010 г.), также переведено в первом выпуске журнала «Практика функционального программирования».

[2] *Hillis W. D., Steele Jr. G. L.* Data parallel algorithms // *Commun. ACM.* — 1986. — Vol. 29, no. 12. — Pp. 1170–1183.

[3] *Hinze R.*, *Paterson R.* Finger trees: a simple general-purpose data structure // *J. Funct. Program.* — 2006. — Vol. 16, no. 2. — Pp. 197–217.

[4] Monoids and finger trees. — URL: http://apfelmus.nfshost.com/monoid-fingertree.html (дата обращения: 29 ноября 2010 г.).

[5] *Piponi D.* Fast incremental regular expression matching with monoids. — URL: http://blog.sigfpe.com/2009/01/fast-incremental-regular-expression.html (дата обращения: 29 ноября 2010 г.). — 2009.

[6] Rope implementation overview. — URL: http://www.sgi.com/tech/stl/ropeimpl.html (дата обращения: 29 ноября 2010 г.).

© 2010 «Практика функционального программирования»

Инкрементальные регулярные выражения

Евгений Кирпичёв
`jkff@fprog.ru`

Аннотация

В этой статье при помощи ряда красивых алгоритмических приемов из мира функционального программирования строится Java-библиотека для инкрементального сопоставления строк с регулярными выражениями. Целевая аудитория статьи: 1) алгоритмисты, любящие пополнить свой арсенал новыми инструментами и 2) люди, интересующиеся тем, как идеи из ФП ложатся на императивные языки.

This article uses a number of beautiful algorithmic techniques from the world of functional programming to build a Java library for incremental regular expression matching. The target audience is: 1) algorithmists who like to extend their toolbox with new gadgets and 2) people interested in the applications of functional ideas to programming in imperative languages.

6.1. Введение

Задача сопоставления строк с регулярными выражениями известна давно (с 1960-х гг.) и имеет множество решений. Сопутствующие алгоритмы очень красивы, и редок тот программист, кто не попытался хотя бы раз в жизни написать для развлечения движок регулярных выражений.

Для понимания дальнейшего материала будет полезно освежить в памяти основные концепции регулярных выражений в одном из общедоступных источников, например [10].

Подходы к решению этой задачи довольно сильно отличаются по области применимости.

Вот некоторые из вопросов, встающих перед разработчиком очередного движка:

- Какие операторы поддерживать? Поддерживать ли «захватывающие группы (capturing groups)», «обратные ссылки (backreferences)», исполнение произвольного кода, «жадное» сопоставление и т. п.? Чем больше возможностей поддерживается, тем труднее реализовать движок эффективно — большинство из них исключают сразу целые классы алгоритмов сопоставления.
- Каков будет размер и количество искомых регулярных выражений? Будут ли это небольшие выражения для валидации данных, или допускается возможность создания полноценного лексера с десятками или сотнями токенов, заданных своими выражениями?
- Как много раз происходит сопоставление с каждым выражением? Допустимо ли очень долго «компилировать» его, но зато очень быстро выполнять сопоставление?
- Насколько велик объем сопоставляемого текста, как часто он меняется и каков характер изменений?
- И т. п.

Перечислим некоторые из существующих подходов и движков.

- Автоматный подход: `awk`, `grep`, `re2` [6] (благодаря автоматному подходу он обладает рядом особенностей, делающих возможным существование Google Code Search) — гарантируется линейное время сопоставления, однако некоторые возможности (например, обратные ссылки) принципиально не реализуемы; затруднено (но возможно) извлечение «захватывающих групп»; иногда возможен экспоненциально большой объем потребляемой памяти (впрочем, `re2` избегает этой проблемы за

счет перехода к недетерминированным автоматам при нехватке памяти);

- Модифицированный автоматный подход, «помеченные (tagged)» автоматы: `libtre` [18], `regex-tdfa` [17] — также гарантируется линейное время сопоставления; возможен нечеткий поиск;
- Подход на основе полуколец: `weighted-regexp` [24] — также гарантированное линейное время сопоставления и линейный (относительно размера выражения) объем требуемой памяти; очень простая, красивая и эффективная реализация;
- Рекурсивный спуск: большинство движков (Perl и PCRE-движки, Java, irregexp и т. п.) — полный спектр возможностей, однако возможны «патологические» случаи, когда время сопоставления резко возрастает;

В блог-посте [22] Дэн Пипони[1] наметил еще один подход, связанный с использованием моноидов и т. н. «подвешенных деревьев (finger trees)[2]». Этот подход работает только для «истинных» регулярных выражений (фактически, в нашем распоряжении лишь символы, скобки и операторы `+`, `*`, `?`, `|`), однако позволяет выполнять сопоставление *инкрементально*, то есть, при изменениях входной строки результат сопоставления пересчитывается очень быстро, без повторного сканирования. К изменениям относятся склейка двух строк и разрезание строки в произвольном месте. Ясно, что через эти операции легко выражаются и все остальные (вставка в середину, дописывание в конец и т. п.).

По мнению автора настоящей статьи, упомянутый блог-пост представляет собой жемчужину программирования. Долгое время автор ограничивался пламенным пересказом его содержания знакомым, коллегам и студентам, однако в конечном итоге желание «копнуть поглубже» возобладало. Данная статья посвящена доработке подхода Дэна Пипони и реализации его в виде библиотеки на языке Java.

Язык Java выбран для того, чтобы достигнуть сразу несколько целей:

- Повысить вероятность того, что разработанная библиотека принесет пользу, а не останется «академической игрушкой» из-за трудностей

---

[1] Дэн Пипони (Dan Piponi) — специалист по графическим спецэффектам, участвовал в создании всех трех «Матриц», «Star Trek» и многих других фильмов (http://www.imdb.com/name/nm0685004/, http://homepage.mac.com/sigfpe/).

[2] Насколько известно автору, общепринятого перевода этого термина на русский язык не существует. Данный перевод выбран исходя из особенностей устройства этой структуры данных.

включения ее в существующие проекты;
- Облегчить понимание кода для весьма широкого на настоящий момент сообщества программистов на Си-подобных языках;
- Показать, что изучение идей из мира функционального программирования приносит плоды и при программировании на нефункциональных языках.

В каких практических задачах может пригодиться подобный подход к сопоставлению с регулярными выражениями?
- Возможность быстро перестраивать результат при изменениях строки была бы полезна в текстовых редакторах для инкрементальной подсветки произвольного синтаксиса, где токены заданы регулярными выражениями (например, vim позволяет достаточно гибко задавать правила синтаксиса, однако не всегда правильно выполняет подсветку и переподсветку при редактировании, поскольку использует «наивный» подход к подсветке).
- Можно представить себе задачу из области биоинформатики, где при помощи, скажем, генетического алгоритма ([13]) из кусков ДНК-последовательностей при помощи склеиваний и разрезов собирается новая последовательность, и оптимизируется какая-то функция, зависящая от наличия и положения в ней определенных паттернов.

Дальнейший текст будет структурирован следующим образом:
- Краткое рассмотрение автоматного подхода к регулярным выражениям и главная идея инкрементального сопоставления — манипулирование переходными функциями автоматов;
- Идея структуры данных «верёвка», предназначенной для представления последовательностей с эффективными операциями разрезания и склейки;
- Связь переходных функций автоматов с понятием моноида; краткая иллюстрация этого понятия на других примерах в программировании;
- Идея комбинирования верёвок и моноидов;
- Общая схема программы;
- Выделение основных алгоритмических и технических трудностей;
- Описание конкретной реализации верёвок и идея чисто функциональной структуры данных;
- Демонстрация программы: примеры и тесты;
- Перечисление нерешенных вопросов и дальнейших направлений развития.

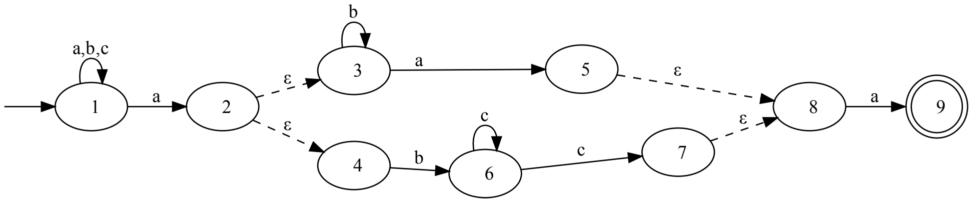


Рис. 6.1. Недетерминированный автомат, соответствующий выражению «`.*a(b*a|bc+)a`»

6.2. Автоматный подход

Наиболее известный подход к реализации сопоставления с регулярными выражениями основан на использовании конечных автоматов и изучается в большинстве университетских курсов по построению компиляторов. Можно ознакомиться с ним подробно, например, в статье Расса Кокса [8]. Мы же лишь напомним его вкратце.

