БИБЛИОТЕКА ПРОГРАММИСТА



Алехандро Серано Мена

ИЗУЧАЕМ HASKELL

Apress®

ПИТЕР®

Alejandro Serrano Mena

Beginning Haskell: A Project-Based Approach

Apress®



БИБЛИОТЕКА ПРОГРАММИСТА

Алехандро Серано Мена

ИЗУЧАЕМ HASKELL



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2015

А. Мена

Изучаем Haskell. Библиотека программиста

Серия «Библиотека программиста»

Перевел с английского Н. Вильчинский

Заведующий редакцией	*П. Щеголев*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*А. Жданов*
Художник	*В. Шимкевич*
Корректор	*С. Беляева, Н. Викторова*
Верстка	*Л. Соловьева*

ББК 32.973.2-018.1
УДК 004.43

Мена А.

М45 Изучаем Haskell. Библиотека программиста. — СПб.: Питер, 2015. — 464 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-01188-4

Эта книга поможет вам быстро освоить базовые концепции языка программирования Haskell, его библиотеки и компоненты, а также заложит основы функциональной парадигмы программирования, которая становится все более значимой в современном мире разработки ПО. Книга предлагает проектный подход к освоению материала, используя в качестве прототипа проект реализации интернет-магазина. Здесь рассматривается экосистема языка Haskell и его вспомогательных средств, инструменты Cabal для управление проектами, модули HUnit и QuickCheck для тестирования программ, фреймворк Scotty для разработки веб-приложений, Persistent и Esqueleto — для управления базами данных и многие другие компоненты и библиотеки Haskell.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1430262503 англ.
ISBN 978-5-496-01188-4

© Apress
© Перевод на русский язык ООО Издательство «Питер», 2015
© Издание на русском языке, оформление
ООО Издательство «Питер», 2015

Права на издание получены по соглашению с Apress. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 28.10.14. Формат 70×100/16. Усл. п. л. 37,410. Тираж 1000. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

Оглавление

Об авторе **9**

О научном редакторе **9**

Благодарности **10**

Введение **11**
Зачем изучать функциональное программирование 11
Почему вам нужна эта книга 11

Часть I. Первые шаги **13**

Глава 1. Настоящий функциональный язык **14**
Почему именно Haskell? 14
История Haskell 19
Ваша рабочая среда 19
Первые шаги в использовании GHCi 24
Магазин, продающий машины времени 26
Выводы 27

Глава 2. Объявление модели данных **28**
Работа с символами, числами и списками 28
Создание нового проекта 36
Определение простых функций 40
Работа с типами данных 46
Сопоставление с образцом 49
Записи 58
Выводы 63

Глава 3. Многократное использование кода с помощью списков **64**
Параметрический полиморфизм 65
Функции в качестве аргументов 68
Подробнее о модулях 76
Углубленное изучение списков 80
Формирователи списков 91

Haskell-оригами 95
Выводы 98

Глава 4. Контейнеры и классы типов 100
Использование пакетов 100
Контейнеры: проекции, множества, деревья и графы 108
Специальный полиморфизм: классы типов 120
Двоичные деревья для минимальной цены 129
Классы типов, связанные с контейнерами 134
Выводы 138

Глава 5. Лень и бесконечные структуры 139
Бесконечное количество машин времени 139
Модель ленивых вычислений 144
Профилирование с помощью GHC 152
Строгие аннотации 157
Выводы 160

Часть II. Анализ данных 161

Глава 6. Ознакомление с клиентами с помощью монад 162
Анализ данных 163
Исследование монад 174
Различные виды состояний 186
Выводы 193

Глава 7. Другие рекомендуемые монады 194
Возвращение более одного значения 194
Неудачи и альтернативы 197
Вывод ассоциативных правил 200
Задачи поиска 207
Монада Logic 208
Возрождение монад и списков 210
Выводы 223

Глава 8. Использование нескольких ядер 224
Параллелизм, одновременность, распределенность 224
Монада Par 226
Программная транзакционная память 232
Облачные решения для Haskell 241
Выводы 248

Часть III. Использование ресурсов 249

Глава 9. Работа с файлами: ввод-вывод и библиотека conduit 250
Базовые ввод и вывод 250
Случайность 254
Работа с файлами 257
Обработка ошибок 261
Потоковый ввод-вывод данных с помощью библиотеки conduit 270
За пределами текстовых файлов 277
Выводы 281

Глава 10. Создание и синтаксический разбор текста 282
Пять текстовых типов данных 282
Создание текста со скоростью ветра 287
Синтаксический разбор с использованием пакета attoparsec 290
Представление новых классов типов 296
Не нужно излишеств: используйте формат JSON 304
Выводы 310

Глава 11. Безопасный доступ к базам данных 311
Средства доступа к базам данных 311
Соединение 314
Схемы и переносы 316
Запросы 323
Вставка, обновление и удаление 330
Выводы 333

Глава 12. Веб-приложения 334
Веб-экосистема Haskell 334
Структура RESTful 338
Разработка внутреннего интерфейса в среде Scotty 339
Разработка внешнего интерфейса с помощью компилятора Fay . . . 349
Выводы 354

Часть IV. Предметно-ориентированные языки 355

Глава 13. Строгие типы для описания предложений 356
Предметно-ориентированные языки 356
Безопасность языка выражений 362
Зависимая типизация 365
Программирование на уровне типов в Haskell 370
Функциональные зависимости 373

Семейства типов 379
Продвижение типа и одноэлементные типы 385
Выводы 394

Глава 14. Интерпретация предложений с помощью атрибутов 395
Варианты интерпретации и атрибутивные грамматики 395
Наша первая атрибутивная грамматика 398
Интеграция UUAGC-кода в пакет 401
Интерпретация выражений 404
Варианты интерпретации предложений 409
Оригами-программирование для любого типа данных 415
Выводы 418

Часть V. Инжиниринг магазина 419

Глава 15. Документирование, тестирование и проверка 420
Документирование двоичных деревьев с помощью утилиты Haddock 421
Блочное тестирование с помощью утилиты HUnit 424
Рандомизированное тестирование с помощью библиотеки QuickCheck 430
Формальная верификация с использованием языка Idris 434
Выводы 438

Глава 16. Создание архитектуры приложения 439
Паттерны проектирования и функциональное программирование . . 439
Рекомендации среднего порядка 441
Утилиты 443
Проекты 446
Выводы 458

Приложение А. Дальнейшие перспективы 459
Haskell-ресурсы 460

Приложение Б. Путешествие во времени с Haskell 462

Об авторе

Алехандро Серано Мена — горячий сторонник функционального программирования, он использует Haskell для личных и коммерческих проектов уже более пяти лет. В настоящее время пишет кандидатскую диссертацию в Утрехтском университете. Область его интересов связана с разработкой систем со строгой типизацией и взаимодействием с такими системами, например, посредством языка Haskell.

Он внес вклад в ряд проектов с открытым кодом, включая Mono и Nemerle. В 2011 году принял участие в программе Google Summer of Code, совершенствуя подключаемый Haskell-модуль EclipseFP для популярной среды разработки Eclipse. В 2012 году он с двумя друзьями основал компанию Nublic, деятельность которой сконцентрирована на предоставлении облачных инструментов для домашних сред. Основная часть программного обеспечения данного проекта была создана с использованием языка Scala, сочетающего объектно-ориентированные и функциональные аспекты.

Диплом по информатике и математике Алехандро получил в родном Мадридском университете. В период обучения он активно продвигал среди студентов идеи свободного программного обеспечения и функциональной парадигмы. Его статьи можно прочитать в испанском журнале «Todo Programación».

О научном редакторе

Жан-Филипп Моресмо пишет программы уже пятнадцать лет. Он занимался созданием веб-приложений с 1996 года и всегда испытывал интерес к интегрированным средам разработки и анализу языка.

Он принимает активное участие в работе сообщества по разработке приложений с открытым кодом на языке Haskell, а также входит в группу поддержки проекта EclipseFP. Его блог находится по адресу http://jpmoresmau.blogspot.com. Ж.-Ф. Моресмо с женой и двумя детьми живет на юге Франции.

Благодарности

Прежде всего, хотел бы отметить грандиозную работу научного редактора и редакции. Их комментарии и предложения представляли исключительную ценность и позволили существенно улучшить качество книги. Еще я хотел бы отдельно отметить великолепную работу Жана-Филиппа Моресмо над EclipseFP.

Написание книги — дело не только благодарное, но и хлопотное. И в те моменты, когда настроение падало до нуля, я всегда ощущал поддержку Елены. Она была со мной и в минуты размышлений, и в минуты оттачивания примеров, и в минуты бесконечных правок текста.

Семья и друзья поддерживали меня все это время, поэтому большое спасибо вам всем. Мои родители, Кармен и Хулиан, заслуживают особого упоминания: они поддерживали меня при выполнении каждого моего проекта, и как бы странно это ни звучало, на протяжении всей моей жизни. Они купили мне первый компьютер, и именно на нем я впервые начал программировать в старой доброй среде Visual Basic. Поэтому хотя бы ради справедливости нужно сказать, что без их помощи вы бы никогда не читали эту книгу.

Нельзя не восхититься всем Haskell-сообществом. Наличие обширных подписок на рассылки и групповых чатов, наполненных собеседников (заметьте, умных собеседников!), всегда готовых прийти на помощь, является хорошим стимулом для проявления любопытства и познания нового. Каждая библиотека и каждый компилятор, о которых рассказывается в книге, были с любовью разработаны этим сообществом: именно они сделали Haskell таким великим языком.

И в завершение хочу поблагодарить своих преподавателей и коллег из Утрехтского университета. Хотя мы были знакомы всего лишь пару месяцев, их страстное увлечение функциональным программированием оказало существенное влияние на книгу.

Введение

Функциональное программирование становится все более и более популярным. Некоторые аспекты этой парадигмы заимствуют даже самые знаменитые языки, в том числе Java и C#, а Haskell, Scala, Clojure или OCaml, олицетворявшие функциональное программирование с самого начала, становятся все более и более востребованными. Haskell — это защищенный от побочных эффектов, чистый функциональный язык с давней историей, огромным количеством библиотек и активным сообществом. Все это превращает Haskell в отличный инструмент для изучения и внедрения функционального программирования.

Зачем изучать функциональное программирование

Взлет функционального программирования поддерживается с двух сторон. В наши дни большинство приложений для увеличения производительности широко используют *одновременное* выполнение заданий или требуют их *параллелизма*. Вспомните любой веб-сервер, которому нужно одновременно обрабатывать тысячи подключений. Haskell-код позволяет легко перейти от однопоточного к многопоточному приложению при весьма незначительных затратах.

Помимо того что приложения все чаще занимаются одновременной обработкой информации, они постоянно *растут в объеме*. Учитывая это, среда разработки должна помогать выявлять дефекты и обеспечивать взаимодействие между всеми модулями создаваемой системы. В Haskell используется очень строгая система типов, следовательно, вы можете задействовать в своем коде широкий диапазон инвариантов, проверяемых в ходе компиляции. Многие дефекты, которые ранее могли выявляться только с помощью тестов, теперь полностью нейтрализуются компилятором. Упрощается и оптимизация кода, поскольку можно гарантировать, что изменения в коде не повлияют на инварианты.

Изучение функционального программирования позволит вам (как разработчику) претендовать на более выгодные вакансии. Функциональное мышление и в дальнейшем останется влиятельной силой на основных направлениях программирования. Вы будете готовы к разработке более объемных и быстродействующих приложений, которые полностью удовлетворят пожелания ваших потребителей.

Почему вам нужна эта книга

Эта книга сконцентрирована как на *идеях*, положенных в основу Haskell-программирования, так и на его *практических аспектах*. В ней показано, как кон-

цепции функционального программирования применяются в реальных сценариях. Здесь вы научитесь работать с утилитами и библиотеками, предоставляемыми Haskell для решения конкретных задач. Однако выиграют от прочтения этой книги не только новички в функциональном программировании. Разработчики, пишущие на языке Scala, Clojure, Lisp или ML, смогут разобраться, что отличает Haskell от других языков программирования.

Материал книги базируется на проекте создания витрины интернет-магазина. В пяти частях этой книги описываются все подсистемы магазина: представление клиентов и товаров в памяти, получение данных (с использованием параллельных и одновременных вычислений), постоянное хранилище, скидки и предложения, общая архитектура приложения. Темы были тщательно отобраны, чтобы вы получили представление обо всей экосистеме Haskell.

Часть I
ПЕРВЫЕ ШАГИ

Глава 1. Настоящий функциональный язык

Глава 2. Объявление модели данных

Глава 3. Многократное использование кода с помощью списков

Глава 4. Контейнеры и классы типов

Глава 5. Лень и бесконечные структуры

Настоящий функциональный язык

1

Добро пожаловать в мир Haskell! Прежде чем углубляться в особенности, давайте посмотрим, что отличает Haskell от других языков и какие преимущества можно извлечь из этих отличий. Haskell принадлежит к довольно обширному семейству *функциональных языков*, к которому относятся такие языки, как ML, Lisp, Scala и Clojure. Если у вас есть опыт работы с императивными или объектно-ориентированными языками, например с C, C++ или Java, то вы познакомитесь с новыми идеями, заложенными в семейство функциональных языков. Если у вас уже есть опыт работы с функциональными языками, то вы увидите, как другие присущие Haskell функциональные возможности, например отложенные вычисления (lazy evaluation) или классы типов (type classes), делают этот язык не похожим ни на какой другой язык программирования.

Книга не требует от вас каких-либо предварительных представлений о функциональной парадигме, но некоторые основные знания в области программирования, например что такое циклы или условия, от вас все же потребуются, как и начальные навыки работы с оболочкой или консолью.

После знакомства с Haskell мы рассмотрим процесс подготовки и полной установки Haskell, включая среду EclipseFP, объединяющую Haskell и интегрированную среду разработки (Integrated Development Environment, IDE) Eclipse. И наконец, мы сделаем первые шаги по освоению языка в интерпретаторе Glasgow Haskell Compiler (GHC), эффективном средстве, выполняющем выражения в интерактивном режиме. В процессе чтения книги мы будем разрабатывать составные части интернет-магазина по продаже машин времени: как и многое другое в этой жизни, Haskell лучше всего изучать, создавая программы.

Почему именно Haskell?

Если вы читаете эту книгу, значит, вы заинтересованы в изучении Haskell. Но что придает ему особый статус? Используемый в этом языке подход к программированию можно свести к пяти пунктам:

- Haskell относится к семейству *функциональных* языков.
- Haskell воплощает понятие *чистоты*, отделяя код с дополнительными эффектами от остального приложения.

- Модель вычислений языка Haskell основана на концепции *лени* (laziness).
- *Типы* подвергаются *статической проверке* со стороны компилятора. Кроме того, особенностью Haskell является система типов, которая намного *строже* и выразительнее, чем обычно.
- Подходы к полиморфизму в Haskell основаны на *параметричности* (parametricity), напоминающей обобщения в Java и C#, и *классах типов*.

Далее вам представится возможность узнать подробнее обо всех этих понятиях и о том, какой в них вкладывается смысл. Мы познакомимся со всей экосистемой Haskell, к которой относятся компилятор, библиотеки и инструменты, доступные в обычном дистрибутиве.

Зачем вообще использовать чисто функциональное программирование?

Функциональное программирование является одним из стилей, или парадигм, программирования. Парадигмой программирования называют множество идей и понятий, используемое различными языками. Например, Pascal и C относятся к императивной парадигме, а Java и C++ объединяют в себе императивную и объектно-ориентированную парадигмы. Основной отличительной чертой функционального программирования являются привелигированные права *функций*. Это означает, что с функциями можно работать так же, как с любыми другими данными, имеющимися в программе. Функции могут передаваться в качестве аргументов другим функциям, возвращаться в качестве результата или присваиваться переменным. Эта возможность рассматривать функции в качестве данных позволяет перейти на более высокий уровень абстракции и, следовательно, дает больше перспектив в плане многократного использования.

Рассмотрим, к примеру, задачу последовательного перебора структуры данных с выполнением какого-либо действия над каждым элементом. В объектно-ориентированном языке создатель структуры, скорее всего, воспользуется образцом итератора, а вы, как потребитель, напишете нечто подобное следующему Java-коду:

```
Iterator it = listOfThings.iterator();
while (it.hasNext()) {
    Element e = it.next();
    action(e); // выполняем действие
}
```

Как видите, в этом примере много шаблонного кода. А в Haskell вы бы воспользовались функцией `map`, которой в качестве аргумента передавалось бы действие, выполняемое над каждым элементом:

```
map listOfThings action
```

Теперь код намного короче, причем благодаря функции `map` намерения программиста стали понятнее. Более того, устраняются любые возможные проблемы, связанные с неверным применением образца итератора, поскольку все тонкости абстрагированы в функции. Вообще-то такие функции, как `map`, встречаются

в функциональном коде довольно часто, и это дает уверенность, что любые недочеты в их реализации будут обнаружены довольно быстро.

Выполнение аналогичных операций на языке Java требует создать на стороне поставщика интерфейс, содержащий функцию для выполнения операции. Затем на стороне пользователя нужно реализовать этот интерфейс посредством класса или задействовать анонимный класс. По объему такой код будет значительно превосходить нашу однострочную версию. Фактически новые версии Java, C++ и C# воспринимают функциональные концепции и позволяют создавать код, гораздо больше похожий на функциональный.

В Haskell фрагмент кода состоит из *выражений*, вычисляемых примерно так же, как и математические выражения. В то же время методы в императивном языке состоят из инструкций, изменяющих глобальное состояние. Это довольно важное отличие, поскольку в императивной программе один и тот же фрагмент кода при своем выполнении может иметь разные результаты в зависимости от исходного состояния. Здесь важно отметить, что элементы, не контролируемые программой и известные как *побочные эффекты* (такие, как ввод-вывод, сетевое взаимодействие, случайность), также являются частью этого глобального состояния, которое может меняться между вызовами одной и той же функции.

Изначально выражения в Haskell не могут иметь побочных эффектов: об этих выражениях принято говорить, что они *чистые*. В отношении функционального программирования бытует заблуждение, что в нем запрещаются любые изменения внешнего состояния. Это неправда: побочные эффекты в Haskell иногда возникают, но язык заставляет программиста отделять чистые (свободные от побочных эффектов) фрагменты кода от нечистых.

Основным преимуществом чистоты является более понятное аргументирование предназначения кода и упрощенные подходы к его тестированию. У разработчика есть уверенность, что итог работы чистой функции зависит только от ее параметров, а каждый запуск этой функции с одними и теми же входными данными приводит к одинаковому результату. Это свойство называется *ссылочной прозрачностью* (referential transparency) и является основой применения формальных методов проверки, краткий обзор которых будет дан в главе 3, а более подробное рассмотрение предстоит в главе 15.

Чистые функции проще составлять, поскольку они гарантируют отсутствие взаимного влияния в ходе выполнения. Фактически вычисление чистых выражений не зависит от порядка этого вычисления, а это открывает двери к различным *стратегиям выполнения* (execution strategies) одного и того же фрагмента кода. Эта особенность используется в Haskell-библиотеках, позволяя осуществлять параллельные и одновременные вычисления, и даже была применена для кода планирования в GPU библиотеки Accelerate.

По умолчанию в Haskell используется стратегия *отложенных*, или *ленивых*, *вычислений*. В режиме отложенных вычислений выражение не вычисляется до тех пор, пока результат его вычисления не понадобится для вычисления более крупного выражения. Как только выражение будет вычислено, результат сохраняется для последующих вычислений или сбрасывается, если он не нужен какому-нибудь другому выполняемому коду. В этом просматриваются как очевидные преимущества

(в ходе выполнения программы производится минимальный объем вычислений), так и недостатки (все отложенные выражения, которые еще не были вычислены, нужно сохранять в памяти). При всей своей высокой эффективности отложенные вычисления могут проявлять немалую долю коварства, в чем мы сможем убедиться, изучая главу 5.

Почему строгая статическая типизация?

В том или ином виде системы типов присутствуют практически во всех языках программирования. *Система типов* — это абстракция, в которой значения, способные появляться в процессе выполнения программы, распределяются по категориям и маркируются как типы. Эти типы обычно служат для ограничения возможного множества действий, применяемых к значению. Например, может быть разрешено объединение двух строк, но запрещено их деление.

В общем, эта маркировка может проверяться в двух определенных моментах: во время выполнения (*динамическая* типизация), как это обычно и бывает при использовании языков со слабой типизацией (looser typing), допускающих, к примеру, неявные преобразования между целочисленными и строковыми значениями; или во время компиляции (*статическая* типизация), когда программы перед формированием на выходе целевого кода (обычно машинного кода, или байт-кода) и получения разрешения на запуск должны подвергаться проверке на наличие полностью определенных типов согласно понятиям языка. Haskell относится к этой второй категории: все программы перед выполнением должны пройти проверку на соответствие типов. Некоторые из языков со статической типизацией, например Java или C#, нуждаются в период выполнения программы в дополнительной проверке типов. В отличие от этого после компиляции Haskell-программы никаких проверок типов больше не проводится, что существенно повышает производительность.

В Haskell очень строгая система типов. Под строгостью в данном случае понимается количество инвариантов, учитываемых в ходе компиляции до того, как будет выдана ошибка при выполнении приложения. Строгость повышает уверенность в коде, прошедшем проверку типов, и в кругах приверженцев языка Haskell нередко раздаются слова: «Раз код откомпилирован, то он будет работать». Строгая типизация порождает такой способ программирования, который называется *программированием с ориентацией на типы* (type-oriented programming). Обычно программист знает тип разрабатываемой им функции и у него есть общая идея насчет структуры кода. Затем он «заполняет пустые места» теми выражениями, которые вписываются в его представления. Этот подход был формализован, существует даже еще один похожий на Haskell язык — Agda (http://wiki.portal.chalmers.se/agda). Agda поставляется вместе с интерактивной средой программирования, позволяющей заполнять такие пустые места и даже делать это в автоматическом режиме, если где-то допустим только один вариант.

В конце книги мы ненадолго перейдем от Haskell к Idris (http://www.idris-lang.org/) — языку с похожим синтаксисом, но *зависимой типизацией* (dependent typing). Он предлагает еще более строгую форму проверки типов, где можно фактически выразить инварианты: например, «если я объединяю список из *n* элементов

со списком из m элементов, то получаю список из $n+m$ элементов» или: «я не могу получить первый элемент пустого списка». Далее вы увидите, как некоторые из этих технологий могут быть перенесены в Haskell в качестве образцов.

Что касается типизации, то последнее отличие Haskell от других языков относится к полиморфизму. Вопрос полиморфизма носит двойственный характер. Прежде всего, нам нужно создавать функции, работающие со списками, не заботясь при этом о типах содержащихся в них элементов. Это так называемый *параметрический полиморфизм* (parametric polymorphism), который рассматривается в главе 3. В других случаях нам нужно выразить тот факт, что некоторые типы позволяют проделывать над своими значениями вполне определенные операции. Например, идея применения функции ко всем элементам списка, как это делалось ранее с функцией `map`, может быть обобщена на понятие операции, применяющей функцию ко всем элементам некой структуры данных, подобной дереву или графу. Таким решением являются *классы типов* (type classes), объединяющие разные типы с общим интерфейсом. Классы типов рассматриваются в главе 4, где также наглядно показано, что это весьма высокоуровневое понятие, позволяющее выразить ряд абстрактных идей (функторы, монады), которые придают Haskell-коду особый колорит.

Экосистема Haskell

До сих пор речь шла о Haskell только как о языке. Однако преимущества Haskell не только в самом языке, но и в большом и постоянно расширяющемся множестве инструментов и библиотек, которые могут использоваться с языком.

Для Haskell доступно несколько компиляторов, которые обычно называются в честь того или иного города: на момент написания книги поддерживались компиляторы GHC (http://www.haskell.org/ghc/) — Глазго, UHC (http://www.cs.uu.nl/wiki/UHC) — Утрехта и JHC (http://repetae.net/computer/jhc/). Из них в качестве стандартного компилятора обычно используется GHC, обладающий самым широким набором свойств. Мы также последуем общей тенденции и при изложении материала книги будем работать с компилятором GHC.

Как и у любого другого популярного языка программирования, у Haskell есть собственное онлайновое хранилище библиотек. Оно называется Hackage и доступно по адресу http://hackage.haskell.org/. Hackage отлично интегрируется с Cabal, инструментом для создания проектов на языке Haskell. В Hackage существуют самые разнообразные библиотеки — от биоинформатики и до создания игровых программ, оконных диспетчеров и многого другого.

Кроме GHC и Cabal, в этой книге рассматриваются инструментальные средства, предназначенные для оказания помощи разработчикам в плане ускоренного создания высококачественного кода. И в первую очередь это GHC-профайлер, о котором рассказывается в главе 5. Он применяется для обнаружения утечек памяти и нерационального использования времени. Также мы уделим внимание средствам Hoogle и Haddock, применяемым для просмотра и создания документации. В главе 14 мы поговорим о системе UU Attribute Grammar System, помогающей создавать предметно-ориентированные языки.

История Haskell

Обычно Haskell считают потомком языка программирования Miranda, который в 1980-х годах был одним из наиболее заметных языков отложенного функционального программирования. Но в то время в мире существовало множество других языков того же рода. Это создавало трудности для исследователей данной парадигмы в плане получения общей базы для экспериментов. Поэтому в конце десятилетия они решили создать совершенно новый язык, который бы стал основным для их исследований.

В 1990-е годы было разработано несколько версий языка Haskell. В этот период Haskell вобрал в себя ряд свойств, придающих ему особый колорит, например классы типов и монады для управления вводом и выводом. В 1998 году появилось сообщение о выходе версии Haskell 98, которая была принята за стандарт для любого Haskell-совместимого компилятора. Именно на эту версию и ориентируется большинство разработчиков библиотек.

И тем не менее в Haskell-компиляторы, главным образом в GHC, исследователями и создателями компиляторов было внедрено множество новых идей. Некоторые из таких расширений получили весьма широкое распространение, что заставило в 2010 году пересмотреть стандарт языка Haskell. На момент написания книги компилятор GHC был ориентирован именно на ту версию языка, которая соответствует данному пересмотренному стандарту.

По мере роста популярности языка к GHC и другим компиляторам добавлялись все новые и новые расширения, которые, как правило, по желанию разработчика можно включать и отключать. В результате появился более-менее упорядоченный график своевременного выпуска пересмотренных стандартов по Haskell.

Ваша рабочая среда

Теперь у вас уже должно появиться желание попробовать поработать с Haskell на своем компьютере. И первым делом, конечно же, нужно обзавестись рабочей копией Haskell на своей системе. Разработчики Haskell озаботились тем, чтобы процесс установки проходил легко и просто, создав дистрибутив Haskell Platform, содержащий компилятор GHC, систему сборок и библиотек Cabal, а также полный набор библиотек. Чтобы получить Haskell Platform, нужно перейти по адресу http://www.haskell.org/platform/, после чего выполнить все инструкции, соответствующие используемой операционной системе.

Установка под Windows

Установка под управлением этой операционной системы осуществляется предельно просто: загружаемый файл является исполняемым, поэтому он сам обо всем позаботится.

Установка под Mac OS X

Чтобы разрабатывать программы на Haskell в Apple, вам необходима среда XCode, содержащая основной инструментарий разработчика. Компоненты XCode распространяются свободно, но компания Apple перед их загрузкой требует регистрации. Для этого следует посетить веб-сайт https://developer.apple.com/xcode/.

Установка пакета Apple XCode даст вам инструменты, работающие в графическом режиме, но вам потребуются также консольные инструменты. Для их получения нужно открыть XCode IDE и перейти в окно Preferences (Предпочтения). В этом окне нужно щелкнуть на вкладке Downloads (Загрузки), и вы увидите элемент под названием Command Line Tools (Средства командной строки), который и нужно будет выбрать для установки на вашей системе. На рис. 1.1 выделена строка с этим инструментом, а также кнопка Install (Установить), на которой нужно щелкнуть для выполнения установки.

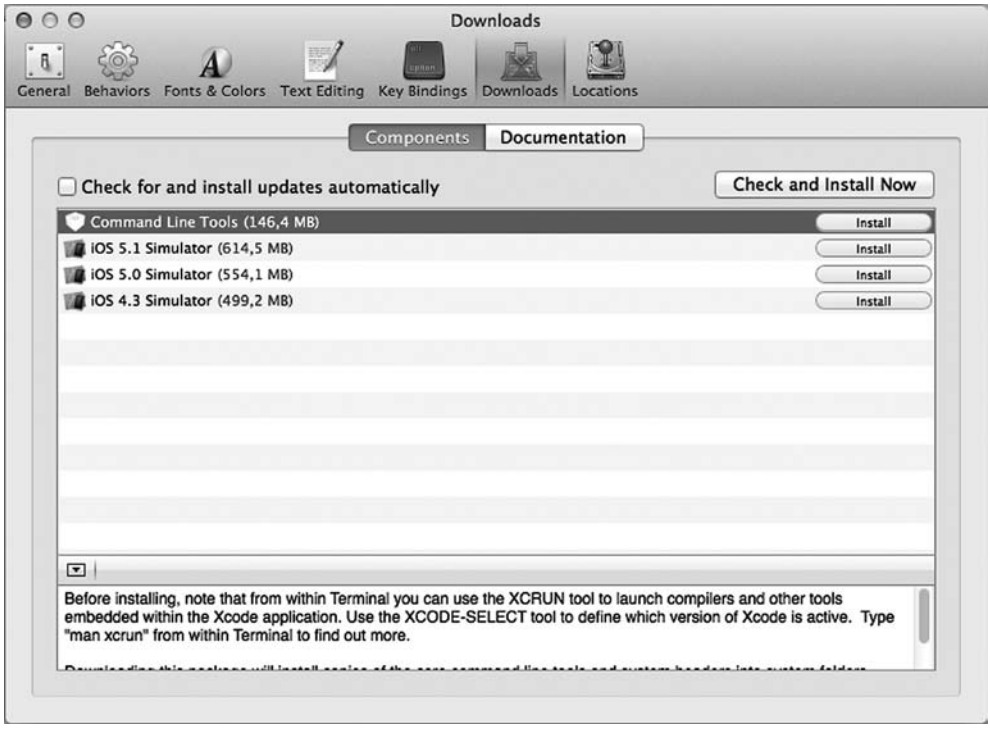


Рис. 1.1. Установка инструментов командной строки XCode

Затем нужно будет решить, какая версия Haskell Platform вам нужна, 32- или 64-разрядная. Существенных различий между ними нет. Но если с помощью компилятора Haskell планируется создание интерфейса для некоторых внешних библиотек, нужно обеспечить загрузку той версии, которая соответствует архитектуре, где были скомпилированы эти библиотеки.

Установка под Linux

Мир дистрибутивов Linux весьма разнообразен, поэтому посоветовать наилучший вариант установки Haskell на этих системах очень сложно. Если используется дистрибутив, поддерживающий какую-нибудь систему управления пакетами, то лучше придерживаться этой системы. Например, системы на основе Debian, такие как Ubuntu, поддерживают команду `apt-get`, что позволяет запустить следующую команду[1]:

```
$ sudo apt-get install haskell-platform
```

В Fedora и других дистрибутивах на основе Red Hat нужно воспользоваться следующей строкой:

```
$ yum install haskell-platform
```

Такие же команды существуют и для остальных широко распространенных дистрибутивов. Просмотреть полный список дистрибутивов, предлагающих установку Haskell Platform, можно на веб-сайте Platform.

Установка под Linux из исходного кода

В случае если вам захочется или потребуется осуществить полную установку из исходного кода, нужно выполнить следующие действия:

1. Перейти на веб-страницу компилятора GHC (http://www.haskell.org/ghc/), щелкнуть на ссылке Download (Загрузка) и получить двоичный пакет для самого последнего стабильного выпуска.
2. Распаковать только что загруженный файл в нужное место. Папку, куда он будет распакован, я рекомендую указать в вашей системной переменной PATH. Чтобы запустить этот двоичный файл, может понадобиться установка ряда библиотек, например GMP.

ПРИМЕЧАНИЕ

Компилятор GHC также можно создать из исходного кода. Но поскольку это довольно трудоемкий и не исключающий ошибок процесс, рекомендуется использовать дистрибутив в виде двоичного файла. Если все же есть желание пойти по этому пути, обратитесь к вики-сайту Haskell (http://hackage.haskell.org/trac/ghc/wiki/Building).

3. Вернуться на страницу Haskell Platform для загрузки его исходного кода.
4. Распаковать, создать и установить платформу, что обычно выполняется запуском следующих команд:

```
$ tar -xzvf haskell-platform-*.tar.gz
$ cd haskell-platform-*
$ ./configure
$ make
$ make install
```

[1] В книге действует соглашение, согласно которому перед всем, что вводится с консоли, ставится знак доллара ($).

Проверка успешности установки

Теперь настало время посмотреть, насколько корректно была установлена среда Haskell Platform. Для этого нужно открыть консоль, набрать команду `ghci -e 5+3` и нажать Enter. В результате на экран должна быть выведена цифра 8.

Установка EclipseFP

Одной из наиболее эффективных сред программирования на Haskell является интегрированная среда разработки EclipseFP. Она предусматривает подсветку синтаксиса, автозавершение кода, встроенную документацию, профилирование и блочное тестирование, а также возможность использования графических мастеров для управления Cabal-проектами и создания новых приложений и модулей. EclipseFP оказывается привычным инструментом для тех разработчиков, которые пришли из других сред программирования, например из Java или .NET, где использование интегрированных сред разработки является нормой. Будучи интегрированной в Eclipse, среда EclipseFP позволяет извлечь выгоду из широкого диапазона доступных подключаемых модулей, в числе которых имеются модули, поддерживающие Subversion и Git. В ходе изучения материала книги освоение командной строки и освойте EclipseFP для управления Haskell-проектами будут идти параллельными курсами.

ПРИМЕЧАНИЕ

Почти все текстовые редакторы, предназначенные для разработчиков, имеют режимы или подключаемые модули (плагины) для работы с Haskell. Весьма эффективные режимы, такие как автозавершение, имеются в Emacs и в Vim. В других редакторах, таких как Sublime Text, Notepad++, Kate или Gedit, их разработчики ограничились главным образом подсветкой синтаксиса.

Чтобы среда EclipseFP заработала, нужно сначала получить Eclipse (EclipseFP работает в любой версии Eclipse начиная с 3.7). Перейдите по адресу http://eclipse.org/downloads. Там вы увидите несколько вариантов, связанных с языковой поддержкой, предлагаемой по умолчанию для каждого пакета. В данном случае выбор не имеет значения, поскольку поддержку Haskell придется добавлять вручную. После загрузки нужно распаковать и запустить двоичный файл Eclipse. При работе под управлением Mac OS X может понадобиться переместить всю папку в Applications. Если после извлечения и установки среда окажется неработоспособной, следует проверить, есть ли на вашей системе виртуальная Java-машина версии 6 или выше.

В Eclipse используется понятие рабочих сред (workspaces). *Рабочая среда* — это множество родственных проектов, работа с которыми проводится одновременно и которые совместно используют активные плагины или имеют одинаковые варианты расположения окон. Когда IDE-среда открывается в первый раз, появляется вопрос о том, где создавать рабочую среду. Для проработки примеров, приводимых в данной книге, мы рекомендуем создать новую рабочую среду.

Следующим шагом будет получение плагина EclipseFP. Для этого нужно раскрыть подменю Help (Помощь) в меню Eclipse и выбрать пункт Install New Software (Уста-

новить новую программу). В поле Work with (Работать с) нужно ввести путь http://eclipsefp.sf.net/updates и нажать Enter. Спустя несколько секунд должен появиться перечень, содержащий элемент Functional programming (Функциональное программирование). Выберите пункт FP: Haskell Support for Eclipse (FP: Поддержка Haskell для Eclipse), как показано на рис. 1.2, и несколько раз нажмите Next (Далее). Eclipse приступит к загрузке этих новых функций и к их интеграции в среду.

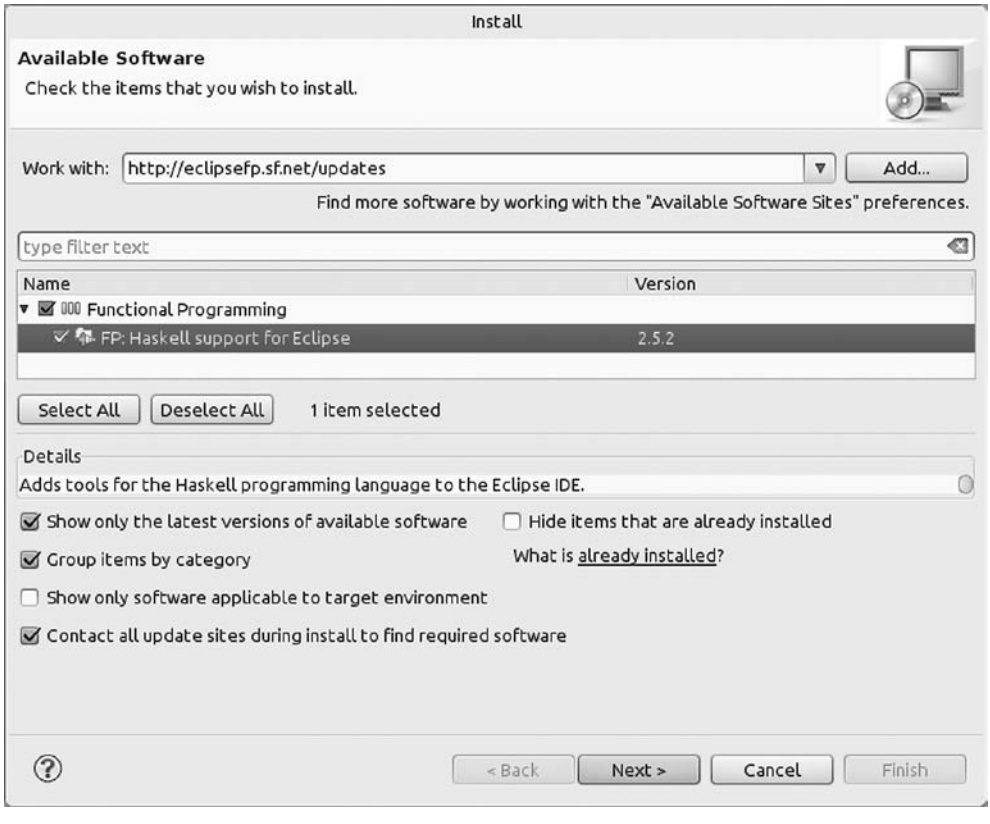


Рис. 1.2. Окно установки плагина Eclipse

После завершения установки будет выдан запрос о перезапуске Eclipse. Чтобы согласиться с перезапуском, нажмите OK. После повторного запуска IDE EclipseFP выдаст запрос на установку вспомогательных исполняемых файлов, и на экране появится диалоговое окно (рис. 1.3). Ядру редактирования Haskell эти программы не понадобятся, но зато они существенно повысят эффективность работы плагина. Установите флажки, указывающие на необходимость их установки. В зависимости от используемой вами машины этот процесс может занять много времени, поскольку должны быть загружены и откомпилированы несколько библиотек и программ. За ходом процесса можно следить на вкладке Console (Консоль).

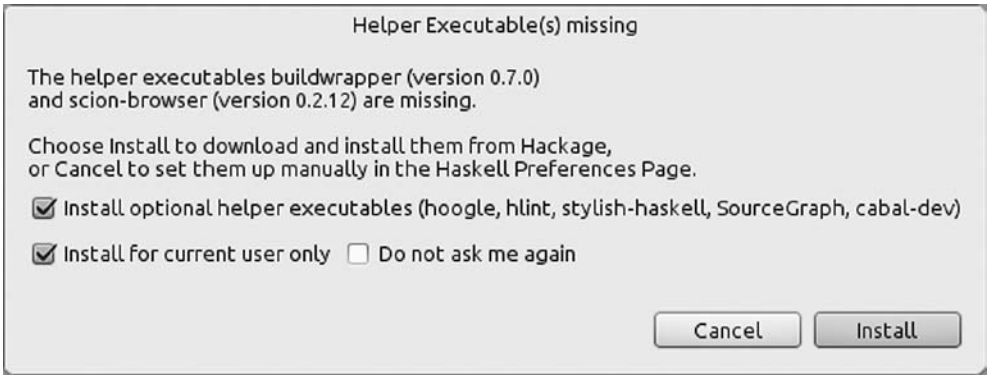


Рис. 1.3. Вспомогательные исполняемые программы для EclipseFP

Заключительным шагом станет вход в режим Haskell. В правом верхнем углу есть небольшая кнопка с надписью Open Perspective (Открыть проекцию). После щелчка на этой кнопке появится перечень поддерживаемых в Eclipse сред. Щелкните на пункте Haskell. При первом запуске EclipseFP вы увидите приглашение на загрузку из Интернета информации для автозавершения. Настоятельно рекомендуется принять приглашение.

Первые шаги в использовании GHCi

Чтобы проверить корректность установки Haskell, запустите приложение GHCi. Оно является одним из экземпляров простой интерактивной среды, работающей по принципу REPL (Read Eval Print Loop: цикл чтение — вычисление — печать), или, проще говоря, интерпретатора. При использовании GHCi вы вводите выражение и нажимаете Enter. Выражение вычисляется, а результат появляется на экране. Это позволяет применить такой метод программирования, когда вы переходите к библиотечным функциям и разбираетесь в особенностях их работы, выдавая команды в интерпретаторе, а также тестируете код в интерактивном режиме.

Чтобы открыть интерпретатор на консоли, просто выполните команду `ghci`. При этом в начале строки должно появиться приглашение командной строки вида `Prelude>`. Это приглашение сообщает вам о готовности к приему команд и о том, что в данный момент загружен только модуль `Prelude`, содержащий самые основные функции и типы данных, используемые в Haskell. В первом приближении GHCi может работать в качестве весьма необычного калькулятора:

```
Prelude> 5 * 3
15
Prelude> 1/2 + 1/3
0.8333333333333333
```

Если сейчас ввести букву `s` и нажать Tab, появится список всех доступных функций, начинающихся на эту букву. Если затем ввести букву `q` и опять нажать Tab, останется только один вариант — вызов функции `sqrt`, чье имя автоматически вводится

в командную строку. Одной из характерных особенностей Haskell является отказ от круглых скобок при вызове функций. То есть чтобы извлечь корень квадратный из 7, нужно просто написать:

```
Prelude> sqrt 7
2.6457513110645907
```

В интерпретаторе можно выполнять и множество других арифметических операций, включая `sin`, `cos`, `log`, `exp` и т. д. В следующей главе мы поговорим об использовании строк и списков, а также научимся определять функции, что существенно обогатит наш опыт работы с интерпретатором.

По умолчанию GHCi не позволяет интерпретировать сразу несколько строк кода. Например, не так-то просто разбить операцию сложения двух рациональных чисел на две строки. Попробуйте ввести выражение, нажав клавишу Enter после ввода знака «плюс». В результате появится сообщение об ошибке:

```
Prelude> 1/2 +
<interactive>:2:6:
    parse error (possibly incorrect indentation or mismatched brackets)
```

Решением проблемы будет открытие *многострочного блока*. Этот блок является выражением, которому разрешено занимать несколько строк. Для открытия такого блока нужно ввести символы `:{` и нажать Enter. Приглашение командной строки изменится на `Prelude|`, показывая тем самым, что ожидается заполнение нескольких строк. Чтобы закрыть блок, нужно ввести сочетание символов, противоположное символам открытия блока, то есть `:}`. Пример:

```
Prelude> :{
Prelude| 1/2 +
Prelude| 1/3
Prelude| :}
0.8333333333333333
```

ВНИМАНИЕ

Чтобы открыть многострочный блок, пара символов :{ должна быть единственным текстом, введенным в строке.

Все внутренние действия интерпретатора (то есть те, которые не являются функциями какой-нибудь библиотеки) начинаются с двоеточия. Например, ввод символов `:?` и нажатие Enter приведет к выводу списка всех доступных команд. Можно также посмотреть на используемую версию стандарта языка: в данном случае это Haskell 2010 с рядом модификаций. Например:

```
Prelude> :show language
base language is: Haskell2010
with the following modifiers:
  -XNoDatatypeContexts
  -XNondecreasingIndentation
```

Ранее утверждалось, что Haskell является системой со строгими статическими типами. Можно удостовериться, что запрещена операция деления двух строк (заключенных в двойные кавычки) и при ее вводе в интерпретатор выдается ошибка:

```
Prelude> "hello" / "world"
<interactive>:2:9:
    No instance for (Fractional [Char]) arising from a use of '/'
    Possible fix: add an instance declaration for (Fractional [Char])
    In the expression: "hello" / "world"
    In an equation for 'it': it = "hello" / "world"
```

Здесь `Fractional` — имя класса типов, предоставляющего поддержку оператору `/`. В сообщении об ошибке говорится, что для получения возможности деления двух строк друг на друга нужно сообщить компилятору, как это должно происходить, добавив объявления с кодом для класса типов `Fractional`.

Чтобы закрыть интерпретатор и вернуться в консоль, можно ввести команду `:quit` или просто нажать Ctrl+D. В обоих случаях это приведет к одинаковому результату:

```
Prelude> :quit
Leaving GHCi.
```

ПРИМЕЧАНИЕ

GHCi является довольно эффективным и гибким (в отношении возможности настройки) инструментом. Массу советов и описаний приемов работы с этим интерпретатором вы можете найти на посвященной ему вики-странице Haskell по адресу http://www.haskell.org/haskellwiki/GHC/GHCi.

Магазин, продающий машины времени

Взглянув на оглавление книги, можно заметить, что в ней есть пять частей, причем первые четыре посвящены отдельным модулям небольшого интернет-магазина.