По регулярному выражению при помощи, например, т. н. «конструкции Томсона» [8] строится конечный автомат, в котором одно или несколько состояний объявлены «начальными», одно или несколько состояний — «терминальными», и некоторые состояния соединены дугами. Дуга из состояния A в состояние B, подписанная символом c, означает: «если состояние A активно и на вход подается символ c, то вместо состояния A активируется состояние B». Есть также ε (эпсилон)-дуги: если A соединено ε-дугой с B, то при активации A немедленно активируется и B. Если же из состояния «некуда» выйти по очередному входному символу, то оно просто деактивируется. Автомат называется «недетерминированным» потому, что в каждый момент могут быть активны сразу несколько состояний.

Для выполнения сопоставления активируется начальное состояние и на вход по очереди подается каждый символ последовательности. Если в итоге оказывается активным хотя бы одно «терминальное» состояние — сопоставление объявляется успешным.

Пример. Регулярное выражение «`.*a(b*a|bc+)a`». Автомат для этого выражения приведен на рис. 6.1, а вот его последовательность активных состояний при подаче на вход строки «`aabcca`».

Поскольку по окончании просмотра строки в активном множестве содержится терминальное состояние s_9, сопоставление объявляется успешным.

Для ускорения сопоставления из автомата иногда убирают ε-дуги и выполняют его *детерминизацию* (из каждого состояния разрешается не более

Увиденная порция строки	Активные состояния
	$\{s_1\}$
a	$\{s_1, s_2, s_3, s_4\}$
a**a**	$\{s_1, s_2, s_3, s_4, s_5, s_8\}$
aa**b**	$\{s_1, s_3, s_6\}$
aab**c**	$\{s_1, s_6, s_7, s_8\}$
aabc**c**	$\{s_1, s_6, s_7, s_8\}$
aabcc**a**	$\{s_1, s_2, s_3, s_4, s_9\}$

одной исходящей дуги, подписанной одним и тем же символом). Для детерминизации существует несколько алгоритмов; они описаны, например, в [14] и [29]. Получается совершенно другой автомат, однако соответствующий тому же регулярному выражению . Тогда в каждый момент при сопоставлении оказывается активно лишь одно состояние, что позволяет построить более эффективную реализацию сопоставления. Однако мы не будем использовать детерминизацию, поскольку в результате нее автомат может экспоненциально разрастись: например, любой автомат для выражения вида $(0|(01*)(01*)(01*)\dots0)*$ будет иметь размер порядка $O(2^n)$, где n — количество повторений $(01*)$, что, как мы увидим далее, в нашей задаче совершенно недопустимо. Мы воспользуемся лишь первым ее этапом — удалением ε-дуг; оно может только уменьшить размер автомата (ср. рис. 6.1 и 6.2). Алгоритм удаления ε-дуг очень прост: конец каждой дуги переставляется на каждый из узлов, достижимых по ε-дугам из «бывшего» конца этой дуги, затем удаляются недостижимые узлы.

Обратим внимание, что можно «расслоить» этот автомат посимвольно. Если для каждого символа из алфавита выбирать из автомата только те переходы, которые активируются при подаче этого символа, то можно разделить исходный автомат на несколько подавтоматов (рис. 6.3).

Изобразим эти подавтоматы несколько иначе (рис. 6.4). Становится видно, что каждый из них можно рассмотреть как таблицу: каждому из возможных входных символов соответствует «переходная функция» — «Каким будет следующее состояние автомата после подачи на вход этого символа, если текущее его состояние — S?». Кроме того, становится видно, что понятие такой «переходной функции» имеет смысл не только для отдельных символов, но и для целых строк. Имея переходную функцию для строки, какой бы длинной эта строка ни была, можно сэмулировать подачу ее на вход автомата, не рассматривая ее символы по отдельности. Переходная функция

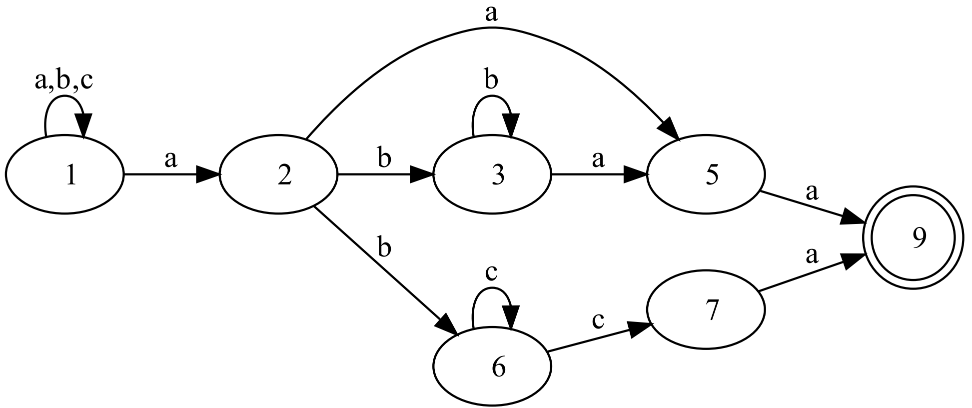


Рис. 6.2. Недетерминированный автомат, соответствующий выражению «`.*a(b*a|bc+)a`», после удаления ε-дуг

Рис. 6.3. Недетерминированный автомат, соответствующий выражению «`.*a(b*|c+)a`», расслоённый посимвольно

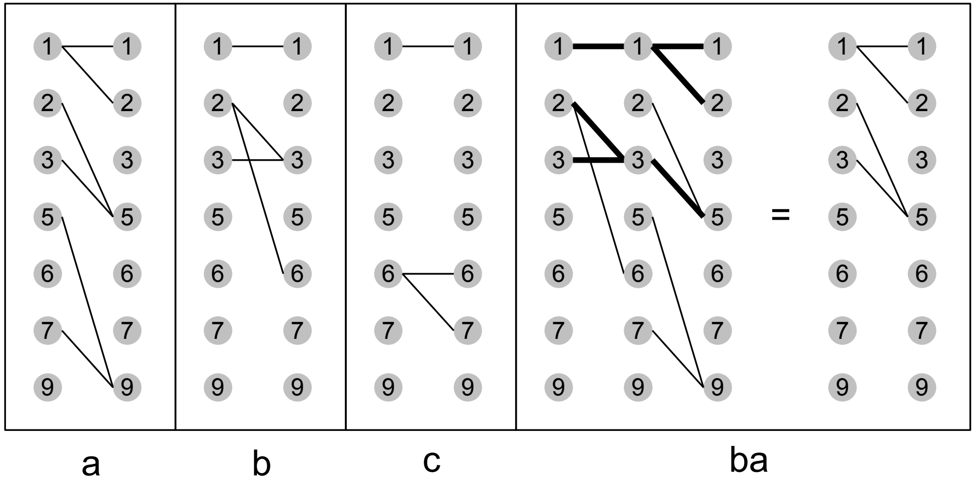


Рис. 6.4. Переходные функции недетерминированного автомата и их композиция

строки легко вычисляется по переходным функциям ее символов; точно так же, имея переходные функции для двух строк, можно вычислить функцию для их конкатенации.[3]

Вот и половина решения нашей задачи инкрементального сопоставления, а именно — быстрый пересчет результата при склейке строк!

Для второй половины (разрезание) ограничимся пока туманным замечанием, что если закэшировать переходные функции для частей строки, то при разрезании можно будет переиспользовать бóльшую часть информации от них, вновь избежав прохода по всей строке.

6.3. Верёвки

Следующая грань задачи — представление строк, допускающее эффективную склейку и разрезание. Такая структура данных давно известна и называется «верёвка (*rope*)». Верёвка представляет собой сбалансированное дерево из «сегментов (*chunks*)» — небольших массивов символов. При

[3] Как отметил один из рецензентов, схожий подход используется в алгоритме разбора контекстно-свободных грамматик Кока-Янгера-Касами [9], который, кроме того, так же как и наш алгоритм, допускает параллельную реализацию — она описана, например, в [11].

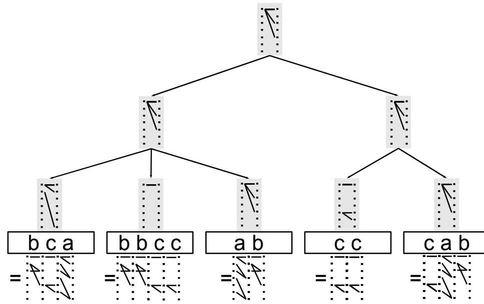


Рис. 6.5. Пример верёвки с закэшированной переходной функцией относительно выражения «`.*a(b*|c+)a`» для строки $bcabbccabcccab$

склейке и разрезании используются операции, аналогичные тем, что используются для сбалансированных деревьев, но с дополнительными действиями на уровне листьев дерева, то есть, сегментов. Разновидности верёвок отличаются способом балансировки. Одна из разновидностей описана в статье [3]. Мы пока не будем останавливаться на этой структуре данных подробно; вполне достаточно уже изложенной основной идеи.