- В части I рассматриваются определения основных блоков нашего приложения. С помощью этих блоков будут представлены клиенты, товары и заказы, а также определен порядок управления ими в памяти компьютера.
- В части II рассматривается разработка ряда алгоритмов извлечения данных с целью получения более широкого представления о клиентах. В частности, мы рассмотрим разработку алгоритма классификации на основе K-средних (K-means), а также рекомендательного алгоритма.
- В части III затрагиваются вопросы хранения данных в постоянном хранилище: для данных о товаре будет использован обычный файловый формат, а для клиентов и заказов — более традиционное решение в виде базы данных. При наличии таких возможностей мы сможем создать в главе 12 первое приложение.
- В части IV показано, как предметно-ориентированный язык может быть использован для моделирования специальных предложений, которые делаются покупателям. Примером может послужить акция: 20 % скидки для всех европейских покупателей в возрасте до 30 лет. Как вы увидите, строгая система типов в Haskell может работать на вас.
- И наконец, в части V разговор идет о принципах разработки программного обеспечения, включая создание документации и блочное тестирование. В среде Haskell имеются замечательные средства, помогающие создавать документацию и проводить качественное тестирование кода.

А что же мы продаем в нашем магазине? Мы продаем машины времени:

> Добро пожаловать в захватывающий мир машин времени! Эти устройства столь необычны, что к нам слетаются клиенты со всей Вселенной. Чтобы обслужить заказы, мы решили создать интернет-магазин и сделать это на языке, который был бы под стать нашим уникальным машинам, то есть на языке Haskell.

Впечатляюще? Язык Haslkell будет использоваться для создания магазина по продаже машин времени. Это весьма забавный пример, который, как мы надеемся, не оставит вас равнодушными.

Выводы

В этой главе мы получили первое представление о языке Haskell:

- Мы узнали о характерных отличиях чистого функционального программирования и о том, как эти отличия помогают создавать более краткий, удобный в сопровождении и менее подверженный ошибкам код.
- Мы увидели преимущества использования строгой, проверяемой в статическом режиме системы типов, воплощенной в Haskell, и узнали, как зависимая типизация обеспечивает очень эффективное выражение инвариантов.
- Мы познакомились с основным инструментарием экосистемы Haskell: компилятором GHC, инструментом компоновки Cabal, хранилищем библиотек Hackage и интерпретатором GHC. Мы также сделали свои первые шаги в использовании последнего из упомянутых инструментов.
- Мы понаблюдали за процессом установки Haskell Platform в наиболее распространенных компьютерных средах, а также за процессом установки плагина Haskell для интегрированной среды разработки Eclipse.
- Мы выяснили основные цели, поставленные в данной книге (не считая изучения языка Haskell): создание интернет-магазина по продаже машин времени с модулями для простого интеллектуального анализа данных и описания предложений.

Объявление модели данных

2

Вы уже знаете, как установить Haskell Platform и среду разработки EclipseFP. Следующим шагом в направлении создания нашего магазина машин времени станет подготовка начального множества значений и функций, представляющих имеющиеся в системе данные: сведения о клиентах, машинах и заказах.

В данной главе мы подготовим все, что для этого нужно. В первом приближении мы создадим функции, работающие с основными типами данных. Числа вам уже знакомы, а теперь мы добавим списки (lists) и кортежи (tuples). Потом вы увидите, как с целью наилучшего представления значений интересующего нас вида создавать собственные *алгебраические типы данных* (Algebraic Data Types, ADT). В ходе обучения для написания лаконичного кода, повторяющего формы типов, впервые будет использована мощная языковая идиома *сопоставления с образцом* (pattern matching).

Иногда при использовании ADT-типов и идиомы сопоставления с образцом получается труднопонимаемый код. Упростить создание и изменение значений можно с помощью *записей* (records), которые вводят ряд способствующих этому синтаксических форм и широко известны Haskell-программистам. Кроме того, будут рассмотрены два часто встречающихся в Haskell-библиотеках образца проектирования: *интеллектуальные конструкторы* (smart constructors) и *значения по умолчанию* (default values).

Эта глава послужит также введением в управление проектами с помощью Cabal и EclipseFP. В частности, будет показано, как с помощью обеих систем создается новый проект, структура которого обычно отражается в папках, и как с целью интерактивного тестирования код загружается в интерпретатор GHC.

Работа с символами, числами и списками

Символы и числа повсеместно воспринимаются как основные разновидности величин, доступные программисту. Следуя этой традиции, Haskell предлагает символьные и числовые типы данных, с которыми мы познакомимся в этом разделе. Затем мы узнаем, как объединять несколько таких значений для создания строк или списков чисел, а также познакомимся с основными операциями над списками любого типа.

Символы

В некоторых языках программирования числа используются еще и для представления символов в какой-нибудь кодировке, например в ASCII или Unicode.

Однако следуя своей традиции четкого разделения различных вариантов значений, Haskell для представления символьных данных имеет специальный тип данных под названием `Char`. Во избежание проблем с локализацией и языками значения `Char` содержат один символ в кодировке Unicode. Эти значения могут создаваться двумя различными способами:

- записью самого символа в одинарных кавычках, например: `'a'`;
- записью указания на код, который определен стандартом Unicode в виде десятичного числа, заключенного между одинарной кавычкой с обратным слешем `'\` и одинарной кавычкой `'`, или в виде шестнадцатеричного числа между символами `'\x` и `'`. Например, символ `'a'` может быть записан как `'\97'` или `'\x61'`.

Воспользовавшись программой GHCi, можно проверить фактический тип каждого выражения, представляемого в системе. Для этого служит команда `:t`, за которой нужно набрать само выражение. Давайте проверим, что символы действительно являются символами:

```
Prelude Data.Char> :t 'a'
'a' :: Char
```

А теперь займемся исследованием некоторых функциональных возможностей, предоставляемых языком Haskell для значений типа `Char`. По умолчанию загружается всего лишь несколько функций, поэтому мы импортируем модуль, содержащий множество дополнительных функций, в данном случае модуль `Data.Char`:

```
Prelude> import Data.Char
Prelude Data.Char>
```

Выводимое интерпретатором на экран приглашение на ввод изменяется, чтобы обозначить факт загрузки еще двух модулей. Более того, если теперь набрать команду `to` и нажать Tab, мы увидим намного больше функций, чем раньше. В Haskell все имеет собственный тип, поэтому давайте попробуем определить тип функции `toUpper`:

```
Prelude Data.Char> :t toUpper
toUpper :: Char -> Char
```

Синтаксис, где два значения разделены стрелкой (`->`), используется для указания типов функций. В данном случае `toUpper` является функцией, которой передается символьное значение (`Char` слева от стрелки), а она возвращает еще одно символьное значение (`Char` справа от стрелки). Разумеется, типы могут и не совпадать. Например, функции `chr` передается целочисленное значение, а она возвращает символ, чей код указан в переданном ей значении:

```
Prelude Data.Char> chr 97
'a'
Prelude Data.Char> :t chr
chr :: Int -> Char
```

Для функций, имеющих более одного аргумента, каждый тип аргумента отделяется от следующего одной стрелкой. Если взять функцию `min`, то ей передаются два целочисленных значения, а она возвращает наименьшее из них, и тип будет иметь следующий вид:

```
min :: Integer -> Integer -> Integer
```

В предыдущей главе упоминалось, что Haskell очень строго относится к проверке типов. И вы можете в этом убедиться: при попытке применения функции `chr` в отношении символьного значения интерпретатор откажется продолжать работу:

```
Prelude Data.Char> chr 'a'
<interactive>:7:5:
    Couldn't match expected type 'Int' with actual type 'Char'
    In the first argument of 'chr', namely 'a'
    In the expression: chr 'a'
    In an equation for 'it': it = chr 'a'
```

Числа

В предыдущей главе мы встречали несколько видов числовых констант. Как и большинство языков программирования, Haskell поддерживает большое количество разнообразных типов, различающихся шириной, точностью и поддержкой дробной части:

- `Int` — ограниченный целочисленный тип. Он как минимум поддерживает значения в диапазоне ±536 870 911 (хотя в GHC используется намного более широкий диапазон). Значения типа `Int` имеют, как правило, исходную ширину, присущую используемой архитектуре, что делает работу с ними наиболее быстрой среди значений всех остальных типов.
- `Integer` — неограниченный целочисленный тип: с его помощью может быть представлено любое значение без дробной части, без переполнения или потери значимости. Это свойство придает данному типу особую ценность при написании кода без оглядки на границы, но при этом приходится жертвовать скоростью выполнения кода.
- `Ratio` — тип, используемый в базовой Haskell-библиотеке для точных рациональных чисел. Рациональные значения создаются с помощью выражения `n % m`.
- `Float` и `Double` — типы чисел с плавающей точкой, соответственно с одинарной и двойной точностью.

В Haskell весьма строгое отношение к типам. Для конвертации значений между различными числовыми представлениями используются функции `fromInteger`, `toInteger`, `fromRational` и `toRational`.

Между представлениями значений рациональных чисел и чисел с плавающей точкой можно переключаться: функция `toRational` пытается создать значение с типом `Ratio`, максимально приближенное к исходному значению (это зависит от его ширины), а перейти от рационального значения к значению с плавающей точкой можно, поделив числитель на знаменатель. Имейте в виду, что эти функции находятся в модуле `Data.Ratio`, поэтому сначала нужно импортировать этот модуль:

```
Prelude> import Data.Ratio
Prelude Data.Ratio> 1 % 2 + 1 % 3
5 % 6
Prelude Data.Ratio> toRational 1.3
5854679515581645 % 4503599627370496
Prelude Data.Ratio> toRational (fromRational (13 % 10))
5854679515581645 % 4503599627370496
```

Как видно из примеров, идеальные круговые переходы между рациональными значениями и значениями с плавающей точкой удаются не всегда.

При попытке определения типа числовой константы может быть получен довольно странный результат:

```
Prelude> :t 5
5 :: Num a => a
Prelude> :t 3.4
3.4 :: Fractional a => a
```

Вместо создания числовой константы определенного типа в Haskell принимается весьма разумное решение для поддержки констант различного типа: они становятся *полиморфными* (polymorphic). Например, `5` является константой, которая может использоваться для создания значений любого типа, поддерживаемого в типовом классе `Num` (который включает все ранее представленные типы). В то же время `3.4` может использоваться для создания значений любого типа, относящегося к классу `Fractional` (который включает типы `Float` и `Double`, но не включает типы `Int` или `Integer`). Более подробно классы типов рассматриваются в главе 4, а сейчас класс типов можно представить себе как некий механизм группировки множества типов, поддерживающих одни и те же операции.

ВНИМАНИЕ

Поскольку применительно к функциям круглые скобки в Haskell не используются, при записи отрицательных чисел нужно проявлять особую осмотрительность. Например, если в GHCi-строке написать atan -4, будет получено сообщение об ошибке отсутствия экземпляра, вызванное применением знака «минус»: No instance for (Num (a0 -> a0)) arising from a use of '-'. Это означает, что интерпретатор счел это попыткой вычислить разницу между atan и 4. Для получения значений арктангенса -4 нужно было использовать выражение atan (-4).

Строки

Поработав немного с символами, можно задаться вопросом: а нельзя ли их собрать в кучу, получив тем самым всем известную *строку*? Синтаксис, используемый для строк в Haskell, похож на синтаксис в C: буквы заключаются в двойные кавычки. Строка создается с помощью следующего кода:

```
Prelude Data.Char> :t "Hello world!"
"Hello world!" :: [Char]
```

А что же тогда интерпретатор выводит в ответ на запрос о получающемся при этом типе данных? Вместо какого-нибудь нового типа вроде `String` мы видим нашего старого знакомого `Char`, но уже заключенного в квадратные скобки. Эти скобки здесь призваны показать, что `"Hello world!"` — это не символ, а *список* символов. В общем, при записи типа `t` в форме `[t]` создается ссылка на тип всех списков, которые состоят исключительно из элементов типа `t`. В функциональном программировании списки являются наиболее часто используемыми структурами данных. Тот факт, что списочный тип зависит от других типов, трактуется как *параметрический полиморфизм*. Соответствующие подробности рассматриваются в следующей главе, а сейчас давайте сосредоточимся на практической стороне дела.

Списки

Каждый из списков является списком *литералов*, то есть списком, чьи значения явно заданы в программном коде, записаны через запятые и заключены в квадратные скобки. Ранее уже было показано, что для списка символов есть специальный строковый синтаксис. Давайте посмотрим на типы некоторых из этих литералов, а также на функцию `reverse`, которая возвращает список, где элементы выстроены в обратном порядке, и на функцию `(++)`, объединяющую два списка.

```
Prelude> :t [1,2,3]
[1, 2, 3] :: Num t => [t]
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> reverse [1,2,3]
[3,2,1]
Prelude> reverse "abc"
"cba"
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

В этом примере следует обратить внимание на то, что существуют такие функции, как `reverse` и `(++)`, которые могут работать со списками любого вида. Это означает, что, зная о них, вы можете применить свои знания к любому списку (включая строки символов). В подтверждение данного факта эти функции показывают в своем типе *переменную типа* (type variable). Это действительно переменная, поскольку она может быть заменена любым типом точно так же, как обычные переменные могут принимать различные значения. Переменные типа должны записываться в коде со строчной буквы, и для них обычно используется одна или две буквы. Здесь переменная типа показана как буква `a`.

ПРИМЕЧАНИЕ

Функции, чье имя образовано целиком из символов наподобие ++, должны вызываться с использованием инфиксного синтаксиса: их следует ставить не впереди аргументов, а между ними. То есть нужно использовать запись a ++ b, а не ++ a b. Если же вам захочется задать функцию обычным образом, потребуется ввести ее в скобках. То есть можно использовать запись (++) a b.

Списки в Haskell являются *однородными* (homogeneous): каждый список может состоять только из элементов одного типа. Поэтому невозможно создать список, содержащий целые числа и символы, а также объединить два списка с элементами разных типов:

```
Prelude> [1,2,3,'a','b','c']
<interactive>:13:2:
    No instance for (Num Char) arising from the literal '1'
    Possible fix: add an instance declaration for (Num Char)
Prelude> "abc" ++ [1,2,3]
<interactive>:11:11:
    No instance for (Num Char) arising from the literal '1'
    Possible fix: add an instance declaration for (Num Char)
```

Операции со списками

Как и в большинстве других функциональных языков, списки в Haskell относятся к категории связанных. Такие списки состоят из ряда ячеек, хранящих значения и ссылки на следующие ячейки в списке, а также специальный маркер конца списка. Основными операциями для создания списков являются `[]` для создания пустого списка и `:` для добавления элементов в уже существующий список. Например, в результате вычисления выражения `elt:lst` получится список, в котором значение `elt` будет помещено перед списком `lst`. То есть список литералов может быть также записан следующим образом:

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
Prelude> 'a' : 'b' : 'c' : []
"abc"
```

К функциям, выдающим информацию о форме и содержимом списка, относятся функции `null`, `head` и `tail`. Функция `null` проверяет, пуст список или нет, функция `head` возвращает первый элемент списка (его голову), а функция `tail` возвращает список без первого элемента (хвост списка, известный также как остальная часть списка). Некоторые примеры применения этих функций:

```
Prelude> null [1,2,3]
False
Prelude> null []
True
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> head []
*** Exception: Prelude.head: empty list
```

На рис. 2.1 показано графическое представление вышеупомянутых списочных операторов и функций. Оператор `:` служит для привязки элемента к какой-нибудь остальной части списка, а снова разбить эти элементы на части можно с помощью функций `head` и `tail`. Также можно увидеть, что список всегда является последовательностью операций конструкторов, неизменно оканчивающейся конструктором пустого списка `[]`.

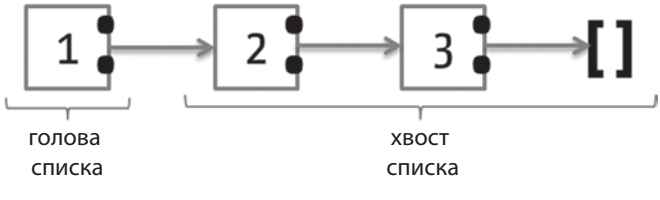


Рис. 2.1. Графическое представление конструкторов и деструкторов списка

При попытке получения головы или хвоста пустого списка мы получим вполне ожидаемую ошибку. Следует знать, что в Haskell исключения не являются предпочтительным средством обработки ошибок (более подробно причины этого изложены в сле-

дующих главах), и по умолчанию при обнаружении ошибки происходит сбой всей программы. Чтобы предотвратить возникновение ошибок при проведении операций над пустыми списками, нужно просто перед применением таких функций, как `head` и `tail`, провести проверку на заполненность списка (или воспользоваться еще одним рассматриваемым далее синтаксическим приемом — сопоставлением с образцом).

Наблюдая за выполнением функции `null`, можно было заметить два новых значения, о которых уже шла речь: `True` и `False`. Это единственные два булевых значения (типа `Bool`). Для их сочетаний в модуле `Prelude` предоставляется ряд стандартных функций: «и» (`&&`), «или» (`||`) и «не» (`not`). Большинство языков программирования, произошедших от языка C, таких как C++ и Java, унаследовали первые два вида булевых операторов. Среди булевых операторов можно найти как «долгоиграющие» операторы (`&` и `|`), всегда вычисляющие обе стороны выражения, так и «урезанные» (`&&` и `||`), способные остановить вычисления и ограничиться только одной стороной выражения. В Haskell в соответствии с моделью отложенного (ленивого) вычисления эти операторы всегда работают в урезанном варианте. Кроме того, существуют функции `and` и `or`, которым передается список булевых значений для выполнения операций:

```
Prelude> (True && False) || (False && not False)
False
Prelude> or [True, False, and [False, True, True]]
True
Prelude> (2 == 2.1) || (2 < 2.1) || (2 > 2.1)
True
```

ВНИМАНИЕ

Стоит напомнить про сравнение значений с плавающей точкой. Компьютеры не в состоянии представить все значения с абсолютно одинаковой точностью, поэтому ожидаемого равенства можно и не получить. Например, в моей системе выражение (4.00000000000000003 – 4) == 0 дает True.

Наряду с этими функциями есть еще одна важная конструкция, имеющая отношение к булевым значениям, — это `if-then-else`. Выражение вида `if b then t else f` дает `t`, если значением `b` является `True`, и `f` в противном случае. Эта структура очень похожа на ту, которая встречается в императивных языках, но в ней имеются существенные отличия:

- `t` и `f` сами по себе должны быть выражениями в соответствии с положениями, принятыми в языке, поскольку у всего выражения `if` в конце должно быть значение. Запись вида `if True then 1 else 2` является в Haskell общепринятым образцом.
- Вместе с `if` должны присутствовать обе ветви, как `then`, так и `else`. В противном случае для части значений `b` выражение вычислено не будет. В некоторых языках принято возвращать для несуществующей ветви `else` значение по умолчанию, но в Haskell такое не практикуется.
- У всего выражения должен быть вполне определенный тип. Haskell справляется с этим, принудительным порядком относя выражения `t` и `f` к одному и тому же типу. Поэтому выражение вида `if True then 1 else "hello"` не будет принято ни компилятором, ни интерпретатором.

Чтобы реально использовать выражения `if`, нужны функции, возвращающие тип `Bool`. Среди них большое количество функций сравнения чисел: `==` (равенство),

`/=` (неравенство — обратите внимание, что синтаксис этой функции отличается от используемого в языках C и Java, где она имеет вид `!=`), `>=` (больше или равно), `>` (больше чем), `<=` (меньше или равно) и `<` (меньше чем). В качестве примера выражения `if` можно привести следующий код:

```
Prelude> if 3 < 4.5 then "3 is less than 4.5" else "3 is not less than 4.5"
"3 is less than 4.5"
```

Давайте заставим интерпретатор вернуть голову списка строк, если этот список не пуст, или строку `"empty"`, если он пуст:

```
Prelude> if not (null ["hello","h"]) then (head ["hello","h"]) else "empty"
"hello"
Prelude> if not (null []) then (head []) else "empty"
"empty"
```

В качестве элементов списки могут содержать другие списки (с любым уровнем вложенности). Таким образом, если `[τ]` — это списки типа `τ`, то списки списков обозначаются как `[[τ]]`. Внутренние списки, находящиеся во внешних списках, не обязаны иметь одинаковую длину (следовательно, они не являются эквивалентами многомерных массивов). Но нужно помнить, что пустой список может быть элементом более крупного списка списков, следовательно, список `[]` не является эквивалентом списка `[[]]`: первый из них является абсолютно пустым списком списков, а вот второй является списком, содержащим всего один элемент, который представляет собой пустой список.

```
Prelude> :t [['a','b','c'],['d','e']]
[['a','b','c'],['d','e']] :: [[Char]]
Prelude> head [['a','b','c'],['d','e']]
"abc"
Prelude> head (head [['a','b','c'],['d','e']])
'a'
Prelude> head [[]]
[]
```

Вам, наверное, надоело многократно набирать в интерпретаторе один и тот же список констант. Чтобы избавиться от этого рутинного занятия, мы собираемся рассмотреть основные варианты многократного использования функциональных возможностей, присущие всем языкам программирования: определение функций, работающих с разными входными значениями, и создание временных привязок. Но перед этим нужно решить ряд включенных в упражнение 2.1 задач и убедиться, что все встречавшиеся понятия вами усвоены. А в следующем разделе мы рассмотрим инструментальное средство Cabal, предназначенное для создания Haskell-проектов и управления ими.

УПРАЖНЕНИЕ 2.1. СПИСКИ СПИСКОВ

Нами уже рассмотрен довольно большой объем материала об основных типах и выражениях в Haskell. Следующие задачи дают возможность применения на практике знаний, полученных в данном разделе. Во всех случаях решениями должны стать выражения, которые можно набрать в командной строке интерпретатора, проверив тем самым их работоспособность.

1. Перепишите предыдущий список литералов, используя только лишь оператор : и конструктор пустого списка [].

2. Напишите выражение, проверяющее список [] на пустоту или же проверяющий на пустоту первый элемент такого списка, как [[],['a','b']].
3. Напишите выражение, проверяющее наличие у списка только одного элемента. Оно должно возвращать True для списка ['a'] и False для списка [] или списка ['a','b'].
4. Напишите выражение, объединяющее два списка, заданные внутри других списков. Например, для списка [«abc», «de»] оно должно возвращать «abcde».

Создание нового проекта

Новый проект создается с использованием *Cabal* — системы компоновки и создания Haskell-проектов. Можно использовать и другие средства, но Cabal была создана с учетом особенностей среды Haskell и хранилища пакетов Hackage. Кроме того, в файле описания Cabal находятся интересные метаданные проекта, такие как имя проекта, название организации, занимающейся его поддержкой, сведения о лицензии. В данном разделе пойдет речь о том, как использовать Cabal не только из командной строки, но и из среды EclipseFP. Можно выбрать любой из этих способов или даже чередовать их, поскольку структуры создаваемых в результате проектов вполне совместимы.

Создание проекта из командной строки

Чтобы разработать проект с помощью командной строки, сначала следует создать папку для размещения файлов, имя которой обычно совпадает с именем пакета. Затем, пользуясь оболочкой командной строки, нужно войти в эту папку (обычно это делается выдачей серии команд `cd`) и запустить команду `cabal init`. После этого придется ответить на ряд вопросов:

```
$ cd path/to/my/haskell/projects
$ mkdir chapter2
$ cd chapter2
$ cabal init
Package name? [default: chapter2]
Package version? [default: 0.1.0.0] 0.0.1
Please choose a license:
 * 1) (none)
   ...
Your choice? [default: (none)]
Author name? Alejandro Serrano
Maintainer email? my@email-address.com
Project homepage URL? http://my-web-page.com
Project synopsis? Project example for Chapter 2
Project category:
 * 1) (none)
   ...
Your choice? [default: (none)]
What does the package build:
   1) Library
   2) Executable
Your choice? 1
Include documentation on what each field means (y/n)? [default: n]
```

Наиболее важными ответами на вопросы являются имя пакета (`Package name`) и создаваемый объект: библиотека (`Library`) или исполняемый файл (`Executable`), поскольку эти ответы влияют на имя и структуру файла проекта. Основным отличием проекта библиотеки от проекта исполняемого файла является создание финальной программы (во втором случае) или кода, который просто будет использоваться другими библиотеками или исполняемыми файлами (в первом случае). В данный момент наш выбор ни на что не повлияет, поскольку мы будем тестировать код с помощью интерпретатора GHC. Более того, проект может быть впоследствии уточнен и к нему могут быть добавлены дополнительные описания библиотек или исполняемых файлов.

Создание проекта из среды EclipseFP

Для запуска нового проекта внутри EclipseFP нужно в меню File (Файл) раскрыть подменю New (Новый) и щелкнуть на пункте Haskell Project (Проект Haskell). После этого появится экран, показанный на рис. 2.2, в котором вам потребуется ввести упомянутую ранее важную информацию. Также вам будет предоставлена возможность выбора места, где должен находиться ваш проект.

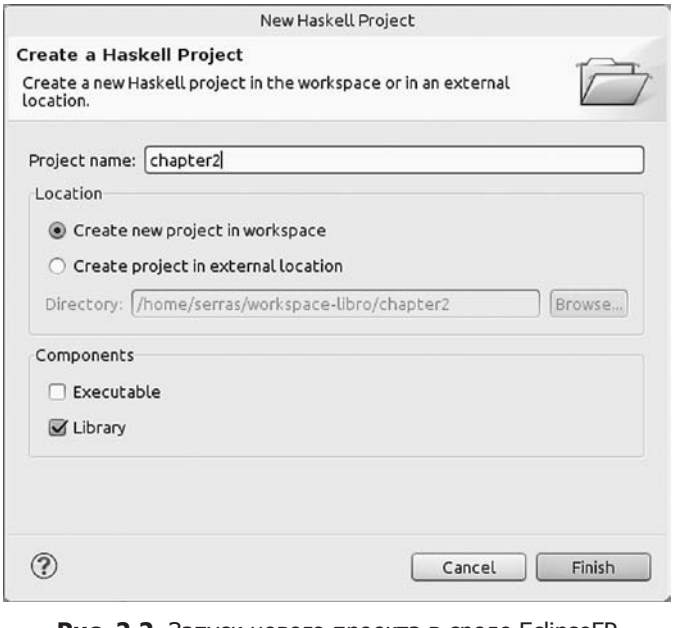


Рис. 2.2. Запуск нового проекта в среде EclipseFP

УПРАЖНЕНИЕ 2.2. ВАШ ПЕРВЫЙ ПРОЕКТ

Создайте новый библиотечный проект chapter2, используя любой из описанных ранее способов.

При выполнении упражнения 2.2 в папке создаются два файла: Setup.hs и package_name.cabal. Setup.hs нам пока не понадобится, поэтому вернемся к нему чуть позже.

А теперь откроем только что созданный файл с расширением .cabal. Если работа ведется в графической среде редактирования EclipseFP, то перед вами откроется картина, показанная на рис. 2.3.

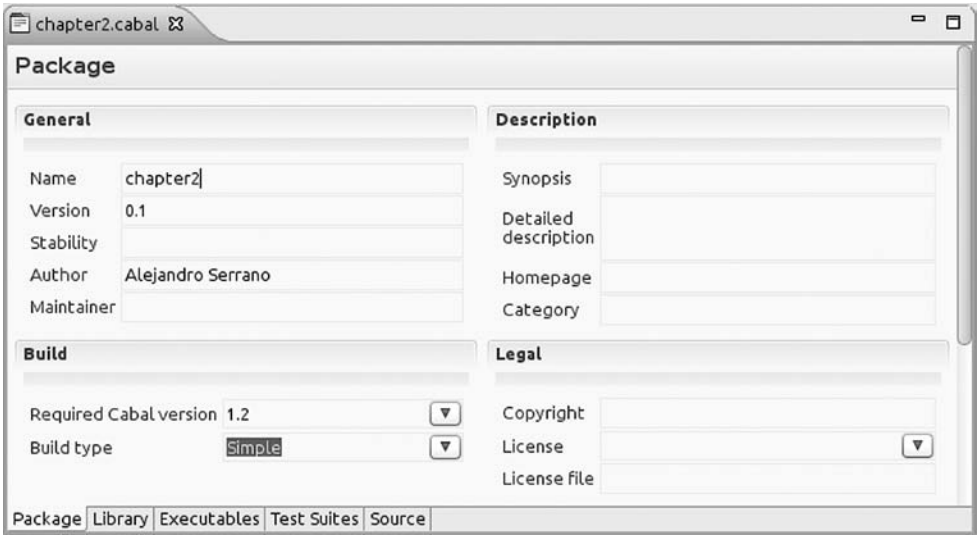


Рис. 2.3. Графическое средство редактирования Cabal в среде EclipseFP

В исходном виде Cabal-проект состоит из ряда свойств пакета, за которыми следует несколько блоков кода, которые в терминологии Cabal носят название *строф* (stanzas). В них определяются создаваемые компоненты (библиотеки и исполняемые файлы), их исходные файлы и параметры компиляции (такие, как флаги или включаемые расширения). Если вы знакомы с форматом JSON или Python-кодом, то распознать синтаксис, используемый в Cabal, будет очень легко — нужно только понять смыл, заложенный в коде. При интерпретации кода нужно придерживаться двух важных правил:

- Каждому свойству задается значение в форме `name: value` (имя: значение). Имя не чувствительно к регистру букв (неважно, как оно будет написано — `name`, `Name` или `nAmE`), а значение записывается без каких-либо кавычек или пометок. Если в качестве значения будет фигурировать список, разделителями элементов должны быть запятые.
- Строфы начинаются с заголовка, обычно это `library` или `executable`, за которым следует имя приложения. Следует учесть, что двоеточие (`:`) после заголовка не ставится. Все составляющие строфу свойства должны иметь отступ с равным количеством пробелов или знаков табуляции.

Например, Cabal-файл, созданный средой EclipseFP для упражнения 2.2, будет иметь следующий вид:

```
name:           chapter2
version:        0.1
cabal-version:  >=1.2
```

```
build-type:     Simple
author:         Alejandro Serrano

library
  hs-source-dirs:  src
  build-depends:   base >= 4
  ghc-options:     -Wall
```

Понятие модуля

Haskell-проекты создаются путем написания *модулей* (modules). В каждом модуле содержится множество таких определений, как функции и типы данных, которые он группирует под общим «колпаком». Имена модулей вкладываются друг в друга в иерархическом порядке: например, внутри модуля `Data` имеется множество различных модулей, таких как `Data.Bool`, `Data.Ratio` и т. д. Эта вложенность делает модули очень похожими на пакеты в Java или на пространства имен в C#.

Каждый модуль определяется в собственном файле. Имя файла должно совпадать с именем последнего компонента в имени модуля (с той частью, которая следует за последней точкой), а сам файл должен быть вложен в папки, названные по именам всех остальных компонентов. Например, путь к модулю `Chapter2.Section2.Example` должен выглядеть так: Chapter2/Section2/Example.hs. То есть на вершине проекта нужно создать папку с именем Chapter2. Внутри этой папки должна быть еще одна папка с именем Section2. И наконец, внутри папки Section2 нужно создать файл Example.hs.

Файл модуля всегда нужно начинать с объявления модуля, задавая ему имя. Например, только что упомянутый модуль в файле Example.hs должен начинаться следующей строкой:

```
module Chapter2.Section2.Example where
```

Затем можно приступать к написанию определений для этого модуля.

Чтобы заставить Cabal откомпилировать файл модуля, нужно включить модуль в какую-нибудь строфу. Для этого под строфой указывается новое свойство, а модуль добавляется либо к свойству `exposed-modules`, либо к свойству `other-modules`. Разница в том, что при использовании вашей библиотеки в другом проекте вам будут доступны только экспонированные (exposed) модули, другие модули останутся внутренними. Например:

```
library
  exposed-modules: Chapter2.Section2.Example
  -- или
  other-modules:   Chapter2.Section2.Example
```

Если используется командная строка, то проект можно откомпилировать, выполнив сначала команду `cabal configure`, а затем команду `cabal build`. На данный момент столкнуться с какими-либо ошибками компиляции вы не должны.

Таким образом, чтобы добавить новый модуль к проекту, нужно выполнить следующие действия:

1. Выбрать имя для модуля, например `A.B.C`.
2. Создать папку для каждого компонента его имени, кроме последнего, в данном случае нужно создать папку A, а внутри нее — папку B.

3. Создать файл, имя которого совпадает с именем последнего компонента, с расширением .hs (в данном случае нужно создать файл C.hs) и записать в него ранее рассмотренное объявление модуля.
4. Заставить Cabal включить этот файл в ваш проект.

Поскольку содержание всех файлов в корневой папке проекта затрудняет управление этими файлами, обычно создается папка для хранения всех исходных файлов проекта, как это делается в других программах создания проектов, например в Maven для Java. Я настоятельно рекомендую помещать ваши файлы в папку src. Но если файлы перемещаются из корневой папки проекта, нужно сообщить об этом Cabal, добавив в строфу новое свойство:

```
library
  hs-source-dirs:  src
```

Если используется среда EclipseFP, управление модулями существенно упрощается. Прежде всего, структура для перемещения исходных файлов из корневой папки создается по умолчанию. Кроме того, можно щелкнуть правой кнопкой мыши на папке src и выбрать в контекстном меню пункт New Haskell Module (Новый модуль Haskell). В текстовом поле Folder (Папка) нужно указать исходные компоненты вашего модуля (в данном случае `Chapter2.Section2`), а в текстовом поле Name (Имя) — последний компонент (в данном случае `Example`). В следующем окне можно выбрать, в какую строфу должен быть включен новый модуль. После щелчка на кнопке Finish (Завершить) в нужном месте создается файл, который попадет в перечень, имеющийся в Cabal-файле.

ПРИМЕЧАНИЕ

С этого момента вам потребуется создавать новый проект при работе над каждой главой этой книги. Также для каждого раздела нужно создавать новый модуль или несколько модулей. Это соглашение придаст вашей работе более организованный характер.

Определение простых функций

Теперь можно приступить к созданию *функций* в файле модуля. В объявление функции включаются:

- *имя* функции — в Haskell оно всегда начинается с буквы *в нижнем регистре*;
- список аргументов — каждый аргумент также должен начинаться с буквы в нижнем регистре и отделяться от других аргументов пробелами (а не запятыми, как в большинстве других языков), кроме того, список не должен заключаться в круглые скобки;
- знак = и само тело функции.

Создание простой функции

Давайте обратимся к последней функции, созданной ранее в разделе «Операции со списками». Получая список из строк, эта функция возвращала либо первую строку списка, либо строку `"empty"`, если в списке ничего не было. Если заменить

список констант именем параметра, то большинством выражений можно будет пользоваться многократно:

```
firstOrEmpty lst = if not (null lst) then head lst else "empty"
```

Для тестирования функции сначала нужно создать новый модуль `Chapter2.SimpleFunctions`, где эта функция будет храниться. Затем нужно загрузить файл в интерпретатор, выполнив команду `:l` с указанием полного пути к файлу. После этого функцию `firstOrEmpty` можно будет вызвать непосредственно из командной строки:

```
Prelude> :l src/Chapter2/SimpleFunctions.hs
[1 of 1] Compiling Chapter2.SimpleFunctions ( src/Chapter2/SimpleFunctions.hs,
interpreted )
Warning: Top-level binding with no type signature:
      firstOrEmpty :: [[Char]] -> [Char]
OK, modules loaded: Chapter2.SimpleFunctions.
*Chapter2.SimpleFunctions> firstOrEmpty []
"empty"
*Chapter2.SimpleFunctions> firstOrEmpty ["hello","hola"]
"hello"
```

В EclipseFP можно получить аналогичный результат — для этого достаточно щелкнуть правой кнопкой мыши на имени загружаемого файла, выбрать в контекстном меню пункт Run As (Запустить как), а затем выбрать пункт GHCi session (Сеанс GHCi). В нижней части экрана получит фокус вкладка Console (Консоль), и в ней будет запущенный интерпретатор.

Вы, наверное, заметили, что для загруженного файла было выведено предупреждение (в случае использования EclipseFP сразу за определением функции появился желтый значок предупреждения). В предупреждении сообщается, что мы не указали сигнатуру типа, то есть не определили тип функции.

Указание типа функции

В главе 1 мы заострили ваше внимание на том, что Haskell является строгим, статически типизированным языком, в то же время сейчас мы создаем функцию без указания ее типа. Как это возможно? Ответ содержится в том же самом предупреждении: интерпретатор не получил от нас на этот счет никакой информации, поэтому сам вывел для этой функции правильный тип. *Вывод типа* (type inference), то есть автоматическое определение типа каждого выражения на основе используемых функций и синтаксических конструкций, является весьма важным для разработчика в таких системах со строгой типизацией, как Haskell, поскольку позволяет сохранить их управляемость. В этом отношении Haskell сильно отличается от других языков программирования, таких как Java или C#, которые вплоть до своих последних редакций требуют от разработчика задавать типы всех переменных в коде.

Однако оставить определение функции без указания типа этой функции не считается правилом хорошего тона, что и стало причиной предупреждения даже при том, что интерпретатор сам сумел определить тип функции. Решить проблему можно добавлением *сигнатуры типа* (type signature): имени определяемой функции, за которым следует двойное двоеточие и тип этой функции. По соглашению

сигнатуры типа добавляются непосредственно перед элементом, тип которого они определяют. Если вспомнить, что типы функций записываются с использованием стрелки `->`, то сигнатура типа для `firstOrEmpty` будет иметь следующий вид:

```
firstOrEmpty :: [[Char]] -> [Char]
firstOrEmpty lst = if not (null lst) then head lst else "empty"
```

ПРИМЕЧАНИЕ

Если вы используете EclipseFP, сигнатуру типа можно добавить автоматически, щелкнув на желтом значке предупреждения и выбрав в меню Add type signature (Добавить сигнатуру типа).

Разработка более сложного примера

Давайте попробуем определить собственные функции для объединения и изменения порядка следования элементов списка на обратный, которые мы соответственно назовем `(+++)` и `reverse2`. Основным средством определения функций, работающих в Haskell со списками (и с большинством других структур данных), является рекурсия. В данном случае определение функции с помощью рекурсии сводится к рассмотрению двух общих случаев:

- Что делать, когда список пуст?
- Что делать, когда у списка есть какой-то начальный элемент и какой-то хвост?

В обоих случаях основная схема одна и та же:

```
if null list then <случай для пустого списка> else <какие-то действия над (head
list) и над (tail list)>
```

Начнем с функции объединения. Прежде всего, из-за ее символического имени `(+++)` придется записать инфиксное имя. То есть в определении нужно написать следующее:

```
lst1 +++ lst2
```

Вспомним два общих случая из ранее представленного списка. Теперь, когда реализуется конкретная функция, эти случаи могут быть определены в более конкретных понятиях:

- При объединении пустого списка с любым другим списком нужно просто вернуть второй список, поскольку первый список не добавляет никаких элементов.
- Когда имеется непустой список, который добавляется ко второму списку, нужно подумать о том, что делать с головой и хвостом первого списка. Используя рекурсию, можно вызвать функцию `(+++)` для добавления хвоста первого списка ко второму списку. Возвращаемое этим вызовом значение и станет требуемым списком, но без первого элемента. Для решения проблемы можно просто вставить голову первого списка, используя оператор двоеточия `(:)`.

Переведенное в код определение приобретает следующий вид:

```
lst1 +++ lst2 = if null lst1  {- проверка на пустоту -}
                then lst2      -- базовый случай
                else (head lst1) : (tail lst1 +++ lst2)
```

Этот пример дает возможность впервые показать использование в Haskell-коде *комментариев*, которые могут содержать любой текст и полностью игнорируются как интерпретатором, так и компилятором (хотя некоторые средства, например Haddock, извлекают из комментариев нужную им информацию). Как и во многих других языках программирования, в Haskell имеется два вида комментариев. Первый из них является многострочным и распространяется от сочетания символов `{-` до ближайшего сочетания символов `-}`. Многострочные комментарии, в отличие от однострочных, на концы строк не реагируют. Однострочные комментарии распространяются от группы символов `--` до первого символа новой строки, встретившегося в исходном коде.

Если у вас возникают трудности с пониманием этого рекурсивного определения, я рекомендую вам попробовать применить его к нескольким небольшим спискам. Например, вычисление выражения `[1, 2] +++ [3, 4]` осуществляется в такой последовательности:

1. Изначально выражение имеет вид:

   ```
   [1,2] +++ [3,4]
   ```

2. Рекурсивно оно дает следующее выражение:

   ```
   1:([2] +++ [3,4])
   ```

3. А затем рекурсивно дает следующее выражение:

   ```
   1:(2:([] +++ [3,4]))
   ```

4. Теперь первый список пуст, следовательно, рекурсия завершается так:

   ```
   1:(2:[3,4])
   ```

5. Оператор двоеточия просто добавляет элементы списка. Поэтому `2:[3,4]` дает `[2,3,4]` и т. д.

6. Окончательный результат:

   ```
   [1,2,3,4]
   ```

Теперь трассировку выполнения мы будем проводить довольно часто. Чтобы примеры стали короче, мы по соглашению будем разделять шаги выполнения символами `=>`. То есть предыдущий пример можно записать следующим образом:

```
[1,2] +++ [3,4] => 1:([2] +++ [3,4]) => 1:(2:([] +++ [3,4])) => 1:(2:[3,4]) =
[1,2,3,4]
```

А сейчас давайте перейдем к функции `reverse2`. Здесь мы опять последуем методологии разделения работы на возможные варианты списка, передаваемого в качестве входных данных. Реверсировать пустой список проще простого: нужно просто вернуть пустой список. Для реверсирования списка, имеющего некое количество элементов, можно воспользоваться следующим подходом:

1. Реверсировать хвост списка.
2. Объединить голову списка с концом реверсированного хвоста.

Рекурсия происходит на шаге 1. Реверсирование хвоста списка означает реверсирование списка, который короче исходного списка на один элемент. Этот укороченный на один элемент список сам передается функции реверсирования, что

приводит к созданию еще одного списка, который короче еще на один элемент. Данный процесс продолжается до тех пор, пока хвост не опустеет.

Поскольку непосредственного способа добавления элементов к концу списка нет, мы воспользуемся функцией `(+++)`, которую только что определили для объединения списка с отдельным элементом. В данном случае получится следующий код:

```
reverse2 list = if null list
                then []
                else reverse2 (tail list) +++ [head list]
```

В главе 1 уже упоминалось, что в экосистеме Haskell есть очень полезное свойство, позволяющее тестировать функции в интерактивном режиме. В упражнении 2.3 описаны действия, которые нужно предпринять в отношении функций данного раздела.

УПРАЖНЕНИЕ 2.3. ТЕСТИРОВАНИЕ ФУНКЦИЙ

Загрузите файл, в котором определены функции, в программу GHCi и с целью тестирования вызывайте эти функции с различными аргументами. Основываясь на появляющихся предупреждениях, добавляйте к своему коду сигнатуры типов.

Возвращение более одного значения

Шаг за шагом мы приближаемся к определению более длинных функций. В следующей функции будет вычисляться максимум и минимум числового списка. Первым же заданным вами вопросом может быть следующий: «А как функция вернет сразу несколько значений?» В других языках программирования для этого потребовалось бы определять для хранения результата какие-нибудь структуры или типы данных. То же самое возможно и в Haskell, но для простых случаев вроде нашего можно воспользоваться встроенным типом, который называется *кортежем* (tuple). Кортеж — это просто тип с фиксированным количеством компонентов, каждый из которых хранит какое-нибудь значение, не обязательно одного и того же типа. Значения кортежа записываются внутри круглых скобок и разделяются запятыми, та же форма записи используется для типов кортежа. Например, при выполнении следующего кода создается кортеж из двух элементов: первый элемент представляет собой строку `"hello"`, а второй — один из результатов вычисления числового условия:

```
Prelude> :t ("hello", True, if 2 > 3 then 'a' else 'b')
("hello", True, if 2 > 3 then 'a' else 'b') :: ([Char], Bool, Char)
```

ВНИМАНИЕ

Типы кортежей разной длины являются совершенно разными типами. Например, функция, работающая с кортежами в форме (a,b), не может применяться ни к кортежам в форме (a,b,c), ни к кортежам, имеющим какое-нибудь другое количество значений.

На данном этапе мы будем работать только с *парами*, то есть с кортежами, состоящими из двух компонентов. Для таких кортежей есть две функции-деструктора: `fst` возвращает первый компонент, `snd` возвращает второй компонент. Теперь у вас есть все составляющие для создания функции, вычисляющей как максимум, так

и минимум списка. Если вы сейчас забыли о случае с пустыми списками, которые не имеют четко определенных максимума или минимума, то можете снова перейти к проработке вариантов. Первый вариант — это список с одним элементом, и этот элемент должен быть возвращен и как максимум, и как минимум, то есть в обоих компонентах кортежа. Если у списка более одного элемента, вы можете получить максимум и минимум хвоста списка, а затем сравнить эти значения с его головой. Таким образом, наше рекурсивное решение приобретает следующий вид:

```
maxmin list = if null (tail list)
              then (head list, head list)
              else ( if (head list) > fst (maxmin (tail list))
                     then head list
                     else fst (maxmin (tail list))
                   , if (head list) < snd (maxmin (tail list))
                     then head list
                     else snd (maxmin (tail list))
                   )
```

Ну надо же! Каким-то образом функция для решения столь несложной задачи превратилась в совершенно непонятный и сложный в поддержке код: здесь масса повторений и, что еще хуже, выражение `maxmin (tail list)` заново вычисляется за один рекурсивный вызов целых четыре раза, что весьма негативно сказывается на производительности. Решением может стать *локальная привязка* (local binding), которая дает имя выражению, предназначенному для использования в более крупном выражении. В Haskell есть два вида конструкторов привязки: `let` и `where`. В обоих случаях привязка представляется в виде *имя* `=` *выражение*. Разница заключается в позиции по отношению к основному выражению: конструктор `let` вводит привязку перед основным выражением и должен заканчиваться ключевым словом `in`, а конструктор `where` вводит привязку после выражения. Следующий код является переписанной версией предыдущего кода, но уже с использованием локальных привязок для ссылки на голову списка и возвращения значений в случае рекурсии:

```
maxmin list = let h = head list
              in if null (tail list)
                 then (h, h)
                 else ( if h > t_max then h else t_max
                      , if h < t_min then h else t_min )
                      where t = maxmin (tail list)
                            t_max = fst t
                            t_min = snd t
```

Специальное расположение кода во всех этих примерах не носит случайного или эстетического характера, как можно было заметить при попытке копирования кода в редактор вручную. Первые догадки о причинах этого могут навести на мысль об использовании такого чувствительного к отступам языка, как Python. Но в Haskell применяется другое решение, называемое *отбивкой* (layout). В синтаксисе, основанном на отбивке, порядок отступа строки не так важен, как тот факт, что все элементы в одном и том же блоке начинаются в одном и том же столбце. Например:

- В блоке `if` строки для `then` и `else` должны иметь одинаковый отступ.
- В блоке `let` или в блоке `where` все локальные привязки должны начинаться в одной и той же позиции.

ПРИМЕЧАНИЕ

При чтении Haskell-кода можно заметить, что его приверженцы склонны в блоках локальной привязки располагать на одной линии и другие символы, например знаки равенства (=). Правила отбивки применяются только к началу выражений, поэтому выравнивать код таким образом вовсе не обязательно. И тем не менее такому ставшему уже общепринятым выравниванию вам нужно следовать или, как минимум, к нему привыкнуть.

И наконец, следует заметить, что в Haskell допускается группировка блоков с помощью символов `{` и `}`, а также отделение выражений друг от друга с помощью символа `;`. Например, в предыдущем примере последнее условие `where` можно переписать следующим образом:

```
where { t = maxmin (tail list) ; t_max = fst t ; t_min = snd t }
```

Здесь нужно отметить, что применять такой синтаксис при написании нового кода не рекомендуется (он был задуман для случаев, когда Haskell-код создается автоматически какой-нибудь другой программой).

Работа с типами данных

Для группировки фиксированного количества компонентов разных типов в Haskell предусмотрены кортежи, а для группировки неограниченного количества элементов одного типа — списки. Казалось бы, этого вполне достаточно, чтобы приступить к моделированию данных для веб-приложения. Например, клиент Paul 25 лет, покупатель двух машин времени, может быть представлен следующим образом:

```
("Paul", 25, ["Super Time Machine 2013", "Medieval Machine"])
```

Однако для такого подхода характерны две проблемы: во-первых, из-за вложенных вызовов `fst`, `snd` или `head` код будет сложно читать, во-вторых, в коде игнорируется строгая типизация, поскольку компилятор не в состоянии отличить описание клиента от, скажем, описания рыбы с ее общепринятым названием, размером и списком морей обитания. Решение проблемы заключается во введении нового типа данных, конкретизированного для представления клиентов. Основными типами данных, которые можно создавать в Haskell, являются *алгебраические типы данных* (Algebraic Data Types, ADT). Именно на таких типах данных и будет сконцентрировано наше внимание в этом разделе. ADT определяется с помощью двух частей:

- имени типа, обеспечивающего представление его значений;
- множества конструкторов для создания новых значений (у этих конструкторов могут быть аргументы, хранящие значения указанных типов).

Во многих языках программирования для типа данных (или для класса в объектно-ориентированных языках) могут определяться различные конструкторы. Но эти конструкторы так или иначе связаны между собой и, как правило, больше похожи на рациональные методы создания значений, предлагаемых по умолчанию. В большинстве функциональных языков, к которым принадлежит и Haskell, разные конструкторы служат для построения совершенно разных вариантов значений.

Чтобы все стало понятнее, приступим к моделированию клиентов. Есть три вида клиентов:

- правительственные организации, идентифицируемые по названиям;
- компании, для которых нужно записывать название компании, ее идентификационный номер, данные контактного лица с указанием его места в иерархии компании;
- отдельные клиенты, известные по имени, фамилии и желанию получать в дальнейшем информацию о предложениях и скидках.

В Haskell эти три типа клиентов могут быть представлены следующим образом:

```
data Client = GovOrg     String
            | Company    String Integer String String
            | Individual String String Bool
```

Как видите, синтаксис объявления типов данных начинается с ключевого слова `data`, за которым следует название типа. После этого идет перечисление конструкторов, разделенных вертикальной чертой (`|`). Каждый из них начинается с имени конструктора, за которым следуют типы его аргументов.

ПРОПИСНЫЕ БУКВЫ В HASKELL

Одной из особых характеристик синтаксиса Haskell является требование, согласно которому имена, даваемые пользователем, должны соответствовать соглашениям об использовании прописных букв. Вкратце эти соглашения можно изложить следующим образом:

- Функции, аргументы и привязки должны начинаться со строчной буквы. При использовании строчных букв в имени оператора оно НЕ ДОЛЖНО начинаться с символа двоеточия (:).
- Типы, конструкторы, классы типов и виды должны начинаться с прописной буквы. При использовании прописных букв в имени оператора оно ДОЛЖНО начинаться с символа двоеточия (:).

Эти правила помогают понять, с каким элементом вы имеете дело.

Используя конструкторы, значения типа `Client` можно создать, написав имя конструктора и значения для каждого аргумента в том порядке, в котором эти аргументы появляются в объявлении:

```
*Chapter2.DataTypes> :t GovOrg "Nasa"
GovOrg "Nasa" :: Client
*Chapter2.DataTypes> :t Company "Pear Inc." 342 "Mr. Sparrow" "CEO"
Company "Pear Inc." 342 "Mr. Sparrow" "CEO" :: Client
```

Однако при попытке вывести сами значения происходит что-то непонятное:

```
*Chapter2.DataTypes> Individual "Jack" "Smith" True
    No instance for (Show Client) arising from a use of 'print'
    Possible fix: add an instance declaration for (Show Client)
    In a stmt of an interactive GHCi command: print it
```

Чтобы показать значения на экране, интерпретатор внутри себя вызывает для этих значений функцию `print`. Однако соответствующий код для этого типа данных не написан, поэтому возникает ошибка. Чтобы решить проблему, нужно воспользоваться доступным в Haskell механизмом *извлечения по умолчанию* (default deriving), который позволяет наделять ADT некоей функциональностью без написания кода. В данном случае нам нужно иметь возможность получения строкового пред-

ставления значений, то есть нам нужно обеспечить извлечение класса `Show`. Класс `Show` представляет собой класс типов, позволяющий получить строку из любого значения этого типа. Можно написать код самостоятельно или позволить это сделать Haskell за вас. В следующем примере используется объявление `deriving Show`, заставляющее Haskell осуществить автоматическую генерацию кода:

```
data Client = GovOrg String
            | Company    String Integer String String
            | Individual String String Bool
            deriving Show
```

Теперь интерпретатор может показать значения на экране:

```
*Chapter2.DataTypes> Individual "Jack" "Smith" True
Individual "Jack" "Smith" True
```

Для использования одного ADT-типа, определяемого внутри другого ADT-типа, нет никаких препятствий. Например, в предыдущем коде есть ряд отличающихся друг от друга вариантов характеристики лица в качестве официального представителя компании и в качестве индивидуального клиента. Вы можете пойти по пути определения совершенно нового типа данных `Person` и использовать его внутри типа данных `Client`:

```
data Client = GovOrg     String
            | Company    String Integer Person String
            | Individual Person Bool
            deriving Show
data Person = Person String String
            deriving Show
```

По поводу этой реорганизации следует сделать ряд замечаний.

- Если создается совершенно новый ADT-тип, например с именем `Client2`, но с теми же самыми именами конструкторов, произойдет ошибка сборки. Причина в том, что внутри модуля все конструкторы должны иметь разные имена. Если задуматься, то в постановке такого условия есть вполне определенный смысл, поскольку в противном случае компилятор не смог бы понять, какой из типов вы хотите создать.
- Типы данных и имена конструкторов живут в совершенно разных мирах. Значит, есть возможность создания конструктора, чье имя совпадает с названием типа данных. Несомненно, это общее соглашение вполне подходит для такого одновариантного типа, как `Person`, позволяя иметь два одинаковых имени.
- Чтобы получить возможность извлечения по умолчанию, все типы, используемые внутри другого типа, должны допускать вывод значений на экран. Например, если в тип `Person` не включить строку `deriving Show`, произойдет ошибка компиляции.

Иногда альтернативные варианты могут быть интересны сами по себе, без сохранения дополнительной информации, кроме конструкторов. Например, можно добавить для людей информацию об их половой принадлежности. Вместо использования простого булева значения (в этом случае есть вероятность просто забыть, какое из значений соответствует мужчине, а какое — женщине) можно создать

новый тип данных `Gender`. Такая разновидность типа данных с пустыми вариантами похожа на перечисления в других языках.

```
data Gender = Male | Female | Unknown
```

В упражнении 2.4 предоставляется пошаговая инструкция по вводу этого нового типа данных `Gender` в существующий основной код и по соответствующей модификации доступной функциональности с целью учета нового типа.

УПРАЖНЕНИЕ 2.4. ДОПОЛНИТЕЛЬНЫЕ ТИПЫ ЗНАЧЕНИЙ

Мы только что определили новый тип данных Gender. Целью его определения было включение информации о половой принадлежности в запись об индивидуальном клиенте, поэтому в Person нужно добавить новое поле.

- Добавьте аргумент Gender к Person и сделайте его выводимым на экран с помощью объявления Show.
- Создайте новые значения новых типов данных Client с расширенным определением, над которым мы работали в данном разделе.

Порядок определения новых типов данных уже изучен, поэтому настало время обратить внимание на другие типы, которые могут пригодиться нашему магазину машин времени. Машины времени определяются по их производителю, модели (представленной целочисленным значением), названию, возможности путешествий в прошлое и будущее и цене (которая может быть представлена в виде числа с плавающей точкой). Определите тип данных TimeMachine, хранящий эту информацию. Попробуйте при структурировании значений воспользоваться более чем одним ADT-типом.

Сопоставление с образцом

Настала пора определить функции, работающие с нашими замечательными типами данных. Только вот плохо, что мы не умеем извлекать информацию из конструкторов по аналогии с использованием функций `head` и `tail` для списков или функций `fst` и `snd` для кортежей. Общим решением этой задачи является *сопоставление с образцом* (pattern matching). Сопоставление значения с образцом позволяет распознать структуру значения, включая конструктор, использованный для создания значений, и создать привязки к значениям, закодированные внутри этих значений. При вводе тела сопоставления фактические внутренние значения будут содержаться в *переменных образца* (pattern variables) и с ними можно будет работать.

Простые образцы

В качестве первого примера давайте создадим функцию, задающую имя клиента. В случае если это компания или правительственная организация, имя клиента будет первым компонентом конструктора. В случае если это отдельная личность, придется заглянуть в `Person` и объединить имя и фамилию. Можно заметить, что в таком случае образцы в точности похожи на ADT-конструкторы, но с аргументами, замененными привязками:

```
clientName :: Client -> String
clientName client = case client of
          GovOrg  name                -> name
```

```
Company name id person resp -> name
Individual person ads       ->
case person of Person fName lName gender -> fName ++ " " ++ lName
```

Давайте посмотрим, как происходит следующий вызов:

```
clientName (Individual [Person "Jack" "Smith" Male]) False
```

Сначала система находит выражение `case`. Соответственно она пытается провести сопоставление с первым и со вторым образцами, но в обоих случаях конструктор не соответствует значению. Подходящий конструктор она находит в третьем образце и привязывает значения: теперь `person` содержит `Person "Jack" "Smith" Male`, а `ads` — значение `False`. В теле сопоставления опять присутствует выражение `case`, из которого происходит сопоставление с конструктором `Person`, привязывающим `fName` к `"Jack"`, `lName` к `"Smith"` и `gender` к `Male`. И наконец, система переходит к самому вложенному телу и выполняет объединение, выдавая результат `"Jack Smith"`.

ПРИМЕЧАНИЕ

При загрузке этого определения в интерпретатор вы получите целую коллекцию предупреждений вида «Defined but not used: 'id'». С их помощью до вас доводится информация о том, что вы создали привязку, не используемую в теле сопоставления. Избавиться от подобного предупреждения можно, сообщив компилятору, что эта привязка в вашем коде использоваться не будет, для этого достаточно заменить привязываемую переменную одинарным знаком подчеркивания (_). Например, образец, не вызывающий предупреждения для Company name id person resp, должен иметь вид Company name _ _ _, потому что в последующем коде сопоставления используется только первая переменная образца.

В этом первом примере мы воспользовались простейшим вариантом сопоставления, который выглядит просто как конструкторы и привязки значений аргументов. Но можно указывать и более сложные образцы, в которых некоторые внутренние части значений должны будут также сопоставляться с другими образцами. Применяя данный подход, можно переписать сопоставление в `clientName`, сделав его короче:

```
clientName :: Client -> String
clientName client = case client of
               -- другие образцы
               Individual (Person fName lName _) _ -> fName ++ " " ++ lName
```

И здесь возникает один весьма важный вопрос: что будет, если не найдется *ни одного образца*, соответствующего заданному значению? Ответ проще всего найти с помощью несложного примера. Рассмотрим функцию `companyName`:

```
companyName :: Client -> String
companyName client = case client of
                       Company name _ _ _ -> name
```

Интерпретатор уже предупреждает о том, что образец не охватывает все случаи, то есть не является *исчерпывающим*:

```
Warning: Pattern match(es) are non-exhaustive
In a case alternative:
    Patterns not matched:
        GovOrg _
        Individual _ _
```

Применим функцию к значению, от которого не ожидается выдача исключения. Это похоже на попытку получить голову пустого списка:

```
*Chapter2.DataTypes> companyName (GovOrg "NATO")
"*** Exception: Non-exhaustive patterns in case
```

Подобные функции, не определенные для всей области своих аргументов, называются *частичными* (partial). Соответственно обратной стороной медали являются *полные* (total) функции. В некоторых случаях, если подходящее значение не получено, может быть возвращено значение по умолчанию, например возвращение строки `"unknown"` (неизвестно) в `companyName`, если ввод не относится к `Company`. Но на практике эта проблема встречается настолько часто, что для нее Haskell Platform предлагает специальный тип данных `Maybe τ`. Будучи списком или кортежем, тип `Maybe` (дословно — «может быть») параметризируется типом того значения, которое в нем содержится, поэтому у вас получается `Maybe Integer`, `Maybe String`, `Maybe [Integer]` и т. д. Есть только два вида значений, которые могут быть у этого типа: значение `Nothing` без аргументов обычно показывает, что функция не имеет ничего подходящего, что можно было бы вернуть конкретно для этого значения, а значение `Just` *v* содержит одиночное значение *v* соответствующего типа. Давайте перепишем `companyName`:

```
companyName :: Client -> Maybe String
companyName client = case client of
                       Company name _ _ _ -> Just name
                       _                  -> Nothing
```

Любопытно, что произвести сопоставление с образцом можно непосредственно в привязках `let` и `where`. В этом случае можно обработать только один образец, но такая обработка будет особенно полезна, если известно, что в данном конкретном месте может быть только один вид значения. Скажем, в какой-то момент вы уверены, что клиент, с которым ведется работа, является компанией. Например:

```
let name = match companyName client of
             Just n -> n
```

Вместо этого не вполне понятного кода можно записать намного более лаконичную версию:

```
let Just name = companyName client
```

Константы также являются образцами, которые в точности соответствуют записанным значениям. Давайте обратимся к весьма типичному примеру, используемому при изучении языков программирования, — к числам Фибоначчи. Число Фибоначчи $F(n)$ определяется как $F(0) = 0$, $F(1) = 1$, а для более крупных значений n — как $F(n) = F(n–1) + F(n–2)$. Это довольно просто выразить в понятиях образцов и рекурсии:

```
fibonacci :: Integer -> Integer
fibonacci n = case n of
                0 -> 0
                1 -> 1
                _ -> fibonacci (n-1) + fibonacci (n-2)
```

Здесь подразумевается, что образцы проверяются в таком же порядке, в котором они появляются в коде. Такое зависящее от порядка поведение может привести

к весьма досадным просчетам, а иногда даже к бесконечно выполняемым программам или же программам, исчерпывающим все ресурсы компьютера. В качестве упражнения давайте перепишем функцию `fibonacci`, поставив последний образец на первую позицию. Если теперь попробовать протестировать функцию в интерпретаторе, мы увидим, что она выполняется бесконечно. К тому же, как только произойдет сопоставление с образцом, она полностью прекратит попытки сопоставления с другими вариантами, даже если последующее сопоставление выдаст ошибку. Например, следующие две функции не являются эквивалентными:

```
f :: Client -> String
f client = case client of
             Company _ _ (Person name _ _) "Boss" -> name ++ " is the boss"
             _                                    -> "There is no boss"

g :: Client -> String
g client = case client of
             Company _ _ (Person name _ _) pos ->
               case pos of "Boss" -> name ++ " is the boss"
             _                                -> "There is no boss"
```

```
*Chapter2.DataTypes> f (Company "A" 5 (Person "John" "Jefferson" Male) "Director")
"There is no boss"
*Chapter2.DataTypes> g (Company "A" 5 (Person "John" "Jefferson" Male) "Director")
"*** Exception: Non-exhaustive patterns in case
```

ПРИМЕЧАНИЕ

Я заостряю ваше внимание на том факте, что если в теле сопоставления что-то пойдет не так, сопоставление с образцом на прежний маршрут не вернется. Не забывайте об этом, особенно если у вас есть опыт логического программирования, в котором нормой является унификация с возвратом на прежний путь.

Когда значение дается функции `f`, первый образец не проходит сопоставление, потому что строка `"Director"` не равна строке `"Boss"`. В результате система переходит в черную дыру второго сопоставления и видит, что босса нет. Однако в функции `g` первое сопоставление происходит с `Company`, чье значение удовлетворяет этому сопоставлению, и в этом месте происходит вход в тело сопоставления, при этом все остальные варианты предаются забвению. Затем внутреннее сопоставление терпит неудачу, вызывая исключение.

Можно было заметить, что большинство выражений `case` представляет собой сопоставления с образцом в отношении какого-нибудь аргумента функции. Для таких случаев Haskell позволяет закодировать образец непосредственно в определении. В определение функции включается несколько строк, каждая из которых определяет ее для того или иного образца. При таком подходе создается код, очень похожий на написание математических функций. Например, новые версии функций `clientName` и `fibonacci` приобретают следующий вид:

```
clientName (GovOrg name)                       = name
clientName (Company name _ _ _)                = name
clientName (Individual (Person fName lName _) _) = fName ++ " " ++ lName
```

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Попробуйте воспользоваться этим новым синтаксисом при написании решений упражнения 2.5, несколько заданий в котором позволяют попрактиковаться в сопоставлении с образцом разнообразных значений как для клиентов, так и для машин времени.

УПРАЖНЕНИЕ 2.5. ТОЧНОЕ СОПОСТАВЛЕНИЕ ДЛЯ ВАШИХ МАШИН ВРЕМЕНИ

Эти упражнения ориентированы на сопоставление с образцами определенных вами типов данных. Для работы со списками придерживайтесь образцов с различными ответвлениями для пустого списка и общего случая. Также тщательно продумайте порядок следования образцов. После этого протестируйте функции в интерпретаторе.

- Для сбора статистики напишите функцию, возвращающую количество клиентов каждого пола. Для хранения результатов выполнения данной функции может понадобиться определение дополнительного типа данных.
- Для привлечения потенциальных покупателей раз в год проводится продажа машин времени с большими скидками. Напишите функцию, которая выдает список машин времени, уменьшая их цену на заданный процент. Используйте определенный в предыдущих упражнениях тип данных TimeMachine.

Списки и кортежи

При выполнении предыдущего упражнения можно было вполне естественно задаться вопросом по поводу сопоставления с образцом в отношении списков и кортежей, поскольку похоже, что это могло бы привести к созданию более компактного кода. Естественно, это возможно, поскольку списки и кортежи ничем особенным не отличаются от любых других определяемых пользователем типов данных. Конструкторами списков являются функции `[]` и `(:)`, которые можно использовать при сопоставлении с образцом. Более того, сопоставление с образцом в списках позволяет избавиться от всех проверок на нулевые списки и от всех вызовов головы и хвоста. Например, ранее определенная функция `(+++)` может быть переписана следующим образом:

```
(+++) :: [a] -> [a] -> [a]
list1 +++ list2 = case list1 of
                    []   -> list2
                    x:xs -> x:(xs +++ list2)
```

Или непосредственно при сопоставлении в объявлении функции:

```
[]     +++ list2 = list2
(x:xs) +++ list2 = x:(xs +++ list2)
```

ПРИМЕЧАНИЕ

В Haskell укоренилась практика написания сопоставлений с образцом в отношении списков, когда используется буква или небольшое слово, сопровождаемое тем же самым идентификатором во множественном числе (с буквой s на конце), например: x:xs.

Внутри функций `empty`, `head` и `tail`, определенных в модуле `Prelude`, нет никаких особых ухищрений. Их можно легко определить путем сопоставления с образцом. А вы способны на это?

Иногда сопоставление требуется для списков, состоящих более чем из одного элемента. Это может понадобиться для функции проверки упорядоченности списка целых чисел. Для этого нужно рассмотреть три случая. Первые два касаются пустых списков и списков из одного элемента, которые всегда можно считать упорядоченными. Но когда в списке больше одного элемента, нужно проводить сравнение первого элемента со вторым, а затем смотреть, является ли список, состоящий из второго и последующих элементов, упорядоченным. Такая проверка второго и последующих элементов проводится рекурсивно. Реализация функции `sorted` может иметь следующий вид:

```
sorted :: [Integer] -> Bool
sorted []       = True
sorted [_]      = True
sorted (x:y:zs) = x < y && sorted (y:zs)
```

Но все же этот код не лишен повторений: в нем проводится сопоставление `y:zs`, а позже воспроизводится точно такое же сопоставление. Такая проверочная последовательность на соответствие какого-то значения какому-то образцу с последующим использованием значения в целом, а не его компонентов, встречается в Haskell-коде довольно часто. Поэтому в Haskell введена синтаксическая форма, которая называется *именованными образцами* (as patterns) и позволяет выполнить привязку некоторого значения в сопоставлении, давая в то же самое время возможность провести сопоставление с внутренними компонентами значения. Для этого нужно весь образец, которому вы хотите дать имя, заключить в круглые скобки, указать перед скобками переменную (она будет использоваться для ссылки на весь образец) и символ `@`. Новое определение функции обнаружения упорядоченности списка, в котором используется форма именованного образца для `y:zs`, имеет следующий вид:

```
sorted []            = True
sorted [_]           = True
sorted (x : r@(y:_)) = x < y && sorted r
```

По поводу сопоставления в отношении списков нужно сделать еще одно, последнее замечание: во многих случаях имеется некая функция, которая на первый взгляд имеет смысл только для непустого списка, например вычисление суммы всех элементов списка. Однако в большинстве случаев эта функция может быть разумным образом распространена и на пустые списки. Например, можно сумме пустого списка присвоить значение 0, поскольку если сложить это значение с любым числом, оно не изменится. Подобные значения, которые могут быть безопасно применены к таким операциям, как сумма, называются *нейтральными элементами* (neutral elements) этой операции. Более подробно нейтральные элементы будут рассмотрены в главе 3 при изучении свертки (fold), а затем еще раз — в главе 4 при изучении моноидов (monoids).

Сопоставление в отношении кортежей производится так же просто: нужно лишь воспользоваться синтаксисом компонентов, разделенных запятыми и заключенных

в круглые скобки. Если пример `maxmi` из последнего раздела переписать в этом стиле, получится алгоритм, намного более понятный читателю или специалисту службы поддержки:

```
maxmin [x]    = (x,x)
maxmin (x:xs) = ( if x > xs_max then x else xs_max
                , if x < xs_min then x else xs_min
                ) where (xs_max, xs_min) = maxmin xs
```

Предохранители

Предохранитель (guard) — это часть синтаксиса сопоставления с образцом, позволяющая с помощью булевых условий уточнить образец, что после успешного сопоставления обеспечивается за счет привязки значений. Предохранители в немалой степени способствуют написанию более понятного кода и решению конкретных проблем, помогая выполнять сопоставление с образцом наиболее эффективно.

Прояснить полезность предохранителей помогут два примера. Первый из них является расширением функции получения чисел Фибоначчи для любого целочисленного значения: если запрашивается получение чисел для неотрицательного числа, значение заключается в `Just`, в противном случае возвращается значение `Nothing`. Используя синтаксис в том объеме, который мы уже рассматривали, можно написать следующий код:

```
ifibonacci :: Integer -> Maybe Integer
ifibonacci n = if n < 0
               then Nothing
               else case n of
                     0 -> Just 0
                     1 -> Just 1
                     n -> let Just f1 = ifibonacci (n-1)
                              Just f2 = ifibonacci (n-2)
                          in Just (f1 + f2)
```

На данный момент присущее вам как разработчику чувство понимания кода сигнализирует о том, что за начальной проверкой числа на отрицательность прячется та часть алгоритма, которая в основном относится к сопоставлению с образцом. Так оно и есть.

Еще одной рассматриваемой математической функцией будет биномиальный коэффициент из n по k, который обычно записывается как $\binom{n}{k}$.

Этот коэффициент дает количество способов, с помощью которых мы можем извлечь из мешка k n шаров без повторений. Используя известный треугольник Паскаля, можно дать окончательное определение этого коэффициента как:

$$\binom{n}{k} = \begin{cases} 1, & k = 0 \vee n = k \\ \binom{n-1}{k-1} = \binom{n-1}{k}, & \text{в противном случае} \end{cases}$$

ваша задача — перевести это математическое определение в Haskell-код. Первое приближение может иметь следующий вид:

```
binom _ 0 = 1
binom x x = 1
binom n k = (binom (n-1) (k-1)) + (binom (n-1) k)
```

К сожалению, этот подход не обрадует наш интерпретатор, который покажет для `x` ошибку конфликтующих определений: `Conflicting definitions for 'x'`. Причина в ограничении, накладываемом на образцы, согласно которому переменная в каждом из них может появляться лишь единожды. Возможное решение состоит в изменении всей структуры функции. Здесь опять возникает ощущение неполной раскрытости всей той эффективности, которую мы ожидаем получить от сопоставления с образцом.

Решение проблем обеих функций — предохранитель. Поскольку сам предохранитель — это часть образца, то в случае неудовлетворения его условия он, в отличие от сопоставления с образцом с последующей проверкой условия, допускает возвращение к продолжению выполнения алгоритма (к выбору других вариантов). Имеющиеся в предохранителе булевы условия отделены от остального образца вертикальной чертой |, что позволяет использовать переменные, привязанные в ходе сопоставления. С помощью предохранителей мы можем переписать наши функции Фибоначчи и биномиального коэффициента следующим образом:

```
ifibonacci n | n < 0     = Nothing
ifibonacci 0             = Just 0
ifibonacci 1             = Just 1
ifibonacci n | otherwise = let (Just f1, Just f2) = (ifibonacci (n-1), ifibonacci
(n-2))
                in Just (f1 + f2)
binom _ 0 = 1
binom x y | x == y = 1
binom n k = (binom (n-1) (k-1)) + (binom (n-1) k)
```

Помимо предохранителей в предыдущем коде можно заметить еще две интересные особенности. Во-первых, в первом образце для функции Фибоначчи имеется привязанная переменная `n'`. У нас может быть многократно используемая переменная `n`, но интерпретатор будет жаловаться на затенение предыдущего определения. Хотя интерпретатор прекрасно знает, на какую именно переменную `n` ссылается код, тот факт, что одно и то же имя использовано дважды, может запутать других разработчиков. В Haskell-коде стало привычным указывать для ссылки на весьма близкую привязку тот же самый идентификатор, но со следующим за ним значком `'` (произносится «прим»). В Haskell есть еще одно соглашение по поводу предохранителей, которое касается применения в последнем образце ключевого слова `otherwise`. Хотя само по себе использование этого ключевого слова не добавляет к коду ничего существенного (его эквивалентом является отсутствие предохранителя), оно абсолютно понятно сигнализирует о том, что образец применяется во всех остальных случаях, не обработанных в других вариантах образцов.

В предохранителях могут указываться любые выражения, возвращающие булево значение. Это означает, что в них можно также вызывать определяемые вами функции. Например, следующий код возвращает особые строки при умножении

числа на 2, 3 или 5, а во всех остальных случаях возвращает строку, предлагаемую по умолчанию:

```
multipleOf :: Integer -> Integer -> Bool
multipleOf x y = (mod x y) == 0

specialMultiples :: Integer -> String
specialMultiples n | multipleOf n 2 = show n ++ " is multiple of 2"
specialMultiples n | multipleOf n 3 = show n ++ " is multiple of 3"
specialMultiples n | multipleOf n 5 = show n ++ " is multiple of 5"
specialMultiples n | otherwise = show n ++ " is a beautiful number"
```

Для этого примера, где проверяются несколько условий в отношении одного и того же аргумента, Haskell допускает использование еще более компактного объявления, в котором каждый раз набирать `specialMultiples n` уже не нужно:

```
specialMultiples n
| multipleOf n 2 = show n ++ " is multiple of 2"
| multipleOf n 3 = show n ++ " is multiple of 3"
| multipleOf n 5 = show n ++ " is multiple of 5"
| otherwise = show n ++ " is a beautiful number"
```

УПРАЖНЕНИЕ 2.6. ДОПОЛНИТЕЛЬНЫЕ СОПОСТАВЛЕНИЯ И ПРЕДОХРАНИТЕЛИ

Мы уже рассмотрели сопоставление для списков и кортежей, а также предохранители. Убедиться в усвоении всех этих понятий вам помогут следующие задания.

- Составьте определение знаменитой функции Аккермана. При этом попробуйте воспользоваться предохранителями:

$$A(m, n) = \begin{cases} n = 1, & m = 0 \\ A(m-1, 1), & m > 0,\ n = 0 \\ A(m-1,\ A(m, n-1)), & m > 0,\ n = 0 \end{cases}$$

- Составьте определение функции unzip, которая принимает список кортежей и возвращает два списка, один — со всеми первыми компонентами, а другой — со всеми вторыми компонентами. Например, unzip [(1,2),(3,4)] = ([1,3],[2,4]).

Отображаемые образцы

Иногда нужно найти образцы в значении, но по тем или иным причинам они напрямую не кодируются. Поэтому возникает необходимость перед сопоставлением провести предварительную обработку значения. Для таких случаев можно воспользоваться *отображаемыми образцами* (view patterns). Такие образцы дополняют все ранее показанное новым синтаксическим элементом (*функция* `->` *образец*), с помощью которого *функция* применяется к значению, а затем результат ее применения сопоставляется с *образцом*. Вспомним, к примеру, функцию `clientName` из начала главы и добавим еще одну функцию, обозначающую ответственность (responsibility):

```
responsibility :: Client -> String
responsibility (Company _ _ _ r) = r
responsibility _ = "Unknown"
```

Теперь можно создать функцию, возвращающую сведения о том, имеет клиент особый статус или нет. Давайте будем считать клиента имеющим такой статус, если он является директором компании или если его имя `"Mr. Alejandro"`. Отображаемые образцы позволяют получить весьма наглядный код:

```
specialClient :: Client -> Bool
specialClient (clientName -> "Mr. Alejandro") = True
specialClient (responsibility -> "Director") = True
specialClient _ = False
```

Похоже, что в спешке мы что-то сделали не так. Интерпретатор почему-то выдает следующую ошибку:

```
Illegal view pattern: clientName -> "Mr. Alejandro"
Use -XViewPatterns to enable view patterns
```

Отображаемые образцы входят не в спецификацию Haskell 2010, а в ее расширение, созданное разработчиками GHC. Поэтому совместимость с этим расширением нужно обозначить явным образом. Это делается добавлением специальных параметров компилятора или интерпретатора, но рекомендуется все же добавить к файлу указание транслятору. Это указание является специальным комментарием, распознаваемым компилятором, который используется для включения или выключения некоторых флагов. В данном случае *в начало* исходного файла нужно включить следующий комментарий:

```
{-# LANGUAGE ViewPatterns #-}
```

При работе в интерпретаторе для включения расширения нужно выполнить команду `:set`. Обратите внимание на то, что перед именем расширения должен ставиться префикс `-X`:

```
Prelude> :set -XViewPatterns
```

ПРИМЕЧАНИЕ

GHC включает в себя большое количество расширений (на момент написания книги их было более 30), от простых расширений синтаксиса (как рассмотренные выше отображаемые образцы) до полной реконструкции системы типов. При существенной разнице в эффективности часть из них была принята сообществом, а насчет других все еще ведутся споры. Все GHC-расширения, представляемые в данной книге, относятся к первому набору: их применение считается полезным, поскольку они делают код более элегантным и простым в понимании, не создавая при этом никаких проблем. Главное возражение против применения расширений, даже тех, которые не вызывают споров, основано на том, что они не являются частью стандарта Haskell 2010 Report и поэтому могут сделать ваш код менее совместимым с некоторыми компиляторами Haskell. Однако такая совместимость практически не создает проблем.

Записи

В большинстве языков программирования используется понятие поля как хранилища какого-нибудь значения в более крупной структуре данных. К тому же к полям нетрудно получить доступ или изменить их значения (например, в C или Java, используя формат *структура*`.`*поле*). Из того, что уже изучено, можно понять,

что сопоставление с образцом в отношении крупной структуры очень скоро может стать неразумной затеей, поскольку потребует создавать длинные образцы для извлечения всего лишь одного значения и создавать заново всю структуру данных для изменения всего лишь одного поля.

Создание и использование

Понятия структуры данных, к полям которой можно обращаться по имени, в Haskell не существует. По сравнению с другими средствами записи существенно облегчают доступ к данным. Записи определяются с помощью `data`, но вместо того чтобы просто использовать тип для каждого аргумента, определение записывается как *имя аргумента* `::` *тип аргумента*. Эти определения являются единственным исключением из правила отбивки: множество полей всегда записывается между знаками `{` и `}`, а сами поля отделяются друг от друга запятыми.

Давайте запишем определения `Client` и `Person`, но теперь уже с использованием синтаксиса записи. Чтобы не вносить изменения во все предыдущие функции данной главы, нужно закодировать новые записи в новых типах `ClientR` и `PersonR`. Следует помнить, что имена конструкторов не должны совпадать, поэтому для типов записей нужно воспользоваться новыми именами. А вот имена полей могут совпадать, поэтому вам разрешается использовать `clientRName` для полей в разных вариантах при условии, что они имеют один и тот же тип. Новые определения записей имеют следующий вид:

```
data ClientR = GovOrgR  { clientRName :: String }
             | CompanyR { clientRName :: String
                        , companyId :: Integer
                        , person :: PersonR
                        , duty :: String }
             | IndividualR { person :: PersonR }
             deriving Show

data PersonR = PersonR { firstName :: String
                       , lastName :: String
                       } deriving Show
```

Значения из этих типов можно создавать, применяя тот же самый синтаксис конструкторов, который уже использовался. Но если объявление `data` сделано в виде записи, можно также задействовать имя конструктора, за которым следует список из имен каждого поля, где за каждым именем располагаются знак `=` и соответствующее значение. Из этого можно извлечь два преимущества: во-первых, создание нового значения лучше документируется, позволяя сразу увидеть, какому полю какое значение соответствует. Кроме того, этот синтаксис дает возможность записывать имена полей в любом порядке, предоставляя вам больше свободы. Пример такой нотации выглядит следующим образом:

```
*Chapter2.DataTypes> IndividualR { person = PersonR { lastName = "Smith", firstName
= "John" } }
IndividualR {person = PersonR {firstName = "John", lastName = "Smith"}}
*Chapter2.DataTypes> GovOrgR "NATO"
GovOrgR {clientRName = "NATO"}
```

Имена полей также используются для создания специальных функций, обращающихся к этим конкретным полям. Например:

```
*Chapter2.DataTypes> clientRName (GovOrgR "NATO")
"NATO"
*Chapter2.DataTypes> :t duty
duty :: ClientR -> String
```

Поскольку эти функции создаются автоматически, Haskell накладывает на имена полей два дополнительных ограничения:

- Они не должны совпадать с именем любого другого поля или с именем функции.
- Как уже ранее упоминалось, вам разрешается использовать одно и то же имя поля в нескольких вариантах вашего типа данных. Но если вы так и сделаете, все эти поля должны иметь один и тот же тип. В противном случае соответствующей функции не может быть задан правильный тип.

Записи пригодятся при сопоставлениях с образцом. В случае традиционного конструктора для каждого имеющегося в нем поля нужно написать привязку или другой образец. Таким образом, во многих случаях код заканчивается коллекцией привязок со знаком подчеркивания (_), что затрудняет поддержку. Запись позволяет воспользоваться новым образцом, напоминающим привязку: конструктор плюс список, состоящий из заключенных в скобки элементов *имя поля* = *образец*. Вам нужно включать не все поля, а только те, которые нужно привязать или сопоставить. Вам даже разрешается записать пустой список полей, что показано в следующем примере:

```
greet :: ClientR -> String
greet IndividualR { person = PersonR { firstName = fn } } = "Hi, " ++ fn
greet CompanyR { clientRName = c } = "Hello, " ++ c
greet GovOrgR { } = "Welcome"
```

Для записи сопоставления в GHC есть два очень интересных дополнения, кодирующих самые обычные образцы, что позволяет использовать меньше текста. Первое дополнение называется *уплотнением записей* (record puns) и включается с помощью указания транслятору `NamedFieldPuns`. При уплотнении записей все образцы полей в виде *имя поля* = *имя поля*, создающие привязку для соответствующего поля, доступную с тем же именем в теле образца, можно заменить одиночным *именем поля*. Этот вид сопоставления может чередоваться с обычным сопоставлением. Например:

```
greet IndividualR { person = PersonR { firstName } } = "Hi, " ++ firstName
greet CompanyR { clientRName } = "Hello, " ++ clientRName
greet GovOrgR { } = "Welcome"
```

Другой весьма распространенной идиомой является подчинение какого-нибудь поля образцу и привязка остальных полей для их использования в теле образца. В этом случае даже с уплотнением записей код может получиться весьма пространным. Однако GHC может воспользоваться своим знанием имен полей и сгенерировать для вас весь этот код автоматически. В частности, расширение `RecordWildCards` позволяет задействовать две точки (`..`) с целью автоматического создания привязок для всех переменных, которые еще не были упомянуты в образце. В результате предыдущий пример мог бы приобрести следующую минимальную форму:

```
greet IndividualR { person = PersonR { .. } } = "Hi, " ++ firstName
greet CompanyR { .. } = "Hello, " ++ clientRName
greet GovOrgR { } = "Welcome"
```

ПРИМЕЧАНИЕ

Запомните, что для использования этих расширений вам нужно включить в начало исходного кода объявление {-# LANGUAGE Имя_расширения #-}.

Обо всем, что касалось возможностей привязки и сопоставления записей, мы уже поговорили, напоследок нам осталось обсудить использование синтаксиса записей для обновления записи: если `r` является привязкой, содержащей значение типа записи, для создания точной копии `r`, в которой соответствующее поле было изменено, можно задействовать выражение `r { `*имя поля* `= `*новое значение*` }`. Рассмотрим, к примеру, функцию, которая гарантирует, что имя клиента `PersonR` всегда начинается с прописной буквы:

```
nameInCapitals :: PersonR -> PersonR
nameInCapitals p@(PersonR { firstName = initial:rest }) =
        let newName = (toUpper initial):rest
        in p { firstName = newName }
nameInCapitals p@(PersonR { firstName = "" }) = p
```

Не пожалейте времени, чтобы разобраться в этом последнем примере, поскольку в нем показано множество нюансов, упоминавшихся в данной главе: для сопоставления с образцом в отношении `PersonR` использован синтаксис записи. Внутри него для сопоставления со списком применена форма *x*`:`*xs*. Если позже понадобится сослаться на все значение с целью его обновления, то это будет возможно, поскольку для привязки его к `p` используется именованный образец. И наконец, внутри выражения `let` вычисляется новое имя, которое служит для обновления `p` с помощью синтаксиса обновления записи.

Закончив внесения изменений в код для клиентов, мы можем также воспользоваться преимуществами синтаксиса записей при описании машин времени. Именно это и является целью упражнения 2.7.

УПРАЖНЕНИЕ 2.7. ЗАПИСИ МАШИН ВРЕМЕНИ

Перепишите ранее данное определение типу данных TimeMachine с использованием записей. Код обновления цен на машины времени станет намного короче.

Идиома значений по умолчанию

Завершается эта глава рассказом об особенно полезном соглашении, к которому пришло Haskell-сообщество. Мы рассмотрим функции, способные получать очень длинный перечень аргументов, но чаще всего эти аргументы имеют значения, предлагаемые по умолчанию. Возьмем, к примеру, сетевую библиотеку. Для создания подключения нужна следующая информация:

- URL-адрес, к которому нужно подключиться;
- тип подключения (TCP или UDP);
- скорость подключения;

- наличие прокси-сервера;
- наличие кэширования;
- наличие проверки активности;
- время истечения тайм-аута.

Эти элементы могут быть закодированы следующим образом:

```
data ConnType = TCP | UDP
data UseProxy = NoProxy | Proxy String
data TimeOut = NoTimeOut | TimeOut Integer

data Connection = ... -- Определение опущено

connect :: String -> ConnType -> Integer -> UseProxy -> Bool -> Bool -> TimeOut ->
Connection
```

Разумеется, многие просто захотят подключиться к целевому URL-адресу, используя TCP-подключение на самой высокой скорости и некие разумные значения в отношении прокси-сервера, кэширования, проверки активности и времени истечения тайм-аута. Первым решением будет создание для этого случая специальной функции:

```
connectUrl :: String -> Connection
connectUrl u = connect u TCP 0 NoProxy False False NoTimeOut
```

Это решение существенно упрощает подключение в простом варианте, но возникают две проблемы:

- Удобство сопровождения. Если в какой-то момент потребуется добавить новый параметр подключения, изменять вызовы `connect` придется всем пользователям этой функции. Или если изменится значение, предлагаемое по умолчанию, придется пересматривать и переписывать все варианты использования.
- Библиотеку легко применять только в самом простом случае. А чтобы подключиться к URL-адресу через прокси-сервер, придется пойти дальше и воспользоваться *полноценной* функцией `connect`, передав ей *все* аргументы. В некоторых случаях выяснение целесообразности тех или иных параметров, предлагаемых по умолчанию, может быть затруднено.

И тут на помощь приходят записи. Вместо последовательной передачи аргументов все они или их большинство могут быть сгруппированы в запись, которая может быть использована в качестве аргумента. Например, для подключения можно задействовать следующий код:

```
data ConnOptions = ConnOptions { connType      :: ConnType
                               , connSpeed     :: Integer
                               , connProxy     :: UseProxy
                               , connCaching   :: Bool
                               , connKeepAlive :: Bool
                               , connTimeOut   :: TimeOut
                               }

connect' :: String -> ConnOptions -> Connection
connect' url options = ...
```

Вторым шагом будет создание константы, кодирующей некоторые разумные умолчания:

```
connDefault :: ConnOptions
connDefault = ConnOptions TCP 0 NoProxy False False NoTimeOut
```

Теперь для создания подключения с параметрами, предлагаемыми по умолчанию, используется немного больше кода, но вы получаете взамен возможность изменения одного параметра (например, типа подключения на UDP) без необходимости переписывать все значения, предлагаемые по умолчанию. В следующем примере показан самый простой случай, а также случай указания в качестве типа подключения варианта UDP:

```
*Chapter2.DefaultValues> connect' "http://apress.com" connDefault
*Chapter2.DefaultValues> connect' "http://apress.com" connDefault { connType = UDP
}
```

Осталась только одна проблема: если мы добавляем новый параметр, а разработчик прибегнул к непосредственному использованию конструктора типа записи, это использование должно быть изменено. Решение заключается в запрещении непосредственного вызова конструктора и принудительном применении тем или иным способом константы `connDefault`. Это может быть сделано путем *отказа от экспорта* конструктора. Как это делается, вы сможете увидеть в следующей главе, где вы также узнаете о том, что такое интеллектуальные конструкторы.

Выводы

В этой главе мы сделали первые шаги на пути изучения программирования на языке Haskell:

- Мы познакомились с основными типами данных: символами, булевыми значениями, списками и кортежами.
- Мы научились определять новые функции, а также применять конструкторы `let` и `where` для создания временных привязок, позволяющих многократно использовать выражения и создавать тем самым более качественный код. Еще мы узнали, как определять функцию по вариантам.
- Мы определили свои первые типы данных, освоили концепции алгебраического типа данных (ADT) и конструктора, поработали над созданием новых значений в интерпретаторе.
- Мы приобщились к фундаментальной идиоме программирования на языке Haskell — сопоставлению с образцом. Мы узнали о том, как выполняется сопоставление для базовых типов данных и для типов, определяемых пользователями, а также о том, как предохранители, именованные и отображаемые образцы позволяют уменьшить объем кода сопоставления.
- Мы выяснили, что представленные в этой главе записи являются в Haskell наиболее подходящим синтаксисом для создания полей, для доступа к этим полям и для их обновления. Мы познакомились с образцом проектирования «значения по умолчанию», в основе которого лежат записи.

Многократное использование кода с помощью списков

3

В главе 1 отмечалось, что функциональные языки, к которым относится и Haskell, характеризуются частым применением функций в качестве аргументов или возвращаемых значений, однако в главе 2 об этом ничего не говорилось. Сейчас мы вернемся к этим темам и познакомимся даже не с одним, а с тремя доступными в Haskell способами многократного использования кода.

Один из способов, ярко высвечивающих возможности Haskell в плане многократного использования кода, заключается в создании функций, способных работать со значениями любого типа, имеющими отношение к конкретной форме или варианту дизайна. Такие списочные функции, как `head`, могут работать с любым типом, имеющим форму `[τ]`, независимо от того, к какому типу данных относится параметр `τ`. Этот принцип, известный как *параметрический полиморфизм* (parametric polymorphism), применим также к типам данных, которые могут хранить значения любого допустимого типа.

Еще один способ, благодаря которому в Haskell обеспечивается многократное использование кода, сводится исключительно к способности этого языка работать с функциями точно так же, как с другими значениями. Функции, манипулирующие другими функциями, называются функциями *высшего порядка* (higher-order). Что же касается Haskell-функций, о которых рассказывалось в предыдущей главе, то это были функции *первого порядка* (first-order).

Третьим способом многократного (и совместного) использования кода является взятие функций или типов данных из одного модуля и применение их в другом модуле. Как это делается, мы рассмотрим с обеих сторон: как с позиции *экспорта* определений, позволяющего сделать функции доступными, так и с позиции их последующего *импорта*.