Вспомним, как мы собирались решить задачу инкрементального сопоставления с регулярным выражением:

- При склейке переходные функции перемножаются;
- Переходные функции для подпоследовательностей кэшируются и используются при разрезании.

Если кэшировать переходные функции в узлах этого дерева-верёвки, то получится структура данных «верёвка, всегда знающая свою переходную функцию» (то есть, фактически, свой результат сопоставления с регулярным выражением). Пример такой структуры показан на рис. 6.5.

Более точно, во время операций перебалансировки верёвок, происходящих при склейке и разрезании, просто будем собирать переходные функции новых узлов из произведений переходных функций узлов, оставшихся нетронутыми (к сожалению, при склейке и разрезании листовых сегментов переходную функцию для получающихся новых сегментов придется всё же считать заново).

6.3.1. Разрезание по монотонному условию

В дополнение к операции разрезания «по индексу» можно реализовать для верёвок еще одну крайне важную операцию: разрезание «по монотон-

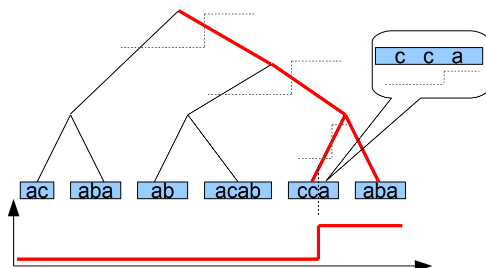


Рис. 6.6. Разрезание верёвки для строки $acabaabacabccaaba$ по монотонному условию «содержит bcc»

ному условию».

Пусть есть некоторый предикат (свойство) f над строками. Пусть f такова, что всякая строка может лишь *приобрести* (но не потерять) свойство f при дописывании символов справа, то есть, $\forall s_1,\ f(s_1) \Rightarrow \forall s_2,\ f(s_1 + s_2)$. В этом случае будем называть свойство f *монотонным*.

Тогда у каждой строки, удовлетворяющей свойству f, существует *минимальный префикс*, удовлетворяющий f. Проиллюстрируем это понятие и алгоритм его быстрого вычисления на верёвке с помощью рис. 6.6.

На рисунке красным отмечены рёбра, удовлетворяющие условию «там, где ребро кончается, условие уже верно». Очевидно, что чтобы найти точку разреза, надо спуститься от корня верёвки до листа, двигаясь вниз и вправо, и на каждом уровне просматривая ребра слева направо, пока не встретится красное ребро, а затем разрезать лист — уже с помощью обычного линейного поиска. Цифры «0» и «1» написаны вокруг просматриваемых ребер и означают, верно ли условие для соответствующего префикса строки (до ребра, между ребрами и т. п.)

Поскольку движение осуществляется вниз и вправо, то в каждый момент просмотренные ребра суммарно покрывают целиком все возрастающий префикс исходной строки, и с каждым просмотренным ребром к этому префиксу прибавляется покрываемая данным ребром подстрока. Если при этом оказывается, что до просмотра ребра условие еще не было верно, а после просмотра — стало, то надо спуститься внутрь узла, в который ведет это ребро.

Чтобы постоянно быть в курсе, верно ли уже условие, надо, чтобы можно было быстро вычислять $f(ps)$, зная $f(p)$ и $f(s)$ для любых строк p и s, поскольку при движении вниз и вправо «текущий префикс» (p) при просмотре каждого очередного ребра увеличивается на покрываемую этим ребром

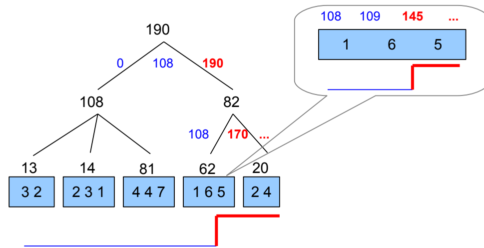


Рис. 6.7. Разрезание верёвки из чисел по монотонному условию «сумма квадратов больше 140»

подстроку (s), а при просмотре каждого очередного символа при линейном поиске внутри листового блока — на строку из одного этого символа.

Проиллюстрируем алгоритм разрезания на еще одном примере — рис. 6.7. На нем изображена верёвка из чисел, знающая сумму их квадратов, и проводится операция разрезания для поиска минимального префикса с суммой квадратов более 140. Алгоритм напоминает игру «горячо —холодно»; на рисунке показаны суммы квадратов префиксов до и после различных ребер и (при просмотре листового блока) символов; те, что еще не удовлетворяют условию, отмечены «холодным» синим цветом, а те, что удовлетворяют — «горячим» красным. На этом рисунке заметно, что при просмотре листовых блоков необходимо перевычислять квадраты некоторых чисел, то есть, информации, «закэшированной» в узлах дерева, уже недостаточно.

Операция разрезания пригодится нам позднее для поиска *позиций* вхождений регулярного выражения в строке.

6.4. Моноиды

Как было сказано выше, имея конечный автомат, можно сопоставить каждой строке «переходную функцию» относительно этого автомата, и при склейке строк их переходные функции комбинируются (обозначим композицию функций f_1 и f_2 как $f_1 \circ f_2$).

Умножение переходных функций (как, кстати, и склейка строк) обладает парой простых и полезных свойств, в истинности которых проще всего убедиться визуально, глядя на рис. 6.4:

- Для любых трех переходных функций f, g, h верно $f \circ (g \circ h) = (f \circ g) \circ h$, то есть значение выражения вида $a \circ b \circ c \circ \ldots$ не зависит от расстановки скобок, поэтому скобки можно опускать и говорить

о таком выражении как о «композиции a, b, c и т. д.». Это свойство называется «ассоциативностью» оператора «$\circ$».

- Существует особая переходная функция u, отображающая каждое состояние автомата в это же состояние. Она называется «единицей» оператора «$\circ$», потому что, как для оператора умножения верно $1 \cdot x = x \cdot 1 = x$, так и для «$\circ$» верно $u \circ f = f \circ u = f$. В графической нотации рис. 6.4 такая функция выглядит как «лесенка» из горизонтальных линий.

Благодаря наличию этих двух свойств говорят, что переходные функции конечного автомата образуют *моноид*.

Более точно: говорят, что *множество M, операция $\otimes$ и элемент $u \in M$ (называемый «единицей» этой операции) образуют моноид*, если выполняются вышеописанные два свойства.

Поскольку понятие моноида настолько простое и общее, неудивительно, что приглядевшись к «повседневным» объектам в программировании, можно увидеть десятки моноидов. Некоторые из них перечислены в табл. 6.1. С некоторыми применениями моноидов в программировании можно ознакомиться, например, в статье [23] (перевод [28]).

Заметим, что приведенный в секции о верёвках способ их использования для инкрементального сопоставления с регулярными выражениями может быть без изменений использован для любого другого моноида: надо всего-навсего вместо операции перемножения переходных функций взять операцию умножения в нужном моноиде. Так можно получить структуры данных «верёвка из чисел, всегда знающая свою сумму (или минимум, НОК и т. п.)», «верёвка из символов, всегда знающая свою гистограмму частотности» и многие другие — с помощью этой методики можно реализовать большинство основных структур данных, основанных на деревьях, например, приоритетные очереди. Можно реализовать обобщенную структуру данных «верёвка над произвольным моноидом», в которую пользователь сможет подставить нужный моноид.

Например, на рис. 6.8 приведена верёвка из чисел, «знающая» свой минимум и максимум. Очевидно, такая верёвка может быть использована, например, для быстрого определения минимума и максимума любого подотрезка. В этой верёвке используется моноид с операцией композиции $(m_1, M_1) \oplus (m_2, M_2) = (min(m_1, m_2), max(M_1, M_2))$ и единицей $(\infty, -\infty)$.

Множество M	Операция $\otimes$	Единица u	Комментарий
Числа	$+$	0	Натуральные, целые, вещественные, комплексные, кватернионы...
Числа	$\times$	1	
Целые числа	НОК	1	
Полиномы	НОК	1	
Числа, строки...	MIN, MAX	Максимальный, минимальный элемент	
Булевские значения	AND	TRUE	
Булевские значения	OR	FALSE	
Матрицы	$+$	0	Над числами $(+, \times)$, над числами $(+, \mathrm{MIN})$, над булевскими значениями (OR, AND), ...
Матрицы	$\times$	1	
Множества	Объединение	Пустое множество	
Множества	Пересечение	Полное множество	Если рассматривать только подмножества «полного» множества
Списки, строки...	Конкатенация	Пустая последовательность	
Словари	Объединение	Пустой словарь	«Конфликты» разрешаются в другом моноиде: $(dic_1 \otimes dic_2)[key] = dic_1[key] \oplus dic_2[key]$
Функции типа $A \to B$	$(f \otimes g)(a) = f(a) \oplus g(a)$	$e(a) = e_B$	$(B, \oplus, e_B)$ моноид
Перестановки	Перемножение	Тождественная перестановка	
Функции	Композиция	Тождественная функция	
Пары (x, y) где $x \in X, y \in Y$	$(x_1, y_1) \otimes (x_2, y_2) = (x_1 \oplus_X x_2, y_1 \oplus_Y y_2)$	(u_X, u_Y)	Если $(X, \oplus_X, u_X)$ и $(Y, \oplus_Y, u_Y)$ моноиды
...	...	...	

Таблица 6.1. Некоторые моноиды из мира программирования

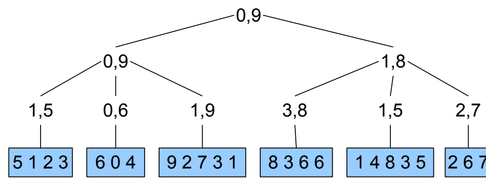


Рис. 6.8. Пример верёвки из чисел с закэшированным минимумом и максимумом

6.5. Схема программы

Теперь сведём представленные алгоритмы воедино и рассмотрим схему устройства всей программы. Графически эта схема представлена на рис. 6.9 и далее. На рисунках использован способ представления знаний «концепт-карта (concept map)»; схемы выполнены с помощью инструмента IHMC CmapTools [15].

Рис. 6.9 «Общая схема программы»: Пользователь задает несколько регулярных выражений в виде строк, в результате компиляции которых (синтаксический разбор, а затем преобразование в автомат) получается объект типа `PatternSet`. Такой объект умеет «индексировать» обычные строки, давая объекты типа `IndexedString`. Они, в свою очередь, позволяют эффективно найти в себе все вхождения шаблонов, а также могут быть склеены или разрезаны (по монотонному предикату).

Рис. 6.10 «Верёвки и моноиды»: «Индексированные строки» реализованы при помощи верёвок (Rope). В программе реализованы только верёвки над символами, поскольку дальнейшее обобщение нецелесообразно в рамках текущей задачи и при реализации в рамках системы типов Java привело бы к большому количеству синтаксического мусора. Верёвка — это строка, знающая свою «меру» некоторого типа `M`. Мера вычисляется как сумма мер отдельных символов (`Function<Character,M>`) в некотором моноиде (`Reducer`). Верёвки реализованы с помощью особого вида сбалансированных деревьев, который будет описан далее в статье. При перебалансировках меры новых узлов складываются из мер старых при помощи заданного моноида. Для сопоставления с регулярными выражениями используется моноид комбинирования переходных функций для автомата, соответствующего заданной системе выражений (точнее, для двух автоматов — прямого и обратного, подробнее об этом будет написано далее в разделе «Поиск позиций вхождений»).

Рис. 6.11 «Конечные автоматы»: Автоматы реализованы таким образом, чтобы было удобно представлять и вычислять их «переходные функции». Автомат позволяет получить свою переходную функцию для любого заданного символа (операция расслоения), а переходная функция — это функция из типа состояний автомата в этот же тип. Состояние — это «черный ящик», про который известен лишь список «завершаемых» им шаблонов: если состояние автомата после просмотра некоторой строки завершает некоторые шаблоны, значит, в конце этой строки имеют место их вхождения. Переходные функции представлены не в виде произвольных функций, а в виде специфических объектов, допускающих эффективное вычисление композиции (уместна аналогия с тем, как в графических приложениях преобразования координат задаются не произвольными функциями, а числовыми матрицами, допускающими эффективное перемножение). Существование такого оператора композиции означает существование «моноида переходных функций» для каждого автомата. Этот моноид подставляется в верёвки, используемые в качестве «индексированных строк», как сказано ранее.

Рис. 6.12 «Связь детерминированных и недетерминированных автоматов»: Абстрактное понятие автомата используется в программе двумя способами: для детерминированных и для недетерминированных автоматов. Строго говоря, детерминированные автоматы в программе не нужны, они использовались лишь при тестировании, и стимулировали создание абстракции автомата, частным случаем которой являются оба перечисленных вида. У детерминированного автомата множество состояний — это множество чисел от 0 до некоторого N, соответственно переходные функции — это отображения из $0 \ldots N$ в $0 \ldots N$. Они реализованы в виде одномерных `int[]`-массивов, и их композиция вычисляется очевидным образом с помощью перемножения перестановок. У недетерминированных же автоматов множество состояний состоит из подмножеств некоторого «базисного» набора состояний, а переходная функция недетерминированного автомата определяется тем, в какие базисные состояния она отображает каждое базисное состояние по отдельности, т. е. она задается отображением вида `int` → `int[]` (см., например, рис. 6.4). Это отображение для эффективности реализовано в виде булевой матрицы, у которой каждый элемент соответствует одному биту в общем массиве.