Основное же внимание в данной главе будет сосредоточено на *списках* и функциях, которые находятся в модуле `Data.List`. Тому есть две причины: списки помогают проиллюстрировать новые понятия и в то же время являются весьма удобным инструментом в повседневной практике Haskell-программирования. В подтверждение их важности язык Haskell имеет даже специальный синтаксис для создания списков и управления ими — это так называемый синтаксис *формирователей* (comprehensions). Подробнее о нем мы расскажем, когда дойдем до последней темы в рамках изучения списков и узнаем, как правильно выполнять *свертку* (fold) и *развертку* (unfold) списков.

Параметрический полиморфизм

Вы, наверное, заметили, что в предыдущих главах все определяемые нами функции обрабатывали какие-то конкретные типы данных. Однако, похоже, что некоторые встроенные функции, например `head` или `empty`, могут работать со списками любого типа: `[Integer]`, `[Client]` и т. д. Вполне уместно задать вопрос: неужели функции, работающие со списками, рассматриваются компилятором или интерпретатором как-то по-другому? Или же у вас, как у разработчика, имеется возможность создавать подобные функции, способные работать с любым типом данных? Правильным будет утвердительный ответ на второй вопрос: мы без проблем можем самостоятельно воспользоваться определенными списочными функциями. Такая возможность у нас есть.

Давайте посмотрим на тип такой функции, как `head`:

```
*Chapter3.ParamPoly> :t head
head :: [a] -> a
```

В предыдущей главе мы узнали, что имена типов должны начинаться с прописной буквы. Но в данной конкретной сигнатуре мы видим идентификатор из строчной буквы. Если помните, по соглашению идентификаторы, начинающиеся со строчной буквы, зарезервированы для определения функций, привязок и аргументов. На самом деле идентификатор `a` — это идентификатор типа аргумента, который можно привязывать к разным типам в зависимости от варианта использования.

Рассмотрим, к примеру, применение функции `head` к строке `"Hello"`. Эта строка имеет тип `[Char]`. На момент применения вам нужно найти значение для аргумента типа `a`. Решением в данном случае будет `a = Char`. Следовательно, тип, получаемый функцией `head` в случае ее применения к `"Hello"`, обозначается как `[Char] -> Char` (как только аргументу типа присваивается какое-нибудь значение, аргумент должен быть заменен во всем типе). Описанную логику иллюстрирует рис. 3.1, где тип переменной и конкретный тип унифицируются, становясь одним типом.

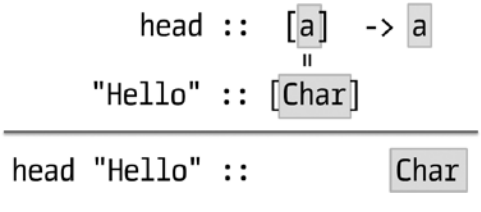


Рис. 3.1. Вывод типа функции head "Hello"

Про функции, подобные `head`, говорят, что они работают с любым значением аргумента типа `a`: их не волнует форма внутренних элементов. Это так называемый параметрический полиморфизм (дословно — «много форм»). На самом деле этимология этого понятия несколько вводит в заблуждение, поскольку полиморфная функция должна работать не со многими, а *со всеми* типами. Haskell также позволяет применять функции только к подмножеству всех типов. Это так называемый специальный полиморфизм, о котором мы поговорим в следующей главе.

ПРИМЕЧАНИЕ

Под разными именами параметрический полиморфизм поддерживается во многих языках программирования: в C++ его реализуют с помощью шаблонов (templates), а в Java и C# — с помощью обобщений (generics). Все они имеют схожую функциональность.

Следует заметить, что у функций может быть более одного аргумента типа, и каждый из них будет получать свое значение независимо от других. Примером может послужить функция `fst`, которая предоставляет первый компонент кортежа, состоящего из двух элементов:

```
*Chapter3.ParamPoly> :t fst
fst :: (a, b) -> a
```

Когда функции `fst` передается конкретный кортеж, тип `(a, b)` выводится из типов, находящихся в этом кортеже. Например, функции можно передать кортеж `([3,8], "Hello")`, и тип `(a, b)` станет типом `([Integer], Char)`.

Специального синтаксиса для написания полиморфных функций, исключая аргументы типа, не существует. Когда при использовании значения его тип не играет никакой роли (например, при сопоставлении с образцом в его возможных конструкторах), Haskell вставляет аргумент типа. Давайте напишем функцию, которая возвращает разные строки в зависимости от значения `Maybe`:

```
maybeString (Just _) = "Just"
maybeString Nothing  = "Nothing"
```

При попытке загрузить эту функцию в интерпретатор появится предупреждение. Тем не менее в соответствующем сообщении для `maybeString` будет показан предполагаемый тип данных:

```
Warning: Top-level binding with no type signature:
      maybeString :: forall t. Maybe t -> [Char]
```

ПРИМЕЧАНИЕ

Как можно видеть, в предупреждении сообщается, что перед типом стоит префикс forall t. Этот префикс явным образом показывает, что t может иметь любой тип, а в остальном не добавляет никакой информации. Более того, в Haskell 2010 этот вид сигнатуры типа не разрешен. В текущем стандарте Haskell нужно всегда записывать тип как Maybe t -> [Char]. Чтобы иметь возможность в явном виде записать forall, нужно дать транслятору директиву ExplicitForAll или любую другую директиву, которая это разрешает, например RankNTypes.

Полиморфизм применим не только в отношении функций, но и в отношении типов данных. Он уже подразумевался, когда для списка из любых возможных типов `t` использовалась запись `[t]`. По приведенным примерам можно видеть, что полиморфный тип записывается с указанием его имени наряду со списком всех своих аргументов типа, например `Maybe Integer`. Определение полиморфных типов похоже на определение основных типов, но после имени типа записываются имена аргументов типов для данного объявления. Затем эти имена можно использовать в конструкторах на месте любого другого типа. Например, в нашей программе можно принять решение о присвоении уникального идентификатора каждому клиенту, но при этом не заострять внимание на том, какого конкретно вида идентификатор

требуется (обычно выбираются варианты в виде целых чисел или строк), поскольку работа напрямую с идентификатором никогда вестись не будет.

В качестве хорошего примера применения полиморфизма можно привести следующий код:

```
data Client i = GovOrg  { clientId :: i, clientName :: String }
              | Company { clientId :: i, clientName :: String
                        , person :: Person, duty :: String }
              | Individual { clientId :: i, person :: Person }
              deriving Show

data Person = Person { firstName :: String, lastName  :: String }
              deriving Show
```

При создании значения аргументу типа будет приписываться конкретное значение, в данном случае — `Char`:

```
*Chapter3.ParamPoly> :t GovOrg 'n' "NTTF" -- National Time Travel Foundation
GovOrg 'n' "NTTF" :: Client Char
```

В типе данных может использоваться более одной переменной типа. В этом случае применяются кортежи. Например, этот кортеж из трех компонентов имеет тип `(a,b,c)`. Если бы вы определяли тройки самостоятельно, объявление `data` имело бы следующий вид:

```
data Triple a b c = Triple a b c
```

Следует заметить, что одну и ту же переменную типа в своем определении ADT-типа можно использовать многократно. Но это не означает, что значения полей должны быть одинаковыми, они просто должны иметь одинаковые типы. Это частое заблуждение тех, кто только начинает изучение Haskell, и соответствующий пример привести нетрудно: `('a','b')` имеет тип `(Char,Char)`, в обеих переменных типа хранится один и тот же тип `Char`, но значение `'a'` отличается от значения `'b'`. Нужно также запомнить еще одну особенность: хотя аргумент типа может появляться в конструкторе несколько раз, он представлен одной переменной типа. Давайте, к примеру, объявим тип для пары элементов, где каждый элемент будет иметь один и тот же тип. Это можно сделать следующим образом:

```
data SamePair a = SamePair a a
```

Такое значение, как `SamePair 1 2`, будет иметь тип `SamePair Integer`, а не `SamePair Integer Integer`. Правда, тот факт, что один и тот же идентификатор обычно используется повторно как для имени типа, так и для его конструктора, добавляет еще больше путаницы, но к этому нужно просто привыкнуть. Такую привычку поможет выработать упражнение 3.1.

УПРАЖНЕНИЕ 3.1. ИСПЫТАЙТЕ СЕБЯ В РОЛИ ПРОВЕРЯЮЩЕГО ТИПОВ

Попробуйте разобраться, чем занимаются следующие функции и какой тип будет подразумеваться интерпретатором. Дайте наиболее полиморфный ответ из всех возможных.

```
swapTriple (x,y,z) = (y,z,x)

duplicate x = (x,x)
```

```
nothing _ = Nothing

index []     = []
index [x]    = [(0,x)]
index (x:xs) = let indexed@((n,_):_) = index xs
               in  (n+1,x):indexed

maybeA [] = 'a'
```

Не забывайте о том, что для проверки ответов можно воспользоваться программой GHCi. Напоминание о том, как это делается, можно найти в главе 2.

Функции в качестве аргументов

В этом разделе мы приступаем к изучению вопроса о том, как трактовать функции в качестве любых других значений, и о тех грандиозных возможностях, которые мы благодаря этому получаем.

Функции высшего порядка

Первой наиболее важной функцией, которую мы собираемся рассмотреть, является функция `map`; она применяет ко всем элементам списка какую-нибудь другую функцию. Пусть это будет функция `succ`, которая прибавляет к числу единицу:

```
*Chapter3.FnsParams> succ 1
2
```

ВНИМАНИЕ

Иногда интерпретатор выводит предупреждения о том, что по умолчанию следующее значение будет ограничено типом Integer: Defaulting the following constraint(s) to type 'Integer'. В предыдущей главе упоминалось, что такие константы, как 1, являются полиморфными числовыми типами, поэтому интерпретатор для запуска кода должен сделать выбор. В предупреждении сообщается, что по умолчанию выбирается тип Integer. Это предупреждение можно спокойно проигнорировать, хотя его можно было бы и вообще отключить, запустив интерпретатор командой ghci -fno-warn-type-defaults. В дальнейшем о подобных предупреждениях мы больше упоминать не будем.

Теперь единицу можно прибавить ко всем элементам списка `[1,2,3]`, воспользовавшись для этого функцией `map` вместе с функцией `succ`:

```
*Chapter3.FnsParams> map succ [1,2,3]
[2,3,4]
```

Как же это все работает? Сначала посмотрим на тип:

```
*Chapter3.FnsParams> :t map
map :: (a -> b) -> [a] -> [b]
```

Здесь можно увидеть форму записи для функций `a -> b`, которая теперь уже находится на месте аргумента. В этих сигнатурах типа закодирован тот факт, что `map` применяет функцию `a` к значению `b` и к списку из значений `a`, после чего возвращает

список значений b. Функции, подобные map, которые получают в качестве аргументов другие функции, называются *функциями высшего порядка*.

В объявлении функций в отношении других функций, передаваемых в качестве аргументов, действуют такие же соглашения об именах, что и для всех других аргументов функций. Никаких других отличительных пометок для аргумента, имеющего тип функции, не требуется. Но так как этот аргумент является функцией, его можно применить к любому другому аргументу или значению, как будто эта функция была определена в каком-нибудь другом месте. Например, пусть определение функции map имеет следующий вид:

```
map _ []     = []
map f (x:xs) = (f x) : (map f xs)
```

Это также является примером параметрического полиморфизма. Но полиморфизм и функции высшего порядка — это совершенно разные понятия: можно определить функцию, которая применяет другую функцию, но в отношении целого числа, увеличенного на две единицы, а после этого умножить результат на три, то есть 3f(x+2). В таком случае будет логичным получение и возвращение функцией f числового значения, следовательно, у функции должен быть тип Integer -> Integer:

```
apply3f2 :: (Integer -> Integer) -> Integer -> Integer
apply3f2 f x = 3 * f (x + 2)
```

Давайте проверим по шагам, что будет при вызове этой функции с использованием функции succ в качестве значения для f:

```
apply3f2 succ 7 => 3 * succ (7 + 2) => 3 * succ 9 => 3 * (9 + 1) => 3 * 10 => 30
```

Теперь, когда мы разобрались с тем, что такое функция высшего порядка, давайте познакомимся с весьма популярной идиомой Haskell-кода. Эта идиома относится к функции ($), которая обеспечивает применение функций (function application):

```
($) :: (a -> b) -> a -> b
f $ a = f a
```

А зачем вообще нужна функция ($)? На первый взгляд она похожа на весьма запутанный инструмент применения некой функции к каким-то аргументам. Но Haskell дает функции ($) очень низкий уровень приоритета, с тем чтобы обе стороны этой операторной функции были вычислены прежде, чем функция f будет применена к a. Поэтому, используя функцию ($), можно избавиться от обилия круглых скобок. Этот прием получил в Haskell довольно широкое распространение. Например:

```
maximum (map succ [1, 2, 3])
```

Этот код обычно можно записать следующим образом:

```
maximum $ map succ [1, 2, 3]
```

Анонимные функции

До сих пор в качестве аргументов всегда использовались другие функции, которые определялись в каком-нибудь другом месте. Но может случиться так, что вам понадобится создать совсем небольшую функцию, применяемую к списку посред-

ством функции `map`. При этом засорять модуль новым определением нет никакого смысла. Решение, заключающееся в определении функции внутри блока `let` или `where`, вам уже известно. Это решение, в котором число 2 прибавляется к каждому имеющемуся в списке числу, показано в следующем примере:

```
*Chapter3.FnsParams> :{
*Chapter3.FnsParams| let f x = x + 2
*Chapter3.FnsParams| in  map f [1,2,3]
*Chapter3.FnsParams| :}
[3,4,5]
```

Но полностью такое решение нас удовлетворить не может: Haskell заставляет нас использовать передающие и возвращающие функции, поэтому с такой конструкцией код будет изобиловать блоками `let`. Вместо этого в Haskell включены *анонимные функции* (anonymous functions). Они представлены только телами функций (без имен), которые могут быть записаны в любом месте кода, где ожидается применение функции. Синтаксис тела функции имеет следующий вид:

```
\аргумент1 аргумент2 ... -> тело
```

Предыдущую операцию `map` можно переписать следующим образом:

```
map (\x -> x + 2) [1,2,3]
```

ИСТОРИЧЕСКОЕ ОТСТУПЛЕНИЕ

Нотация `\... -> ...` родилась из математической теории, которая называется лямбда-исчислением. С этой формальной точки зрения такое выражение, как \x -> x + 2, называется абстракцией и записывается как $\lambda x.x + 2$ (создатели Haskell выбрали символ обратного слеша, поскольку он похож на λ, но его намного проще набирать). Благодаря таким вот историческим корням анонимные функции иногда называют лямбда-абстракциями, или просто абстракциями.

В анонимных функциях, как и во всех других, сопоставление с образцом можно осуществлять непосредственно в отношении аргументов. Например, можно создать функцию, проверяющую равенство целочисленных пар:

```
equalTuples :: [(Integer,Integer)] -> [Bool]
equalTuples t = map (\(x,y) -> x == y) t
```

Однако при использовании анонимных функций допустимы не все формы объявлений обычных функций. У анонимных функций нет имени, поэтому они не в состоянии вызывать самих себя, что делает рекурсию невозможной. Кроме того, можно осуществить сопоставление только лишь с одним образцом. А чтобы осуществить сопоставление с несколькими образцами, следует обратиться к инструкции `case`:

```
sayHello :: [String] -> [String]
sayHello names = map (\name -> case name of
                                 "Alejandro" -> "Hello, writer"
                                 _           -> "Welcome, " ++ name
                     ) names
```

При использовании GHC и включении расширения `LambdaCase` это последнее ограничение снимается. В результате для создания анонимной функции только с одним аргументом сопоставления можно воспользоваться специальным синтаксисом `\case`. Например:

```
sayHello names = map (\case "Alejandro" -> "Hello, writer"
                            name        -> "Welcome, " ++ name
                     ) names
```

Абстракции могут также применяться для возвращающихся функциональных значений. К примеру, пусть нам нужно определить функцию, получающую число `n` и возвращающую другую функцию, которая производит умножение на `n`:

```
multiplyByN :: Integer -> (Integer -> Integer)
multiplyByN n = \x -> n*x
```

Теперь эту возвращаемую функцию можно использовать в тех местах, где берутся одиночные значения, например там, где используется функция `map`:

```
*Chapter3.FnsParams> map (multiplyByN 5) [1,2,3]
[5,10,15]
```

Как видите, функция `multiplyByN 5` при своем применении «помнит» значение, заданное для `n`. В этом случае говорят, что функция содержит наряду с телом значение из окружения (в данном случае — только `n`). Поэтому такие функции почти во всех языках, поддерживающих функциональные свойства, обычно называют *замыканиями* (closures).

Еще одной функцией, работающей со списками, является `filter`. В данном случае `filter` получает функцию вида `type a -> Bool`, то есть функцию, возвращающую булево значение, и применяет ее к каждому элементу, возвращая только те элементы, который отвечают условию. Например, передав упомянутой функции `filter` в качестве аргумента функцию `even` из стандартной библиотеки, можно отфильтровать список чисел и оставить только четные числа:

```
*Chapter3.FnsParams> filter even [1,2,3,4,5]
[2,4]
```

УПРАЖНЕНИЕ 3.2. РАБОТА С ФИЛЬТРАМИ

Используя функцию filter в качестве основы вашего решения, напишите следующие функции:

- функцию filterOnes, возвращающую только элементы, равные константе 1;
- функцию filterANumber, возвращающую только элементы, равные тому числу, которое будет передаваться в качестве аргумента;
- функцию filterNot, выполняющую действие, обратное действию функции filter, то есть возвращающую только те элементы списка, которые не отвечают условию;
- функцию filterGovOrgs, получающую список клиентов Clients (определенный ранее) и возвращающую только клиентов, имеющих отношение к правительственным организациям (напишите ее, используя как вспомогательную функцию isGovOrg, так и выражение \case).

Подсказка: поищите документацию по функции not :: Bool -> Bool (или просто наберите в браузере http://www.haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:not).

Частичное применение функции

Вернемся к функции `map`. Нам уже известны два способа написания этой функции, удваивающей значения всех элементов в списке:

```
double list = map (\x -> x * 2) list
double       = \list -> map (\x -> x * 2) list
```

Haskell позволяет применить еще один подход. Как видите, в конце обоих списков аргументов находится ключевое слово `list`, поэтому его можно просто опустить:

```
double      = map (\x -> x * 2)
```

Чтобы лучше понять смысл такого упрощения, давайте посмотрим на процесс, в ходе которого компилятор выстраивает предположение о типе функции `double`. Прежде всего, начнем с `map :: (a -> b) -> [a] -> [b]`. Функция `(\x -> x * 2)` получает и возвращает число, поэтому ее тип, к примеру, может быть определен как `Integer -> Integer`. Теперь, если применить числовую функцию к `map`, сначала делается сопоставление `a = b = Integer`, а затем первый аргумент в типе заменяется, поскольку для него уже предоставлено значение. Получается следующий результат:

```
map (\x -> x * 2) :: [Integer] -> [Integer]
```

То есть если передать список целочисленных значений, будет возвращен новый список целочисленных значений.

Следуя по пути частичного применения функций, этот подход можно использовать и для анонимной функции `\x -> x * 2`. Нужно сделать лишь одно синтаксическое замечание: когда имя функции представлено оператором (то есть в нем используются только символы вроде *), мы не можем просто задействовать ее имя и аргументы, но можем указать ее *секцию* (section). Эта секция для полностью безымянной анонимной функции представляет собой просто спецификацию выполняемой операции, заключенную в круглые скобки. В данном случае новое определение для `double` имеет следующий вид:

```
double      = map (*2)
```

Здесь уместно сделать обычные предупреждения о коммутативных операциях. В вопросе о том, где именно опускается аргумент, нужно быть предельно осмотрительными, поскольку это может оказать большое влияние на результат. Присмотритесь, например, к разнице в результатах, получаемых в следующих двух примерах:

```
*Chapter3.FnsParams> map (/2) [1,2,3]
[0.5,1.0,1.5]
*Chapter3.FnsParams> map (2/) [1,2,3]
[2.0,1.0,0.6666666666666666]
```

Следует заметить, что конструкторы типов в любом возможном смысле также ведут себя как функции, разница касается только использования прописных букв (вспомним, что имена функций должны начинаться со строчной буквы, а конструкторов типа — с прописной). Запросить тип конструктора или частично применить его можно обычным порядком:

```
*Chapter3.FnsParams> :t Just
Just :: a -> Maybe a
*Chapter3.FnsParams> :t ('a' :)
('a' :) :: [Char] -> [Char]
```

Теперь, когда вы узнали о возможности частичного применения функций, настало время поближе познакомиться со значениями типов функций в том виде, в кото-

ром они написаны. Прежде всего, символ `->` обозначает привязку вправо. То есть тип `a -> b -> c -> d`, хотя и смотрится лучше, является эквивалентом версии `a -> (b -> (c -> d))`. Следовательно, в своей основе каждая функция с более чем одним аргументом является просто функцией, которая берет один аргумент и возвращает замыкание с количеством аргументов, уменьшенным на единицу. Это замыкание, несомненно, может принять еще один аргумент, и так далее, пока дело не дойдет до нефункционального типа, когда у нас уже имеется вся информация по применению функции и по ее телу. Итак, теперь у вас есть как минимум четыре равнозначных способа объявления одной и той же функции с двумя аргументами:

```
f x y = ...
f x   = \y -> ...
f     = \x y -> ...
f     = \x -> \y -> ...
```

Давайте через эту новую призму взглянем на функцию `map`. Ранее о функции `map` говорилось, что «берутся функция и список, после чего функция применяется к списку». Но если теперь тип записан в виде `(a -> b) -> ([a] -> [b])`, возникает новый взгляд на вещи: `map` получает функцию и возвращает версию этой функции, которая работает со списком!

Частичное применение стимулирует использование такого стиля программирования, при котором функции объединяются без какого-либо упоминания об их аргументах. Это называется *бесточечным стилем* (point-free style), поскольку в математике аргументы функций называются точками. Вне всякого сомнения, наиболее важным из комбинаторов является символ точки[1] (`.`), объединяющий две функции. Под объединением я подразумеваю, что указание точки приводит к применению одной функции после другой. Например, следующим образом функцию `f` можно применить к выходным данным функции `g`:

```
f . g = \x -> f (g x)
```

Напомним, что по сравнению с другими системами записи сочетание функций записывается в обратном порядке: стачала идет самая внешняя функция, которая будет применяться к результатам выполнения внутренней функции. Эта особенность опять исходит из основ языка, базирующихся на лямбда-вычислениях и математике, где композиции именно так и обозначаются.

Предположим, к примеру, что нужно создать функцию, которая удваивает все нечетные числа в списке. Казалось бы, естественнее всего воспользоваться для этого следующим кодом:

```
duplicateOdds list = map (*2) $ filter odd list
```

Но на самом деле для получения желаемого результата нужно сначала применить функцию `filter odd`, извлекающую нечетные числа, а затем удвоить каждый элемент полученного списка с помощью функции `map (*2)`. Это и есть композиция,

[1] Поскольку символ (.) называется точкой, может возникнуть путаница из-за эпитета «бесточечный». На самом деле в бесточечном стиле использование точки (.) поддерживается, а не поддерживается явное указание аргументов.

составленная из двух функций, поэтому в соответствии с бесточечным стилем можно написать следующую строку кода:

```
duplicateOdds = map (*2) . filter odd
```

Во многих случаях выражение может быть записано в бесточечном стиле в виде последовательности преобразований некоторых данных, и когда вы привыкнете к такой системе записи, код станет для вас предельно понятным.

В остальной части этого раздела мы познакомимся с дополнительными функциями, реализующими бесточечный стиль. Поскольку эти функции выполняют задачу объединения других функций, их иногда называют *комбинаторами* (combinators). Следующие два комбинатора служат для прямого и обратного преобразования представлений функций для работы с несколькими аргументами и для работы с одним аргументом, представляющим собой кортеж:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f = \(x,y) -> f x y

curry :: ((a,b) -> c) -> a -> b -> c
curry f = \x y -> f (x,y)
```

Функции, получающие последовательность аргументов, называются *каррированными* (curried) версиями тех функций, которые получают кортеж. Здесь следует подчеркнуть одну особенность: некаррированная версия функции получает только *один* аргумент, но этот аргумент является кортежем, следовательно, в нем содержится более одного информационного элемента. Например, функция `max`, возвращающая максимум два числа, получает два аргумента:

```
*Chapter3.FnsParams> max 3 2
3
```

Однако при каррировании этой функции ее нужно вызывать только с одним аргументом, представляющим собой кортеж:

```
*Chapter3.FnsParams> (uncurry max) (3,2)
3
```

Обычно предпочтение отдается каррированным версиям, поскольку они поддаются частичному применению. Но иногда некаррированная версия также заслуживает рассмотрения. Предположим, к примеру, что у вас есть список пар чисел, и нужно получить список максимумов этих пар. Использовать сигнатуру `map max` напрямую нельзя, поскольку `max` требует двух аргументов. Решение состоит в каррировании функции перед ее применением:

```
*Chapter3.FnsParams> map (uncurry max) [(1,2),(2,1),(3,4)]
[2,2,4]
```

ИСТОРИЧЕСКОЕ ОТСТУПЛЕНИЕ

И сам язык Haskell, и понятие «каррированная функция» названы в честь американского логика Хаскелла Брукса Карри (Haskell Brooks Curry) (1900–1982), преподававшего комбинаторную логику, которая легла в основу последующих разработок в области функционального программирования.

Теперь введем еще несколько комбинаторов: (***) выполняет параллельную композицию из двух функций (то есть каждая из них применяется к компоненту кортежа).

```
(***) :: (a -> b) -> (c -> d) -> ((a,c) -> (b,d))
f *** g = \(x,y) -> (f x, g y)
```

Используя следующий комбинатор, можно, к примеру, вычислить корень квадратный из 4 и арктангенс 3:

```
*Chapter3.FnsParams> (sqrt *** atan) (4, 3)
(2.0,1.2490457723982544)
```

А вот комбинатор `duplicate` порождает два значения из одного. Польза от применения этой функции может быть не столь очевидна, но она станет весьма важной составляющей последующих примеров.

```
duplicate :: a -> (a,a)
duplicate x = (x,x)
```

Например, с помощью комбинаторов (***) и `duplicate` можно вычислить несколько функций от одного и того же значения:

```
*Chapter3.FnsParams> (sqrt *** atan) (duplicate 4)
(2.0,1.3258176636680326)
```

Давайте, используя эти комбинаторы, перепишем математическую формулу в бесточечном стиле. Первоначальной целью станет переписывание формулы $3x + 7(x + 2)$. Пренебрегая упрощениями, графически формулу можно представить так, как показано на рис. 3.2.

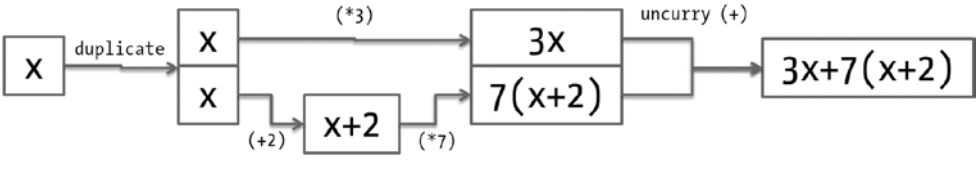


Рис. 3.2. Графическая запись формулы 3x+7(x+2)

Работая на основе диаграммы, изображенной на рис. 3.2, эту формулу можно переписать в бесточечном стиле. Показанные на рисунке параллельные части объединяются с помощью комбинатора (***). Каждая из частей получается путем объединения нескольких функций с помощью комбинатора (.). При этом получается следующий результат:

```
formula1 :: Integer -> Integer
formula1 = uncurry (+) . ( ((*7) . (+2)) *** (*3) ) . duplicate
```

Можете попробовать преобразовать в диаграммы другие, более короткие (или длинные!) арифметические выражения, приведя их в конечном счете к бесточечной нотации. Для изменения порядка следования аргументов функции на обратный может понадобиться определение дополнительного комбинатора. Чаще всего для этого комбинатора используется имя `flip`, и он имеет следующую сигнатуру типа:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

В заключение следует заметить, что все рассмотренные до сих пор комбинаторы тесно связаны с понятием *Haskell-стрелки* (Haskell arrow), то есть схемы, использованной для моделирования широкого спектра концепций, в том числе автоматов. Основные сведения о схеме стрелок даны в статье Джона Хьюза (John Hughes) «Generalising Monads to Arrows», вышедшей в 1998 году. Но прежде чем углубляться в данную статью, я предлагаю дочитать эту книгу до конца.

ПРИМЕЧАНИЕ

Статью Джона Хьюза можно найти на сайте http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf.

Подробнее о модулях

В предыдущей главе мы говорили о создании модулей, призванных систематизировать наши функции и типы данных. Следующим логическим шагом станет изучение возможностей получения в модуле определений, которыми можно было бы воспользоваться в другом модуле. Такая возможность называется *импортом* (importing) модуля, она очень похожа на импорт пакетов в Java или пространств имен в C#.

Импорт модулей

Импортируемые модули перечисляются после объявления `module`, но перед любым другим определением. Существует несколько способов импорта модуля. Наиболее распространенный подход заключается в том, чтобы ввести все функции модуля в область видимости и сделать их доступными для использования, как будто они были определены в импортирующем их модуле. Поскольку в данной главе нам наиболее интересны списочные функции, мы можем импортировать модуль `Data.List` и воспользоваться функцией перестановок. Например, функцией `permutationsStartingWith`, которая возвращает все перестановки строки, начинающейся с указанной буквы.

```
module Chapter3.MoreModules
import Data.List
permutationsStartingWith :: Char -> String -> [String]
permutationsStartingWith letter = filter (\l -> head l == letter) . permutations
```

ПРИМЕЧАНИЕ

Несмотря на то что модули в Haskell имеют иерархическую структуру, импорт модуля не приводит к тому, что в области видимости появляются какие-либо из его дочерних модулей. Например, импорт модуля Data не даст доступа к функции Data.List.permutations, потому что функция permutations находится в модуле Data.List, а этот модуль не был импортирован.

В некоторых случаях возникает конфликт имен, находящихся в разных модулях. То есть при импорте определений из двух модулей, включающих одинаковые функции или типы, компилятор не знает, что из них ему нужно использовать. Наиболее разумным решением станет конкретизация импортируемых определений.

Для этого нужно включить перечень требуемых элементов в список, заключенный в круглые скобки. Например, можно указать, что требуется импортировать только функции `permutations` и `subsequence`:

```
import Data.List (permutations, subsequence)
```

Иногда возникает прямо противоположная ситуация: требуется импортировать весь модуль за исключением конкретных элементов (обычно тех, из-за которых может возникнуть конфликт). Как правило, это нужно, когда возникает конфликт некоторых имен в импортируемом и в разрабатываемом модулях. Для подобных случаев Haskell предоставляет механизм *скрытого импорта* (hiding import). Те объявления, которые не следует вводить в область видимости, также заносятся в список, но перед ними ставится ключевое слово `hiding`. Например, чтобы импортировать все, кроме функций `head` и `tail`, можно воспользоваться следующим объявлением:

```
import Data.List hiding (head, tail)
```

Для выбора или скрытия импортируемых типов данных нужен дополнительный синтаксис. Эта потребность обусловлена тем, что в ADT фактически заключены два информационных компонента: сам тип и его конструкторы. Поэтому Haskell-сообществом было принято решение воспользоваться формой записи *Тип*(*Перечень конструкторов*). Вот несколько способов, позволяющих импортировать тип данных `Client` из первого раздела:

```
import Chapter3.ParamPoly (Client()) -- только тип без конструкторов
import Chapter3.ParamPoly (Client(GovOrg,Individual)) -- подмножество конструкторов
import Chapter3.ParamPoly (Client(..)) -- импорт всех конструкторов
```

До сих пор речь шла о том, как импортировать модули *без квалификации*. После импорта определений какой-либо дополнительный синтаксис для их использования не нужен. Обратной стороной медали является *импорт с квалификацией* (qualified imports). Такой импорт требует указания префикса функции в виде имени модуля, из которого она была взята. Тогда появляется возможность без всяких проблем использовать функции или типы с одинаковыми именами, но из разных модулей. Например, можно выполнить импорт с квалификацией функций `permutations` и `filter`:

```
import qualified Data.List (permutations, filter)
permutationsStartingWith :: Char -> String -> [String]
permutationsStartingWith letter = Data.List.filter (\l -> head l == letter).
                                  Data.List.permutations
```

Как видите, выбор подмножества функций можно комбинировать с квалификацией модуля. Разумеется, эти понятия не зависят друг от друга, поэтому они допускают свободное сочетание.

В некоторых случаях у модуля бывает слишком длинное и мало подходящее для префикса имя. Чтобы избавиться от рутинного набора текста, модуль можно *переименовать*, используя ключевое слово `as`. После этого в префиксах объявлений можно будет указывать новое имя модуля:

```
import qualified Data.List as L

permutationsStartingWith :: Char -> String -> [String]
```

```
permutationsStartingWith letter = L.filter (\l -> head l == letter) .
L.permutations
```

Как и в предыдущем случае, импорт с квалификацией, указание конкретного списка импортируемых функций и переименование можно смешивать в одном объявлении. Импорт модуля с включением только функций `permutations` и `subsequence` будет иметь следующий вид:

```
import qualified Data.List (permutations, subsequences) as L
```

МОДУЛЬ PRELUDE

По умолчанию при импорте любого Haskell-модуля всегда без квалификации импортируется модуль Prelude. В этом модуле содержатся самые основные и востребованные функции, доступные в Platform, например (+) или head. Такой автоматический импорт играет важную роль, предоставляя возможность писать Haskell-код, не заботясь об импорте каждой вызываемой функции. Мы уже использовали модуль Prelude во всех встречавшихся в этой книге примерах.

Потребность в отключении режима автоматического импорта этого модуля возникает крайне редко. Это можно сделать в GHC, включив расширения NoImplicitPrelude. При этом следует помнить, что если понадобится какая-нибудь функция из модуля Prelude, вам придется импортировать ее явным образом.

Интеллектуальные конструкторы и представления

Вы можете управлять не только импортом объявлений из какого-нибудь модуля, но и тем, какие объявления нужно сделать публичными в ваших собственных модулях для того, чтобы ими можно было пользоваться в других модулях. То есть вы можете *экспортировать* требуемые объявления. По умолчанию экспортируется каждое отдельное имеющееся в модуле объявление. Для ограничения доступности ваших функций и типов данных сразу же после имени модуля нужно явно перечислить все публичные объявления. Например, из следующего модуля экспортируется только функция `f`:

```
module M (f) where

f = ...
g = ...
```

Разумеется, можно также управлять экспортом типов данных и конструкторов типа. Как и в случае со списками импорта, у вас есть несколько вариантов для экспорта типов данных: экспорт только типа без конструктора (тем самым запрещается создание значений путем непосредственного вызова конструкторов), экспорт только некоторого подмножества конструкторов или экспорт всех имеющихся конструкторов.

Помните, в предыдущей главе, когда упоминался образец «значения по умолчанию», утверждалось, что разговор еще не закончен, поскольку на тот момент мы еще не умели ограничить создание значений `ConnOptions`? Теперь у вас есть все необходимое, чтобы доработать образец. Вы можете экспортировать только тип данных `ConnOptions` без конструкторов, а также константу `connDefault`, уточняемую путем изменения значений по умолчанию. Например:

```
module Chapter2.DataTypes (ConnOptions(), connDefault) where
```

Эта идея скрытия конструкторов заданного типа данных открывает дверь к новому образцу проектирования, который обычно называют *интеллектуальным конструктором* (smart constructors). Сценарий использования состоит в следующем: иногда для моделируемого понятия правильными являются не все значения, которые могут быть получены с помощью конструктора. В таких случаях нужно, чтобы разработчик при создании значений ограничивался только значениями из нужного пространства.

Например, может понадобиться представить замкнутое целочисленное пространство, то есть множество значений между каким-нибудь целым числом `a` (в самой нижней границе) и каким-нибудь другим целым числом `b` (в самой верхней границе). Во всех случаях логичным инвариантом будет $a \leq b$. Но определение ADT-типа `Range` имеет следующий вид:

```
data Range = Range Integer Integer deriving Show
```

Это определение не предотвращает появления неправильных значений. Поэтому можно предоставить вместо него функцию `range`, которая будет осуществлять проверку. Если проверка будет пройдена, выполнение функции `range` продолжится конструированием. В противном случае будет выдана ошибка.

```
range :: Integer -> Integer -> Range
range a b = if a <= b then Range a b else error "a must be <= b"
```

ПРИМЕЧАНИЕ

Встроенная функция error может быть использована в любом месте программы для оповещения о том, что в этой точке программа не может продолжить свое выполнение и должна быть остановлена. При этом выводится конкретное сообщение об ошибке. Это один из инструментов обработки ошибок в Haskell. О других способах оповещения об ошибках вы узнаете по мере чтения книги.

Такая функция, как `range`, называется *интеллектуальным конструктором*. В основном она работает как обычный конструктор, но выполняет при этом некоторые дополнительные проверки его аргументов. Принудительное использование этого конструктора во всех случаях можно организовать, отказавшись от экспорта конструктора `Range` с экспортом одного только типа. Например:

```
module Chapter3.Ranges (Range(), range) where
```

Но тут возникает проблема! Поскольку конструктор скрыт, любые сопоставления с образцом, имеющие следующую форму и находящиеся за пределами закрытого кода модуля, не будут даже компилироваться:

```
case ... of Range x y -> ...
```

Компиляция не будет проводиться по причине недоступности конструктора. Решение проблемы заключается в создании нового типа данных, в котором кодируются обозреваемые значения этого типа, а затем при сопоставлении с образцом используются представления. Разумеется, это не остановит пользователей от создания неверных значений `RangeObs`, но в случае, когда все функции работают с `Range` и не работают с `RangeObs`, не останется ничего иного, как применять их правильно.

В этом случае обозреваемый тип данных и функция преобразования могут иметь следующий вид:

```
data RangeObs = R Integer Integer deriving Show
r :: Range -> RangeObs
r (Range a b) = R a b
```

Теперь при экспорте конструктора `RangeObs` можно осуществлять сопоставление с образцом, используя представление. При этом следует помнить о необходимости включения в исходный файл расширения `ViewPatterns`:

```
prettyRange :: Range -> String
prettyRange rng = case rng of
                    (r -> R a b) -> "[" ++ show a ++ "," ++ show b ++ "]"
```

Наконец-то мы нашли решение, позволяющее не показывать конструкторы для создания значений и в то же время не препятствовать возможности сопоставления с образцом для работы с ними.

Углубленное изучение списков

Пользоваться наиболее востребованными списочными функциями, а именно функциями `map` и `filter`, мы уже научились. В данном разделе мы рассмотрим дополнительные примеры списочных функций высшего порядка и изучим ряд весьма полезных для Haskell-кода образцов проектирования, таких как свертки. Большинство рассматриваемых функций находится в модуле `Prelude`, поэтому явно импортировать их не нужно. Остальная часть этих функций находится в модуле `Data.List`.

УГЛУБЛЕНИЕ В КОД СПИСКОВ

Читая этот раздел, пытайтесь писать определения для каждой списочной функции, с которой вы познакомились. Это будет для вас неплохим упражнением для закрепления знаний о параметрическом полиморфизме и функциях высшего порядка. Начать можно с написания определения для функции filter.

Свертки

Первой из рассматриваемых функций будет функция `foldr`, которая откроет для нас мир сверток. *Свертка* (fold) в такой структуре данных, как список, является функцией, которая собирает вместе все значения, содержащиеся в структуре, с целью выдачи единого результата. Свертка — это весьма выразительный и эффективный инструмент, роль которого часто недооценивается. В качестве примера сверток можно привести суммирование всех целочисленных значений, имеющихся в списке, и нахождение максимума значений в узлах дерева (более подробно деревья будут рассмотрены чуть позже).

Определение функций `foldr` включает три аргумента: бинарную функцию `f`, используемую для пошагового объединения элементов, начальное значение `initial` и, наконец, сам список:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f initial []     = initial
foldr f initial (x:xs) = f x (foldr f initial xs)
```

Начальное значение плюс бинарная операция получили в Haskell-коде довольно широкое распространение. Обычно начальное значение выбирается таким образом, чтобы его использование в бинарной операции не приводило к изменению результата. Возьмем, к примеру, задачу суммирования всех элементов списка. Интуитивно выбирается операция сложения (+). Тогда начальное значение должно выбираться таким образом, чтобы не повлиять на эту операцию, то есть вроде бы кажется, что этим значением должен быть ноль. Давайте проследим за вычислениями при вызове функции `foldr`, выполняющей суммирование всех элементов списка:

```
foldr (+) 0 [1,2,3] => 1 + foldr (+) 0 [2,3]
                    => 1 + (2 + foldr (+) [3])
                    => 1 + (2 + (3 + foldr (+) 0 []))
                    => 1 + (2 + (3 + 0))
                    => 1 + (2 + 3) => 1 + 5 => 6
```

Как видите, функция `foldr` осуществляет поэлементный перебор списка до тех пор, пока не достигнет его конца. В этот момент для начала вычисления всего созданного стека вызовов функция `foldr` использует исходное значение от конечного до первого применения соответствующей объединяющей функции, в данном случае — функции (+). Если трактовать список как комбинацию из конструкторов (:) и [], алгоритм можно перефразировать следующим образом: `foldr` заменяет все экземпляры (:) функцией `f`, а все экземпляры [] — исходным значением. Такое представление алгоритма показано на рис. 3.3.

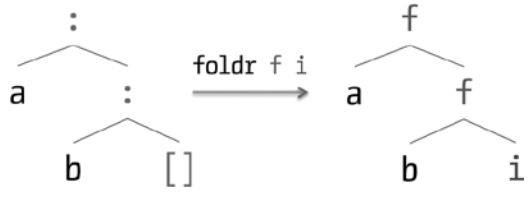


Рис. 3.3. Визуальное представление работы функции foldr

Еще одним примером свертки является функция `maximum`, которая находит в списке самое большое значение. В этом случае начальное значение еще более иллюзорно, поскольку требуется понять, что собой представляет максимум пустого списка. Чтобы ответить на этот вопрос, нужно вспомнить о втором свойстве начального значения: оно не должно изменять итог бинарной операции, которая в данном случае представлена функцией `max`. Это означает, что нам нужно найти такое значение `z`, чтобы для любого значения `x` соблюдалось условие `max(z,x) = max(x,z) = x`. Можете проверить, что значением, удовлетворяющим этому условию, является минус бесконечность (-∞).

По умолчанию целочисленные значения в Haskell не позволяют представлять бесконечность, поэтому для такого случая возникает потребность в определении пользовательского типа данных. После некоторых размышлений можно прийти

к выводу, что понятие добавления бесконечных значений не относится исключительно к целым числам. Оно применимо также к рациональным значениям, к значениям с плавающей точкой и т. д. Чтобы получить от нового типа данных больше пользы, можно определить тип данных `InfNumber` полиморфным:

```
data InfNumber a = MinusInfinity
                 | Number a
                 | PlusInfinity
                 deriving Show
```

За счет его полиморфности мы получаем возможность использования типа не только для целочисленных значений. Хотя наша непосредственная задача требует только бесконечных целочисленных значений, будущие задачи могут потребовать, скажем, бесконечных значений с плавающей точкой. Полиморфизм в данном случае является нашим вкладом в будущее.

Следующим шагом является определение новой бинарной операции `infMax`, предназначенной для объединения двух таких чисел:

```
infMax MinusInfinity x       = x
infMax x MinusInfinity       = x
infMax PlusInfinity  _       = PlusInfinity
infMax _  PlusInfinity       = PlusInfinity
infMax (Number a) (Number b) = Number (max a b)
```

Давайте попробуем записать нашу свертку:

```
*Chapter3.Lists> foldr infMax MinusInfinity [1,2,3]
    No instance for (Ord t0) arising from a use of 'infMax'
```

Очевидно, задача еще не решена. Мы получили сообщение об ошибке, потому что записанное выражение не прошло проверку типов. Операция `infMax` объединяет элементы типа `InfNumber`, но `[1,2,3]` является списком целочисленных значений. Первым решением могло бы быть преобразование списка в `InfNumbers` за счет проецирования конструктора `Number` на список:

```
*Chapter3.Lists> foldr infMax MinusInfinity $ map Number [1,2,3]
Number 3
```

Однако если приглядеться к типу функции `foldr`, можно заметить, что объединительной функции (аргумент `f` в `foldr`) не нужно получать значения одного и того же типа, поскольку она не имеет типа `a -> a -> a`. Ее тип записывается как `a -> b -> b`, следовательно, в качестве первого аргумента функция `f` должна получать значение того типа, к которому принадлежат элементы списка, а второй аргумент должен относиться к аккумулируемому типу, совпадающему с типом начального значения (в показанном здесь определении функции `foldr` он называется `initial`). В нашем случае это означает, что объединительная функция должна иметь тип `Integer -> InfNumber Integer -> InfNumber Integer`, поскольку начальное значение имеет сигнатуру типа `MinusInfinity :: InfNumber Integer`. Как конвертировать существующие числа в `InfNumbers`, вы уже знаете, и это единственная особенность, которая нам нужна для свертки:

```
*Chapter3.Lists> foldr (\x y -> infMax (Number x) y) MinusInfinity [1,2,3]
Number 3
```

В имени `foldr` содержится напоминание об алгоритме, реализуемом функцией. Это правоассоциативная (`r`) свертка (`fold`). То есть самые внутренние скобки будут находиться в правой стороне выражения. По аналогии с этим можно создать левоассоциативную свертку, которая включена в Haskell Platform под именем `foldl`.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ initial [] = initial
foldl f initial (x:xs) = foldl f (f initial x) xs
```

Теперь самые внутренние скобки будут в левой стороне выражения, как показано в следующей трассе вычислений:

```
foldl (+) 0 [1,2,3] => foldl (+) (0 + 1) [2,3]
                    => foldl (+) ((0 + 1) + 2) [3]
                    => foldl (+) (((0 + 1) + 2) + 3) []
                    => ((0 + 1) + 2) + 3 => (1 + 2) + 3 => 3 + 3 => 6
```

В приведенных примерах получающееся в результате свертки значение не зависит от того, в какую сторону она выполняется, вправо или влево. Но для подобной независимости выбранный оператор объединения должен быть коммутативным. Иными словами, должно быть верным следующее условие: `f(x,y) = f(y,x)`. Раз порядок следования входных аргументов не имеет значения, можно сделать так, чтобы направление свертки, влево или вправо, также не имело значения.

Однако не все операции можно превратить в коммутативные. Вычитание, например, не является коммутативной операцией, поэтому результат от свертки к свертке меняется:

```
*Chapter3.Lists> foldr (-) 0 [1,2,3]
2
*Chapter3.Lists> foldl (-) 0 [1,2,3]
-6
```

И еще одна, последняя, версия сверток представляет собой функции `foldr1` и `foldl1`, не требующие начального значения. В них начальным значением является последний (в `foldr1`) или первый (в `foldl1`) элемент списка. Если вы знакомы с каким-нибудь языком, родившимся из языка Lisp, например с Common Lisp, Racket или Clojure, такие операции свертки будут вам известны под именем `reduce`. В Haskell они используются нечасто, но могут пригодиться в тех случаях, когда обработка пустого списка гарантированно не должна происходить или когда обработка сильно запутывает код. Например, ранее определенная функция `maximum` при использовании функции `foldr1` становится намного проще:

```
maximum' :: [Integer] -> Integer
maximum' = foldr1 max
```

УПРАЖНЕНИЕ 3.3. ВАША ПЕРВАЯ СВЕРТКА

Рассмотрите задачу создания функций product, minimumClient и all. Функция product вычисляет произведение элементов целочисленного списка. Функция minimumClient вычисляет клиента с самым коротким именем. И наконец, функция all вычисляет конъюнкцию (&&) списка булевых значений. С учетом описания данных функций сделайте следующее:

- Напишите функции, используя сопоставление с образцом и не прибегая к какой-либо функции высшего порядка.

- Напишите функции в виде сверток. В каждом случае попытайтесь найти объединительную операцию и исходя из этого подберите подходящее начальное значение.

Можете ли вы определить общую для всех этих функций структуру при написании их в первом стиле? В каких случаях справедливо утверждение, что использование функций foldr и foldl дает одни и те же результаты?

Дополнительно попробуйте написать функцию minimumBy таким образом, чтобы порядок брался путем первоначального применения к результату функции g. Например, вызов minimumBy (\x -> -x) [1,2,3] должен вернуть значение 3.

Списки и предикаты

Другое большое семейство списочных функций включает в себя функции, принимающие логические предикаты, то есть функции, имеющие тип `a -> Bool`. Представителем данного семейства является уже упоминавшаяся функция `filter`. Мы уже пытались написать двойственную версию функции `filter`, которая извлекает из списка только те элементы, которые не отвечают поставленным условиям. Потребности в группировке элементов списка в зависимости от удовлетворения или неудовлетворения их тем или иным условиям возникают довольно часто. Проще всего это можно сделать следующим образом:

```
bothFilters :: (a -> Bool) -> [a] -> ([a],[a])
bothFilters p list = (filter p list, filter (not . p) list)
```

Определение правильное, но не без проблемы: при таком определении последовательный перебор элементов списка будет осуществляться дважды. Именно для такого случая в Haskell-модуле `Data.List` есть определение функции `partition`, которая разбивает список за один проход:

```
*Chapter3.Lists> import Data.List
*Chapter3.Lists Data.List> partition (> 0) [1,2,-3,4,-5,-6]
([1,2,4],[-3,-5,-6])
```

Если нужно получить только первый удовлетворяющий условию элемент списка, вместо функции `filter` следует использовать функцию `fine`. Есть вероятность того, что в списке нет такого элемента. По этой причине `find` возвращает значение `Maybe`:

```
*Chapter3.Lists Data.List> find (> 0) [1,2,-3,4,-5,-6]
Just 1
*Chapter3.Lists Data.List> find (> 7) [1,2,-3,4,-5,-6]
Nothing
```

А теперь давайте перейдем к следующему варианту использования: у вас есть процессинговая система для магазина, выстраивающая клиентов в такую очередь, которая представлена в виде списка, где головой является следующий обслуживаемый клиент. Когда нагрузка на систему велика, хочется установить следующую политику: игнорировать всех клиентов, не являющихся правительственными организациями. Модуль `Data.List` предоставляет функцию `dropWhile`, возвращающую список с того места, в котором некий предикат становится ложным:

```
skipUntilGov :: [Client a] -> [Client a]
skipUntilGov = dropWhile (\case { GovOrg {} -> False ; _ -> True })
```

ПРИМЕЧАНИЕ

Чтобы предыдущий код воспринимался компилятором GHC, не забудьте включить расширение LambdaCase.

Противоположностью этой функции служит функция `takeWhile`, которая извлекает начальный элемент до тех пор, пока предикат не станет ложным. Функцию `takeWhile` можно использовать для получения списка всех команд в перечне, пока не попадется команда `"stop"`, на которой обработка завершается. Код, реализующий данный замысел, имеет следующий вид:

```
*Chapter3.Lists Data.List> takeWhile (/= "stop") ["hello", "send", "stop",
"receive"]
["hello","send"]
```

Функции `takeWhile` и `dropWhile` являются двумя составляющими функции `span`, которая возвращает как принимаемый, так и отбрасываемый списки:

```
*Chapter3.Lists Data.List> span (/= "stop") ["hello", "send", "stop", "receive"]
(["hello","send"],["stop","receive"])
```

Родственной ей является функция `break`, которая делает то же самое, что и `span`, но перед этим инвертирует предикат. Фактически функция `break` может быть определена как `span (not . p)`.

Последними двумя функциями, получающими унарные предикаты, являются `any` и `all`. По смыслу имен они, соответственно, проверяют, отвечает ли заданному условию хотя бы один из элементов списка или все его элементы. Они подобны логическим кванторам «существует» (∃) и «для всех» (∀). Например, в ежемесячной аналитике требуется убедиться в наличии зарегистрированного в интернет-магазине индивидуального клиента, а также в наличии в системе как минимум компании или правительственной организации, то есть какого-нибудь клиента, не являющегося индивидуальным клиентом. Для начала можно определить функцию `isIndividual`:

```
isIndividual :: Client a -> Bool
isIndividual (Individual {}) = True
isIndividual _               = False

checkIndividualAnalytics :: [Client a] -> (Bool, Bool)
checkIndividualAnalytics cs = (any isIndividual cs, not $ all isIndividual cs)
```

Давайте теперь перейдем к другому виду предикатов — к бинарным предикатам. То есть к таким функциям, которые получают два аргумента и после того или иного сравнения этих аргументов возвращают булево значение. Первой разновидностью сравнения является равенство или неравенство двух элементов: к этому семейству принадлежат функции `(==)` и `(/=)`. И именно таких предикатов ожидает функция `nubBy`: она извлекает элементы, не оставляя в возвращаемом списке двух одинаковых элементов. В следующем примере для каждого равенства мы получаем только одного представителя:

```
*Chapter3.Lists Data.List> nubBy (\x y -> (even x && even y) || (odd x && odd y))
[1,2,3,4,5]
[1,2]
```

Если в nubBy использовать функцию (==), то из списка будут удалены все дубликаты:

```
*Chapter3.Lists Data.List> nubBy (==) [1,2,1,1,3,2,4,1]
[1,2,3,4]
```

Во многих случаях типы оснащены сравнением, предлагаемым по умолчанию. В следующей главе мы увидим, как сравнение, предлагаемое по умолчанию, добавляется к нашим собственным типам, а затем поговорим о классах типов. Если значения поддерживают эту возможность, как те, которые принадлежат типу Integer, то можно просто использовать функцию nub и отбросить функцию равенства:

```
*Chapter3.Lists Data.List> nub [1,2,1,1,3,2,4,1]
[1,2,3,4]
```

ПРИМЕЧАНИЕ

Особой производительностью функции nub и nubBy не отличаются, поскольку они должны проверить на равенство все возможные пары элементов. Это означает, что порядок у функции четверной. В следующей главе мы изучим более быстрый способ удаления дубликатов из списка.

Проверки на равенство или в более широком смысле — на эквивалентность могут быть использованы для сохранения списков в виде множеств, в которых содержится только одна копия каждого значения. Перечислим основные функции. Пара функций union(By) возвращает новое множество со всеми элементами, взятыми из исходных элементов; пара функций intersect(By) возвращает множество, содержащее только те элементы, которые встречаются в обоих множествах; пара функций insert(By) добавляет к множеству только один элемент; функция \\ определяет разницу между множествами, например: x1 \\ x2 содержит все элементы в x1, которых нет в x2. В каждом случае та версия, которая заканчивается окончанием By, получает аргумент, сообщающий о том, как элементы проверяются на эквивалентность, в то время как другие версии используют сравнение, предлагаемое по умолчанию:

```
*Chapter3.Lists Data.List> :{
*Chapter3.Lists Data.List | let x1 = [1,2,3,4]
*Chapter3.Lists Data.List |     x2 = [2,3,5]
*Chapter3.Lists Data.List | in (x1 `union` x2, x1 `intersect` x2, x1 \\ x2)
*Chapter3.Lists Data.List | :}
([1,2,3,4,5],[2,3],[1,4])
```

В этом примере показана также интересная особенность Haskell-синтаксиса — *инфиксная нотация* (infix notation). Когда есть функция, получающая два аргумента и не имеющая имени, которое состоит только из символов (вроде union или intersect), то имя можно записать между аргументами, заключив его в обратные апострофы (``).

И наконец, функция elem указывает на наличие элемента в списке:

```
*Chapter3.Lists Data.List> 2 `elem` [1,2,3]
True
*Chapter3.Lists Data.List> 4 `elem` [1,2,3]
False
```

МИНИ-УПРАЖНЕНИЕ

Напишите функцию elem, используя функцию find и сопоставление по образцу.

Другим привычным значением для бинарных предикатов является упорядочение: `p x y` означает, что `x` в некотором смысле предшествует `y`. Однако для прояснения ситуации и повышения производительности упорядочение в Haskell определяется не возвращением значения типа `Bool`, а возвращением значения типа `Ordering`, каковым может быть `LT` (less than — меньше чем), `EQ` (equal — равно) или `GT` (greater than — больше чем). Например, можно определить функцию, представляющую, что компании и правительственные организации следуют первыми среди клиентов, а затем следуют индивидуальные заказчики. На каждом уровне решение о представлениях принимается на основе имен клиентов. Эта функция реализуется с помощью следующего кода, который пишется на основе знания о том, что встроенная функция `compare` определяется для строк с типом `Ordering`:

```
compareClient :: Client a -> Client a -> Ordering
compareClient (Individual{person = p1}) (Individual{person = p2})
                              = compare (firstName p1) (firstName p2)
compareClient (Individual {}) _ = GT
compareClient _ (Individual {}) = LT
compareClient c1 c2             = compare (clientName c1) (clientName c2)
```

В следующих примерах код будет использовать часть нашего списка клиентов. Как вы, возможно, догадываетесь, покупают или читают книги в нашем магазине многие наши известные современники: ученые, писатели и авантюристы.

```
listOfClients = [ Individual 2 (Person "H. G." "Wells")
                , GovOrg 3 "NTTF"  -- National Time Travel Foundation
                , Company 4 "Wormhole Inc." (Person "Karl" "Schwarzschild")
"Physicist"
                , Individual 5 (Person "Doctor" "")
                , Individual 6 (Person "Sarah" "Jane")
                ]
```

Используя вспомогательную функцию `compareClient`, весь список клиентов можно отсортировать с помощью функции `sortBy`:

```
*Chapter3.Lists Data.List> sortBy compareClient listOfClients
[ GovOrg { clientId = 3, clientName = "NTTF" }
, Company { clientId = 4, clientName = "Wormhole Inc."
          , person = Person { firstName = "Karl", lastName = "Schwarzschild" }
          , duty = "Physicist"}
, Individual { clientId = 5, person = Person { firstName = "Doctor", lastName = ""
}}
, Individual { clientId = 2, person = Person { firstName = "H. G.", lastName =
"Wells" }}
, Individual { clientId = 6, person = Person { firstName = "Sarah", lastName =
"Jane" }} ]
```

Некоторые типы уже имеют предопределенный исходный способ упорядочения значений. Примерами таких типов могут служить легко поддающиеся сравнению числа и символы. В таких случаях можно вызвать функцию `sort`, которая не нуждается в функции сравнения. Например:

```
*Chapter3.Lists Data.List> sort [1,4,2,-3]
[-3,1,2,4]
```

Очень интересно здесь то, что упорядочение также определено для кортежей и списков, если содержащиеся в них элементы обладают исходной возможностью сравнения. В обоих случаях упорядочение носит лексикографический характер: значения сравниваются поэлементно. Лексикографическое сравнение означает, что при разнице в первых компонентах кортежей упорядочение этих двух значений определяет порядок следования элементов кортежа, в случае совпадения ведущих значений сравниваются вторые элементы и т. д. Для списков используется тот же подход, что и для кортежей. Также при упорядочении списков более короткий список считается предшествующим более длинному, который в итоге содержит более короткий список в качестве префикса. Давайте посмотрим на аналогичные примеры сравнений, в которых четко прослеживается лексикографический характер: в первом и третьем случаях кортежи или списки до некоторой позиции равны друг другу, а во втором случае первый список короче второго.

```
*Chapter3.Lists> (compare (1,2) (1,1), compare "Hello" "Hello world", compare
"This" "That")
(GT,LT,GT)
```

Наряду с определением функции `compare` Haskell предоставляет реализацию операторов `(>)`, `(<)`, `(>=)` и `(<=)`. Обычно эти операторы помогают сделать создаваемый код понятнее, поскольку отпадает необходимость в вызове функции сравнения с последующим сопоставлением полученного на выходе результата. К тому же эти операторы более известны. В следующей строке кода показано, что прежний пример также может быть выражен с помощью оператора `(<=)`. Следует заметить, что этот оператор возвращает вместо значения типа `Ordering` простое булево значение.

```
*Chapter3.Lists> ( (1,2) <= (1,1), "Hello" <= "Hello world", "This" <= "That" )
(False,True,False)
```

Это может пригодиться при группировке клиентов в зависимости от определенных характеристик. Функция `groupBy`, имеющая тип `(a -> a -> Bool) -> [a] -> [[a]]`, помещает в единый список все элементы, для которых предикат равенства возвращает значение `True`, то есть все эти элементы должны попасть в одну группу.

Например, может потребоваться определить, какие должности в компаниях встречаются в базе данных (которая сейчас представлена просто списком) чаще других. Чтобы добиться желаемого результата, сначала нужно с помощью функции `filter` отфильтровать те элементы, которые не являются компаниями. Затем можно сгруппировать клиентов в зависимости от занимаемых ими должностей (с помощью функции сравнения, применяемой к функции `groupBy`). Третьим шагом должна стать сортировка списков в зависимости от их длины. Чтобы после сортировки сначала шли наиболее распространенные должности, нужно отсортировать списки по длине в обратном порядке, чтобы первым шел самый длинный список. И наконец, путем обращения к головному элементу из каждого списка извлекается должность. Это можно делать без опаски, поскольку все списки будут непустыми. Также вам известно, что у всех элементов в том или ином списке

указана одна и та же должность, поэтому элемент, к которому мы обращаемся, для нас ничуть не хуже любого другого элемента. Функция, которая все это делает, должна иметь следующий вид:

```
companyDutiesAnalytics :: [Client a] -> [String]
companyDutiesAnalytics = map (duty . head) .
                           sortBy (\x y -> compare (length y) (length x)) .
                           groupBy (\x y -> duty x == duty y) .
                           filter isCompany
                         where isCompany (Company {}) = True
                               isCompany _            = False
```

Есть и более изящный вариант написания этой функции. Нетрудно заметить наличие двух образцов, в которых сравниваются два элемента, но только после применения к значениям какой-нибудь операции. Как показано в следующем фрагменте кода, функция высшего порядка `on`, которая находится в модуле `Data.Function`, позволяет объединять функции сравнения и извлечения значения именно так, как это нам нужно. Чтобы получить обратный порядок для длин списков, пригодится прием, при котором функция сравнения вызывается с аргументами, расположенными в обратном порядке. Для вызова функции двух аргументов с аргументами, следующими в обратном порядке, имеется специально разработанный комбинатор `flip :: (a -> b -> c) -> (b -> a -> c)`. Следующий код является бесточечной версией предыдущего:

```
companyDutiesAnalytics :: [Client a] -> [String]
companyDutiesAnalytics = map (duty . head) .
                           sortBy (flip  (compare `on` length)) .
                           groupBy ((==) `on` duty) .
                           filter isCompany
                         where isCompany (Company {}) = True
                         isCompany _ = False
```

ДЕКЛАРАТИВНЫЙ ХАРАКТЕР ЯЗЫКА HASKELL

Наличие в Haskell такого большого количества разнообразных списочных функций, в то время когда другие языки вполне обходятся итераторами и циклами for, может вызвать удивление. Идея здесь в том, чтобы вместо явного поэлементного преобразования списка объявлять преобразования в качестве высокоуровневой абстракции. Языки, поддерживающие подобную идею, к числу которых относится и Haskell, называются декларативными.

В Haskell-программировании обычно считают, что этот более высокий уровень абстракции вредит производительности. Однако компиляторы декларативных языков способны на оптимизацию в довольно широком диапазоне, поскольку, сохраняя исходное поведение кода, они могут изменять код многими дополнительными способами. Типичный пример кода может выглядеть так: map f . map g. При выполнении этого кода выполняются многочисленные проходы по данным, но он может быть спокойно преобразован компилятором в код map (f . g), делающий то же самое за один проход.

Списки кортежей

Еще одно семейство списочных функций ориентировано на списки, содержащие кортежи. Все эти списочные типы в конечном счете основаны на каком-нибудь экземпляре типа `[(a,b)]`. Как уже упоминалось, сравнения по умолчанию проводятся

над кортежами лексикографически. Функции множеств, например `nub`, и функции сортировки, такие как `sort`, работают аналогичным образом.

Ранее в книге давалось задание на написание функции, преобразующей два списка в список кортежей. Эта функция известна под именем `zip`, поскольку она чередует элементы подобно застежке-молнии (zipper). Одним из примеров использования функции `zip` является включение позиции каждого элемента сразу же за самим элементом. Например, применяя `zip` к списку `['a', 'b', 'c']`, можно получить `[(1,'a'),(2,'b'),(3,'c')]`. Фактически происходит чередование элементов исходного списка со списком, начинающимся с числа 1 и доходящим до числа, представленного длиной исходного списка. Рассмотрим две стороны застежки-молнии, одна из которых является списком чисел, а вторая — списком символов. По мере того как бегунок молнии будет подниматься вверх, каждое число связывается с символом.

В качестве примера давайте определим функцию `enum`, генерирующую список чисел:

```
enum :: Int -> Int -> [Int]
enum a b | a > b = []
enum a b = a : enum (a+1) b
```

Входящая в модуль `Prelude` функция `length` возвращает количество содержащихся в списке элементов. Имея эти два ингредиента, можно создать искомую функцию:

```
withPositions :: [a] -> [(Int,a)]
withPositions list = zip (enum 1 $ length list) list
```

Чтобы составить списки для типов, обладающих исходной упорядоченностью, таких как целые числа или символы, существует особый инструмент, который называется *диапазоном* (range). Он имеет синтаксис `[ a .. b ]` для получения списка со всеми элементами между `a` и `b`, включая сами эти элементы. Например, вместо нашей функции `enum` можно написать следующий код:

```
withPositions list = zip [1 .. length list] list
```

Существует также функция `unzip`, совершающая действие, обратное действию функции `zip`, выдавая из списка кортежей два списка. Давайте, к примеру, разобьем список пар на страны и их столицы:

```
*Chapter3.Lists> unzip [("France","Paris"),("Spain","Madrid"),("Portugal","Lisbon")]
(["France","Spain","Portugal"],["Paris","Madrid","Lisbon"])
```

В последнем примере показан один из возможных вариантов использования списка кортежей — реализация проецирования между ключами и значениями. Список, имеющий такие характерные черты, называется *ассоциативным списком* (association list) и является в функциональном программировании хорошо известной структурой. Функция `lookup` позволяет вести поиск значения, связанного с конкретным ключом. Здесь опять возможность того, что ключ найден не будет, подразумевает, что возвращаемое значение заключено в «оболочку» `Maybe`[1]:

[1] Выражение «заключено в оболочку Maybe», означает, что вместо значения типа `τ` мы используем значение типа Maybe `τ`. Кроме того, предполагается, что обычно функция будет возвращать значение, построенное с помощью Just.

```
*Chapter3.Lists> lookup "Spain" [("France","Paris"),("Spain","Madrid"),("Portugal",
"Lisbon")]
Just "Madrid"
*Chapter3.Lists> lookup "UK" [("France","Paris"),("Spain","Madrid"),("Portugal",
"Lisbon")]
Nothing
```

ВНИМАНИЕ

Мы уже видели, как список может быть использован для представления множеств и проекций. Однако подобные реализации весьма неэффективны, поскольку они требуют для большинства операций перебора элементов списка. В следующей главе мы рассмотрим другие контейнеры, например из модулей Data.Set и Data.Map. Эти контейнеры главным образом ориентированы на строго конкретные варианты применения с учетом достижения высокой производительности.

Формирователи списков

Тот факт, что многие списочные функции включены в стандартную библиотеку, а большинство из них входят даже в модуль `Prelude` (и, стало быть, по умолчанию доступны любому исходному Haskell-коду), подтверждает важность списков в функциональном программировании. Мы уже видели, как для сугубо декларативного стиля программирования предоставляется композиция функций с пошаговым определением преобразований. Например, функция `doubleOdds` для вычисления удвоенных значений всех имеющихся в списке `list` нечетных чисел записывается следующим образом:

```
doubleOdds :: [Integer] -> [Integer]
doubleOdds list = map (*2) $ filter odd list
```

Однако если вы еще помните уроки алгебры, у математиков есть весьма сжатый, но интуитивно понятный язык для работы с множествами. Предыдущий пример может быть записан в системе записи множеств так:

$$\{2x \mid x \in \text{list.odd}(x)\}$$

Разработчикам Haskell также нравится этот синтаксис, поэтому для его имитации они предложили синтаксис формирователей списков. При использовании этого синтаксиса наш пример приобретет следующий вид:

```
doubleOdds list = [ 2 * x | x <- list, odd x ]
```

В сигнатуре формирователей списков есть две части, разделенные вертикальной чертой | и заключенные в квадратные скобки. Первая часть представляет собой *выражение*, которое определяет преобразование, применяемое ко всем возвращаемым элементам, а вторая часть составлена из перечня *спецификаторов*, указывающих, откуда берутся элементы и какие к ним применяются ограничения.

Первой разновидностью спецификаторов являются генераторы вида `e <- list`. Они показывают, какие элементы из списка `list` будут обрабатываться, и во всем остальном формирователе списка на каждый из элементов будет ссылка как на `e`. При желании часть `e` может быть образцом, определяющим, что будут

включены только те значения, которые с ним сопоставимы. Например, с помощью следующего кода можно получить имена клиентов всех правительственных организаций:

```
*Chapter3.Comprehensions> [ clientName x | x@(GovOrg _ _) <- listOfClients ]
["NTTF"]
```

В формирователе списка может быть несколько генераторов. Проще всего несколько генераторов реализовать путем итерации в двух ничем не связанных друг с другом разных списках, получая все возможные комбинации элементов, поступающих из каждого списка. Такой результат из всех возможных комбинаций называется *произведением* (product) этих списков. В качестве примера можно привести следующий код, в котором произведение двух списков применяется к решению задачи генерации таблиц умножения от 1 до 4:

```
*Chapter3.Comprehensions> [(x,y,x*y) | x <- [1 .. 4], y <- [1 .. 10]]
[(1,1,1),(1,2,2),(1,3,3),(1,4,4),(1,5,5),(1,6,6),(1,7,7),(1,8,8),(1,9,9),(1,10,10)
,(2,1,2),(2,2,4),(2,3,6),(2,4,8),(2,5,10),(2,6,12),(2,7,14),(2,8,16),(2,9,18),(2,10,20)
,(3,1,3),(3,2,6),(3,3,9),(3,4,12),(3,5,15),(3,6,18),(3,7,21),(3,8,24),(3,9,27),(3,10,30)
,(4,1,4),(4,2,8),(4,3,12),(4,4,16),(4,5,20),(4,6,24),(4,7,28),(4,8,32),(4,9,36),(4,10,40)]
```

Однако генератор может также зависеть от других значений в формирователе, в частности от элемента другого генератора. Например, может понадобиться перечислить все возможные костяшки домино. Однако, как вы знаете, что если уже учтена костяшка `(1,6)`, то костяшка `(6,1)` будет той же самой и ее учитывать не нужно. Способ вычисления правильного результата заключается в получении для каждого первого компонента в списке чисел в костяшках домино только тех значений второго компонента, которые равны ему или больше его по значению. Тогда вариант `(6,1)` будет исключен, поскольку 1 меньше 6. Такой подход реализован в следующем коде:

```
*Chapter3.Comprehensions> [(x,y) | x <- [0 .. 6], y <- [x .. 6]] [(0,0),
(0,1),(0,2),(0,3),(0,4),(0,5),(0,6), (1,1),(1,2),(1,3),(1,4),(1,5),(1,6),
(2,2),(2,3),(2,4),(2,5),(2,6), (3,3),(3,4),(3,5),(3,6), (4,4),(4,5),(4,6),
(5,5),(5,6), (6,6)]
```

И наконец, элемент, находящийся в списке, сам может быть списком, что позволяет ему находиться в правой стороне спецификатора. Имея список слов (вспомним, что строка сама по себе является списком символов), вы можете объединить все слова и вывести их в верхнем регистре путем проведения двойной итерации:

```
*Chapter3.Comprehensions> import Data.Char
*Chapter3.Comprehensions Data.Char> [ toUpper c | s <- ["This","is","a","list"], c
<- ' ':s ]
" THIS IS A LIST"
```

Бывает, что внутри формирователя списков нужно задать локальные привязки, — обычно это делается, чтобы сделать код понятнее. Синтаксис у этой второй формы спецификаторов похож на синтаксис, применяемый в выражениях: `let` *b* `=` *выражение*. Например, может возникнуть заинтересованность в вычислении норм списка векторов, представленных в виде кортежей[1]:

[1] Нормой вектора (x,y) является $\sqrt{x^2 + y^2}$.

```
*Chapter3.Comprehensions> [ sqrt v | (x,y) <- [(1,2),(3,8)], let v = x*x+y*y ]
[2.23606797749979,8.54400374531753]
```

И наконец, формирователи списков позволяют фильтровать некоторые элементы посредством *предохранителей* (guards). Предохранители являются третьей формой спецификаторов и синтаксически представляют собой просто вызов предиката. В возвращаемый список будут попадать только те элементы, которые удовлетворяют условию предохранителя. Предохранители позволяют выразить инвариант для костяшек домино по-другому:

```
*Chapter3.Comprehensions> [(x,y) | x <- [1 .. 6], y <- [1 .. 6], x <= y]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(2,2),(2,3),(2,4),(2,5),(2,6),
(3,3),(3,4),(3,5),(3,6),(4,4),(4,5),(4,6),(5,5),(5,6),(6,6)]
```

ПРИМЕЧАНИЕ

Если вы знаете язык Scala, то формирователи списков в Haskell покажутся вам знакомыми. Изменения касаются только синтаксиса: [e | q] превращается в for (q) yield e, а генераторы записываются точно так же. Локальные привязки вводятся без какого-либо ключевого слова, а вот предохранителям должно предшествовать ключевое слово if.

До сих пор мы рассматривали формирователи списков как нотацию, пришедшую из математической формы записи множеств. Однако если присмотреться, можно уловить некоторое сходство с SQL-запросами. Нотацию `[ x | x <- list, b x ]` можно встретить в SQL-запросе в виде `select x from list where b x`. Однако чтобы получить полноценный язык запросов, понадобятся еще группировка и сортировка. И здесь нас непременно обрадует наличие в GHC таких операций. Вам понадобится только включить расширение `TransformListComp`.

Первым спецификатором, предоставляемым расширением `TransformListComp`, является `then`. Спецификатор `then f` выполняет преобразование входного списка с применением функции `f` к результату формирования списка до указанной позиции. Ограничение состоит в том, что у функции `f` должен быть тип `[a] -> [a]`, чем и сдерживается ее применение. И тем не менее мы можем воспользоваться этим спецификатором для изменения порядка следования элементов списка на обратный:

```
*Chapter3.Comprehensions> :set -XTransformListComp
*Chapter3.Comprehensions> [x*y | x <- [-1,1,-2], y <- [1,2,3], then reverse]
[-6,-4,-2,3,2,1,-3,-2,-1]
```

Более эффективным расширением является `then f by e`, которое должно преобразовать список в зависимости от значения некоторого выражения. Наиболее часто это требуется для сортировки списка. Прежде чем добиться желаемого результата, нужно импортировать модуль `GHC.Exts`, в котором содержится функция `sortWith`. После этого для сортировки в зависимости от значений в `v` нужно включить спецификатор `then sortWith by v`. Можно вернуть предыдущий список, но теперь уже отсортированный по значениям `x`:

```
*Chapter3.Comprehensions> import GHC.Exts
*Chapter3.Comprehensions GHC.Exts> [x*y | x <- [-1,1,-2], y <- [1,2,3], then
sortWith by x]
[-2,-4,-6,-1,-2,-3,1,2,3]
```

И последнее расширение касается группировки. Для него используется синтаксис `then group by e using f`, где `f` — это функция типа `(a -> b) -> [a] -> [[a]]`. Чаще всего функция `groupWith`, также находящаяся в модуле `GHC.Exts`, используется с целью вычисления `e` для каждого элемента в списке и группировки вместе тех значений, для которых `e` возвращает одинаковый результат. После спецификатора группировки все предыдущие привязки рассматриваются в качестве списков, составленных из предыдущих элементов. Это весьма важное положение, поскольку все преобразования группируемых элементов к этому моменту должны быть уже сделаны. Во многих случаях все группируемые элементы будут равны друг другу, поэтому GHC предоставляет функцию `the`, которая просто извлекает из списка один элемент.

Например, числа из предыдущего примера можно сгруппировать по признаку их принадлежности к положительным или отрицательным числам:

```
*Chapter3.Comprehensions GHC.Exts> :{
*Chapter3.Comprehensions GHC.Exts| [ (the p, m) | x <- [-1,1,-2]
*Chapter3.Comprehensions GHC.Exts| , y <- [1,2,3]
*Chapter3.Comprehensions GHC.Exts| , let m = x*y
*Chapter3.Comprehensions GHC.Exts| , let p = m > 0
*Chapter3.Comprehensions GHC.Exts| , then group by p using groupWith ]
*Chapter3.Comprehensions GHC.Exts| :}
[(False,[-1,-2,-3,-2,-4,-6]),(True,[1,2,3])]
```

Обратите внимание на то, как в этом примере кода вычисляется произведение элементов перед группировкой, для чего используется выражение `let m = x*y`. Затем происходит группировка на основе значения `m > 0`, и в этот момент вы получаете список:

```
[([False,False,False,False,False,False],[-1,-2,-3,-2,-4,-6]),([True,
True,True],[1,2,3])]
```

И наконец, для объединения первых компонентов в один элемент применяется функция `the`.

Чтобы закрепить в сознании идею формирования списков, давайте попробуем построить формирователь списков для анализа клиентов вашего предприятия. Возможно, вы помните, что у каждой компании, фигурирующей в базе данных, может быть более одного контактного лица. Замысел состоит в том, чтобы сгруппировать все записи, принадлежащие одной компании, по должностям, а затем отсортировать компании по номерам записей. Эти задачи решаются с помощью следующего кода:

```
companyAnalytics :: [Client a] -> [(String, [(Person, String)])]
companyAnalytics clients = [ (the clientName, zip person duty)
                           | client@(Company { .. }) <- clients
                           , then sortWith by duty
                           , then group by clientName using groupWith
                           , then sortWith by length client
                           ]
```

ПРИМЕЧАНИЕ

Если вы занимались программированием на языке C#, то могли заметить, что синтаксис формирователей списков похож на синтаксис выражений запросов в C# 3.0.

GHC поддерживает еще одно расширение, называемое *параллельным формированием* (parallel comprehension). При параллельном формировании списков обеспечивается одновременный перебор элементов нескольких списков, что невозможно в SQL-запросах. Это расширение включается с помощью специального распознаваемого компилятором комментария `ParallelListComp`. Использование этой функциональной возможности позволяет установить в формирователе списка более одной ветви спецификаторов, каждая из которых отделяется от других вертикальной чертой (`|`). Вместо вложенных итераций происходит поочередный выбор элементов из двух списков, результат которого становится доступен выражению. В следующем примере производится перемножение пар чисел, где каждый компонент берется из другого списка. Сравните результат при использовании обычного вложения и при поочередном выборе элементов из двух списков:

```
*Chapter3.Comprehensions> :set -XParallelListComp
*Chapter3.Comprehensions> [ x*y | x <- [1,2,3], y <- [1,2,3] ] -- обычное вложение
[1,2,3,2,4,6,3,6,9]
*Chapter3.Comprehensions> [ x*y | x <- [1,2,3] | y <- [1,2,3] ] – поочередный выбор
[1,4,9]
```

Haskell-оригами

Оригами — это японское искусство складывания фигурок из бумаги. Мы уже знаем, как происходит свертка списков. В этом разделе мы еще раз взглянем на свертку и встретимся с ее коллегой — разверткой. Цель всего этого заключается в более глубоком усвоении структуры списочных функций, в осознании того факта, что все множество уже рассмотренных нами функций может быть отнесено к весьма компактному семейству схем. Поскольку эти схемы основаны на функциях свертки и развертки, они известны как «Haskell-оригами». В этом разделе содержится дополнительный и более сложный материал. Поэтому если с первого раза будет что-то непонятно, волноваться не стоит, просто через некоторое время попробуйте разобраться в нем снова.

Давайте начнем с одного наблюдения: эффективность свертки превосходит ваше воображение. С использованием свертки `foldr` можно написать практически все списочные функции. Например, можно написать функцию `filter` в виде свертки, аккумулирующей значения списка:

```
filterAsFold :: (a -> Bool) -> [a] -> [a]
filterAsFold p = foldr (\x l -> if p x then x : l else l) []
```

Но как убедиться в том, что определение функции `filter` при наличии обычного сопоставления с образцом и рекурсии списков эквивалентно указанному определению со сверткой? Ответ заключается в *эквациональном рассуждении* (equational reasoning) — технологии формальной проверки кода на предмет отождествления функций. В данном конкретном случае мы воспользуемся универсальным свойством функции `foldr`, согласно которому доказательство того, что `g = foldr f v`, эквивалентно доказательству следующего набора уравнений[1]:

[1] Большинство идей взято из статей «A tutorial on the universality and expressiveness of fold» Грэма Хаттона (Graham Hutton) и «Origami programming» Джереми Гиббонса (Jeremy Gibbons).

```
g [] = v
g (x:xs) = f x (g xs)
```

Давайте воспользуемся этим свойством в нашем примере. Выражение `filterAsFold p` согласно этому универсальному свойству эквивалентно следующему набору:

```
filterAsFold p [] = []
filterAsFold p (x:xs) = (\x l -> if p x then x : l else l) x (filterAsFold p xs)
```

Во втором уравнении можно применить в отношении аргументов следующую абстракцию:

```
filterAsFold p (x:xs) = if p x
then x : (filterAsFold p xs)
else (filterAsFold p xs)
```

Вуаля! В итоге мы вернулись к коду, использовавшемуся для определения свертки `filter`, в которой имели место сопоставление с образцом и рекурсия. Каждый из шагов является простым применением универсального свойства или замены аргументов в теле функции, поэтому мы можем гарантировать, что оба определения эквивалентны.

Мы можем также в понятиях `foldr` определить функцию `map`:

```
mapAsFold :: (a -> b) -> [a] -> [b]
mapAsFold f = foldr (\x l -> f x : l) []
```

Есть еще одно универсальное свойство, имеющее отношение к `map` и `foldr`. Согласно этому свойству, `foldr f v . map g = foldr (\x xs -> f (g x) xs) v` (то есть операцию `map` можно ввести непосредственно в свертку). Чтобы это доказать, давайте приступим к расширению определения `map` с использованием универсального свойства. Обратимся для ясности к `h = foldr f v . map g`:

```
h []     = foldr f v (map g [])     = foldr f v []
h (x:xs) = foldr f v (map g (x:xs)) = foldr f v ( (\x l -> g x : l) x (map g xs) )
```

Во втором уравнении можно в тело функции ввести аргументы, а затем к каждому из аргументов применить универсальное свойство `foldr`:

```
h []     = v
h (x:xs) = foldr f v ( g x : map g xs) )
         = f (g x) ( foldr f v (map g xs) )
         = f (g x) (h xs) = (\y ys -> f (g y) ys) x xs
```

На последнем шаге указано, что `h l = foldr f v (map g l)`. И опять в универсальном свойстве свертки можно увидеть образец, позволяющий нам сказать, что:

```
foldr f v . map g = h = foldr (\y ys -> f (g y) ys) v
```

Этот результат и был нашей конкретной целью. Если у вас нет полной уверенности в наших рассуждениях, попробуйте расписать все поэтапно с помощью карандаша и бумаги. На каждом этапе обращайте внимание на то, как применяются правила игры. Если вы уверены в деталях, попробуйте доказать общий закон для функции `map`:

```
map f . map g = map (f . g).
```

ПРИМЕЧАНИЕ

Осознание этих законов может показаться простой теоретической игрой. Однако у них есть весьма важное применение к программам на языке Haskell, поскольку они используются компилятором для преобразования кода в его более эффективную версию, гарантирующую при этом точное воспроизведение исходного поведения. Например, код map(f . g) осуществляет последовательный перебор элементов списка всего один раз, а код map f . map g делает это дважды и нуждается в промежуточной структуре данных в памяти. Поэтому компилятор пытается выполнить замену каждого экземпляра последнего фрагмента кода предыдущим фрагментом.

До сих пор речь шла только о свертке, использовавшей списки для предоставления одного значения. Однако есть еще понятие *развертки* (unfold), когда списки создаются из какого-нибудь начального значения. Как и в случае со свертками, существует как правая, так и левая развертка. Наше основное внимание будет уделено функции правой развертки `unfoldr`, которая доступна в модуле `Data.List`. Давайте начнем с выяснения ее типа:

```
*Chapter3.Origami Data.List> :t unfoldr
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

Алгоритм развертки следующий: начать с начального значения типа `b`. Применить функцию, заданную в качестве первого аргумента. На выходе можно получить два разных значения: `Nothing` сигнализирует о необходимости прекращения генерирования элементов, `Just (x, s)` прикрепляет `x` к новому списку и продолжает процесс с новым начальным значением `s`. Давайте, к примеру, создадим список от `n` до `m`: функция на каждом этапе должна выдавать число, увеличивая его на единицу для следующей итерации, и останавливаться, когда начальное значение станет больше `m`. Вот соответствующий код:

```
enumUnfold :: Int -> Int -> [Int]
enumUnfold n m = unfoldr (\x -> if x > m then Nothing else Just (x, x+1)) n
```

Пошаговое выполнение этого кода проиллюстрировано на рис. 3.4.

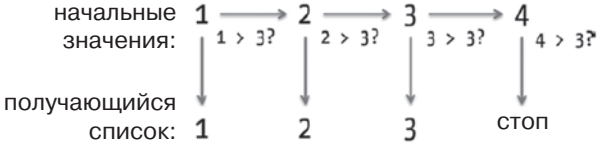


Рис. 3.4. Шаги вычислений для enumUnfold 1 3

Еще один алгоритм, который может быть выражен в форме развертки, — сортировка списка по минимальным значениям. При такой сортировке производятся серии действий, в каждой из серий во входном списке ищется элемент с минимальным значением, после чего он извлекается из этого списка и добавляется в выходной список, который в конечном итоге будет отсортирован. Для реализации этого алгоритма с помощью развертки список, содержащий сортируемые элементы, будет использоваться в качестве начального значения. На каждом шаге из списка берется элемент с минимальным значением, а предыдущий список без этого элемента становится новым начальным значением. Когда в качестве начального значения

оказывается пустой список, генерирование новых элементов должно останавливаться. В Haskell-коде это выглядит следующим образом:

```
minSort :: [Integer] -> [Integer]
minSort = unfoldr (\case [] -> Nothing
xs -> Just (m, delete m xs) where m = minimum xs)
```

ПОЧЕМУ СВЕРТКИ И РАЗВЕРТКИ ИМЕЮТ ДВОЙСТВЕННЫЙ ХАРАКТЕР?

Понятия свертки и развертки имеют двойственный характер, но чем это подтвердить? Ключевым моментом является то, что функция unfoldr возвращает либо Nothing для остановки, либо Just для продолжения, а функция foldr для пустых и общих случаев получает разные аргументы.

Исходное значение и функция объединения могут быть собраны вместе в единую функцию типа Maybe (a,b) -> b, которая вернет исходное значение, если ей не дано ничего (nothing), или выполнит объединение в случае Just.

```
foldr2 :: (Maybe (a,b) -> b) -> [a] -> b
foldr2 f [] = f Nothing
foldr2 f (x:xs) = f $ Just (x, foldr2 f xs)

mapAsFold2 :: (a -> b) -> [a] -> [b]
mapAsFold2 f = foldr2 (\case Nothing -> []
                             Just (x,xs) -> f x : xs)
```

Теперь в этих двух функциях можно увидеть отраженные типы:

```
foldr2  :: (Maybe (a,b) -> b) -> [a] -> b
unfoldr :: (b -> (Maybe (a,b)) -> b -> [a]
```

Эту двойственность можно считать весьма красивой и элегантной, к тому же это яркий пример того, как функции высшего порядка позволяют находить связи между различными абстракциями.

Выводы

Эта глава завершилась разделом, посвященным списочным оригами. Но этому предшествовало рассмотрение многих других концепций, связанных со списками и многократным использованием кода:

- Мы рассмотрели идею параметрического полиморфизма, позволяющего определять функции, работающие с несколькими типами данных, а также записывать типы данных, содержащие значения различных типов.
- Мы научились использовать функции в качестве аргументов или возвращаемых значений, поднявшись до функций высшего порядка, существенно расширяющих возможности многократного использования кода.
- Мы усвоили понятие анонимных функций как средства написания кода непосредственно на месте, что избавляет от необходимости определения новой функции всякий раз, когда нужно передать эту функцию в качестве аргумента.
- Мы узнали, что идея функций, возвращающих функции, обогащает язык Haskell, и познакомились с синтаксисом их частичного применения.

- Мы познакомились с бесточечным стилем программирования, который поощряет использование комбинаторов между функциями с целью написания более лаконичного кода. Основное внимание при этом было уделено объединяющему оператору `(.)`.
- Мы рассмотрели импорт определений из других модулей проекта и их экспорт в другие модули. В частности, мы узнали, как скрывать определения для образца проектирования «интеллектуальные конструкторы».
- Мы изучили наиболее интересные функции модуля `Data.List`, использующие важное понятие свертки.
- Во многих случаях формирователи списков предлагают интуитивно понятный синтаксис преобразования списков; мы узнали основные характеристики формирователей списков, а также конкретные GHC-расширения, обеспечивающие сортировку и группировку списков по типу SQL-запросов.
- И наконец, мы увидели, что в основе многих списочных функций лежат свертки и развертки, а также выяснили, как и зачем их использовать.

Контейнеры и классы типов

4

Мы уже знаем, как параметрический полиморфизм и функции высших порядков помогают при создании абстракций. В данной главе представлена новая разновидность полиморфизма — *специальный полиморфизм* (ad-hoc polymorphism), который занимает нишу между параметрическим полиморфизмом и его полным отсутствием. С его помощью можно выразить тот факт, что определенные типы проявляют общее для них поведение. Попутно мы узнаем, как Haskell позволяет выполнять сложение разных числовых типов, сохраняя при этом строгую систему типов.

Примеры, приводимые в данной главе, работают благодаря *контейнерам*. Контейнер — это имя, которое дается любой структуре данных, такой как списки или деревья, предназначенной для хранения элементов других типов. Помимо написания собственных реализаций двоичных деревьев с механизмом кэширования, мы познакомимся с реализациями, доступными в хранилище пакетов *Hackage*, принадлежащем Haskell-сообществу. Это позволит лучше узнать возможности Cabal и понять, как они могут использоваться для управления не только проектами, но и их зависимостями. Кроме Hackage, Haskell-сообщество предлагает массу различных инструментов поиска кода и документации, и в данной главе мы познакомимся с одним из таких инструментов под названием Hoogle.

При использовании и реализации контейнеров мы встретимся с множеством новых *образцов* (patterns) проектирования. И с этого момента такая ситуация станет доминирующей: после знакомства с какими-либо полезными типами ваше внимание будет привлечено к тому, что их объединяет. В частности, в данной главе вводятся понятия функторов, свертываемых классов и моноидов.

Использование пакетов

До сих пор мы использовали функции и типы данных из пакета `base`. Однако в других пакетах также доступно множество функций. В этой главе мы в первую очередь выясним, как управлять пакетами. Затем мы сосредоточим свое внимание на изолированных программных средах, благодаря которым можно разрабатывать проект, никак не влияющий ни на другие проекты, ни на общесистемную базу данных пакетов.

Управление пакетами с помощью Cabal и EclipseFP

Пакет (package) является единицей распространения кода, воспринимаемой таким инструментом создания проектов, как Cabal, с которым мы уже познако-

мились. Каждый пакет идентифицируется по имени и номеру версии и включает в себя множество модулей. Дополнительно пакет может включать в себя ряд метаданных, например описаний или способов связи с поддерживающей его организацией. Разумеется, все ранее созданные вами с помощью Cabal проекты являются пакетами.