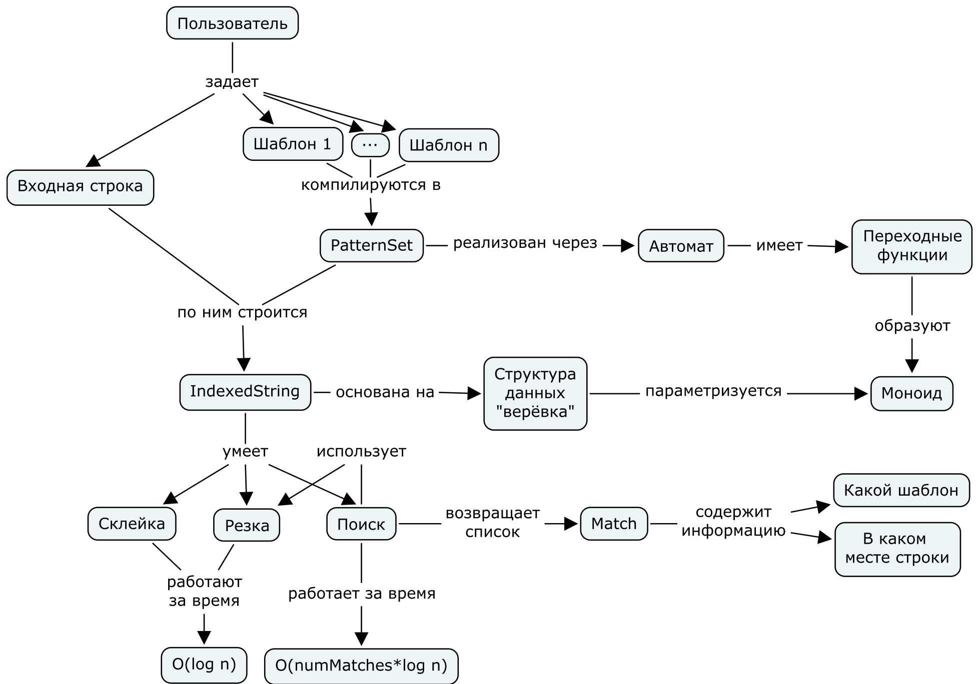


Рис. 6.9. Общая схема программы

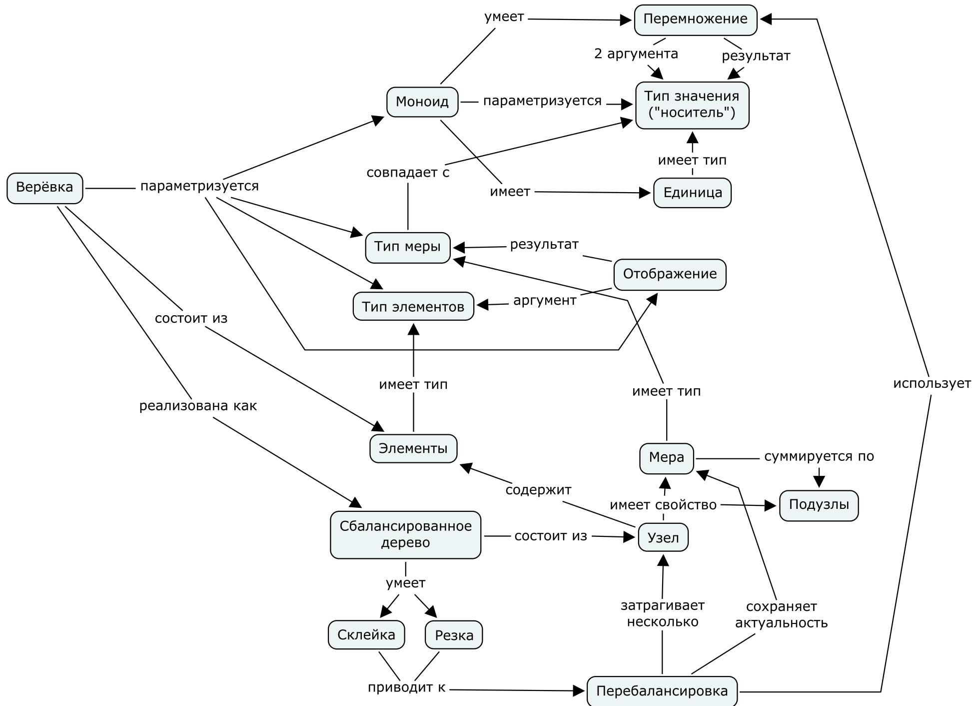


Рис. 6.10. Схема программы: Верёвки и моноиды

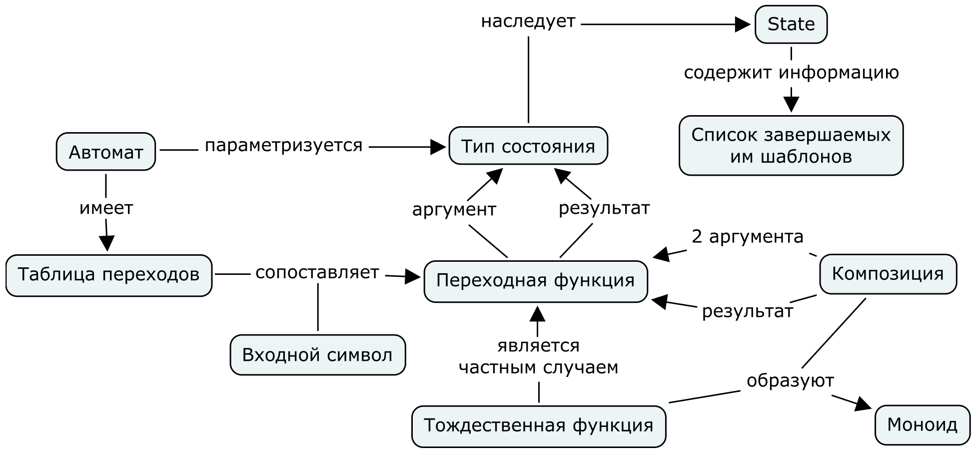


Рис. 6.11. Схема программы: Конечные автоматы

Тип состояния = int
Детерминированный автомат
специализирован
Переходная функция :: int -> int
реализована как
Массив
2 аргумента
результат
2 аргумента
результат
Композиция: (f * g)(s) = f(g(s))
реализована как
Умножение перестановок

2 аргумента
Булева матрица BitSet<n>[n]
Умножение матриц
результат
реализована как
Переходная функция :: Set<int> -> Set<int>
задается через
Переходная функция :: int -> Set<int>
Недетерминированный автомат
специализирован
2 аргумента
результат
2 аргумента
результат
реализована как
Композиция
задается через
Композиция: (f * g)(s) = объединение g(q) по q из f(s)
Тип состояния = Set<int>
соответствует множеству
Базисное состояние
содержит информацию
Завершаемые шаблоны
объединяется из
Список завершаемых им шаблонов

Рис. 6.12. Схема программы: Связь детерминированных и недетерминированных автоматов

6.6. Design challenges

У читателя уже должно было сложиться представление о механизме работы программы, однако остается еще несколько «тёмных мест».

- **Поиск позиций вхождений**. Мы рассмотрели структуру данных «верёвка, кэширующая свою переходную функцию относительно автомата некоторого регулярного выражения». Однако имея строку и ее переходную функцию, можно лишь узнать, *подходит ли* она (или хоть какая-нибудь из ее подстрок) под регулярное выражение, но нельзя узнать, *где именно* расположено вхождение.
- **Сразу несколько регулярных выражений**. Рассмотренная структура данных позволяет сравнить строку лишь с одним регулярным выражением. Напрашивается значительно более практически полезное обобщение: сравнение сразу с большим количеством выражений (к этой задаче известны и другие подходы, например, [4]). Результатом такого сравнения является набор фактов вида «под k-е выражение подошел отрезок строки $i..j$».
- **Реализация верёвок**. Выбор разновидности верёвок, наиболее подходящей для реализации наших алгоритмов — также задача, достойная обсуждения. Встают вопросы о размере листовых «блоков», о способе балансировки и т. п.

6.6.1. Позиции вхождений

Во-первых, напомним, что речь идет не просто о сравнении строки с регулярным выражением R, а о *поиске* вхождения R в строке. Эту задачу можно отчасти переформулировать как сравнение строки с выражением «`.*R.*`», однако при этом получается лишь ответ на вопрос «есть ли в строке вхождение R?», но не на вопрос «где именно расположено вхождение».

С ответом на второй вопрос нам помогут две ключевых идеи.

- Заметим, что ответ на первый вопрос (о наличии вхождения) «монотонен» в смысле, указанном в разделе «Разрезание по монотонному условию». То есть, начиная с некоторого префикса строки, для всех дальнейших префиксов ответ также положителен. *Именно там, где заканчивается первый из таких префиксов, заканчивается и первое вхождение R.*

- Известно,[4] что имея регулярное выражение R или соответствующий ему автомат A, можно построить регулярное выражение R' и автомат A', распознающие зеркальные отражения строк, распознаваемых R и A, просто перевернув все последовательности внутри R, и, соответственно, все стрелки в A. Например: под выражение «`a+(b|c*d)x*`» подходит строка «`abbdxx`», а под выражение «`x*(b|dc*)a+`» — «`xxdbba`». Поэтому можно найти *начало* вхождения, запустив перевёрнутый автомат (автомат для перевёрнутого выражения) в обратную сторону строки от места, где вхождение заканчивается.

Таким образом, с помощью быстрой операции разрезания по монотонному предикату мы найдем конец первого вхождения (разрезание будет производиться по монотонному условию «переходная функция префикса отображает начальное состояние автомата в одно из терминальных» — при этом, как и в случае с суммой квадратов, в листовых блоках работы будет гораздо больше, чем во внутренних узлах), а затем с помощью перевёрнутого автомата найдем его начало. Если выражение таково, что подходящие под него строки обычно невелики, то можно просто запустить перевернутый автомат посимвольно, а если нет, то можно кэшировать в узлах верёвки переходную функцию как строки, так и ее зеркального отражения (легко убедиться, что при этом лишь тривиальным образом изменяется моноид, подставляемый с структуру данных «верёвка»). Тогда поиск начала вхождения также выразится через операцию разрезания по монотонному предикату (хотя при большом количестве вхождений посимвольный просмотр, скорее всего, будет эффективнее; возможно, оптимален какой-то гибридный подход).

На самом деле здесь имеется ряд осложнений, связанных с возможными перекрытиями вхождений различных элементов заданной системы регулярных выражений (или даже одного и того же выражения), однако общая идея сохраняется — с помощью одного разрезания находится конец вхождения, с помощью второго — начало. С подробностями можно ознакомиться в исходном коде (класс `DFAMatcher`).

---

[4] Эта идея почерпнута из статьи Расселла Кокса [7] и используется в его движке `re2`.

6.6.2. Эффективные по памяти деревья

Как уже говорилось, при представлении строки сбалансированным деревом для обеспечения разумного потребления памяти нужно сопоставлять каждому листу дерева не один символ, а целый блок. Поэтому структура данных, предложенная в [22] («подвешенные деревья (finger trees)», описанные, например, в [21] и [12]), нам не подходит: в ней создается по узлу на каждый элемент последовательности.[5]

Остается выбрать один из видов сбалансированных деревьев, подходящий под требования. Основные требования же будут таковы:

- Возможность хранить в узле дерева сумму поддерева согласно некоторого моноида;
- Дешевизна обновления этой суммы при перебалансировках дерева (и малое число перебалансировок);
- Логарифмическая (от числа элементов) высота;
- Эффективные операции разрезания и склейки.

Одни из самых простых в реализации, но в то же время достаточно эффективных сбалансированных деревьев — это деревья постоянной высоты, к которым относятся, например, 2–3-деревья [1] и B-деревья [2] (часто используемые для индексов в СУБД). В таких деревьях длина пути от корня до всех листьев одинакова, поэтому (поскольку у каждого нелистового узла есть хотя бы два ребенка) они имеют логарифмическую высоту. Обычно они решают совершенно другой класс задач — задачи представления множеств и поиска в них — однако отлично подходят и для представления последовательностей (строк). Основная идея таких деревьев в том, что узлу разрешается иметь от K до $2K-1$ детей (для некоторого значения K), и при этом большинство операций, такие как вставка, разрезание или склейка, не нарушают это свойство — а когда все-таки нарушают, то производится перебалансировка и переполнившийся узел разбивается на два, или, наоборот, два «иссякших» узла склеиваются в один.

Этой идеей мы и воспользуемся: будем использовать вариацию на тему 2–3 деревьев с блоками определенного размера в листьях, где размер блока может варьироваться от N до $2N-1$, и все данные хранятся лишь в листьях, но не во внутренних узлах. На рис. 6.13 схематически показаны реализации всех операций с такими деревьями. Фактически, достаточно

[5] Похожая структура используется для похожей цели в модуле «`Data.Sequence`» [26] из стандартной библиотеки языка Haskell.