ПРИМЕЧАНИЕ

В EclipseFP вместо терминологии Cabal применяется терминология Eclipse, в которой пакеты называют проектами.

Пакетами можно управлять вручную, но есть более удобный способ их получения и установки в вашей системе, делающий их доступными в проектах. Если Cabal определяет, что пакет в вашей системе недоступен, Cabal связывается с базой данных пакетов *Hackage*, которая находится по адресу http://hackage.haskell.org, после чего загружает, компилирует и устанавливает соответствующий пакет. Изначально база данных Hackage была задумана просто как хранилище пакетов, но теперь она предоставляет ряд дополнительных сервисов, например по созданию документации. Выгружать новые пакеты в Hackage разрешается всем, кто имеет в этом хранилище учетную запись. Учитывая довольно высокую активность Haskell-сообщества, это означает доступность широкого спектра библиотек.

ИССЛЕДУЙТЕ

Перейдите на веб-страницу Hackage и щелкните на ссылке Packages. Уделите время просмотру различных категорий и пакетов, доступных в каждой из категорий. Затем попробуйте в категории Data Structures найти пакет containers и щелкнуть на нем. Перед вами откроется список экспортируемых модулей и их описание. Очень важно освоиться на этом сайте, поскольку он является основным входом в мир Haskell-библиотек.

Ваша локальная копия Cabal поддерживает список всех пакетов, доступных в удаленном хранилище Hackage. Но этот список пакетов автоматически не обновляется: для загрузки новой версии нужно время от времени делать явный запрос в адрес Cabal. При использовании командной строки это делается с помощью команды `cabal update`:

```
$ cabal update
Downloading the latest package list from hackage.haskell.org
```

Как и в предыдущих главах, мы параллельно будем управлять пакетами в среде разработки EclipseFP. Если вы находитесь в окне Haskell, то сможете заметить вкладку Cabal Packages (Cabal-пакеты), которая изначально располагается в самой правой позиции (рис. 4.1). В правом верхнем углу вкладки имеются две кнопки. Действие первой кнопки (на рисунке она обведена кружком), Update package list (Обновить список пакетов), эквивалентно действию предыдущей команды.

В этой вкладке приводится ряд сведений с веб-страницы Hackage, а именно информация о доступных версиях каждого пакета и метаданные, связанные с каждым из них. Кроме того, чтобы открыть окно с документацией, можно щелкнуть на ссылке More information (Дополнительные сведения), хотя для этого должно быть установлено соединение с Интернетом.

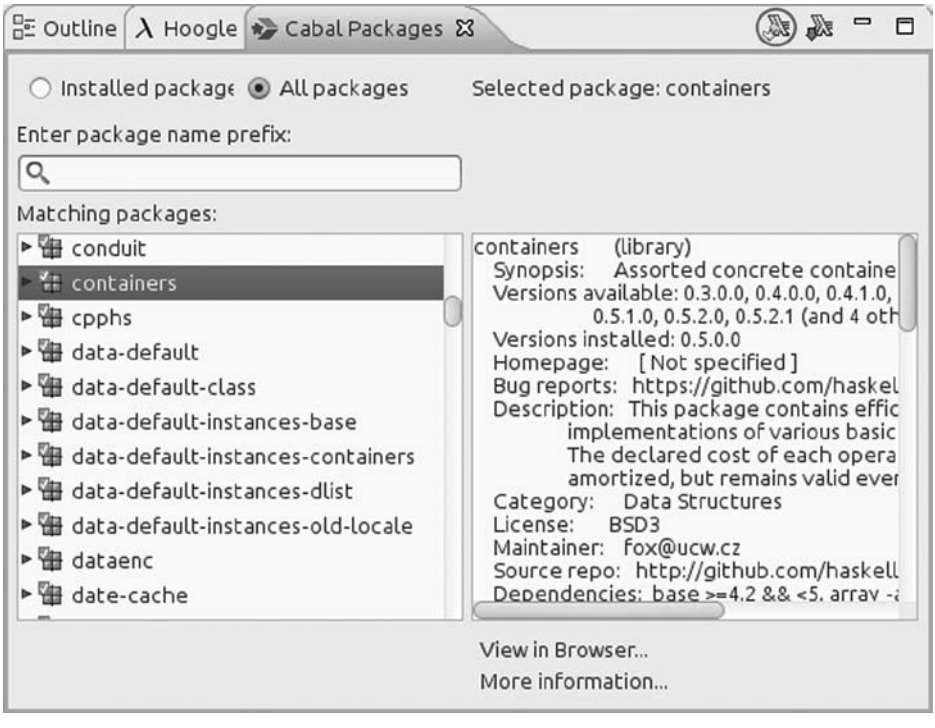


Рис. 4.1. Вкладка Cabal Packages в EclipseFP

А теперь давайте разберемся, как установить пакет со страницы Hackage. Когда нужный пакет будет найден, просто выполните в командной строке команду `cabal install` *имя_пакета*. Если же вы предпочитаете работать в среде EclipseFP, щелкните сначала на имени пакета на вкладке Cabal Packages (Cabal-пакеты), а затем — на второй справа кнопке этой вкладки, которая называется Install selected package (Установить выбранный пакет). В последнем случае, как показано на рис. 4.2, появится запрос о том, хотели бы вы установить пакет только для текущего пользователя или для всех пользователей в системе. Установка для текущего пользователя подразумевается по умолчанию и не требует каких-либо дополнительных разрешений

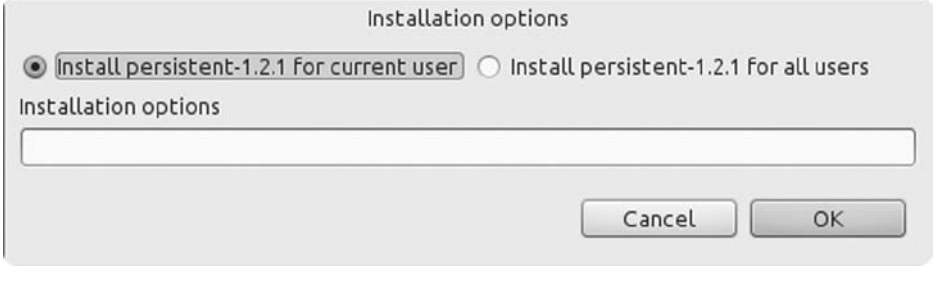


Рис. 4.2. Установка пакета в среде EclipseFP

системы. Попробуйте установить пакет `persistent`, предоставляющий некий уровень доступа к базе данных; с этим пакетом мы будем работать при изучении последующих глав.

Вот как выглядит часть данных, выводимых программой Cabal при загрузке, компиляции и установке пакета `persistent`. Чтобы не загромождать вывод ненужными подробностями, сообщения со 2 по 21 были опущены (это сообщения о компиляция файлов).

```
Resolving dependencies...
... # Вывод о зависимостях опущен
Downloading persistent-1.2.1...
Configuring persistent-1.2.1...
Building persistent-1.2.1...
Preprocessing library persistent-1.2.1...
[ 1 of 22] Compiling Database.Persist.Class.PersistConfig ( Database/Persist/Class/
PersistConfig.hs, dist/build/Database/Persist/Class/PersistConfig.o )
... # Сообщения о компиляции файлов со 2 по 21 опущены
[22 of 22] Compiling Database.Persist.Sql ( Database/Persist/Sql.hs, dist/build/
Database/Persist/Sql.o )
In-place registering persistent-1.2.1...
Installing library in /home/serras/.cabal/lib/persistent-1.2.1/ghc-7.6.3
Registering persistent-1.2.1...
Installed persistent-1.2.1
```

Как можно заметить, был установлен не только сам пакет, но и все, от чего зависит его работа, то есть все необходимое, но пока недоступное на вашей системе. Распространение идет в виде исходного кода, а это означает, что полученный из хранилища пакет нуждается в компиляции. Следовательно, нужно настроиться на продолжительное ожидание окончания компиляции. Кроме того, имейте в виду, что при наличии зависимостей режим работы немного изменяется: в том случае, если пакет `A` требует использования пакета `B`, а вам нужно задействовать модули из пакета `B`, потребуется явное включение именно этого пакета, даже если он фигурирует в перечне зависимостей пакета `A`.

ПРИМЕЧАНИЕ

Если нужна конкретная версия пакета, следует выполнить команду (со слешем между именем пакета и версией):

```
cabal install имя_пакета / версия
```

В случае использования EclipseFP нужно выбрать желаемую версию в списке на вкладке Cabal Packages. Если версия не указана, то и Cabal, и EclipseFP установят самую последнюю из версий.

Однако наиболее распространенным способом получения пакета является не его загрузка вручную, а его добавление в качестве зависимости другого пакета. Это достигается путем введения соответствующей строфы в свойство `build-depends` вашего файла с расширением .cabal. Зависимости пакета могут добавляться либо в строфы библиотеки, либо в строфы исполняемого файла. Давайте, к примеру, создадим новый проект для главы 4 — если используется EclipseFP, нужно в качестве вида проекта выбрать вариант Library (Библиотека) — и добавим в качестве зависимости пакет `containers`, поскольку он потребуется нам на протяжении всей главы. Соответствующие части файла описания пакета будут иметь следующий вид:

```
name:           chapter4
version:        0.1

library
  hs-source-dirs:  src
  build-depends:   base >= 4, containers
```

После добавления зависимости в строфу библиотеки при запуске для этого пакета команды `cabal build` будет получено следующее предупреждение:

```
./chapter4.cabal has been changed. Re-configuring with most recently used
options. If this fails, please run configure manually.
```

Для данной ситуации EclipseFP также предоставляет графическое средство. Откройте файл с расширением .cabal в Eclipse и перейдите на вкладку Library (Библиотека), которая должна быть похожа на ту, что изображена на рис. 4.3. В первый столбец, который озаглавлен как Dependencies, считываются зависимости, и в нем имеются кнопки как для добавления, так и для удаления имен пакетов. Если щелкнуть на кнопке в форме знака «плюс», появится список всех возможных зависимостей (взятый из базы данных Hackage), включающий небольшое описание каждой из них. Нужно просто выбрать необходимую зависимость и щелкнуть на кнопке OK.

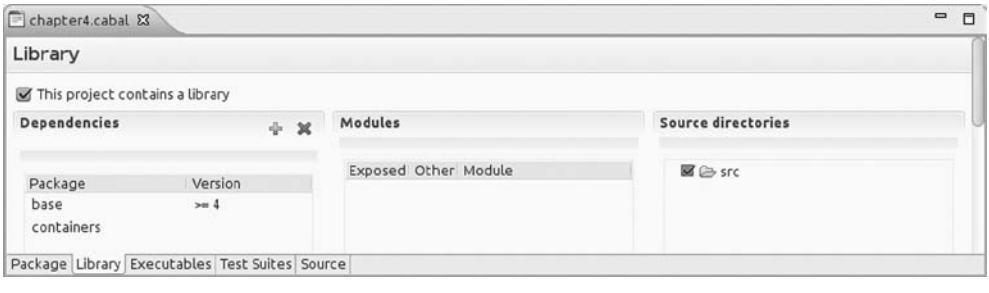


Рис. 4.3. Редактор Cabal-файла в EclipseFP

Как видно в редакторе, работающем в графическом режиме, зависимости могут ссылаться на пакет, а также указывать на ограничения, касающиеся требуемых версий. Версии имеют форму $a.b.c.d$, где каждый компонент представлен цифрой. Порядок имеет лексикографическую основу: для сравнения версий проверяется первый компонент (a), и только при его совпадении проверяется второй компонент. Если он также совпадает, аналогично проверяются следующие компоненты. Могут использоваться различные операторы сравнения: `==`, `>=`, `>`, `<=` и `<`, а также их комбинации, составленные с помощью оператора `&&` для конъюнкции и оператора `||` для альтернативных ограничений. При всей высокой эффективности системы ограничений обычно используется следующее основное правило: добавьте ограничение, требующее минимальной версии, с которой, как вам известно, ваш пакет компилируется и запускается (обычно это версия, установленная на вашей системе), а также ограничение, не допускающее перехода к следующей полноценной версии, то есть к следующему номеру $a.b$ в лексикографическом порядке. Например, на момент написания книги текущей версией пакета `containers` была

0.5.2.1, следовательно, следующей полноценной версией будет 0.6. Предлагаемое объявление зависимости будет иметь следующий вид:

```
containers >= 0.5.2.1 && < 0.6
```

ПОЛИТИКА ВЫПУСКА ВЕРСИЙ ПАКЕТОВ

Трактовка номеров версий для Haskell-пакетов долгое время не была стандартизована. Это затрудняло принятие решений о том диапазоне, который нужно было показать для конкретной зависимости. Но в недавнем прошлом появилось соглашение о политике управления версиями (Versioning Policy)[1]:

- Если тип или поведение какой-либо функции, типа данных, класса типов или экземпляра изменились либо эта функция, тип данных, класс типов или экземпляр были удалены, должен быть увеличен номер полноценной версии, то есть номер первых двух компонентов.
- Если же были внесены только некоторые дополнения, разрешается увеличить номер только остальных компонентов. Это правило сохраняется и для новых модулей, кроме тех, для которых высока вероятность конфликта с модулем в другом пакете.

Помимо этих рекомендаций, касающихся создателей пакетов, было также введено ранее упомянутое правило для указания зависимостей.

Как видите, имя пакета и его версия являются весьма важной частью файла Cabal-проекта. Кроме того, если ваш пакет предназначается для публикации в Hackage или имеет публичную доступность иного рода, также важно включить в него определенные метаданные. В файле описания пакета метаданные определяются высокоуровневыми свойствами. Обязательными являются только свойства name (имя) и version (версии), но зачастую включаются также свойства license (лицензия), author (автор), maintainer (лицо, отвечающее за поддержку), synopsis (небольшая аннотация), description (более обширное описание), homepage (домашняя страница проекта) и categories (список категорий в хранилище Hackage, в которые будет включен пакет). Для пакета chapter4 могут быть созданы следующие метаданные:

```
name:           chapter4
version:        0.1
cabal-version:  >=1.2
build-type:     Simple
author:         Alejandro Serrano
synopsis:       Package for chapter 4
maintainer:     Alejandro Serrano <my@email.com>
homepage:       http://haskell.great.is
```

ПРИМЕЧАНИЕ

Вам могут встретиться и дополнительные свойства, например cabal-version и build-type. Они предназначены для использования в том случае, если разработчик нуждается в настройке системы компоновки или в сохранении совместимости с более старыми версиями Cabal. Если в исходной версии эти свойства были созданы, то мы советуем оставить их в покое.

Для изменения наиболее важных из этих свойств пакета может быть использован также доступный в EclipseFP Cabal-редактор, работающий в графическом

[1] Просмотреть полный текст политики управления версиями пакетов (Package Versioning Policy) и найти обновления можно по URL-адресу http://www.haskell.org/haskellwiki/Package_versioning_policy

режиме. Свойства в редакторе представлены на первой вкладке Package (Пакет) (рис. 4.4).



Рис. 4.4. Изменение свойств пакета в EclipseFP

Владея данной информацией, можно приступить к созданию пакета для магазина. Точно выполните все задания в упражнении 4.1, необходимые для создания нового пакета.

УПРАЖНЕНИЕ 4.1. ПАКЕТ ДЛЯ МАГАЗИНА МАШИН ВРЕМЕНИ

Создайте новый пакет, который станет ядром нашего магазина машин времени, используя либо Cabal, либо EclipseFP. Поскольку магазин будет веб-приложением, сделайте этот пакет исполняемым. В качестве зависимостей добавьте пакеты containers и persistent (не забудьте следовать правилу выпуска версий), а затем сконфигурируйте проект и осуществите его сборку. Поэкспериментируйте с различными полями метаданных.

Изолированные программные среды

Изначально Cabal устанавливает пакеты для использования их текущим пользователем. Это означает, что при разработках все создаваемые вами или устанавливаемые в силу зависимостей пакеты сохраняются в одном и том же месте и являются общими для всех других проектов. При этом возникают две проблемы:

- У разных разработанных пакетов могут быть конфликтующие множества зависимостей. Например, вами может поддерживаться приложение, использующее последние стабильные версии зависимостей, и в то же время его улучшение в другой ветви нацеливается на новую версию, зависящую от менее стабильных библиотек.
- В рамках больших проектов очень часто разрабатывается сразу несколько пакетов. Если все они будут установлены таким образом, чтобы стать доступными пользователю, вы можете столкнуться с проблемами конфликта версий и засорения базы данных в целом.

В обоих случаях решение заключается в использовании *изолированных программных сред* (sandboxed environments), или так называемых *песочниц* (sandboxes), которые находятся в изоляции и хранят собственные множества пакетов.

Возможность работы в изолированных программных средах стала доступна в Cabal начиная с версии 1.18 (номер своей версии можно проверить, выполнив в команд-

ной строке команду `cabal --version`). Хотя эта версия появилась совсем недавно, есть позитивная новость для всех: до того как эта функциональная возможность была включена в Cabal, она была доступна в cabal-dev.

ПРИМЕЧАНИЕ

Если утилита cabal-dev на вашей машине недоступна, установите ее, выполнив в командной строке команду cabal install cabal-dev.

Если вы используете Cabal-команды, то сначала нужно создать для вашего проекта структуры песочниц. Это делается запуском в корневом каталоге проекта команды `cabal sandbox init`. С этого момента каждая выполняемая внутри данного проекта Cabal-команда будет действовать только внутри песочницы. В частности, если запустить команду `cabal install`, все зависимости проекта будут загружены и скомпонованы внутри этой песочницы.

Имеется также возможность использования в качестве песочницы папки, отличающейся от папки проекта. Это может пригодиться для совместного тестирования множества пакетов. О подобной ситуации уже упоминалось при описании важности изолированных программных сред. Единственное отличие при использовании общей папки песочницы заключается в том, что каждую песочницу проекта придется инициализировать путем ссылки на общую папку. Это делается с помощью следующей команды:

```
cabal sandbox init --sandbox=/путь/к/песочнице
```

Использование команд утилиты cabal-dev очень похоже на использование обычных Cabal-команд. Для этого нужно просто выполнять команды, начинающиеся со слова `cabal-dev`, а не `cabal`. Например, чтобы загрузить и установить зависимости и скомпилировать программу, нужно просто выполнить команду `cabal-dev install`.

Важным отличием является отсутствие необходимости в запуске специальной команды для инициализации песочницы: при первом же запуске любой команды утилиты cabal-dev будет создана песочница, предназначенная конкретно для данного проекта. Как и при использовании Cabal-команд, можно также задействовать общую песочницу для нескольких проектов. Однако путь к песочнице придется указывать при каждом вызове утилиты cabal-dev. Например, компиляция и установка в общую песочницу производится с помощью следующей команды:

```
cabal-dev install --sandbox=/путь/к/песочнице
```

В Cabal есть еще одна дополнительная возможность, позволяющая вести локальную базу данных Hackage, из которой будут браться зависимости, используемые в других проектах. Это пригодится для хранения исправленных или невыпущенных версий библиотеки, которая еще не попала в основное хранилище Hackage. Чтобы воспользоваться данной возможностью, к вашей песочнице нужно добавить *исходную папку* (source folder): место, в котором Cabal будет искать зависимости для вашего проекта. В корневом каталоге того проекта, в котором требуется наличие такой исходной папки, нужно выполнить в командной строке следующую команду:

```
cabal add-source /путь/к/папке/пакета
```

С этого момента Cabal при поиске пакетов будет использовать именно эту папку. Кроме того, Cabal будет отслеживать изменения файлов в этой папке, проводя при необходимости их перекомпиляцию. Функциональные возможности исходной папки доступны также командам утилиты cabal-dev. Похоже, на данный момент имеется два способа управления пакетами, принадлежащими одному и тому же проекту. Один из них связан с использованием общей песочницы, а другой заключается в наличии основного проекта, зависящего от других проектов, добавленных к исходным папкам. Обычно когда эти пакеты сильно связаны, а вы одновременно проводите тестирование и вносите изменения во все эти пакеты, предпочтительнее воспользоваться первым вариантом (общей песочницей). Но при явном применении библиотеки, отсутствующей в Hackage, интенсивной модификации которой не предвидится, лучше работать с исходной папкой.

EclipseFP также поддерживает утилиту cabal-dev[1]. В данном случае для каждого Haskell-проекта (который является пакетом) используется собственная песочница. Кроме того, EclipseFP замечает, когда проект в вашем рабочем пространстве ссылается на один из других проектов, выполняя нужные команды для поддержания общей работоспособности. Но этот режим по умолчанию выключен. Чтобы его включить, следует в меню Window (Окно) выбрать пункт Preferences (Предпочтения), в новом окне перейти к Haskell, затем — к Helper executables (Помощник исполняемых команд) и после этого — к cabal-dev. Если щелкнуть на второй кнопке (рядом с текстовым полем), то при правильной установке cabal-dev будет обнаружен правильный путь. После щелчка на кнопке OK все проекты будут изолированы.

Иногда возникает необходимость удалить все находящиеся в песочнице файлы, например, чтобы создать пакет с нуля. Для подобного удаления нужно выполнить команду `cabal sandbox delete`. Если изоляция организована не с помощью Cabal, а с помощью cabal-dev, вам потребуется лишь удалить всю папку .cabal-dev, находящуюся в папке песочницы (которая по умолчанию является корневой папкой вашего проекта, если посредством ключа `--sandbox` не указано иное).

Изоляция является весьма полезным решением, предотвращающим большинство проблем Haskell-программирования, возникающих из-за несовместимости версий библиотек. Вместо глобального множества зависимостей в каждом пакете могут использоваться его реально изолированные версии. Чтобы подчеркнуть важность применения сред такого рода, следует сказать, что разработчики Cabal думают о том, чтобы они устанавливались по умолчанию (то есть все запуски Cabal будут происходить в изолированной программной среде, если не указано обратное).

Контейнеры: проекции, множества, деревья и графы

В этом разделе мы собираемся рассмотреть ряд контейнерных типов, получивших широкое распространение в программировании. Как уже было сказано, *контейнер*

[1] Во время написания книги была реализована поддержка изолированности обычными средствами Cabal.

является типом данных, основной целью которого является хранение любого количества элементов одного типа. В частности, вам предстоит рассмотреть проекции, деревья, графы и множества. Все эти структуры могут быть реализованы с использованием списков и кортежей (например, мы уже видели, как для представления проекций могут применяться ассоциативные списки). Но у специализированных типов данных есть два преимущества:

- Они работают гораздо быстрее, поскольку специально разработаны для конкретного образца. Например, поиск в списке значения по ключу требует перебора всего списка, в то время как в проекции на это всегда тратится практически одинаковое время.
- Библиотеки, реализующие эти структуры, предоставляют функции, изначально предназначенные для конкретного применения каждой из них. Например, есть функции для обхода узлов или получения сильно связанных компонентов графа. Эти функции можно было бы реализовать при использовании списков, но в Haskell Platform они пока недоступны.

Все контейнеры, о которых пойдет речь, предоставляются пакетом `containers`, поэтому, чтобы проверить примеры, следует включить этот пакет в качестве зависимости, как это было показано в предыдущем разделе.

Проекции

Давайте начнем с *проекций* (maps), позволяющих эффективно связывать значения с ключами. Использовать повторяющиеся ключи не разрешается, поэтому если нужно изменить значение конкретного ключа, предыдущее значение будет заменено новым. В отличие от этого при реализации проекций с помощью ассоциативных списков, где основой служит список кортежей, вся ответственность за обслуживание такого инварианта ложится на вас.

Реализация проекций находится в модуле `Data.Map`. Однако имена многих функций этого модуля конфликтуют с именами функций из встроенного модуля `Prelude`. Поэтому при импорте модуля его, конечно же, нужно квалифицировать. Мы сделаем это с помощью имени `M`, то есть любое объявление из этого модуля будет предваряться этим префиксом. Сокращение квалификации до небольшого однобуквенного имени с целью экономии набираемого текста является весьма распространенной практикой. В последующих примерах предполагается, что модуль был импортирован с помощью такой строки кода:

```
import qualified Data.Map as M
```

Сам тип имеет вид `Map k a`. В качестве аргумента берется тип `k`, относящийся к ключам, которые будут индексировать значения типа `a`. Например, проекция между клиентами и списком товаров, приобретенных каждым клиентом, будет иметь тип `Map Client [Product]`. В примерах мы будем работать с несложными проекциями, начиная от строк и заканчивая целыми числами, которые намного короче.

В предыдущих главах для создания списков был введен специальный синтаксис: либо использование конструктора `[]` для пустого списка, либо перечисление

элементов в квадратных скобках. В Haskell нет специального синтаксиса для проекций или для экспорта любого из конструкторов данных проекций. Вместо этого новые проекции должны создаваться с помощью либо функции `empty` для создания проекции без пар, либо с помощью функции `singleton`, которой передаются ключ и значение и которая генерирует проекцию с этим одним элементом.

```
*Chapter4.Containers M> M.empty
fromList []
*Chapter4.Containers M> M.singleton "hello" 3
fromList [("hello",3)]
```

По умолчанию проекции показываются как списки пар. Преобразование между таким списочным стилем и реальным стилем проекции можно осуществить с помощью функции `fromList`. Если две пары имеют один и тот же ключ, остается только последнее значение:

```
*Chapter4.Containers M> M.fromList [("hello",1),("bye",2),("hello",3)]
fromList [("bye",2),("hello",3)]
```

При вставке новых значений следует помнить, что с конкретным ключом может быть связано только одно значение. Это приводит к двум различным способам работы, если значение уже связано с ключом:

- Можно полностью проигнорировать это старое значение и просто заменить его новым. Это делается с помощью функции `insert` — ей передаются только новые ключ и значение, и в этом порядке проекция, в которой находится связь, должна быть изменена.
- Можно объединить старое значение с новым. Для этого используется функция `insertWith` со следующим типом:

  ```
  (a -> a -> a) -> k -> a -> Map k a -> Map k a
  ```

  Первый аргумент является объединительной функцией, которая будет вызвана со старым и новым значениями, когда соответствующий ключ уже присутствует. В отдельных случаях в качестве аргумента объединительной функции может потребоваться ключ: тогда придется воспользоваться функцией `insertWithKey`, чьим первым аргументом является значение типа `k`: `k -> a -> a -> a`. Это общий образец для модуля `Data.Map`: при наличии функции, вызываемой со значением проекции, существует альтернативная функция, заканчивающаяся на `WithKey`, которой также передается ключ.

Давайте посмотрим на пример нескольких последовательных вставок:

```
*Chapter4.Containers M> :{
*Chapter4.Containers M| let m1 = M.singleton "hello" 3
*Chapter4.Containers M|     m2 = M.insert "bye" 2 m1
*Chapter4.Containers M|     m3 = M.insert "hello" 5 m2
*Chapter4.Containers M|     m4 = M.insertWith (+) "hello" 7 m3
*Chapter4.Containers M|  in (m1,m2,m3,m4)
*Chapter4.Containers M| :}
( fromList [("hello",3)]
, fromList [("bye",2),("hello",3)]
, fromList [("bye",2),("hello",5)]
, fromList [("bye",2),("hello",12)] )
```

На последнем шаге обратите внимание на пару `("hello",5)`, которая существовала в проекции, и пару `("hello",7)`, которую мы туда собрались вставить. В качестве объединителя мы указали сложение, поэтому получили в итоговой проекции пару `("hello",12)`.

ПРИМЕЧАНИЕ

Если вы ранее работали с такими императивными языками, как C или Java, то, наверное, привыкли к тому, что содержимое контейнеров изменяется функцией напрямую. Однако Haskell является чистым языком, поэтому все эти функции возвращают новую проекцию с соответствующими изменениями. На практике это не создает проблем, поскольку при работе с проекциями, свертками и им подобными структурами мы обычно имеем дело с проекциями «в целом», а не с их «поэлементной» обработкой, которая встречается нечасто.

Помимо хранения элементов проекции используются для выполнения запросов по ключу. Функция `null` позволяет проверить, является проекция пустой или нет, а функция `member` сообщает о том, доступен ли в ней конкретный ключ. Для получения значения по ключу можно воспользоваться функцией `lookup`, возвращающей при наличии подходящего значения `Just value`, или `Nothing`, если ключ отсутствует. В качестве альтернативы можно задействовать функцию `findWithDefault`, которой передается значение, возвращаемое в случае отсутствия запрашиваемого ключа. В следующем сеансе работы с интерпретатором показано несколько примеров применения этих функций:

```
*Chapter4.Containers M> M.null M.empty
True
*Chapter4.Containers M> M.null $ M.fromList [("hello",3),("bye",4)]
False
*Chapter4.Containers M> M.member "hello" $ M.fromList [("hello",3),("bye",4)]
True
*Chapter4.Containers M> M.lookup "hello" $ M.fromList [("hello",3),("bye",4)]
Just 3
*Chapter4.Containers M> M.lookup "welcome" $ M.fromList [("hello",3),("bye",4)]
Nothing
*Chapter4.Containers M> M.findWithDefault 0 "welcome" $ M.fromList
[("hello",3),("bye",4)]
0
```

Можно также удалять пары из проекции, используя для этого функцию `delete`:

```
*Chapter4.Containers M> M.delete "hello" $ M.fromList [("hello", 3), ("bye", 4)]
fromList [("bye",4)]
```

Помимо вставки или удаления можно изменить значение, соответствующее конкретному ключу, воспользовавшись функцией `adjust`: эта функция получает другую функцию, которая применяется к старому значению для получения нового значения. Если ключ отсутствует, проекция остается без изменений:

```
*Chapter4.Containers M> M.adjust (+7) "hello" $ M.fromList [("hello", 3), ("bye",
4)]
fromList [("bye",4),("hello",10)]
```

Функции `insert`, `delete` и `adjust` являются экземплярами общей функции `alter`, которая все их объединяет. Первый аргумент — это функция типа `Maybe a -> Maybe a`:

если ключа в проекции еще нет, ввод будет иметь значение `Nothing` или же он будет иметь предыдущее значение, заключенное в `Just`. Что именно делать с этим ключом, определяет возвращаемое значение данной функции: если это значение `Nothing`, ключ удаляется, а если это значение `Just v`, то оно будет новым значением ключа. Следующий код делает то же самое, что и код в предыдущем примере:

```
*Chapter4.Containers M> :{
*Chapter4.Containers M> M.alter (\(Just v) -> Just (v+7)) "hello"
*Chapter4.Containers M>       $ M.fromList [("hello", 3), ("bye", 4)]
*Chapter4.Containers M> :}
fromList [("bye",4),("hello",10)]
```

В упражнении 4.2 перед вами ставится задача проверки утверждения, что функция `alter` является общей формой представленных ранее функций.

УПРАЖНЕНИЕ 4.2. ВНЕСИТЕ ИЗМЕНЕНИЯ В СВОИ ПРОЕКЦИИ

В Haskell-библиотеках нередко содержатся такие функции общего характера, как alter, которые впоследствии приобретают более конкретное очертание в других функциях, зачастую предоставляющих более простой способ работы с API-интерфейсом и запоминания входящих в него функций. Поставьте себя на место разработчика библиотеки и напишите функции insert, delete и adjust, используя для этого функцию alter.

Можно также комбинировать целые проекции, используя для этого функции `union`, `intersection` и `difference`, создающие соответственно новый ключ проекции из обеих проекций (даже если ключ имеется только в одной из них), только тот ключ, который имеется в обеих проекциях, или только тот ключ, который имеется в первой проекции, но отсутствует во второй. В том случае, если у ключа в каждой проекции имеются разные связанные с ним значения, приоритет будет иметь первая проекция и ее значение. Уточнить характер комбинирования можно с помощью функций `unionWith`, `intersectionWith` и `differenceWith`, которым передается дополнительный аргумент в виде функции, комбинирующей элементы с одним и тем же ключом.

```
*Chapter4.Containers M> :{
*Chapter4.Containers M| let m1 = M.fromList [("hello",3),("bye",4)]
*Chapter4.Containers M|     m2 = M.fromList [("hello",5),("welcome",6)]
*Chapter4.Containers M|  in (m1 `M.union` m2, M.intersectionWith (-) m1 m2)
*Chapter4.Containers M| :}
( fromList [("bye",4),("hello",3),("welcome",6)]
, fromList [("hello",-2)] )
```

Зная основные функции для работы со списками, вы, как правило, сможете применить эти знания к другим структурам данных. Что касается проекций, среди всех остальных функций функции `map`, `foldr`, `foldl`, `filter` и `partition` будут вести себя так же, как и при работе со списками, но возвращать проекции. Здесь также для каждой функции имеется соответствующая функция с суффиксом `WithKey`, в состав аргументов которой тоже входит ключ, подвергаемый модификации, свертке или фильтрации. Давайте удвоим все значения в проекции, а затем вернем их сумму:

```
*Chapter4.Containers M> let m = M.fromList [("hi",3),("bye",4)] in (M.map (*2) m,
M.foldr (+) 0 m)
( fromList [("bye",8),("hi",6)], 7 )
```

Мы уже встречали преобразование списка кортежей в проекцию с помощью функции `fromList`. Обратное преобразование можно осуществить с помощью функции `assocs`. Как можно было заметить, проекции всегда показываются с ключами, выстроенными по порядку. Этот инвариант поддерживается самой проекцией, поэтому в проекции совсем не трудно получить доступ к максимальному и минимальному элементам. Разумеется, данным обстоятельством пользуются такие функции, как `findMin`/`findMax`, `deleteMin`/`deleteMax` и `updateMin`/`updateMax`, что позволяет им осуществлять быстрое извлечение, удаление или обновление значений по ключам.

Множества

Множества (sets) находятся в модуле `Data.Set`. В основном они ведут себя так же, как списки, но не допускают повторений. Набор функций для этого типа данных практически идентичен набору функций для проекций, но им в качестве аргумента передается только значение (у элемента множества нет ключа). В следующих примерах импортируемый модуль `Data.Set` квалифицируется как `S`:

```
Prelude> import qualified Data.Set as S
```

Множество создается с помощью функций `empty` и `singleton`. Как и соответствующие функции для проекций, первая функция создает множество без элементов, вторая — множество с одним элементом. После этого новые элементы можно добавлять с помощью функции `insert`. В следующем примере демонстрируется способ создания множества с элементами `"welcome"` и `"hello"`:

```
Prelude S> S.insert "welcome" $ S.singleton "hello"
fromList ["hello","welcome"]
```

В качестве альтернативы множество можно создать непосредственно из списка его элементов, воспользовавшись для этого функцией `fromList`. Дублированные элементы списка будут сведены в один элемент, поскольку множества могут содержать только один экземпляр каждого элемента.

```
Prelude S> S.fromList ["hello","bye","hello"]
fromList ["bye","hello"]
```

Есть также функция `toList`, осуществляющая обратное преобразование множества в список его элементов. Поведение этих двух функций позволяет реализовать удаление дубликатов из списка (а также сортировку его в восходящем порядке) с намного более высокой производительностью, чем функция `nub`.

```
Prelude S> S.toList $ S.fromList ["duplicate","boom","duplicate"]
["boom","duplicate"]
```

Как уже упоминалось, интерфейс модуля `Data.Set` очень похож на интерфейс модуля `Data.Map`. Следующий код демонстрирует пример использования операций над множествами (в данном случае `intersection`, но также доступны операции `union` и `difference`). Вы можете увидеть, как с помощью функции `member` проверяется принадлежность к множеству. И наконец, как и при работе со списками и проекциями, используя функцию `map`, к каждому элементу множества можно применить какую-нибудь функцию (но делать это нужно осмотрительно: поскольку дублированные результаты сводятся в один элемент, порядок может быть нарушен).

```
Prelude S> :{
Prelude S| let set1 = S.insert "welcome" $ S.singleton "hello"
Prelude S|     set2 = S.fromList ["hello","bye"]
Prelude S|  in ( set1 `S.intersection` set2
Prelude S|     , "welcome" `S.member` set1
Prelude S|     , S.map length set2 )
Prelude S| :}
(fromList ["hello"], True, fromList [3,5])
```

ТИПЫ INTMAP, INTSET, HASHMAP И HASHSET

Эффективность работы с проекциями может существенно возрасти, если в качестве ключей использовать только целые числа. То же самое справедливо и для множеств, содержащих только целочисленные значения. Поэтому для данных целей в библиотеке containers предоставляются специальные типы данных IntMap и IntSet.

Однако ключи в проекции или значения в множестве могут и не быть целыми числами, но тем не менее проецироваться практически однозначно. Такая проекция называется хешем исходного значения. Типы HashMap и HashSet в пакете неупорядоченных контейнеров предоставляют реализацию проекций и множеств, чьи ключи и элементы соответственно могут быть хэшированы. Если тип допускает хеширование, он будет использоваться гораздо эффективнее, чем его обычный аналог.

Как и любые другие значения, эти контейнеры могут вкладываться один в другой, образуя списки множеств, проекции со строковыми ключами и значениями, представляющими собой числовые списки, и т. д. В частности, в упражнении 4.3 мы воспользуемся проекцией с множествами в качестве значений с целью классификации списка клиентов магазина.

УПРАЖНЕНИЕ 4.3. КЛАССИФИКАЦИЯ КЛИЕНТОВ

Для аналитических исследований интересно будет классифицировать клиентов в соответствии с той категорией, к которой они принадлежат: правительственные организации, компании или простые покупатели. Прежде всего, создайте для представления этих категорий новый тип данных:

```
data ClientKind = GovOrgKind | CompanyKind | IndividualKind
```

Теперь создайте функцию classifyClients, которая осуществляет последовательный перебор списка клиентов (с типом [Client Integer], где клиент определен так же, как и в предыдущей главе) и генерирует значение типа Map ClientKind (Set (Client Integer)). Вам нужно будет создать две различные реализации:

- Первая должна осуществить поэлементный перебор списка и, классифицируя каждый элемент, решить, какие элементы проекции нужно модифицировать, а затем добавить к множеству.
- Вторая должна сначала создать списки, соответствующие трем категориям, а в завершение конвертировать эти списки в множества и создать из них упомянутую проекцию.

Будет интересно создать очень большой список клиентов и запустить две реализации, чтобы сравнить, какая из них быстрее справится со своей задачей.

Деревья

Деревья (trees) состоят из узлов, в которых хранятся значения, и могут иметь в качестве дочерних структур другие деревья. В модуле `Data.Tree` такие дочерние структуры представлены в виде пустого списка деревьев, который иногда называ-

ют *лесом* (forest). Следует понимать, что это представление не приспособлено для какой-то конкретной цели: при реализации ряда алгоритмов может потребоваться использование другого вида деревьев, например AVL или черно-красных деревьев. Для таких случаев Hackage предоставляет специализированные пакеты, поддерживающие эти типы данных, такие как `TreeStructures`, `AvlTree` или `RBTree`. Давайте посмотрим на код, определяющий `Data.Tree.Tree`:

```
data Tree   a = Node { rootLabel :: a, subForest :: Forest a }
type Forest a = [Tree a]
```

Ключевое слово `type`, с которым мы пока еще не сталкивались, служит для создания синонима типа, то есть для присвоения типу альтернативного имени. Обычно оно применяется для вызова типа с длинным именем с помощью более короткого или более выразительного имени. Например, следующий синоним можно ввести для тех функций, которые возвращают булево значение:

```
type Predicate a = a -> Bool
```

Синоним типа и его полная форма взаимозаменяемы в любых обстоятельствах. То есть вы также можете записать тип `filter` как `Predicate a -> [a] -> [a]`, и компилятор это вполне устроит. В отличие от этого, другой способ определения альтернативных имен, при котором используется функция `newtype`, не создает эквивалента типа. В каких случаях применяется этот второй вариант, мы узнаем чуть позже при обсуждении классов типов.

Как вы уже, возможно, знаете, для обхода дерева (то есть для последовательного перебора всех его элементов) есть несколько способов, которые по большому счету делятся на два семейства: обход в глубину (depth-first traversal) и обход в ширину (breadth-first traversal). В первом случае в каждом узле дерева рекурсивно посещаются его поддеревья. Там еще выбирается вариант, определяющий, когда нужно посетить значение в самом узле: перед посещением поддеревьев (прямой порядок) или после посещения всех поддеревьев (обратный порядок). Каждый из способов обхода элементов дерева показан на рис. 4.5.



Рис. 4.5. Обход дерева при прямом порядке, обратном порядке и в ширину

Давайте попробуем реализовать функцию, обходящую дерево в прямом порядке, применяя другую функцию к каждому значению и возвращая результат в список:

```
import Data.Tree

preOrder :: (a -> b) -> Tree a -> [b]
preOrder f (Node v subtrees) = let subtreesTraversed = concat $ map (preOrder f)
subtrees
                                in f v : subtreesTraversed
```

Обратите внимание на то, как код использует функцию `map` для выполнения частично вычисляемого выражения `preOrder f` в отношении каждого из поддеревьев. Таким образом вы получите список элементов для каждого поддерева, поскольку `map` возвращает список списков. Поэтому ситуацию нужно выровнять, чтобы получить только один список: это достигается применением функции `concat`. В действительности этот образец, применяемый к списку с последующим выравниванием ситуации и получением одного списка, встречается настолько часто, что в модуль `Prelude` включена функция `concatMap f`, точное определение которой имеет вид `concat . map f`. Можно проверить, как функция работает с деревом, показанном на рис. 4.5:

```
pictureTree :: Tree Int
pictureTree = Node 1 [ Node 2 [ Node 3 []
                              , Node 4 []
                              , Node 5 [] ]
                     , Node 6 [] ]
-- в GHCi
*Chapter4.Containers> preOrder show pictureTree
["1","2","3","4","5","6"]
```

Этот прямой обход можно осуществить, используя функцию `flatten`, определенную в модуле `Data.Tree`. Но она не применяет какие-либо операции в отношении значений узлов, а просто возвращает их в исходном состоянии. Обход в ширину доступен с помощью функции `levels`, где также каждый уровень возвращается в виде списка:

```
*Chapter4.Containers> flatten pictureTree
[1,2,3,4,5,6]
*Chapter4.Containers> levels pictureTree
[[1],[2,6],[3,4,5]]
```

Как и любой другой контейнер, деревья в Haskell поддерживают последовательный перебор своих элементов и свертку. Эти операции доступны через функции `fmap` модуля `Prelude` и функции `foldr` модуля `Data.Foldable`. Причина описана в этой главе чуть позже.

```
*Chapter4.Containers> fmap (*2) pictureTree
Node { rootLabel = 2
     , subForest = [ Node { rootLabel = 4
                          , subForest = [ Node { rootLabel = 6,  subForest = [] }
                                        , Node { rootLabel = 8,  subForest = [] }
                                        , Node { rootLabel = 10, subForest = [] } ]
}
                   , Node { rootLabel = 12, subForest = [] } ] }
*Chapter4.Containers> Data.Foldable.foldr (+) 0 pictureTree
21
```

Графы

Деревья являются частным случаем более общей структуры данных под названием *граф* (graph). Графы состоят из множества вершин (узлов), объединенных множеством ребер (дуг). В реализации, находящейся в модуле `Data.Graph`, узлы всегда идентифицируются с помощью целочисленных значений, а ребра являются направленными (ребро от `a` к `b` — не то же самое, что ребро от `b` к `a`) и не имеют веса.

Граф создается двумя способами:

- Когда есть список узлов, каждый из которых идентифицируется с помощью ключа и содержит значение, и для каждого узла также имеется список соседей, то есть список любых других узлов, принимающих от него ребро, вызывается функция `graphFromEdges`. В этом случае она получает список триад (*значение*, *ключ*, [*ключ*]), где самый последний компонент является вышеупомянутым списком соседей. В обмен вы получаете не только граф, но и две функции: первую типа *вершина* `->` (*узел*, *ключ*, [*ключ*]), которая проецирует идентификатор вершины из графа на соответствующую информацию об узле, и вторую типа *ключ* `->` `Maybe` *вершина*, которая реализует обратное проецирование: ключей на идентификаторы вершин.
- Если у вас уже есть граф в форме, где имеются целочисленные идентификаторы, можно использовать функцию `buildG`. Эта функция получает в качестве аргументов кортеж с минимальным и максимальным идентификаторами (границы) и список кортежей, соответствующих каждому направленному ребру в графе.

Существует множество функций, предназначенных для исследования самих графов; например, функции `vertices` (вершины) и `edges` (ребра) возвращают множества, соответствующие их названиям. Однако реально эффективным этот модуль делает все множество функций, необходимое для обхода элементов в графах и работы с ними, что, как правило, достаточно сложно реализовать вручную. Например, предположим, что у вас есть список того, что нужно сделать для создания машины времени. Но для решения этих задач требуется соблюсти некий порядок действий: для создания дверей машины времени сначала нужно купить алюминий, из которого они сделаны. Этот порядок может быть представлен с помощью графа, где есть ребро от `a` до `b`, причем `a` должно предшествовать `b`. Создание первого графа, показанного на рис. 4.6, осуществляется с помощью следующего кода:

```
import Data.Graph

timeMachineGraph :: [(String, String, [String])]
timeMachineGraph =
  [("wood","wood",["walls"]), ("plastic","plastic",["walls","wheels"])
  ,("aluminum","aluminum",["wheels","door"]),("walls","walls",["done"])
  ,("wheels","wheels",["done"]),("door","door",["done"]),("done","done",[])]

timeMachinePrecedence :: (Graph, Vertex -> (String,String,[String]), String ->
Maybe Vertex)
timeMachinePrecedence = graphFromEdges timeMachineGraph
```

Можно составить план конструирования машины времени, запрашивая *топологическую сортировку* элементов. В этой схеме сортировки любой узел `n` всегда

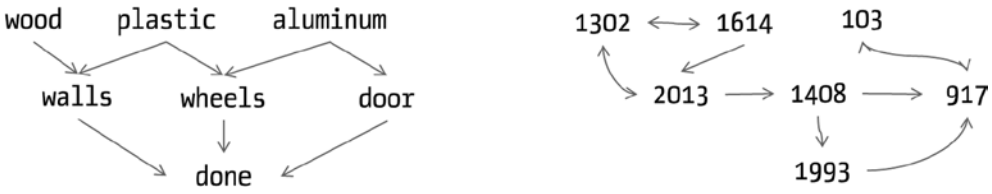


Рис. 4.6. Графы, относящиеся к машинам времени

встречается раньше любого другого узла, который получает ребро от `n`. Обратите внимание на то, как в примере проекция между вершинами и ключами служит для записи результатов с использованием строкового представления, а не внутренних целочисленных идентификаторов:

```
*Chapter4.Containers> :{
*Chapter4.Containers> let (g,v,_) = timeMachinePrecedence
*Chapter4.Containers>  in map (\x -> let (k,_,_) = v x in k) $ topSort g
*Chapter4.Containers> :}
["wood","plastic","walls","aluminum","door","wheels","done"]
```

Большинство людей не знает о машинах времени одной детали: вы не можете попасть в любую точку времени. Вместо этого у каждой машины времени есть ряд точек, в которые можно попасть, и может случиться так, что вы сможете попасть в одну точку только в одном направлении. Поэтому при путешествии во времени нужно быть уверенными в том, что вы сможете попасть в нужное для вас время и вернуться в исходную точку. Эти ограничения можно смоделировать с помощью графа: из каждого года у вас будут ребра к каждому году, в который вы можете прибыть. Этот замысел применительно ко второму графу, изображенному на рис. 4.6, реализуется с помощью следующего кода:

```
timeMachineTravel :: Graph
timeMachineTravel = buildG (103,2013)
                [(1302,1614),(1614,1302),(1302,2013),(2013,1302),(1614,2013)
                ,(2013,1408),(1408,1993),(1408,917),(1993,917),(907,103),(103,917)]
```

Можно спросить: есть ли возможность попасть из 1302 в 917 год? Ответ на этот вопрос даст функция `path`. А если нужно узнать о каждой вершине, достижимой из этого года, можно воспользоваться функцией `reachable`. Давайте взглянем на ряд примеров начиная с 1302 года:

```
*Chapter4.Containers> path timeMachineTravel 1302 917
True
*Chapter4.Containers> reachable timeMachineTravel 1302
[1302,2013,1408,917,1993,1614]
```

Интересно узнать, как разделить вершины, чтобы всегда можно было путешествовать между всеми годами в каждом множестве: каждый компонент этого раздела называется *сильно связанным компонентом* (strongly connected component). Его можно получить, используя функцию `scc`, которая вернет множество деревьев, каждое из которых будет задавать один из таких компонентов. Но если эту функцию выполнить напрямую, будет получен слишком большой объем выходной информации. Это обусловлено тем, что при создании графа с помощью функции `buildG` библиотека

создает вершины для всех промежуточных идентификаторов. Поэтому мы собираемся отфильтровать деревья, имеющие только один узел. Такая фильтрация уберет те вершины, которых не было в исходном списке, а также связанные только с одним элементом компоненты. Наш код фильтрации имеет следующий вид:

```
*Chapter4.Containers> filter (\(Node { subForest = s }) -> s /= []) $ scc
timeMachineTravel
[ Node { rootLabel = 2013
       , subForest = [ Node { rootLabel = 1302
                            , subForest = [ Node { rootLabel = 1614
                                                 , subForest = [] } ] } ] } ]
```

Очевидно, управлять такими выходными данными не слишком просто. Если вместо функции `buildG` ваш граф будет представлен с помощью функции `graphFromEdges`, выходные данные получатся более удобными. Останется только воспользоваться функцией `stronglyConnComp`. Для представления каждого компонента используется специальный тип `SCC`: чтобы получить пригодную для печати версию, нужно, как показано в следующем примере, выполнить функцию `flattenSCC`:

```
*Chapter4.Containers> map flattenSCC $ stronglyConnComp timeMachineGraph
[["done"],["door"],["walls"],["wood"],["wheels"],["plastic"],["aluminum"]]
```

Получение справочной информации

Как уже упоминалось, на веб-сайте Hackage содержится документация обо всех пакетах, доступных в хранилище, включая описания модулей, функций, типов и классов. Это хороший источник информации как для случаев, когда нужно найти сведения о какой-нибудь конкретной функции, так и для случаев, когда достаточно получить обзорные сведения о модуле. Более того, все пакеты в Haskell Platform имеют на этом веб-сайте довольно подробные пояснения и примеры.

EclipseFP предлагает еще один инструмент навигации по содержимому пакетов — Haskell Browser. Он доступен в качестве дополнительного представления (переключиться на него можно с помощью кнопки Haskell Browser в верхнем правом углу). При переходе в этот браузер компоновка страницы меняется, и на экран в верхней части окна выводятся ярлычки четырех вкладок: Packages (Пакеты), Modules (Модули), Types (Типы) и Functions (Функции), как показано на рис. 4.7. При щелчке на имеющихся там элементах в нижней части вкладки появляется описание соответствующего элемента, а все остальные вкладки обновляются в соответствии с выбранным элементом. Таким образом, если вы щелкнете на названии пакета, изменится список модулей, а если вы выберете модуль, обновятся типы и функции. И наконец, если дважды щелкнуть на элементе, на экране будет показано окно с Hackage-документацией.

При повседневном программировании на языке Haskell может помочь еще одно очень хорошее средство под названием Hoogle, которое доступно по адресу www.haskell.org/hoogle/. Эффективный поисковый движок Hoogle позволяет вести поиск не только по имени, но и по типу. Более того, поиск понимает Haskell-идиомы: например, если вы ищете функцию с конкретным типом, может быть найдена функция более общего характера, использующая класс типов, реализующий в сигнатуре

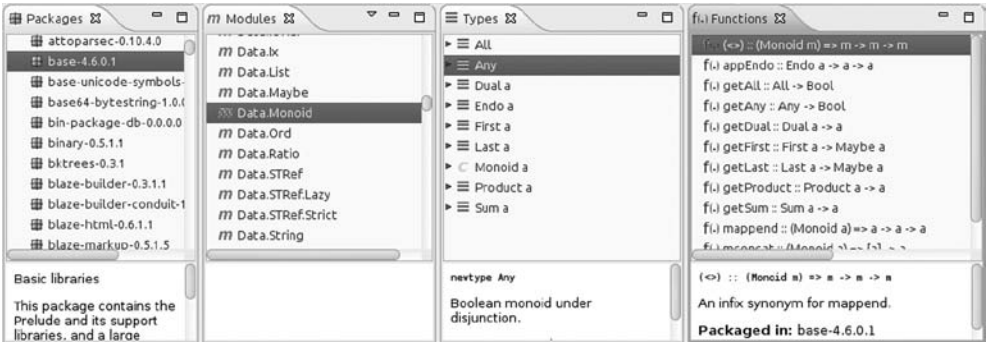


Рис. 4.7. Haskell Browser в EclipseFP

искомый тип, или же в найденной версии может быть другой порядок аргументов. Это средство доступно также в виде программы командной строки `hoogle`, которую можно получить, выполнив команду `cabal install hoogle`, а также через вкладку Hoogle в EclipseFP. Перед тем как приступить к созданию запросов, нужно выполнить консольную команду `hoogle data` или щелкнуть на соответствующей кнопке Update (Обновить) в EclipseFP, загрузив из Hackage самую свежую версию базы данных.

Вот как выглядят выходные Hoogle-данные для функции, подобной функции `map`:

```
$ hoogle '(a -> b) -> ([a] -> [b])'
Prelude map :: (a -> b) -> [a] -> [b]
Data.List map :: (a -> b) -> [a] -> [b]
Control.Parallel.Strategies parMap :: Strategy b -> (a -> b) -> [a] -> [b]
Control.Applicative liftA :: Applicative f => (a -> b) -> f a -> f b
Data.Traversable fmapDefault :: Traversable t => (a -> b) -> t a -> t b
Prelude fmap :: Functor f => (a -> b) -> f a -> f b
Data.Functor fmap :: Functor f => (a -> b) -> f a -> f b
# в продолжении находится еще 80 результатов
```

HOOGLE В WINDOWS

Чтобы такое средство, как Hoogle, заработало под управлением операционной системы Windows, требуется предпринять дополнительные усилия по сравнению с Linux или Mac OS X. Перед тем как появится возможность выполнять команду hoogle data, нужно загрузить несколько программ и сделать их доступными через системную переменную PATH:

- wget: http://gnuwin32.sourceforge.net/packages/wget.htm;
- gzip: http://gnuwin32.sourceforge.net/packages/gzip.htm;
- tar: http://gnuwin32.sourceforge.net/packages/gtar.htm.

Также при составлении поисковых запросов в Windows нужно использовать вместо одинарных кавычек двойные.

Специальный полиморфизм: классы типов

До сих пор в данной книге мы рассматривали типы нескольких функций, доступных в Platform. Но если посмотреть на некоторые функции в модуле `Data.Map` или `Data.Set`, вы увидите нечто такое, что еще не получило объяснения:

```
*Chapter4.Containers> :t M.insert
M.insert :: Ord k => k -> a -> M.Map k a -> M.Map k a
```

Обратите внимание на то, что выражение `Ord k` отделено от остального определения типа комбинацией символов `=>` (эту комбинацию не следует путать со стрелкой вида `->` в типах функций). Выражение `Ord k` ограничивает то множество возможных типов, которые сможет принимать переменная типа `k`. Это отличается от параметрического полиморфизма списочных функций, рассмотренных в предыдущих главах, поскольку здесь мы требуем, чтобы тип поддерживался некоторыми функциями. Эта разновидность полиморфизма называется *специальным полиморфизмом* (ad-hoc polymorphism). В данном случае класс типов `Ord` сообщает о том, что тип должен обеспечить реализацию операторов сравнения, таких как `<` или `==`. Таким образом, с его помощью формализуется нотация «порядка, предлагаемого по умолчанию», о котором уже шла речь.

Объявление классов и экземпляров

Класс типов (type class), или просто *класс* (class), является объявлением множества функций наряду с их типами, которое получает имя (в предыдущем случае — `Ord`). Объявление класса типов имеет следующий синтаксис:

```
class ИмяКласса переменная where
  однаФункция   :: одинТип
  ...
  другаяФункция :: другойТип
```

Переменная, введенная в объявление, может использоваться в функциях для ссылки на тип, поддерживающий класс типов. Например, и у клиентов, и у машин времени есть имена, следовательно, можно ввести класс типов, который мы назовем `Nameable`, для выражения понятия «у значений этого типа есть имена». Посмотрите, как тип переменной `n` используется в типе имени функции:

```
class Nameable n where
  name :: n -> String
```

Если посмотреть на тип `name`, в нем определяется ограничение, связанное с его отношением к `Nameable`:

```
*Chapter4.TypeClasses> :t name
name :: Nameable n => n -> String
```

С этого момента использование функции `name` также сопровождается связанным ограничением, которое должно быть указано в соответствующем объявлении типа. Класс `Nameable` мог бы содержать функцию, объявленную за пределом класса типов и возвращающую инициал имени:

```
initial :: Nameable n => n -> Char
initial n = head (name n)
```

Разумеется, основной целью класса типов является объявление о том, что некий конкретный тип поддерживает операции, введенные классом. Такой тип называется *экземпляром* (instance) класса типов. Объявление этого факта должно включать реализацию функций, объявленных в классе:

```
instance ИмяКласса Тип where
  однаФункция   = ... -- реализация
  ...
  другаяФункция = ... -- реализация
```

Следуя примеру, мы получаем конкретизацию класса типов `Nameable` с помощью класса `Client`. Вспомним, как этот тип выглядел в конце главы 2:

```
data Client i = GovOrg  { clientId :: i, clientName :: String }
              | Company { clientId :: i, clientName :: String
                        , person :: Person, duty :: String }
              | Individual { clientId :: i, person :: Person }
              deriving Show
```

В объявление экземпляра нужно включить весь тип. Это означает, что нам нужно также написать параметры типа, которые должны применяться в объявлении (в данном случае — параметр `i`).

```
instance Nameable (Client i) where
  name Individual { person = Person { firstName = f, lastName = n } } = f ++ " " ++ n
  name c = clientName c
```

ПРИМЕЧАНИЕ

Классы типов в Haskell не следует путать с классами в объектно-ориентированном программировании. Вообще-то если нам нужно создать связь, классы типов можно представить как интерфейсы в этих языках, но имеющие намного более широкий спектр применения, например связывание типов (указание на то, что IntSet содержит элементы типа Int). Слово «экземпляр» также имеет в этих параллельных «мирах» программирования совершенно разные значения: в объектно-ориентированных языках оно относится к конкретному значению класса, а в Haskell оно относится к реализации класса с помощью типа. Это указывает на третье отличие: в объектно-ориентированном языке объявление класса включает список всех реализуемых им интерфейсов, а в Haskell объявление типа и реализация класса типов отделены друг от друга. В действительности в некоторых случаях они находятся даже в разных модулях — такие экземпляры называются сиротами (orphan).

При использовании типа, реализующего класс, компилятор Haskell должен найти соответствующее объявление `instance`. Он делает это путем просмотра всех импортируемых модулей независимо от способа, которым они импортированы. В настоящее время предотвратить применение импортированного объявления невозможно. Например:

```
import Module ()
```

Если в каком-нибудь исходном коде встретится такая строка, это может быть не ошибкой (зачем объявление, если ничего не импортируется?), а импортом объявлений экземпляров, находящихся в модуле `Module`.

В упражнении 4.4 первая задача решается за счет применения классов типов и экземпляров. Это делается для осуществления нашей главной цели: создания полноценного магазина машин времени.

УПРАЖНЕНИЕ 4.4. ЦЕНЫ ДЛЯ МАГАЗИНА

В интернет-магазине будут продаваться не только машины времени, но и путеводители, а также средства для обслуживания машин. И у всего этого есть нечто общее — цена. Создайте класс

типов Priceable, состоящий из типов, поддерживающих цену, и создайте такие типы данных, которые реализуют этот класс.

Следующим шагом станет создание функции totalPrice, которая при получении списка товаров с ценами вычисляет общую сумму денег, которую нужно заплатить. У этой функции должен быть следующий тип:

```
totalPrice :: Priceable p => [p] -> Double
```

Нужно учесть, что смысл данной сигнатуры может противоречить вашим интуитивным представлениям, особенно если у вас есть опыт объектно-ориентированного программирования. Когда компилятор обрабатывает этот код, он ищет для переменной типа p конкретный тип. Это означает, что он может работать только с однородными списками: можно вычислить цену списка машин времени, [TimeMachine], или списка книг, [Book]. Но типизировать или создать неоднородный список, содержащий как значения типа TimeMachine, так и значения типа Book, невозможно.

Класс типов `Nameable` очень похож на класс типов `Show`, предоставляющий функцию `show` для преобразования значения в строку. В определении предыдущих типов данных нам такой класс уже встречался, но мы не создавали для этих типов данных никаких экземпляров. Причина в том, что компилятор Haskell способен автоматически записывать экземпляры множества классов типов исходя из формы типа данных. Эта способность называется извлечением (deriving) экземпляра, поскольку генерируемые экземпляры указываются в конце определения типа данных после ключевого слова `deriving`.

ПРИМЕЧАНИЕ

К настоящему времени Haskell поддерживает извлечение экземпляров только для некоторых классов типов, и эта способность жестко запрограммирована в компиляторе. Однако та часть функционального программирования, которая называется обобщенным программированием (generic programming), предоставляет способы написания функций, зависящих от структуры типов данных. Таким образом, появляется возможность расширения механизма вывода. Дополнительные сведения можно найти на веб-странице http://www.haskell.org/haskellwiki/Generics, где показано несколько способов возможного применения обобщенного программирования в Haskell.

У класса `Show` есть парный класс `Read`, выполняющий обратную функцию: он преобразует строковое представление в фактическое значение типа, делая это с помощью функции `read`. Реализация экземпляра `Read` обычно не обходится без трюков, поскольку вам нужно позаботиться о разрядке, правильной обработке скобок, о разных форматах для чисел и т. д. Но не огорчайтесь, Haskell позволяет извлекать экземпляры `Read` также и в автоматическом режиме, что гарантирует обратное чтение любого значения, выданного функцией `show`. Если извлекаются экземпляры того и другого, вы можете быть уверены в идентичности значений типа данных `read . show`, то есть в том, что итоговое значение будет таким же, как и исходное.

Давайте извлечем экземпляр `Read` для нашего типа данных `Person`:

```
data Person = Person { firstName :: String, lastName  :: String }
            deriving (Show, Read)
```

А теперь попытаемся проанализировать строку, представляющую человека (person):

```
*Chapter4.TypeClasses> read "Person { firstName = \"A\", lastName = \"S\" }"
<interactive>:
    No instance for (Read a0) arising from a use of 'read'
```

```
    The type variable 'a0' is ambiguous
    Possible fix: add a type signature that fixes these type variable(s)
```

В сообщении об ошибке говорится, что тип переменной `a0`, создаваемой компилятором, непонятен, а также указывается на то, что компилятор не знает, какой из экземпляров `Read` нужно использовать: вы не задали какую-либо дальнейшую операцию над результатом, которая могла бы помочь компилятору Haskell сделать предположение относительно окончательного типа. Решение заключается в явном указании возвращаемого типа. Это достигается путем аннотирования выражения с помощью группы символов `::`, за которой следует тип:

```
*Chapter4.TypeClasses> read "Person { firstName = \"A\", lastName = \"B\" }" :: Person
Person {firstName = "A", lastName = "B"}
```

Здесь вы снова можете проверить тот факт, что компилятор Haskell всегда делает логический вывод о наиболее общем типе на основе функций, используемых в выражении. Например, функция `read . show` будет работать с любым типом данных, поддерживающим и `Show`, и `Read`. Но в целом она также работает, если какой-нибудь тип данных поддерживает `read`, что относится к более общему случаю по сравнению с каким-нибудь отдельным типом, поддерживающим обе функции.

```
*Chapter4.TypeClasses> :t read . show
read . show :: (Read c, Show a) => a -> c
```

Встроенные классы типов

В предыдущих главах уже говорилось о некоторых списочных функциях, использующих типы, имеющие «сравнение, предлагаемое по умолчанию» и «равенство, предлагаемое по умолчанию». Теперь, когда мы знаем о классах типов, настало время познакомиться с конкретными классами, в которых закодированы эти понятия, а именно `Ord` и `Eq`.

Класс `Eq` является классом типов, объявляющим, что тип поддерживает проверку на равенство (и неравенство) своих значений. Давайте посмотрим на его определение из исходного GHC-кода. Доступ к нему можно получить, просматривая пакет `base` в Hackage, исследуя модуль `Prelude` и щелкая на ссылке Source (Источник), которая следует за информацией о классе.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)
```

В начале раздела утверждалось, что классы типов включают только функции, подлежащие реализации, но здесь находится и определенная реализация кода. Причина в том, что он относится к *определениям, предлагаемым по умолчанию* (default definitions): к коду функции, которая работает независимо от того, реализованы или нет остальные функции. Например, если мы реализуем функцию `(==)`, то, как было показано, есть также простой способ реализации функции `(/=)`. При создании экземпляра класса типов можно оставить для этих функций реализацию, предлагаемую по умолчанию.

Это означает, что, реализуя `Eq`, можно обойтись без какой бы то ни было фактической реализации, поскольку для всех функций есть реализации, предлагаемые по умолчанию. В таком случае любое сравнение войдет в бесконечный цикл, поскольку функция `(/=)` вызывает функцию `(==)`, которая в свою очередь вызывает функцию `(/=)`, и так до бесконечности. Это может привести к тому, что программа исчерпает всю память или просто зависнет. Чтобы исключить подобные случаи, в классах типов Haskell обычно указывается *минимальное полное определение* (minimal complete definition): то есть то множество функций, которое должно быть реализовано, чтобы все работало без проблем. Для `Eq` минимальным полным определением будет либо `(==)`, либо `(/=)`, следовательно, нужно реализовать как минимум одну из этих функций.

ВНИМАНИЕ

Понимание необходимости минимального полного определения играет для класса типов весьма важную роль, поскольку является единственным средством навязать программам правильное поведение. Компилятор GHC, начиная с версии 7.8, способен проверить, что все необходимые функции определены. Но поскольку эта возможность отсутствовала в начальных версиях компилятора, в некоторых библиотеках явная поддержка минимального полного определения отсутствует. Поэтому перед их использованием нужно перепроверить документацию по классам типов.

А теперь у вас может появиться вопрос: поскольку при наличии функции `(==)` определение функции `(/=)` всегда можно записать как `not . (==)`, зачем включать определение обеих этих функций в класс типов, чтобы затем приходилось вводить понятие минимального полного определения? Не должна ли функция `(/=)` быть определена *за пределами* класса типов? Наличие всех этих функций в одном и том же типовом классе объясняется двумя причинами:

- Простота определения экземпляра. Для некоторых типов естественнее записать экземпляр `Eq` путем определения функции `(==)`, а для других — функции `(/=)`. Возможность поступить двояко упрощает потребителям создание экземпляров классов. Возможно, это не столь очевидно для `Eq`, но для более сложных классов типов эта возможность играет весьма существенную роль.
- Производительность. Наличие обеих функций в типовом классе позволяет при желании реализовать обе эти функции. Так зачастую делается из соображений повышения производительности: возможно, ваш тип быстрее выполнит проверку на неравенство, когда попытка проверки на равенство не проходит.

Случай с равенством приводит к другим интересным особенностям имеющейся в Haskell системы классов типов: создание экземпляра для типа с переменными и ограничения для создания экземпляров класса типов. Например, можно реализовать экземпляр `Eq` для любого возможного списочного типа в общем виде: нужно только провести поэлементную проверку на равенство. Однако справедливости ради нужно сказать, что вы должны потребовать, чтобы внутренние элементы также реализовывали класс `Eq`. Посмотрим на следующий код:

```
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False
```

Сконцентрируем внимание на выделенных частях: прежде всего здесь вводится ограничение на элементы с использованием синтаксиса, очень похожего на синтаксис функций. Затем в объявлении в имени типа имеется параметрическое указание `[a]` с переменной типа. В итоге этот экземпляр применим к спискам любого типа `a`, который также относится к классу `Eq`. В Haskell Platform также включаются такие объявления для многих широко распространенных контейнеров, а не только для списков. Экземпляры `Eq` определены для кортежей, проекций, множеств, деревьев, типа `Maybe` и т. д.

Эффективный способ создания экземпляра класса типов путем использования параметрических типов не является исключением для специального множества встроенных типов, и, как показано в упражнении 4.5, его можно применить и в вашем собственном типе данных.

УПРАЖНЕНИЕ 4.5. ТОТ ЖЕ САМЫЙ КЛИЕНТ

Реализуйте класс Eq для типов Person и Client i, введенных в предыдущих главах, чтобы он проверял на равенство все поля. Учтите, что для Client может потребоваться добавление некоторых ограничений, касающихся типа идентификаторов.

Хорошо, что в повседневной работе вам не придется создавать подобные экземпляры, поскольку компилятор Haskell также может строить выводы на основе Eq, как он это делал с Show и Read.

В некоторых случаях кроме равенства нужно вводить понятие упорядоченности — этим занимается класс типов `Ord`:

```
class Eq a => Ord a where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

Здесь опять очень много составляющих, но благодаря определениям, предлагаемым по умолчанию, минимальным полным определением будет либо реализация `compare`, либо реализация `<=`. Но если посмотреть на код в `compare`, можно заметить, что в нем используется функция `(==)`, входящая в определение `Eq`. Действительно, в самом начале определения класса `Ord` есть необходимое условие для его реализации: каждый тип, принадлежащий классу `Ord`, должен также принадлежать и классу `Eq`. Синтаксис опять похож на включение ограничений в функции или в реализации экземпляров: `Eq a =>`.

В данном случае мы говорим, что `Eq` является *суперклассом* (superclass) класса `Ord`. И снова нужно предостеречь вас от возможной путаницы при сравнении этого

понятия с аналогичным понятием в объектно-ориентированных языках. Класс типов не наследует ничего у своего суперкласса, кроме обещания, что некоторые функции будут реализованы в их экземплярах. Что касается всего остального, это классы абсолютно разных типов. В частности, реализации типов будут помещаться в отдельных объявлениях `instance`.

Что же касается класса `Eq`, то компилятор Haskell также обеспечивает автоматическое извлечение экземпляров класса `Ord`. При этом им берется в расчет, что различные варианты следуют в том же порядке, что и в объявлении `data`. Например, при объявлении клиентов в начале раздела правительственные организации всегда предшествуют компаниям, а те всегда предшествуют индивидуальным клиентам. Затем, если два значения имеют одинаковый конструктор, компилятор продолжит просмотр каждого поля в порядке объявления. В данном случае это означает, что после проверки вида клиента следующее сравнение будет касаться его идентификатора. Но как указывается в упражнении 4.6, такое поведение может быть не самым удачным.

УПРАЖНЕНИЕ 4.6. ВЫСТРАИВАНИЕ КЛИЕНТОВ ПО ПОРЯДКУ

Автоматическое извлечение экземпляров класса Ord для класса Client не имеет особого смысла: как уже отмечалось, при этом сначала осуществляется проверка вида клиента, а затем — его идентификатора. Напишите взамен новый экземпляр со следующими свойствами: сначала должна идти проверка имени клиента. Затем, при совпадении, первыми должны идти индивидуальные клиенты, потом — компании и в конце — правительственные организации. Для выстраивания нужного порядка вам может потребоваться добавление дополнительных сравнений в остальных полях (например, две компании, у которых разные ответственные лица, не считаются равными друг другу, поэтому нужно решить, какую из них поместить первой).

Решите заранее, нужно ли включать в экземпляр какие-нибудь ограничения.

К другим классам типов, использовавшимся нами без каких-либо оговорок, относятся `Num` и его подклассы. Как уже можно было догадаться, `Num` является классом для всех типов, представляющих числа, будь то целые числа, рациональные числа или числа с плавающей точкой. Каждый подкласс добавляет некоторую детализацию: например, в `Num` включено только сложение, вычитание и умножение, а в классе `Integral` к этому добавляются целочисленное деление и получение остатка. Операции, принадлежащие каждому классу, а также связанным с ним подклассам, сведены в табл. 4.1.

Таблица 4.1. Классы типов, связанные с числами

Класс типов	Родительский класс	Описание
`Num`	Отсутствует	Основной числовой тип. Поддерживает сложение (+), вычитание (−), умножение (*), унарное отрицание (negate), получение абсолютного значения (abs), получение знака (signum) и преобразование из целочисленного значения (fromInteger)
`Real`	`Num`	Подкласс, поддерживающий преобразование в рациональные значения Rational (с помощью toRational). Integer является экземпляром этого класса, поскольку целочисленные значения могут быть внедрены как дроби со знаменателем 1

Таблица 4.1 (продолжение)

Класс типов	Родительский класс	Описание
`Integral`	`Real`	Подкласс для целочисленных значений, поддерживающих целочисленное деление и получение целочисленного остатка от деления. Связанные с ним функции представлены триадами: quot (деление), rem (получение остатка от деления), quotRem (то и другое с округлением до нуля), функции div, mod и divMod делают то же самое с округлением до минус бесконечности (-∞)
`Fractional`	`Num`	Подкласс для дробных значений. Поддерживает деление (/), получение обратного значения (recip) и преобразование из значения Rational (fromRational)
`Floating`	`Fractional`	Подкласс для значений с плавающей точкой. Поддерживает такие широко распространенные значения, как pi и e. Позволяет вычислять квадратные корни (sqrt), натуральные логарифмы (log), возведение в степень (exp для ex, (**) для общего вида ab,), а также круговые и гиперболические тригонометрические функции

Одним из наиболее интересных в Haskell-трактовке чисел является способ трактовки констант. Например, при попытке получения в интерпретаторе типа для 1 результат поначалу может привести в замешательство:

```
*Chapter4.TypeClasses> :t 1
1 :: Num a => a
```

Нетрудно заметить, что сама по себе константа является полиморфной: она может быть представлена в любом типе, создающем экземпляр класса `Num`. Причина этого кроется в функции `fromInteger`, позволяющей извлекать значение из целочисленной константы.

Например, предположим, мы собираемся создать тип данных для комплексных чисел и реализовать его экземпляр класса `Num`[1]. Из курса алгебры мы помним, что комплексное число имеет форму $a + ib$, где a является вещественной, а b — мнимой частью. Правила, регулирующие операции с комплексными числами, могут быть выведены из того факта, что $i^2 = -1$. И наконец, для каждого комплексного числа имеется понятие абсолютного значения $\|x\|$ и аргумента θ_x, который удовлетворяет условию $\|x\|\,\theta_x = x$ — именно это нам и нужно для функций `abs` и `signum` в экземпляре класса `Num`.

```
data Complex = C Double Double deriving (Show, Eq)

instance Num Complex where
  (C a1 b1) + (C a2 b2) = C (a1 + a2) (b1 + b2)
  (C a1 b1) - (C a2 b2) = C (a1 - a2) (b1 - b2)
  (C a1 b1) * (C a2 b2) = C (a1*a2-b1*b2) (a1*b2+b1*a2)
  negate (C a b)        = C (negate a) (negate b)
  fromInteger n         = C (fromInteger n) 0
```

[1] В Haskell Platform есть все, что нужно, там также есть тип для комплексных чисел, который можно найти в модуле Data.Complex. Поэтому наше определение — не более чем иллюстрация.

```
  abs (C a b)           = C (sqrt $ a*a+b*b) 0
  signum c@(C a b)      = let C n _ = abs c in C (a / n) (b / n)
```

Вы убедились в том, что классы типов являются весьма эффективным средством для абстрагирования понятий и структур. В предыдущих главах мы рассмотрели идиому «значений, предлагаемых по умолчанию», поэтому может возникнуть вопрос: а нет ли класса типов и на этот случай? Ответ находится в модуле `Data.Default` в пакете `data-default`, который предоставляет класс `Default` с единственной составляющей в виде функции `def`, возвращающей значение, предлагаемое по умолчанию. Если вернуться к рассмотрению вариантов подключения к Интернету из главы 2, то вместо `connDefault` будет использоваться решение с классом типов:

```
instance Default ConnOptions where
  def = ConnOptions TCP 0 NoProxy False False NoTimeOut
```

Двоичные деревья для минимальной цены

До этой главы мы сохраняли всю информацию о клиентах в списках. Несмотря на то что вам теперь уже известно, где искать готовые типы, в данном разделе мы собираемся сделать шаг назад и посмотреть на конструкцию собственного контейнерного типа, который полностью удовлетворяет конкретным потребностям приложения. В частности, цель состоит в поддержке скидочного модуля (discount module), которому требуется доступ к самым дешевым элементам, имеющимся в контейнере. В процессе работы вы увидите, как классы типов позволяют достичь более высокой степени обобщения, повышая возможности многократного использования кода.

В данном разделе контейнер будет содержать путеводители. Каждый путеводитель состоит из названия, списка авторов и цены. Это можно смоделировать с помощью записи. Чуть позже вы увидите, что нам понадобится понимание того, что меньше и что больше, поэтому нам нужен экземпляр `Ord`, а с ним и экземпляр `Eq`:

```
data TravelGuide = TravelGuide { title :: String, authors :: [String], price ::
Double }
                 deriving (Show, Eq, Ord)
```

Шаг 1. Простые двоичные деревья

Первое решение заключается в использовании простых списков. Самой большой проблемой, связанной с ними, является то, что запрос элементов внутри этих списков обходится слишком дорого, потому что единственным имеющимся вариантом является поэлементный обход списка. В самом худшем случае для получения ответа понадобится обойти весь список. Решением проблемы является использование двоичных деревьев, которые предлагают лучший показатель сложности (не линейный, а логарифмический).

Двоичное дерево (binary tree) — это структура данных, состоящая из узлов. В каждом узле содержится значение и две ссылки на поддеревья. Чтобы показать, что у дерева отсутствуют некоторые поддеревья, используется специальный маркер *листа* (leaf). Особое свойство наших двоичных деревьев заключается в том, что

любой узел в левом поддереве будет содержать только те значения, которые меньше значения узла, а в правом поддереве будут находиться значения, которые больше значения узла. Именно поэтому нам нужно извлечь `Ord` для типа `TravelGuide`. На рис. 4.8 показан пример двоичного дерева с ограничениями, применяемыми к узлам, расположенным по краям.

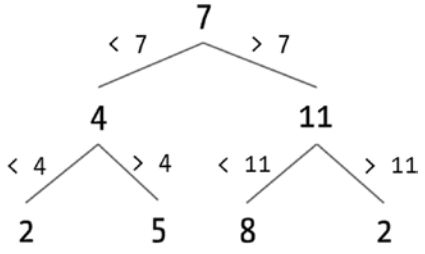


Рис. 4.8. Графический пример двоичного дерева

Теперь можно создать структуру данных двоичных деревьев для путеводителей:

```
data BinaryTree = Node TravelGuide BinaryTree BinaryTree
                | Leaf
                deriving Show
```

Как уже объяснялось, поиск в двоичном дереве проводится намного быстрее, поскольку сравнение искомого элемента с текущим исследуемым узлом дает возможность принять решение, в какое поддерево нужно углубляться, будучи уверенным, что этого элемента нет в другом поддереве:

```
treeFind :: TravelGuide -> BinaryTree -> Maybe TravelGuide
treeFind t (Node v l r) = case compare t v of
                              EQ -> Just v
                              LT -> treeFind t l
                              GT -> treeFind t r
treeFind _ Leaf         = Nothing
```

Вам также нужен способ начального создания пустых деревьев и вставки значений в дерево при сохранении инварианта. В последнем случае алгоритм довольно прост: обход дерева, как при поиске вставляемого значения. Если удалось достичь листа `Leaf`, значит, значение можно вставить на это место.

```
treeInsert :: TravelGuide -> BinaryTree -> BinaryTree
treeInsert t n@(Node v l r) = case compare t v of
                                  EQ -> n
                                  LT -> Node v (treeInsert t l) r
                                  GT -> Node v l (treeInsert t r)
treeInsert t Leaf = Node t Leaf Leaf
```

Шаг 2. Полиморфные двоичные деревья

Основная структура двоичных деревьев просто наталкивает на обобщение. Любая информация внутри `TravelGuide` используется только для ее упорядочения и ни для чего другого. Это наводит на мысль о том, что мы должны работать с экземпля-

рами `Ord`. Параметрическая версия двоичного дерева и связанной с ним функции `treeFind` теперь имеет следующий вид:

```
data BinaryTree2 a = Node2 a (BinaryTree2 a) (BinaryTree2 a)
| Leaf2
deriving Show

treeFind2 :: Ord a => a -> BinaryTree2 a -> Maybe a
treeFind2 t (Node2 v l r) = case compare t v of
EQ -> Just v
LT -> treeFind2 t l
GT -> treeFind2 t r
treeFind2 _ Leaf2 = Nothing
```

ПРИМЕЧАНИЕ

Здесь может возникнуть вопрос, можем ли мы закодировать ограничения, накладываемые на класс элементов, которые могут содержаться в двоичном дереве, непосредственно в объявлении BinaryTree2? Конечно можем, но делать это не рекомендуется: лучше закодировать ограничения в каждой операции, работающей со структурой, как это было сделано в данном примере. Следует знать, что в целях наложения этих ограничений мы должны скрыть конструкторы Node2 и Leaf2 от общедоступного применения.

Функция `treeFind` была обобщена, но вам все же необходимо внести ряд изменений в функцию `treeInsert`, чтобы сделать ее полностью универсальной. Более детально этот вопрос рассматривается в упражнении 4.7.

УПРАЖНЕНИЕ 4.7. УВЕЛИЧЕНИЕ КОЛИЧЕСТВА ОПЕРАЦИЙ НАД ТИПИЧНЫМИ ДЕРЕВЬЯМИ

Внесите изменения, необходимые для того, чтобы функция treeInsert работала с новым деревом BinaryTree2.

Попробуйте также реализовать объединение двоичных деревьев путем многократной вставки всех элементов в одно из них.

Здесь следует заметить, что при автоматическом извлечении экземпляра `Ord` для `TravelGuide` проводится сравнение сначала названия, затем списка авторов и, наконец, цены. Но вам нужно не это: приложению требуется упорядочить путеводители по цене. Сначала можно попытаться написать новый экземпляр `Ord`:

```
instance Ord TravelGuide where
(TravelGuide t1 a1 p1) <= (TravelGuide t2 a2 p2) =
p1 < p2 || (p1 == p2 && (t1 < t2 || (t1 == t2 && a1 <= a2)))
```

Конечно же, будет получено сообщение об ошибке, связанное с дублированием экземпляров:

```
Duplicate instance declarations:
     instance Ord TravelGuide
       -- Defined at MinimumPrice.hs:4:38
     instance Ord TravelGuide
       -- Defined at MinimumPrice.hs:6:10
```

Первое, что может прийти в голову, — это создание типа данных, имеющего одно поле для хранения упорядоченных по ценам путеводителей с последующим созданием экземпляра для этого типа данных:

```
data TravelGuidePrice = TravelGuidePrice TravelGuide
instance Ord TravelGuideByPrice where ...
```

Проблема в том, что создается новый конструктор, который в процессе исполнения должен подвергаться сопоставлению с образцом и извлечению из оболочки, что отнимает время и снижает производительность. Вам же нужен способ как-то помечать значения с новым типом, чтобы компилятор мог распознать, какой из экземпляров нужно применить. Но это хотелось бы сделать без перезаписи исходного типа и без проблем с производительностью. И в Haskell есть нужное решение: объявление `newtype` дает уже существующему типу другое имя, но в отличие от объявления `type` новое имя не является синонимом, считается, что тип с новым именем совершенно не связан с предыдущим типом. Плюс здесь в том, что использование объявления `newtype` не вызывает потерь производительности, поскольку в ходе своей работы компилятор знает, что значения этого типа всегда будут равны значению исходного типа, поэтому он может удалить все оболочки конструктора и все сопоставления с образцами. В следующем фрагменте кода для `TravelGuide` объявляется тип `newtype` и осуществляется привязка к нему нового экземпляра класса типов `Ord`, но теперь уже со следующим первым сравнением по цене:

```
newtype TravelGuidePrice = TravelGuidePrice TravelGuide deriving Eq

instance Ord TravelGuidePrice where
(TravelGuidePrice (TravelGuide t1 a1 p1)) <= (TravelGuidePrice (TravelGuide t2 a2
p2)) =
p1 < p2 || (p1 == p2 && (t1 < t2 || (t1 == t2 && a1 <= a2)))
```

Теперь давайте предположим, что выстраивание по цене используется для всех остальных примеров.

Шаг 3. Двоичные деревья с моноидальным кэшем

И все же нахождение наименьшей цены в дереве требует некоторого времени, потому что приходится заходить в левое поддерево до тех пор, пока не попадется лист. Однако на веб-странице вам нужно показывать этот элемент очень часто. Решение заключается во включении кэша в каждый узел, в котором хранится цена самого маленького элемента в дереве. Давайте создадим новую версию двоичных деревьев, в которых для большей универсальности тип кэша также сделан параметрическим:

```
data BinaryTree3 v c = Node3 v c (BinaryTree3 v c) (BinaryTree3 v c)
                     | Leaf3
                     deriving (Show, Eq, Ord)
```

Далее создадим соответствующую реализацию функции `treeInsert`, где кэш обновляется на каждом шаге:

```
treeInsert3 :: (Ord v, Ord c) => v -> c -> BinaryTree3 v c -> BinaryTree3 v c
treeInsert3 v c (Node3 v2 c2 l r) = case compare v v2 of
                                EQ -> Node3 v2 c2 l r
                                LT -> Node3 v2 (min c c2) (treeInsert3 v c l) r
                                GT -> Node3 v2 (min c c2) l (treeInsert3 v c r)
treeInsert3 v c Leaf3             = Node3 v c Leaf3 Leaf3
```

На определенном этапе вам могут сказать, что кроме минимальных цен маркетинговая команда также хочет иметь информацию о средней цене путеводителей, чтобы проанализировать, какой процент людей покупает книги по более низкой и более высокой ценам. Для этого нам придется создать новую функцию вставки, которая вместо минимума вычисляет сумму всех цен с последующим делением на общее количество путеводителей. Но в целом структура функции остается такой же.

Давайте попробуем разобраться в структуре в обоих случаях: то, чем мы занимаемся, является кэшированием некоторой информации, поступающей из кэшированных значений в поддеревьях и из значения самого узла. В случае с минимальной ценой операцией, создающей новое кэшированное значение, является `min`, а в случае с суммой — `+`. Можно ли уточнить конкретные требования к такой функции?

- Для любых двух элементов нужно найти еще один элемент того же типа. То есть нам нужна функция `f` типа `c -> c -> c`.
- Также в зависимости от того, как именно элементы были вставлены в дерево, структура может не быть одинаковой. Но это не должно влиять на финальное кэшированное значение. Следовательно, нужно убедиться в том, что структура в скобках не имеет значения. Иными словами, операция должна быть ассоциативной.
- И еще одно, последнее обстоятельство вытекает из наблюдения, что при объединении двух двоичных деревьев у вас должна быть возможность восстановить новое кэшированное значение для корневого узла из кэшированных значений исходных корневых узлов. Это означает, что пустому дереву, не содержащему элементов, должно быть присвоено такое значение `e`, при котором `f e x = f x e = x`.

Эта структура, являющаяся ассоциативной бинарной операцией с элементом, не влияющим на итог (с так называемым нейтральным элементом), называется *моноидом* (monoid) и имеет соответствующий класс в модуле `Data.Monoid`:

```
class Monoid a where
   mempty :: a             -- нейтральный элемент
   mappend :: a -> a -> a -- ассоциативная бинарная операция
   mconcat :: [a] -> a
```

Обычно, чтобы сократить запись, вместо `mappend` используют синоним `(<>)`.

Теперь можно написать более универсальную функцию `treeInsert`. Обратите внимание: в этой универсальной версии нужно применять оператор `(<>)` как для поддеревьев, так и для информации в каждом узле. В версии, вычисляющей минимальные элементы, можно воспользоваться тем фактом, что значения в дереве выстроены по порядку, но в универсальной версии этим воспользоваться нельзя.

```
treeInsert4 :: (Ord v, Monoid c) => v -> c -> BinaryTree3 v c -> BinaryTree3 v c
treeInsert4 v c (Node3 v2 c2 l r) = case compare v v2 of
                      EQ -> Node3 v2 c2 l r
                      LT -> let newLeft = treeInsert4 v c l
                                newCache = c2 <> cached newLeft <> cached r
                            in Node3 v2 newCache newLeft r
                      GT -> let newRight = treeInsert4 v c r
                                newCache = c2 <> cached l <> cached newRight
                            in Node3 v2 newCache l newRight
```

```
treeInsert4 v c Leaf3             = Node3 v c Leaf3 Leaf3

cached :: Monoid c => BinaryTree3 v c -> c
cached (Node3 _ c _ _) = c
cached Leaf3           = mempty
```

Класс `Monoid` относится к такому типовому классу, у которого только для одного типа может быть несколько реализаций, что привело к созданию объявления `newtype` для некоторых широко распространенных типов. К наиболее важным можно отнести тип `All`, в котором реализуется моноидальная структура `Bool` под операцию `(&&)` с нейтральным элементом `True`, а также тип `All`, в котором делается то же самое с `(||)` и нейтральным элементом `False`. Числа также допускают две моноидальные структуры: в структуре `Sum` в качестве операции используется сложение, в качестве нейтрального элемента — нуль, а в структуре `Product` — умножение и единица соответственно.

Фактически, если вы желаете использовать этот универсальный алгоритм вставки в кэш, для чисел должна быть предоставлена еще одна моноидальная структура. В следующем фрагменте показан код, необходимый для объявления `newtype` наряду с новым экземпляром. Обратите внимание на то, что в этом коде для чисел с плавающей точкой используется элемент бесконечности, который может быть получен через операцию 1/0:

```
newtype Min = Min Double deriving Show

instance Monoid Min where
  mempty = Min infinity where infinity = 1/0
  mappend (Min x) (Min y) = Min $ min x y
```

Классы типов, связанные с контейнерами

Зачастую при разработке приложения появляется потребность в смене контейнера, используемого для работы со значениями. Поэтому представляется весьма интересной идея сделать шаг назад и подумать о том, что между ними общего, поскольку может появиться возможность абстрагироваться от них и исследовать ряд полезных классов типов.

Функторы

Давайте попробуем написать функцию, применяющую скидку к каждому путеводителю в списке:

```
modifyTravelGuidePrice :: Double -> [TravelGuide] -> [TravelGuide]
modifyTravelGuidePrice m = map (\tg -> tg { price = m * price tg })
```

Теперь то же самое в проекции или в дереве:

```
modifyTravelGuidePriceMap :: Double -> M.Map a TravelGuide -> M.Map a TravelGuide
modifyTravelGuidePriceMap m = M.map (\tg -> tg { price = m * price tg })

modifyTravelGuidePriceTree :: Double -> T.Tree TravelGuide -> T.Tree TravelGuide
modifyTravelGuidePriceTree m = fmap (\tg -> tg { price = m * price tg })
```

Здесь уже проявляется некий образец: все эти контейнеры позволяют применять функцию внутри структуры данных:

```
map   :: (a -> b) -> ([a]       -> [b])
M.map :: (a -> b) -> (M.Map k a -> M.Map k b)
fmap  :: (a -> b) -> (T.Tree a  -> T.Tree b) -- версия для деревьев
```

Тип данных, поддерживающий такую функцию, как, например, `map`, называется *функтором* (functor). Соответствующий класс определяется следующим образом:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Итак, теперь вы можете написать самую универсальную функцию для изменения цен путеводителей:

```
modifyTravelGuidePrice' :: Functor f => Double -> f TravelGuide -> f TravelGuide
modifyTravelGuidePrice' m = fmap (\tg -> tg { price = m * price tg })
```

Здесь можно заметить весьма странный факт, касающийся класса `Functor`: в определении `fmap` переменная типа, соответствующая экземпляру, применяется к другой переменной типа, вместо того чтобы использоваться такой, как она есть. Это означает, что те типы, которые предполагается сделать функторами, должны принимать один параметр типа. Например, тип `IntSet`, который ничего не принимает, не может иметь такой экземпляр (даже при том, что концептуально это функтор).