реализовать лишь две операции: разрезание и склейку; все остальные выражаются через них. При осознании картинок нужно помнить, что мы имеем дело с деревьями постоянной высоты. Также отметим, что инвариант размера блока может нарушаться, но только в том случае, если в дереве содержится менее N элементов — в этом случае оно представляется всего одним таким маленьким блоком.

Один из самых важных аспектов нашей реализации этой структуры данных — ее «чистота»: операции над ней не изменяют существующий экземпляр, а формируют новый, то есть, являются функциями в математическом смысле. У такого подхода есть масса преимуществ (см. также статью [27]):

Исключительная простота реализации. Фактически, можно взять вышеизображенные графические схемы операций и один-в-один перенести их в код. За счет отказа от изменяемости код становится намного проще и его корректность (или, наоборот, некорректность) становится более очевидной, поскольку в нем пропадает измерение *времени* и чтобы понять механизм его работы, не нужно мысленно отслеживать последовательность операций[6] — код представляет собой всего-навсего разбор случаев, где для каждой ветки декларируется «из таких-то входов получаются такие-то выходы». И, действительно, первая же компилирующаяся версия кода прошла все тесты после исправления пары глупых ошибок.

Удобство отладки. При отладке часто возникает необходимость «заблаговременно» посмотреть на результаты тех или иных выражений, чтобы понять, нужна ли их пошаговая отладка, или же их результат верен и ошибка где-то дальше.[7] Когда же эти выражения являются «чистыми» (т. е. не имеют побочных эффектов), такой подход вполне возможен. Если же побочные эффекты присутствуют (например, изменяется какая-то структура данных), то вычисление выражения в отладчике изменит внутреннее состояние программы и дальнейшая отладка будет бессмысленной.

Полная потокобезопасность. Хорошо известно, что большинство стандартных изменяемых структур данных не допускает одновременное чтение и изменение, и доступ к ним в многопоточной программе необходимо координировать. Нередко все же бывает желательно обеспечить возможность неблокирующего чтения, пусть и не самого «актуального» состояния структу-

[6] Читателю предлагается для сравнения ознакомиться, например, с какой-либо реализацией перебалансировки в изменяемых красно-черных деревьях.

[7] Впрочем, ни для кого не секрет, что Настоящие Программисты не используют отладчик — что ж, они не смогут оценить данное преимущество.

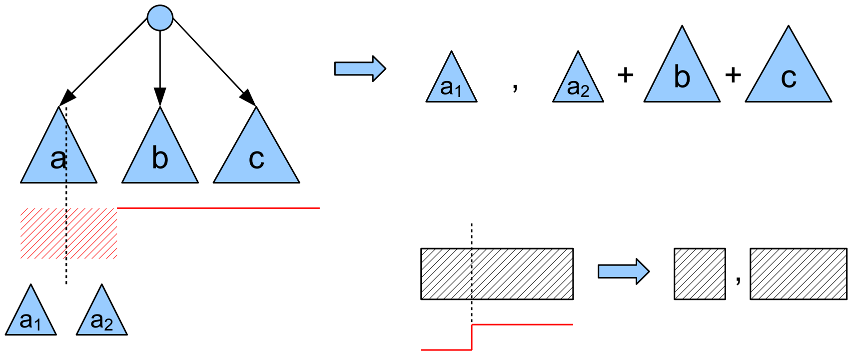


Рис. 6.13. Операции над веревками

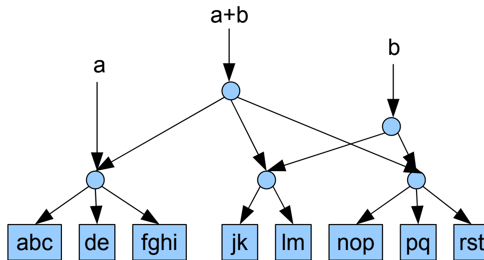


Рис. 6.14. Совместное использование памяти при склейке верёвок

ры. Существуют хитроумные трюки, позволяющие этого добиться и для изменяемых структур (см., например, реализацию класса `ConcurrentHashMap` или `ConcurrentSkipListMap` в стандартной библиотеке Java — [19], [20]), однако для чистых структур никаких трюков не нужно — любой экземпляр такой структуры можно безопасно считывать, не боясь, что он будет одновременно изменен извне.

Высокая производительность и низкое потребление памяти в некоторых сценариях. Существуют ситуации, когда полезно после применения некоторой операции к структуре не терять и ее «первоначальную» версию (например, после выполнения склейки двух веревок не терять возможности использовать склеенные части по отдельности). Например, сюда относятся алгоритмы перебора с бэктрекингом, или генетические алгоритмы (например, если два генома можно скомбинировать несколькими разными способами, или хочется оставить в живых и сами геномы, и результат их комбинирования). Конечно, можно просто скопировать первоначальную структуру — но это может быть очень неэффективно, если структура велика. В случае же чистой структуры в копировании нет необходимости: получается выигрыш в производительности. Кроме того, как видно из рис. 6.13, при выполнении многих операций над веревками выделяется лишь мизерное (константное или логарифмическое) количество дополнительной памяти. На рис. 6.14 изображен граф объектов для двух веревок и их склейки. Как видно, большая часть памяти используется этими тремя объектами совместно, и в то же время каждый объект доступен по отдельности.

6.7. Примеры и тесты

Приведем теперь пример использования получившейся библиотеки.

```
PatternSet pat = RegexCompiler.compile("007","008")

IndexedString s1 = pat.match("as00haklsdjhfla00");

IndexedString s2 = pat.match("7jhd7dsh008dsfa");

System.out.println(s1.append(s2).getMatches());
```

Программа выводит на экран:

```
[0@(15,3), 1@(25,3)]
```

Это означает, что были найдены вхождения первого и второго шаблонов соответственно в позициях 15–17 и 25–27.

В этом коде задействованы следующие элементы API:

- `RegexCompiler.compile` — компиляция нескольких регулярных выражений в автомат, распознающий каждое из них.
- `PatternSet.match` — индексирование «обычной» строки, т. е. подготовка ее к поиску заданных шаблонов.
- `IndexedString.append` — склейка двух строк, индексированных одним и тем же набором шаблонов.
- `IndexedString.getMatches` — поиск в индексированной строке.

Теперь поговорим о производительности получившейся библиотеки.

Этот разговор получится непростым, поскольку на производительность библиотеки влияет большое количество факторов.

- Размер автомата, зависящий примерно линейно от количества и размера отдельных регулярных выражений: он влияет линейно и на производительность всех операций, и на потребление памяти (чем больше, тем хуже). В программе используется алгоритм минимизации недетерминированных автоматов, описанный в [16] (правда, он реализован крайне неэффективно, но на время сопоставления это не влияет), однако он обычно уменьшает размер автомата лишь на десятки процентов.
- Размер листовых блоков верёвок линейно влияет на производительность операций поиска (чем больше, тем хуже), практически не влияет на производительность склейки, и линейно влияет на потребляемый объем памяти (чем больше блоки, тем меньше потребляется памяти).
- Особенности конкретного регулярного выражения влияют на «форму» автомата, от которой зависит скорость операций над ним (чем выражение менее «ветвистое», тем лучше производительность, поскольку

«ветвистые» выражения приводят к плотным матрицам для переходных функций, которые медленнее перемножаются в текущей реализации).
- Количество вхождений: в текущей реализации линейно влияет на время поиска (чем больше, тем хуже), однако здесь возможна оптимизация.

Кроме того, необходимо балансировать и соотношение времени на индексирование строки, на операции склейки/резки и на поиск. Индексирование выполняется довольно долго, и нужно большое количество операций склейки/резки и поиска, чтобы суммарно получить прирост по сравнению с использованием какого-либо обычного движка регулярных выражений.

Ввиду вышесказанного рассмотрим лишь один тест и проанализируем его производительность. Возьмем набор регулярных выражений из задачи «regex-dna» с Language Shootout [5] и посмотрим на производительность операций поиска в сравнении со стандартным движком регулярных выражений языка Java (`java.util.regex.Pattern` [25]) при различных длинах входной ДНК-строки (но при постоянном общем количестве вхождений) и при различных размерах листового блока: 8, 16, 32, 64, 128, 256, 512 символов. Производительность резки отдельно не измеряется, поскольку резка используется при поиске, а производительность склейки не измеряется потому, что склейка выполняется настолько быстро (выделение нескольких объектов, плюс перемножение нескольких матриц), что трудно придумать сценарий, где ее производительность будет определяющим фактором.

Вот набор входных паттернов:

```
[cgt]gggtaaa|tttaccc[acg]

a[act]ggtaaa|tttacc[agt]t

ag[act]gtaaa|tttac[agt]ct

agg[act]taaa|ttta[agt]cct

aggg[acg]aaa|ttt[cgt]ccct

agggt[cgt]aa|tt[acg]accct

agggta[cgt]a|t[acg]taccct

agggtaa[cgt]|[acg]ttaccct
```

В качестве входной строки возьмем случайную последовательность из символов «a,g,c,t» длины $50000N$ (N будет варьироваться от 1 до 10), где каждые два последовательных символа различны (поэтому вхождения указанных шаблонов там встретиться не могут), и вставим в 100 ее случайно выбранных мест вхождения случайно выбранных строк из $8 \times 2 \times 3 = 48$ строк, задаваемых указанным набором паттернов. Программа будет подсчитывать, сколько раз встречается каждый из паттернов.

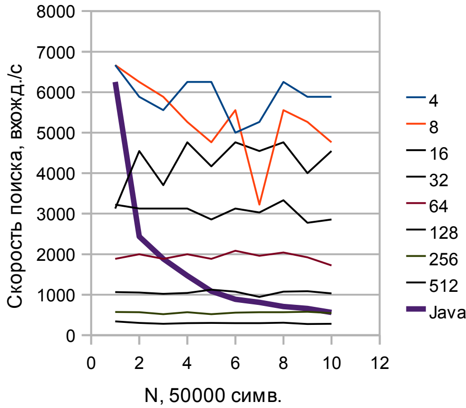
На рис. 6.15 изображены результаты тестирования. Характеристики производительности каждой из двух программ (рассматриваемый движок и стандартный движок регулярных выражений Java) показаны в тех терминах, которые наиболее осмысленны для них: для нашего движка указана скорость индексирования (символов в секунду — поскольку длительность индексирования пропорциональна числу символов на входе) и скорость поиска (вхождений в секунду — поскольку скорость поиска пропорциональна числу вхождений). Для движка Java более осмыслена была бы характеристика «количество обрабатываемых символов в секунду»; она указана на одном графике со «скоростью индексирования» нашего движка, хотя это сравнение и не совсем корректно.

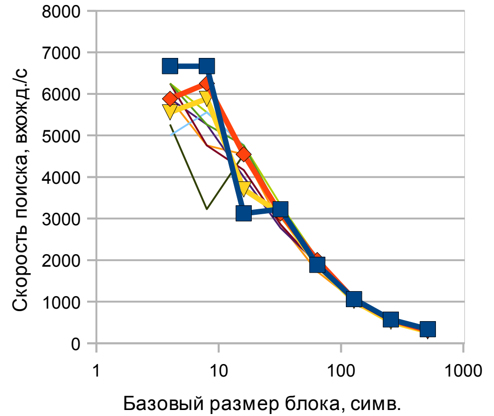
На графиках в левой части различные кривые соответствуют различным *базовым размерам листового блока* в структуре данных «верёвка», а жирная кривая соответствует использованию стандартного движка Java.

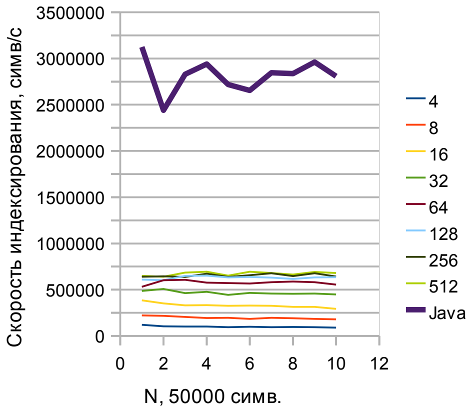
Наилучший ответ на вопрос «Когда наш движок лучше, чем стандартный движок Java» дает график зависимости скорости поиска от длины строки (слева вверху). Видно, что время поиска для движка Java пропорционально длине строки, а для нашего движка — числу вхождений. При малых базовых размерах блока (4–32 символа) наш движок существенно быстрее для больших строк.

На графиках в правой части различные кривые соответствуют различным *длинам* входной строки. Они приведены, чтобы показать, как влияет на скорость поиска и индексирования базовый размер блока. Видно, что с ростом этого размера сильно (но ограниченно) растет скорость индексирования и так же сильно падает скорость поиска.

Можно сделать вывод, что для больших строк с малым числом вхождений наш движок более эффективен, особенно если настроить его на малый размер листовых блоков. При этом, однако, резко возрастает потребление памяти — см. рис. 6.16: видно, что потребление памяти на листовой блок не зависит от размеров блока, однако, например, 4-байтовых блоков в строке







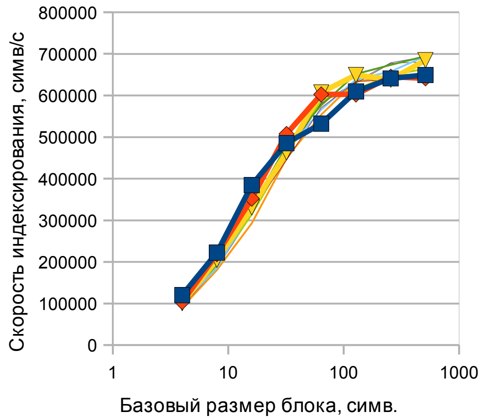


Рис. 6.15. Тестирование производительности

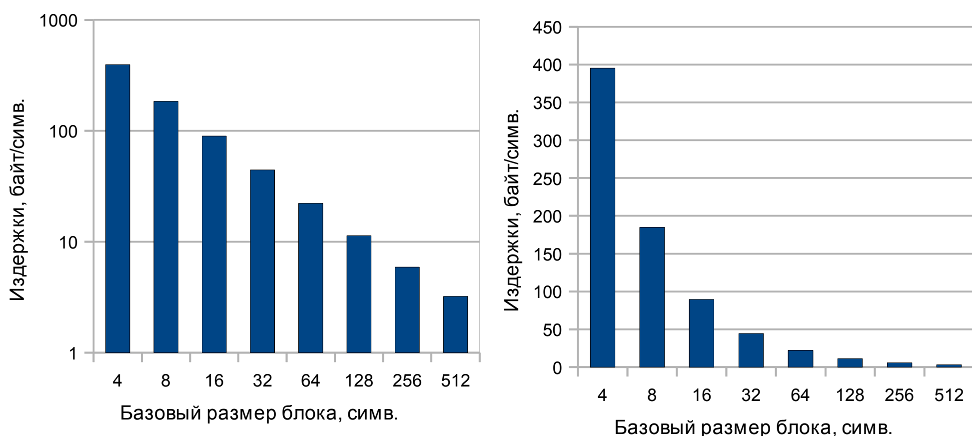


Рис. 6.16. Издержки потребления памяти

в 128 раз больше, чем 512-байтовых, поэтому в 128 раз больше и потребление памяти.

Практически всё время расходуется на перемножение булевых матриц для пересчета переходных функций листовых блоков при разрезании (например, если базовый размер листового блока равен 512, то листовые блоки будут иметь размер от 512 до 1023 символов, и при резке такого блока надо будет умножать в среднем 768 матриц); практически вся «дополнительная» память — на хранение этих матриц.

Мы не рассмотрели зависимость производительности от сложности регулярных выражений и от числа вхождений. Полное рассмотрение вопросов производительности заняло бы слишком много места; читателям предлагается самим «поиграться» с библиотекой.

Кроме того, индексирование строки можно тривиальным образом распараллелить или даже распределить по нескольким машинам с практически линейной масштабируемостью (строка бьется на несколько частей, они параллельно индексируются по отдельности, а затем склеиваются). Эта возможность в текущей версии программы никак не задействована, и сравнение параллельной реализации нашего движка с последовательной реализацией движка Java было бы некорректным, однако на практике распараллеливание почти наверняка будет оправдано.

6.8. Что дальше?

Существующая реализация не лишена ряда недостатков. Пока неясно, какие из них исправимы, а какие нет, однако они, во всяком случае, представляют собой интересные алгоритмические проблемы, достойные размышления.

- В программе вовсе не уделено внимание таким вопросам, как, например, «жадность» сопоставления. Неясно, какие из распространенных решений (POSIX, Perl, ...) поддаются эффективной реализации в рамках автоматного подхода.
- Не поддерживаются «захватывающие группы». В [7] описано, как можно реализовать их поддержку в движке на основе автоматного подхода, однако предложенный алгоритм не сочетается с подходом настоящей статьи.
- Для умножения булевских матриц (используемого при комбинировании переходных функций) используется хорошо оптимизированный, но довольно наивный алгоритм; вероятно, для некоторых сценариев использования были бы более эффективны другие алгоритмы (например, использующие разреженные представления матриц).
- При сопоставлении с помощью разрезания выполняется довольно много лишней работы: например, при обратном разрезании (выполняемом для поиска *начала* вхождения) нет никакой необходимости вычислять переходную функцию для полученных двух строк. Исправление этого недостатка позволило бы увеличить производительность на несколько десятков процентов.
- Время сопоставления пропорционально количеству вхождений, размеру листового блока и размеру автомата. Это делает программу бесполезной в качестве инкрементального лексера, поскольку в этой ситуации количество вхождений очень велико. Один из возможных способов исправления этой проблемы — модификация алгоритма разрезания строки на две части по монотонному условию, при которой разрезание будет производиться сразу на много частей, например, по возрастанию монотонной целочисленной функции.

6.9. Заключение

Итак, перед нами библиотека, осуществляющая инкрементальное сопоставление строк с набором регулярных выражений при помощи сбалансированных деревьев и моноидов. Библиотека очень эффективна в ситуации «длинные строки, не очень много регулярных выражений, мало вхождений, частый инкрементальный пересчет, много свободной памяти» и весьма неэффективна в остальных случаях. Трудно сказать, для каких еще ситуаций ее *можно* сделать эффективной — можно ли, например, все-таки сделать из нее высокопроизводительный инкрементальный лексер, и можно ли, вообще, признать этот эксперимент однозначно удачным с алгоритмической точки зрения. Автор, во всяком случае, выражает надежду, что рассмотренные техники стимулируют полет мысли у читателей-алгоритмистов и пригодятся им в других задачах.

Однако можно точно сказать, что проведенная разработка — интересный и удачный опыт скрещивания миров функционального и императивного программирования.

Рассмотрим, какие техники, идеи и «обычаи» из мира функционального программирования были использованы, и насколько удачно они сочетались с императивной природой целевого языка.

- Главная структура данных — верёвка — является «чистой» (неизменяемой). Это решение прекрасно легло на язык Java и кардинально упростило разработку и отладку, несмотря на отсутствие таких языковых средств, как алгебраические типы или сопоставление с образцом.

- Вообще, практически весь API библиотеки — «чистый» (т. е. не обладает побочными эффектами). Но отказ от изменяемого состояния производится лишь на «архитектурном» уровне, а внутри реализаций этого API изменяемое состояние используется достаточно обильно — в целях повышения производительности (например, перемножение булевских матриц), а кое-где, как ни парадоксально, в целях упрощения и сокращения кода (например, построение недетерминированного конечного автомата из регулярного выражения). За деталями предлагается обратиться непосредственно в исходный код. В целом это означает, что «чистый» подход к программированию прекрасно ложится и на императивные языки, не мешая использованию изменяемого состояния в тех задачах, где оно полезно.

- Вопреки распространенному мифу «чистое программирование не может быть эффективным и приводит к излишнему потреблению памяти», единственное узкое место производительности находится в императивном алгоритме перемножения переходных функций, а память расходуется на хранение этих функций для узлов верёвки в виде битовых масок. По-видимому, никакой связи издержек с «чистой» природой алгоритмов в данном случае нет.[8]

- В основе всей программы лежит манипулирование функциями высшего порядка: например, верёвка параметризована моноидом, а самая важная ее операция — разбиение по предикату. Поскольку Java не поддерживает компактный синтаксис определения функций (например, лямбда-выражения) и вывод типов, то использование этих сущностей сопряжено с большим количеством синтаксического мусора (особенно это касается типов в объявлениях). Однако использование их хоть и крайне важно для всей программы, но сосредоточено в сравнительно небольшом количестве кода, изолированном от конечных клиентов библиотеки. Впрочем, если бы система типов Java была чуть мощней и чуть лаконичней, можно было бы без ущерба для производительности обобщить библиотеку до поиска не только в строках, но и в произвольных последовательностях.

Автор благодарит Дмитрия Демещука, Юлию Астахову, Алексея Отта, Сергея Пластинкина и других рецензентов за предоставленные замечания.

Проект опубликован на GitHub: http://github.com/jkff/ire.[9]

Литература

[1] *Aho A. V., Hopcroft J. E.* The Design and Analysis of Computer Algorithms. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974.

[2] *Bayer R., McCreight E.* Organization and maintenance of large ordered indexes. — 2002. — Pp. 245–262.

[8]На начальных этапах разработки существовала проблема, связанная с созданием N промежуточных матриц для вычисления переходной функции блока из N символов, однако проблему удалось вылечить, сохранив чистоту интерфейса и лишь слегка изменив его.

[9]*ire* — 1) incremental regular expressions, 2) гнев, ярость (англ.)

[3] *Boehm H.-J., Atkinson R., Plass M.* Ropes: an alternative to strings // *Software — Practice and Experience.* — 1995. — Vol. 25, no. 12. — Pp. 1315–1330.

[4] Compact dfa structure for multiple regular expressions matching / W. Lin, Y. Tang, B. Liu et al. // ICC'09: Proceedings of the 2009 IEEE international conference on Communications. — Piscataway, NJ, USA: IEEE Press, 2009. — Pp. 899–903.

[5] Computer language benchmarks game: regex-dna. — URL: http://shootout.alioth.debian.org/u32q/performance.php?test=regexdna (дата обращения: 29 ноября 2010 г.).

[6] *Cox R.* Re2: an efficient, principled regular expression library. — Проект на Google Code, URL: http://code.google.com/p/re2/ (дата обращения: 29 ноября 2010 г.).

[7] *Cox R.* Regular expression matching in the wild. — URL: http://swtch.com/~rsc/regexp/regexp3.html (дата обращения: 29 ноября 2010 г.).

[8] *Cox R.* Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). — URL: http://swtch.com/~rsc/regexp/regexp1.html (дата обращения: 29 ноября 2010 г.). — 2007.

[9] Cyk algorithm. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/CYK_algorithm (дата обращения: 29 ноября 2010 г.).

[10] *Friedl J. E. F.* Mastering Regular Expressions / Ed. by A. Oram. — 2 edition. — Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[11] *Grune D., Jacobs C. J.* Parsing techniques — a practical guide. — Веб-сайт книги, URL: http://www.few.vu.nl/~dick/PTAPG.html (дата обращения: 29 ноября 2010 г.).

[12] *Hinze R., Paterson R.* Finger trees: a simple general-purpose data structure // *J. Funct. Program.* — 2006. — Vol. 16, no. 2. — Pp. 197–217.

[13] *Holland J. H.* Adaptation in natural and artificial systems. — Cambridge, MA, USA: MIT Press, 1992.

[14] *Hopcroft J. E.* An $n \log n$ algorithm for minimizing the states in a finite automaton // The Theory of Machines and Computations / Ed. by Z. Kohavi. — Academic Press, 1971. — Pp. 189–196.

[15] IHMC CmapTools. — URL: http://cmap.ihmc.us/ (дата обращения: 29 ноября 2010 г.).

[16] *Ilie L., Navarro G.* On NFA reductions // THEORY IS FOREVER. — Springer, 2004. — Pp. 112–124.

[17] *Kuklewicz C.* regex-tdfa. — Пакет на hackage, URL: http://hackage.haskell.org/package/regex-tdfa (дата обращения: 29 ноября 2010 г.).

[18] *Laurikari V.* TRE: The free and portable approximate regex matching library. — URL: http://laurikari.net/tre/ (дата обращения: 29 ноября 2010 г.).

[19] *Lea D.* Реализация класса java.util.concurrent.concurrenthashmap. — URL: http://www.docjar.com/html/api/java/util/concurrent/ConcurrentHashMap.java.html (дата обращения: 29 ноября 2010 г.).

[20] *Lea D.* Реализация класса java.util.concurrent.concurrentskiplistmap. — URL: http://www.docjar.com/html/api/java/util/concurrent/ConcurrentSkipListMap.java.html (дата обращения: 29 ноября 2010 г.).

[21] Monoids and finger trees. — URL: http://apfelmus.nfshost.com/monoid-fingertree.html (дата обращения: 29 ноября 2010 г.).

[22] *Piponi D.* Fast incremental regular expression matching with monoids. — Блог-пост, URL: http://blog.sigfpe.com/2009/01/fast-incremental-regular-expression.html (дата обращения: 29 ноября 2010 г.).

[23] *Piponi D.* Haskell monoids and their uses. — Пост в блоге, URL: http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html (дата обращения: 29 ноября 2010 г.).

[24] *Wilke T., Huch F., Fischer S.* Weighted regexp matching. — URL: http://sebfisch.github.com/haskell-regexp/ (дата обращения: 29 ноября 2010 г.).

[25] Документация к классу java.util.regex.pattern. — URL: http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html (дата обращения: 29 ноября 2010 г.).

[26] Документация к модулю `Data.Sequence`. — URL: http://www.haskell.org/ghc/docs/6.12.2/html/libraries/containers-0.3.0.0/Data-Sequence.html (дата обращения: 29 ноября 2010 г.).

[27] *Кирпичёв Е.* Изменяемое состояние: опасности и борьба с ними // — *Практика функционального программирования*2009. — July. — Т. 1.

[28] *Пипони, Дэн.* Моноиды в haskell и их использование (перевод Кирилла Заборского) // — *Практика функционального программирования*2009. — July. — Т. 1.

[29] *Советов, П.* Алгоритм Бржозовского для минимизации конечного автомата. — URL: http://sovietov.com/txt/minfa/minfa.html (дата обращения: 29 ноября 2010 г.).