Способ, который используется компилятором Haskell для проверки правильности применения параметров типа, зависит от системы *видов* (kinds). Знание этого факта может помочь вам понять смысл некоторых сообщений об ошибках. До сих пор нам было известно, что значения, функции и конструкторы имеют связанный с ними тип, но сами типы также упорядочиваются на основе уровня применения. Для начала все основные типы, такие как `Char` или `Integer`, имеют вид `*`. Типы, для полного применения которых требуется один параметр, например `Maybe`, имеют вид `* -> *`. Этот синтаксис напоминает тот, что использовался для функций по назначению: если теперь у нас есть `Maybe Integer`, у нас есть тип вида `* -> *`, к которому применяется тип вида `*`. Следовательно, финальным видом для `Maybe Integer` несомненно является `*`.

УПРАЖНЕНИЕ 4.8. ПРОВЕРЯЕМ ФУНКТОР В ДЕЛЕ!

Класс Functor — это один одним из наиболее широко используемых классов в Haskell. Например, к этому семейству относятся как тип Maybe, так и двоичные деревья из последнего раздела. Экземпляр функтора типа Maybe довольно интересен, поскольку он позволяет сократить код, применяющий функцию, когда значением является Just (сопоставление с образцом плюс создание нового значения просто заменяются вызовом функции map).

Напишите соответствующие экземпляры как для Maybe, так и для двоичных деревьев. Чтобы не обидеть компилятор, вам понадобится создать для значений типа Maybe объявление newtype. Из всех просмотренных до сих пор двоичных деревьев выберите в качестве типа для создания экземпляра Functor дерево BinaryTree2. В этом последнем случае следует помнить, что вы должны считаться с инвариантом порядка в дереве, поэтому наилучшим способом написания функции map могут быть повторяющиеся вызовы функции treeInsert2 применительно к пустому дереву.

Хотя понятие функтора пришло через понятие контейнера, само оно гораздо шире. Одним из экземпляров функтора, который не укладывается в эти рамки, являет-

ся (->) r: элементы этого типа представляют собой функции вида r -> a, то есть функции, получающие в качестве входных данных значение конкретного типа r (синтаксис Haskell в этом случае помочь не сможет, просто нужно запомнить, что выражение a f b может быть записано как (f) a b, если f полностью состоит из символов, как в случае с ->). Для закрепления идеи попробуем написать тип для соответствующей версии fmap:

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Наипростейшим решением для получения результата нужного типа будет применение первой функции после второй:

```
instance Functor ((->) r) where
fmap f g = f . g
```

В данном случае в основу реализации класса типов положено понятие *вычислительного контекста* (computational context): добавление к любому выражению дополнительного значения типа r, которое может использоваться для управления поведением такого выражения. Как мы увидим в главе 6, это имитирует существование в вашем коде константы типа r.

ПРИМЕЧАНИЕ

Следует заметить, что класс Set не может быть экземпляром Functor. Причина в том, что функция обхода (типа map) для множеств имеет тип Ord b => (a -> b) -> Set a -> Set b, который несовместим с типом Functor, не имеющим никаких ограничений. В настоящее время язык Haskell предоставляет достаточное количество инструментов для создания другого класса типов для функторов, допускающего присутствие класса Set внутри себя. Но это усложнит механизм использования класса типов Functor и приведет в негодность большие объемы уже существующего кода. Поэтому в библиотеки включается простая версия Functor.

Свертываемые классы

Еще одной основной операцией, которую можно проделать с контейнерами, является вычисление некой итоговой информации из всех содержащихся элементов, то есть получение свертки. Для этой реализации концепции есть соответствующий класс типов, который называется Foldable и находится в модуле Data.Foldable. Разница между свертками и функторами состоит в том, что после введения нами такого понятия, как моноиды, свертку можно представить себе двумя способами:

- Подобно списочной функции foldr свертка получает начальное значение и объединительную функцию, после чего применяет эту функцию для каждой пары элементов, начиная с исходного значения.
- Как мы могли видеть, тип с бинарной функцией и специальным значением в точности соответствует определению моноида. Кроме того, мы видели, что объединительная функция в foldr должна быть ассоциативной, как функция (<>) из Monoid.

Эти два определения позволяют получить два способа создания экземпляра класса Foldable: вам нужно определить либо функцию foldr (версия с объединительной функцией), либо функцию foldMap (версия с моноидами):

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr   :: (a -> b -> b) -> b -> t a -> b

  fold    :: Monoid m => t m -> m
  foldr'  :: (a -> b -> b) -> b -> t a -> b
  foldl   :: (a -> b -> a) -> a -> t b -> a
  foldl'  :: (a -> b -> a) -> a -> t b -> a
  foldr1  :: (a -> a -> a) -> t a -> a
  foldl1  :: (a -> a -> a) -> t a -> a
```

Остальные операции соответствуют определениям, предлагаемым по умолчанию, которые могут быть заменены из соображений производительности: `fold` является такой версией `foldMap`, которая просто сочетает в себе контейнер, полный моноидных значений без предварительного применения какой-либо функции. Функция `foldl` соответствует свертке элементов, начиная с «обратной стороны» структуры: мы уже видели, как различаются результаты `foldr` и `foldl`, если объединительная функция не является коммутативной. Версии, чьи имена заканчиваются знаком прим (`'`), являются строгими версиями функций: они будут играть центральную роль в следующей главе.

Свертываемые классы (foldables) имеют в Haskell-коде такое же широкое распространение, как и функторы. В упражнении 4.9 вам предлагается предоставить экземпляры тех же классов, что и в упражнении 4.8.

УПРАЖНЕНИЕ 4.9. ПРОВЕРЯЕМ СВЕРТЫВАЕМЫЙ КЛАСС В ДЕЛЕ!

Тип Maybe и двоичные деревья можно также свертывать. Напишите соответствующие экземпляры класса Foldable. В данном упражнении учитывайте подсказки и предупреждения из упражнения 4.8.

Как мы уже видели в предыдущей главе, с помощью сверток могут быть выражены самые разные алгоритмы. Модуль `Data.Foldable` включает большинство из них, например `maximum` и `elem`. Одним из простых способов сделать ваши функции более универсальными является скрытие функций с теми же именами из модуля `Prelude` и их импорт с помощью класса `Foldable`.

ПРИМЕЧАНИЕ

То, что модуль Prelude включает специализированные определения для списков вместо универсальных версий, использующих классы Functor и Foldable, может вызвать удивление. Причина в том, что для новичка наличие функций, работающих только с [a], помогает разобраться с первыми сообщениями об ошибках, с которыми он может столкнуться, поскольку в них не применяются классы типов. Но теперь, когда вы знаете об их существовании, вы должны стремиться достичь как можно более высокой степени абстракции.

В данном разделе мы написали экземпляры классов `Functor` и `Foldable` для различных типов данных. Из-за их довольно интенсивного использования разработчики GHC решили также обеспечить автоматическое извлечение экземпляров этих классов типов. Однако поскольку эта возможность в Haskell Report не упомянута, для того чтобы данный механизм заработал, вам нужно включить ряд расширений, а именно расширения `DeriveFunctor` и `DeriveFoldable`. Следует заметить, что в слу-

чае типа с несколькими параметрами варианты для проекции или свертки всегда располагаются в объявлении на последнем месте.

TYPECLASSOPEDIA

Множество рассмотренных здесь классов типов, включая Monoid, Functor и Foldable, подробно документированы в Typeclassopedia по адресу http://www.haskell.org/haskellwiki/Typeclassopedia.

Выводы

В этой главе мы рассмотрели ряд механизмов экосистемы Haskell:

- Мы изучили способ создания пакетов и управления ими с помощью инструментального средства Cabal, а также способ задания зависимостей, позволяющий многократно использовать массу доступных в Hackage библиотек.
- Мы познакомились с рядом новых контейнеров, включая проекции, множества, деревья и графы. Стремление к осмыслению их общих черт привело к открытию таких понятий, как функтор и свертываемый класс.
- Мы выяснили причину, по которой оператор `(+)` способен работать с несколькими числовыми типами. Ответ кроется в специальном полиморфизме и классах типов. Мы не только научились объявлять новый класс типов, описывающий общие представления для нескольких типов, но и изучили способы создания экземпляров этого класса для того или иного типа. Кроме того, мы увидели, что классы и экземпляры могут зависеть от ограничений класса в отношении других типов или параметров.
- Мы познакомились со встроенными классами: класс `Eq` описывает равенство, класс `Ord` — упорядоченность, класс `Num` и его производные — числа, класс `Default` — значения, предлагаемые по умолчанию для типа данных.
- На примере разработки специального двоичного дерева с кэшем мы узнали о нескольких неочевидных этапах создания нового типа данных. В частности, мы увидели, как классы типов позволяют обобщать принимаемые типом значения, рассматривая конкретную моноидальную структуру, которая весьма востребована среди типов данных в Haskell.

Лень и бесконечные структуры

5

В предыдущих главах мы познакомились с несколькими столпами Haskell-программирования, которые, несмотря на чистую функциональную парадигму и строго типизированную природу языка, позволяют создавать эффективные конструкции типов, реализуя параметрический полиморфизм и классы типов. Эта глава посвящена уникальной вычислительной модели Haskell, основанной на *лени* (laziness), и последствиям выбора данной модели.

Если говорить коротко, то в данном контексте лень, или концепция отложенных вычислений, означает, что вычисляется только необходимая часть выражения, причем делается это по возможности в самый последний момент. Например, если взять такое выражение, как `head [2+3, 5*7]`, то в процессе вычисления кода умножение не выполняется никогда, поскольку его значение не имеет отношения к результату выражения: функция `head` задействует только первый элемент списка. Интуитивно можно понять, что данный способ выполнения вычислений существенно отличается от такового в других языках программирования. Это открывает возможности для ряда интересных новых идиом, например для работы с бесконечными и цикличными структурами, не заботясь при этом об их особой природе.

Однако вам как разработчику также следует знать обо всех издержках этой модели, особенно о тех, которые касаются вопросов расходования памяти и производительности. Поэтому мы рассмотрим наиболее распространенные проблемы, возникающие в Haskell-коде из-за отложенных вычислений, и познакомимся с идеей *строгости* (strictness), позволяющей добиться преодоления этих проблем. Строгую аннотацию можно применять при сопоставлении с образцом и в объявлениях `data`. Для выявления существующих проблем отличным средством станет профайлер расходования времени и памяти, который поставляется в комплекте с GHC: в данной главе рассмотрены основные приемы его использования.

Бесконечное количество машин времени

Прежде всего, мы собираемся выяснить, как Haskell справляется с бесконечными и цикличными структурами, не создавая трудностей для разработчика. В данном разделе мы рассмотрим объявления и порядок использования подобных структур, чтобы это было интуитивно понятно. А в следующем разделе мы узнаем, как язык Haskell обеспечивает представление этой информации при сохранении работоспособности кода.

Как уже упоминалось, некоторые виды машин времени допускают путешествия в прошлое только в один строго определенный год. Однако магазин известен тем,

что в нем всегда найдется нужная вам машина времени для путешествия в любой заданный год. Проблема здесь в том, что история нашего мира может не закончиться никогда, следовательно, и множество машин времени бесконечно! Прежде всего, давайте объявим компактный тип данных, содержащий наши машины времени, с указанием производителя и года, в который разрешено путешествие:

```
data TimeMachine = TM { manufacturer :: String, year :: Integer } deriving (Eq, Show)
```

Вы можете написать функцию для возвращения всех машин времени, начиная с года *n*. Убедитесь в том, что здесь нет никакого предохранителя или некоего базового случая, останавливающего дальнейшую выдачу информации о машинах времени:

```
timeMachinesFrom :: String -> Integer -> [TimeMachine]
timeMachinesFrom mf y = TM mf y : timeMachinesFrom mf (y+1)
```

Исходя из этого можно получить информацию обо всех машинах времени, изготовленных компанией Timely Inc., начиная с 100-го года:

```
timelyIncMachines :: [TimeMachine]
timelyIncMachines = timeMachinesFrom "Timely Inc." 100
```

Теперь, если загрузить файл в GHC-интерпретатор, мы сможем получить первый элемент списка, используя встроенную функцию `take`. При вычислениях, касающихся машин производства Timely Inc., система не входит в какой бы то ни было бесконечный цикл:

```
*Chapter5.Infinite> take 3 timelyIncMachines
[TM {manufacturer = "Timely Inc.", year = 100}
,TM {manufacturer = "Timely Inc.", year = 101}
,TM {manufacturer = "Timely Inc.", year = 102}]
```

Можно также попробовать найти первую из этих машин, на которой можно отправиться в следующий после 2018-го год:

```
*Chapter5.Infinite> import Data.List
*Chapter5.Infinite Data.List> find (\(TM { year = y }) -> y > 2018)
timelyIncMachines
Just (TM {manufacturer = "Timely Inc.", year = 2019})
```

Но если попытаться вычислить длину списка или найти несуществующий элемент (в данном случае машину времени, позволяющую отправиться в 10-й год), интерпретатор войдет в бесконечные вычисления и никогда не вернет управление вызвавшей его программе. Для его остановки нужно на консоли нажать комбинацию клавиш Ctrl+C. В EclipseFP закрыть программу можно, щелкнув на кнопке с красным квадратом.

```
*Chapter5.Infinite> length timelyIncMachines
-- Никогда не остановится
*Chapter5.Infinite> find (\(TM { year = y }) -> y == 10) timelyIncMachines
-- Никогда не остановится
```

Так или иначе, Haskell знает, как обращаться с бесконечными списками, учитывая, что во время выполнения программы вы наблюдаете только их ограниченную часть. В то же время вычисление такого выражения, как `length timelyIncMachines`, требует обхода всего списка, следовательно, оно никогда не закончится.

Бесконечные списки могут пригодиться и в некоторых других ситуациях. Например, в предыдущей главе мы написали функцию, возвращающую при получении входящего списка новый список кортежей, в которых каждый элемент наделяется его позицией в списке. Для этого использовалась функция `zip`:

```
*Chapter5.Infinite> (\list -> zip [1 .. length list] list) "abcd"
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

Но для получения нужного результата приходилось обходить список дважды: один раз для получения его длины, второй — для объединения исходных и полученных данных в кортежи с помощью функции `zip`. Один из более удачных способов решения данной задачи основан на том, что выполнение функции `zip` останавливается, когда один из списков заканчивается. Значит, в качестве первого аргумента функции `zip` можно указать бесконечный список номеров. Прежде всего, давайте напишем функцию, содержащую список всех номеров от единицы и больше:

```
allNumbers :: [Integer]
allNumbers = allNumbersFrom 1

allNumbersFrom :: Integer -> [Integer]
allNumbersFrom n = n : allNumbersFrom (n+1)
```

Теперь можно написать ту же самую функцию, затратив меньше усилий:

```
*Chapter5.Infinite> zip allNumbers "abcd"
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

А еще лучше воспользоваться имеющимися в Haskell диапазонами, задающими бесконечные списки: форма записи `[1 ..]` описывает список, начинающийся с 1 и продолжается до конца целочисленных элементов (которого в данном случае не существует, потому что целых чисел бесконечное множество):

```
*Chapter5.Infinite> zip [1 .. ] "abcd"
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

ПРИМЕЧАНИЕ

Форма записи [e ..] не обязательно предполагает создание бесконечного списка, скорее она показывает, что в списке содержатся все элементы, которые больше чем e. Например, [False ..] является эквивалентом записи [False, True], поскольку в типе Bool используется порядок, определяющий, что False < True, а других элементов данного типа не существует.

Бесконечные списки позволяют применить еще больше приемов. Давайте посмотрим теперь на интересный способ определения чисел Фибоначчи. Если вы помните из предыдущих глав, n-е число Фибоначчи определяется как сумма чисел Фибоначчи $n-1$ и $n-2$. Давайте взглянем на это с разных точек зрения: предположим, что имеется список всех чисел Фибоначчи. Позиция n в этом списке содержит число Фибоначчи. Если взять хвост списка (`tail`), то список переместится на один шаг вперед: теперь в позиции n будет храниться число Фибоначчи, которое находилось в позиции $n+1$. И наконец, прикосновение к волшебству: если провести поэлементное суммирование, будут получены числа Фибоначчи, но смещенные на две позиции. Эта ситуация показана на рис. 5.1.

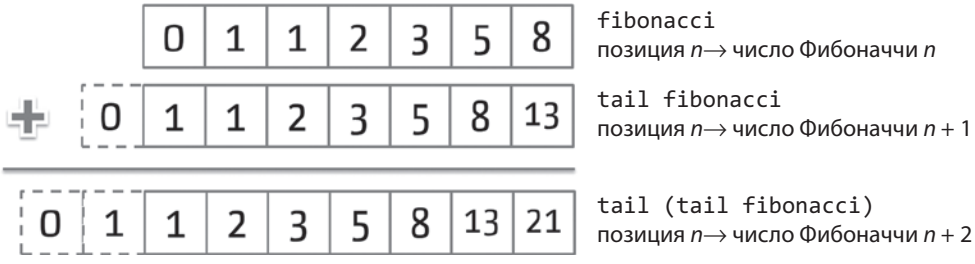


Рис. 5.1. Графическое представление свойств списка чисел Фибоначчи

Этим наблюдением можно воспользоваться для определения списка всех чисел Фибоначчи: первыми двумя элементами будут 0 и 1 (это установлено определением). А остальная часть списка получается путем поэлементного сложения со смещением списка на один элемент вперед. Это поэлементное сложение — именно то, что может быть получено при использовании кода `zipWith (+)`.

```
fibonacci :: [Integer]
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

Получение n-го числа Фибоначчи теперь эквивалентно получению элемента в позиции $n-1$ (как и в языке C или Java, списки в Haskell индексируются начиная с 0). Функция `(!!)` в модуле `Data.List` — именно то, что вам нужно:

```
*Chapter5.Infinite> import Data.List
*Chapter5.Infinite Data.List> fibonacci !! 20
6765
```

Кроме использования списочных диапазонов и конструирования вручную функций, возвращающих бесконечные списки, для их генерации можно воспользоваться модулем `Prelude`, содержащим ряд необходимых встроенных функций. В качестве специального предложения компания Timely Inc. снабдила нас бесконечным количеством машин времени для путешествия в 2020 год. Один из способов определить это может быть реализован так:

```
infinite2020Machines :: [TimeMachine]
infinite2020Machines = TM "Timely Inc." 2020 : infinite2020Machines
```

Однако задачу можно решить и другим способом — за счет использования комбинатора `repeat`, который просто создает в списке бесконечные копии одного и того же значения:

```
*Chapter5.Infinite> take 3 $ repeat $ TM "Timely Inc." 2020
[TM {manufacturer = "Timely Inc.", year = 2020}
,TM {manufacturer = "Timely Inc.", year = 2020}
,TM {manufacturer = "Timely Inc.", year = 2020}]
```

Кроме одного значения можно также повторять множество значений по порядку. В качестве специального предложения у нас может быть множество машин времени с 20-процентной скидкой, но с конкретной особенностью, что нам нужно продать одну машину для отправки в 2005 год, одну в 1994 год, одну в 908 год, а затем снова

начать продажу с машины для отправки в 2005 год. Этот бесконечный список объявляется с помощью функции `cycle`:

```
specialOffer :: [TimeMachine]
specialOffer = cycle [TM m 2005, TM m 1994, TM m 908]
               where m = "Timely Inc."
```

Увидеть, как значения повторяются, можно, просмотрев первые четыре значения:

```
*Chapter5.Infinite> take 4 specialOffer
[TM {manufacturer = "Timely Inc.", year = 2005}
,TM {manufacturer = "Timely Inc.", year = 1994}
,TM {manufacturer = "Timely Inc.", year = 908}
,TM {manufacturer = "Timely Inc.", year = 2005}]
```

Значения не обязательно всегда должны быть одинаковыми. Функция `iterate` генерирует значения, применяя к значению функцию для получения второго значения, затем применяя ту же самую функцию к этому второму значению для получения третьего значения, и т. д. Мы можем увидеть бесконечный список, который будет генерироваться следующим образом:

```
iterate f x = [ x, f x, f (f x), f (f (f x)), ... ]
```

Это дает еще один способ реализации чисел Фибоначчи. В частности, список `fibonacci2` будет содержать пары значений, позволяющие в позиции *n* найти кортеж (число Фибоначчи *n*, число Фибоначчи *n*+1). Из одного такого кортежа можно создать следующий элемент, смещаясь на позицию влево и добавляя два числа, чтобы получить число Фибоначчи *n*+2. В коде это транслируется в следующие строки:

```
fibonacci2 :: [Integer]
fibonacci2 = map fst $ iterate (\(n,n1) -> (n1,n+n1)) (0,1)
```

Эту новую функцию можно использовать точно так же, как предыдущую:

```
*Chapter5.Infinite> fibonacci2 !! 20
6765
```

В упражнении 5.1 нам предстоит увидеть, что бесконечные списки позволяют даже заглянуть в историю.

УПРАЖНЕНИЕ 5.1. РЕШЕТО ЭРАТОСФЕНА

Греческий математик Эратосфен жил в третьем тысячелетии до нашей эры. Одно из его наиболее известных изобретений — решето простых чисел. Это решето дает алгоритм получения списка простых чисел, который работает следующим образом:

1. Сначала создается список всех чисел начиная с 2.
2. Берется первое число (в данном случае — 2), и из списка выбрасываются все кратные ему числа, то есть все числа *n*, остатком от деления которых на 2 является нуль.
3. Берется следующее число (в данном случае — 3) в отфильтрованном списке, и операция повторяется: отфильтровываются все числа, кратные этому числу.
4. Повторяется последний шаг с первым числом слева от предыдущего.

Реализуйте решето Эратосфена, используя технологии, рассмотренные в данном разделе. В частности, решение должно принять форму primes :: [Integer], содержащую все простые числа.

Модель ленивых вычислений

Теперь уже вы должны были убедиться в том, что Haskell действительно может работать с бесконечными значениями (или, по крайней мере, с бесконечными списками). Тем не менее это может выглядеть как нечто нереальное. Разумеется, никакой магии здесь нет: эта возможность является результатом стратегии, которой Haskell следует для вычисления выражений и которая существенно отличается от стратегий в других языках программирования. В данном разделе мы познакомимся с этой стратегией и рассмотрим некоторые связанные с ней проблемы.

Понятие вычислений в Haskell

Большинство языков программирования следуют *модели строгих вычислений* (strict evaluation model): как только встречается составное выражение, оно тут же преобразуется в более простую версию (возможно, включающую вызовы функций или методов), и это делается перед вычислением более длинного выражения. В частности, аргументы функций вычисляются перед передачей управления телу самой функции. Давайте посмотрим на примере шаги, которые выполняются в рамках этой модели для вычисления простого выражения:

```
head [3+2, 7*5] => head [5, 35]  -- вычисляются аргументы функции
                => 5             -- затем выполняется сама функция
```

Следуя этой модели вычислений, выражение вида `head timelyIncMachines` вызовет вхождение в бесконечный цикл, потому что здесь отсутствует точка, в которой нужно остановиться, и происходит перемещение все дальше и дальше по списку. В следующем примере воспроизводятся первые шаги этого бесконечного цикла, который, судя по многоточию, продолжится. Не пожалейте времени на разбор этого примера, чтобы понять, почему код входит в бесконечный цикл.

```
head timelyIncMachines => head (timeMachinesFrom "Timely Inc." 100)
   => head (TM "Timely Inc." 100 : timeMachinesFrom "Timely Inc." 101)
   => head (TM "Timely Inc." 100 : TM "Timely Inc." 101 : timeMachinesFrom "Timely
Inc." 102)
   => head (TM "Timely Inc." 100 : TM "Timely Inc." 101 : TM "Timely Inc." 102 : ...)
```

В отличие от этого в Haskell вычисление выражений максимально откладывается. В ранее приведенном примере Haskell изначально не вычисляет выражения, составляющие элементы списка. Когда обнаружится вызов функции `head`, Haskell получит первый элемент, представляющий собой еще не вычисленное выражение 3+2. Поскольку результат этого выражения должен выводиться на экран, произойдет вычисление только сложения с получением конечного значения, то есть значения 5. Такой вид вычислений называется *нестрогим* (non-strict), или *ленивым* (lazy).

Идея «вычислить как можно позже» приходит интуитивно. Но, к примеру, с бесконечным списком такой подход применить напрямую нельзя, поскольку для получения головы списка нужно войти в тело `timeMachinesFrom`, что обернется вхождением в цикл. Здесь есть еще один нюанс: по умолчанию Haskell вычисляет выражение до тех пор, пока не будет найден конструктор. Остальная часть значения не вычисляется: ее место занимает местозаполнитель, показывающий, как может быть вычислено это конкретное поле. Такой местозаполнитель называется *переходником* (thunk).

Если применить только что описанный порядок действий к вычислению головы бесконечного списка, для `timeMachinesFrom` будут созданы лишь конструктор `(:)` с переходником для элемента и еще с одним переходником для всего остального списка. Когда к списку будет применена функция `head`, вернется первый из переходников. Если нужно показать значение на экране, переходник должен быть развернут, и за рецептом создания значения должно последовать создание этого значения. На рис. 5.2 эти этапы представлены графически. Первые три перехода фактически являются вычислением переходников. На четвертом этапе достигается точка, в которой можно вычислить выражение `case`, поскольку мы уже можем выбрать образец на основе вычисленного значения. Завершающим этапом является обычное вычисление переходника. Очередной переходник, вычисляемый на каждом этапе, обведен более жирной рамкой.

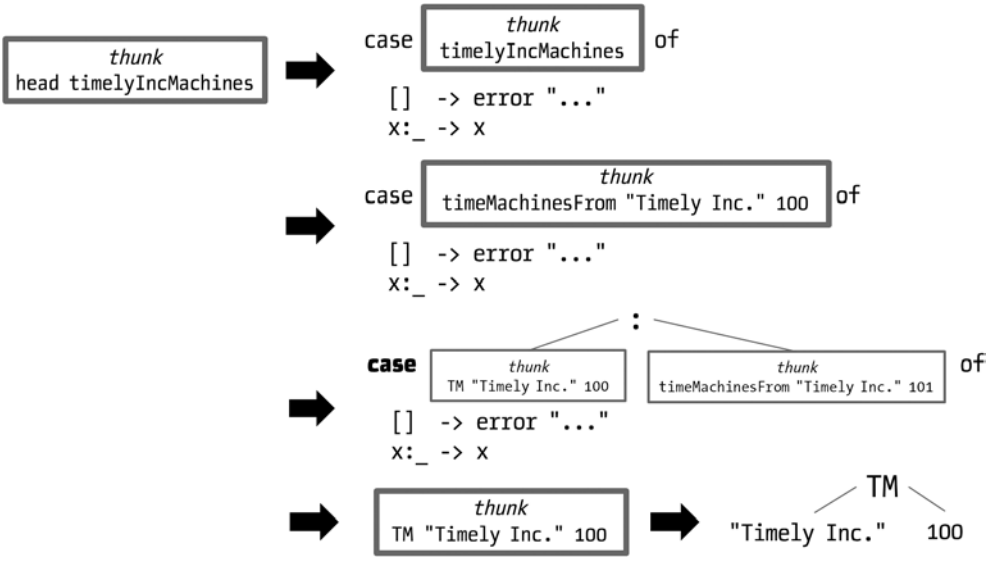


Рис. 5.2. Вычисление выражения head timelyIncMachines

Одна весьма важная особенность ленивых вычислений заключается в том, что после вычисления переходника результат сохраняется, поэтому если он где-нибудь снова понадобится, второй раз его вычислять уже не нужно. Это великолепное свойство, поскольку оно означает, что вычислять каждое выражение в приложении нужно только один раз. Кроме того, базовой особенностью Haskell является то, что этот язык помогает в совместном использовании переходников, ссылающихся на одни и те же выражения, а это означает, что он позволяет многократно использовать вычисления из какой-нибудь части программы в других местах. Например, на рис. 5.3 показано, как меняется распределение памяти при выполнении кода (`head allNumbers`, `head (tail allNumbers)`, `tail allNumbers`). Поскольку `allNumbers` — это список, исполняющая среда Haskell сохраняет ссылки на данное выражение во всех местах, где появляется это значение. На рис. 5.3 это показано в виде разных стрелок, указывающих на одно и то же выражение. Упражнение 5.2 призвано проверить ваше понимание этой концепции совместного использования.

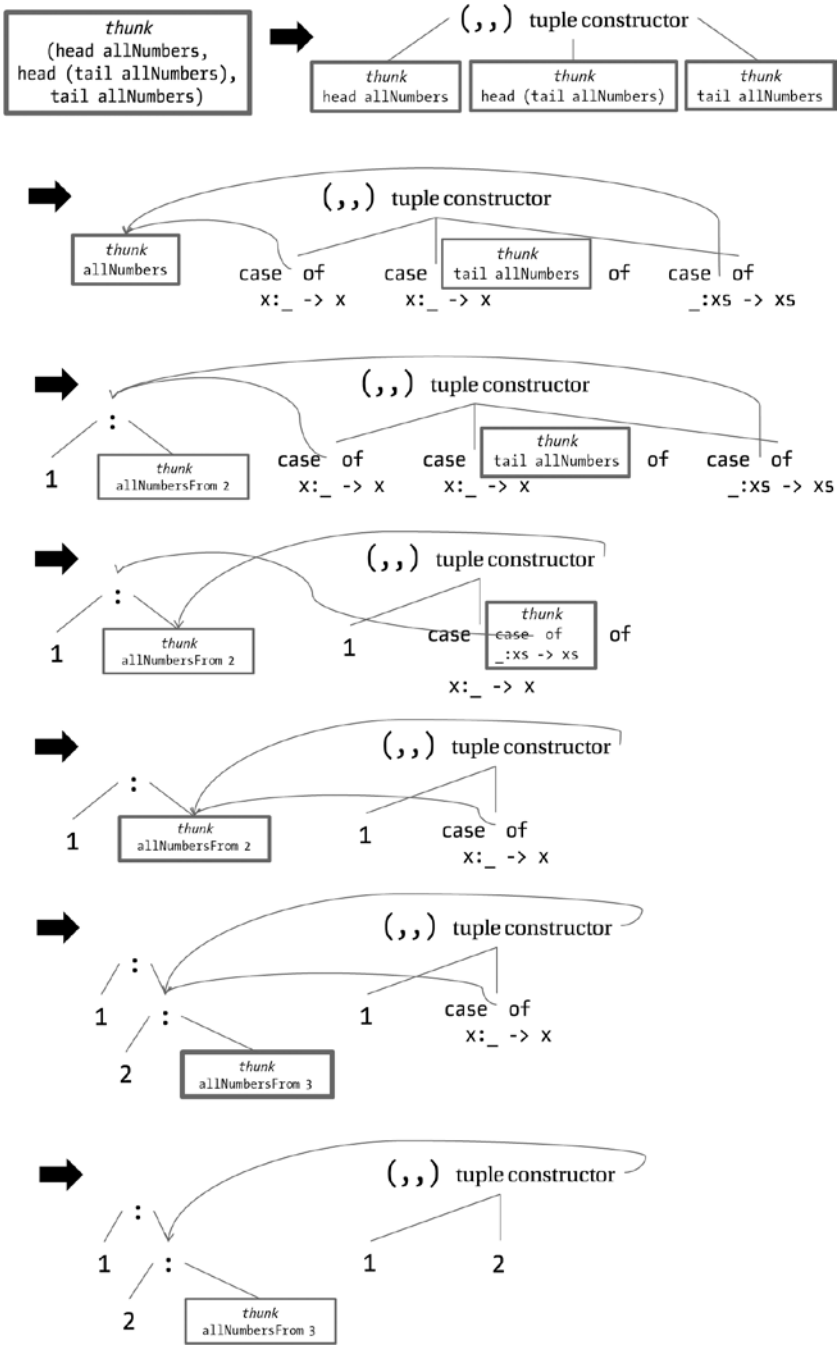


Рис. 5.3. Вычисление выражения (head allNumbers, head (tail allNumbers), tail allNumbers)

УПРАЖНЕНИЕ 5.2. ВЫЧИСЛЕНИЕ ЧИСЕЛ ФИБОНАЧЧИ

Запишите этапы вычисления выражения fibonacci !! 3, где fibonacci является бесконечным списком чисел Фибоначчи, определенным ранее в данной главе.

Это было бы невозможно сделать на языке, который допускает вывод значения на экран в ходе его вычисления. Давайте предположим, что в ходе вычисления список `allNumbers` выводит строку `"Natural numbers rule!"`. Если для `allNumbers` используется одно и то же значение, строка будет выведена только один раз. Но во многих языках, включая C и Java, эта строка должна появиться три раза, по одному разу для каждой ссылки на `allNumbers`. Мы уже видели, что побочные эффекты не позволяют применять эти оптимизации совместного использования, что является ключевым моментом в достижении высокой производительности Haskell-программ.

Следует заметить, что совместно использоваться будут только выражения. Это не нужно путать с запоминанием функции, то есть с кэшированием результатов для аргументов, которые уже были предоставлены. Например:

```
(allNumbersFrom 1, allNumbersFrom 2)
```

При таком вызове даже если из `allNumbersFrom 1` будет вызываться `allNumbersFrom 2`, вычисление `allNumbersFrom 2` в `allNumbersFrom 1` и в предыдущем выражении совместно использоваться не будет.

Осталось ответить еще на один, последний, вопрос: как представлять цикличные структуры? Ответ состоит в том, что при одних и тех же объявлениях Haskell обслуживает цикл в памяти самостоятельно. Например, для случая `repeat e` вычисления показаны на рис. 5.4.

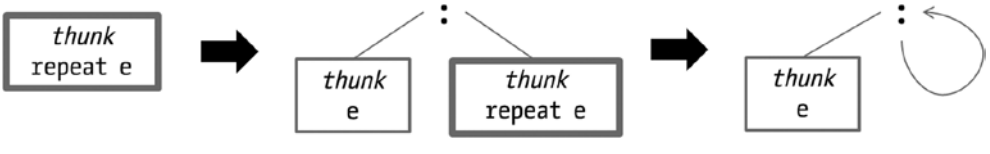


Рис. 5.4. Вычисление выражения repeat e

ВЫЧИСЛИТЕЛЬНЫЕ СТРАТЕГИИ

В этом разделе мы увидели два примера вычислительных стратегий: то, как продолжаются вычисления, и то, как вычисляются части выражений. Кроме этих двух были разработаны и другие стратегии.

То что мы назвали «строгими вычислениями», известно также как «вызов по значению». Иногда, особенно в объектно-ориентированных языках, это определение меняется на «вызов по ссылке», где вы получаете значения не как аргументы, а как блоки, содержащие эти значения.

О ленивых вычислениях иногда говорят как о «вызове по необходимости», что является частным случаем более общей стратегии «вызова по имени», в которой аргументы функции не вычисляются до тела функции, а непосредственно подставляются. Разница в том, что при «вызове по имени» одно и то же выражение может вычисляться более одного раза, а вот при «вызове по необходимости» для однократного вычисления используются переходники.

Проблемы стратегии ленивых вычислений

Стратегия ленивых вычислений зачастую является благом, но иногда может принести и неприятности. Как это обычно бывает в компьютерной науке, свои компромиссы есть и в стратегии ленивых вычислений: в данном случае откладывание вычислений до тех пор, пока не понадобится их результат, может привести к меньшему объему вычислений, а также дает право на жизнь некоторым идиомам программирования, недоступным в других языках. В то же время при этом может создаваться множество переходников, что вызывает излишнее расходование памяти и вынуждает операционную систему активно использовать подкачку страниц памяти, а это может замедлить выполнение программы. Давайте изучим эту проблему, воспользовавшись нашими старыми друзьями — свертками.

Сначала изобразим, как вычисляется выражение `foldr (+) 0 [1,2,3]`, четко обозначая переходники: каждый переходник будет содержать внутри себя рецепт по его преобразованию в конкретное значение. Все это показано на рис. 5.5.

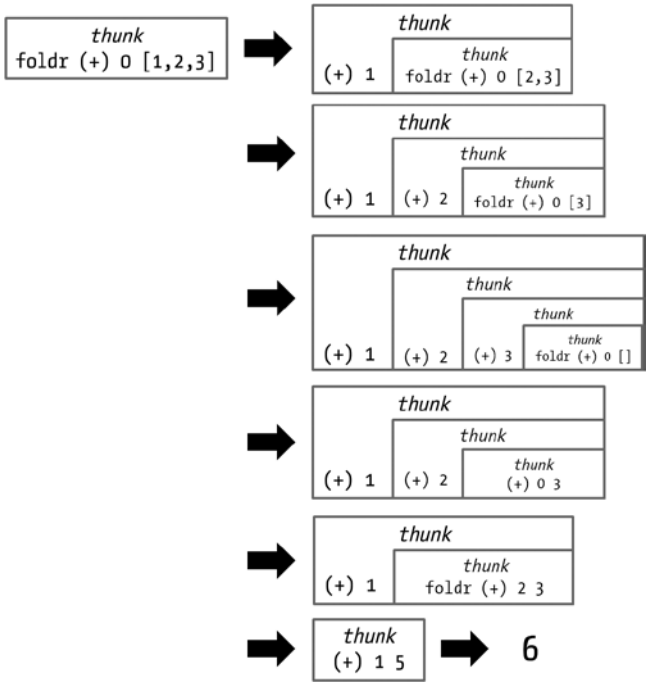


Рис. 5.5. Вычисление выражения foldr (+) 0 [1,2,3]

Пока интерпретатор не доберется до финального этапа вычисления функции `foldr`, он не может перейти к сложению. Это означает, что для каждого элемента в списке создается новый переходник. Теперь вы понимаете, почему при запросе на интерпретацию следующей строки кода компьютер начинает усердно трудиться, а потом останавливается, выдавая ошибку:

```
*Chapter5.Problems> foldr (+) 0 [1 .. 1000000000]
<interactive>: out of memory (requested 1048576 bytes)
```

ПРИМЕЧАНИЕ

На первый взгляд может показаться, что причиной ошибки является большая длина списка. Но если с ним провести какие-нибудь другие вычисления, не создающие промежуточных переходников, например length [1 .. 1000000000], можно увидеть, что система реагирует на это вполне корректно (фактическое быстродействие будет зависеть от того, какой объем памяти на вашей машине выделен для хранения больших целых чисел).

Порядок вычисления с использованием функции `foldr` примерно следующий: `(1 + (2 + (3 + (... + <переходник>))))`. Значит, вычисления не могут продолжаться, потому что в каждой точке в ходе вычислений известен только один аргумент для `(+)`. Следовательно, если использовать скобки по-другому, придав выражению вид `((((1 + 2) + 3) + ...) + <переходник>)`, проблема может быть устранена. Вы уже знаете, как это сделать: нужно воспользоваться функцией `foldl`.

```
*Chapter5.Problems> foldl (+) 0 [1 .. 1000000000]
<interactive>: out of memory (requested 1048576 bytes)
```

Но тут мы сталкиваемся с похожей ситуацией: функция `(+)` на каждом этапе получает все свои аргументы, но поскольку результат нам не требуется до конца списка, создается большое количество переходников.

Решение состоит в *принуждении* к вычислениям: вам нужно приказать компилятору вычислить переходники `(n+m)` до продолжения всех остальных вычислений, отменив тем самым исполнение стратегии ленивых вычислений, предлагаемой по умолчанию. Сделать это позволяет функция `seq` из модуля `Prelude`. В наиболее общей форме принудительное выражение записывается как `` `seq` b``. В результате Haskell гарантирует вычисление выражения, находящегося перед `b`. Обычно `a` является частью выражения `b`. Давайте запишем операцию сложения, используя оператор принуждения, который позволяет избавиться от проблем с памятью:

```
sumForce :: [Integer] -> Integer
sumForce xs = sumForce' xs 0
        where sumForce' []    z = z
              sumForce' (y:ys) z = let s = z + y in s `seq` sumForce' ys s
```

При выполнении выражения `sumForce [1 .. 1000000000]` у интерпретатора может уйти много времени, но проблем с памятью не возникнет, и в конечном итоге ответ будет получен. Идиома `x `seq` f x` встречается настолько часто, что для нее есть специальный оператор *соблюдения строгости* (strict application) `$!`, решающий данную задачу. Таким образом, выделенную в предыдущем примере часть кода можно переписать следующим образом:

```
sumForce' ys $! (z+y)
```

ПРИМЕЧАНИЕ

В предыдущем коде нам опять встретилась уже знакомая схема, похожая на свертку. Модуль Prelude включает в себя функцию foldl', которая принудительно аккумулирует значение перед его передачей на следующий этап. Следовательно, предыдущему примеру можно было бы придать вид foldl' (+) 0 [1 .. 1000000000].

Здесь еще раз нужно подчеркнуть, что Haskell что-либо вычисляет только до того, как обнаруживает конструктор: поля слева остаются переходниками, пока какие-нибудь дополнительные вычисления не потребуют заключенной в них информации. Это справедливо также и для `seq`: если вы хотите, чтобы перед продолжением вычислений была получена какая-то часть большого значения, нужно принудительно предоставить это значение в явном виде (в конце главы будет показано, что если внутри значения не нужны *никакие* переходники, то можно воспользоваться глубоким строгим вычислением, которое задается оператором `deepseq`). Для данного случая этого достаточно, поскольку первый конструктор, который будет вычисляться, после сложения превратится в целое число.

Узнав о принудительных вычислениях, вы не должны впасть в искушение применять их везде, где, по вашему мнению, могут возникать проблемы с памятью. Принудительное вычисление выражений сводит на нет ленивую природу языка и может приводить к случаям, когда предварительного завершения выражения больше не происходит. Подумайте о получении головы бесконечного списка с помощью функции `head`: если заставить Haskell проводить вычисления для всего списка, вычисления никогда не закончатся, войдя в бесконечную цепочку. Если есть подозрения на утечку памяти, сначала нужно провести профилирование для выявления точной причины, а затем хорошенько подумать, не помешает ли использование функции `seq` применимости ваших функций к разным видам ожидаемых аргументов.

Сопоставление с образцом и лень

Как видите, умение «играть» откладыванием вычислений с помощью переходников и принуждений к вычислениям имеет очень большое значение. Поэтому вам нужно четко понимать, когда это требуется. Помимо непосредственного использования оператора `seq` или `($!)`, еще одной ситуацией, когда компилятору или интерпретатору требуется вычислять переходники, является сопоставление с образцом: его нужно вычислять вплоть до той точки, в которой становится понятно, куда должна идти соответствующая ветвь.

В GHC для образцов имеется расширение под названием `BangPatterns`, позволяющее задавать принудительное вычисление некоторых частей образца. Точнее говоря, можно перед любой частью образца поставить знак `!`, в результате при сопоставлении с образцом сначала будет вычислено выражение в этой точке и только потом произойдет сопоставление. Например, можно написать функцию, суммирующую все годы из списка машин времени, используя этот синтаксис как для принудительного сложения на каждом этапе, так и для гарантии того, что также будет вычисляться достижимый для каждой машины времени год (чтобы сложение не имело в качестве второго аргумента переходник):

```
{-# LANGUAGE BangPatterns #-}

sumYears :: [TimeMachine] -> Integer
sumYears xs = sumYears' xs 0
        where sumYears' []            z = z
              sumYears' (TM _ !y :ys) z = let !s = z + y in sumYears' ys s
```

Интересно, что по причине таких принудительных вычислений при сопоставлении с образцом Haskell поддерживает также *отсрочку* вычислений на этапах сопоставления. Для этого служит так называемый *бесспорный образец* (irrefutable pattern): сопоставление с ним всегда удачно, но он деструктурируется только при необходимости в каких-нибудь его составных частях. Один из случаев использования бесспорных образцов предполагает, что функция всегда возвращает значение одних и тех же входных данных. Например, вы разыскиваете элемент в списке и убедились в том, что он существует, поэтому функция `find` всегда будет возвращать один и тот же результат. В данном случае можно ненадолго отсрочить вычисление функции `find`, а вычислять ее только при необходимости в составной части значения.

Чтобы привести более понятный пример, предположим, что у вас есть следующая функция:

```
lengthyOperation = if lengthyPredicate then Just something else Nothing
```

Пусть вы знаете, что `lengthyPredicate` в некоторой ситуации возвращает значение `true`. Напишем обычное сопоставление:

```
case lengthyOperation of
  Just something -> ...
  Nothing        ->
```

Для выбора правильной ветви в этом сопоставлении нужно потребовать принудительного вычисления функции `lengthyOperation`. Однако поскольку нам известно, что выбрана должна быть именно первая ветвь, вычисления можно немного отсрочить, используя бесспорный образец:

```
case lengthyOperation of
  ~(Just something) -> ...
```

Следует помнить, что сопоставление с подобным образцом никогда не может быть неудачным, поэтому в ситуации, когда `lengthyOperation` возвращает `Nothing`, а в теле сопоставления используется `something`, вы получите ошибку:

```
Prelude> case Nothing of ~(Just e) -> "hello, " ++ e ++ "!"
"hello, *** Exception: Irrefutable pattern failed for pattern (Data.Maybe.Just e)
```

ПРИМЕЧАНИЕ

Бесспорные образцы используются довольно редко, намного реже принудительных вычислений, но в некоторых случаях они становятся ключом для достижения высокой производительности кода. Не обязательно знать обо всех случаях, в которых они могут понадобиться, но знание об их существовании может пригодиться, особенно при изучении содержимого некоторых встроенных функций.

СТРОГИЕ ФУНКЦИИ

Рано или поздно в документации по какому-либо пакету вы прочитаете, что функция строга к одному или нескольким своим аргументам. На высоком уровне это означает, что аргумент нужно будет вычислять, если он имеет форму переходника, и вам придется позаботиться о том, чтобы на этом месте было выражение, не вызывающее прекращения работы программы.

Формально в Haskell имеется каноническое значение под название undefined (неопределенное значение), представляющее все те вычисления, которые не заканчиваются. Поскольку это зна-

чение никогда не возвращается, для него может быть определен какой угодно тип, и вот что мы имеем: undefined :: a. Кстати, такое определение типа делает из undefined прекрасный местозаполнитель для той части кода, которая еще не написана, когда вам нужно убедиться в том, что ваш текущий код проходит проверку типов.

Функция f называется строгой к своему аргументу, если f undefined = undefined. То есть если дан аргумент, не вызывающий прекращения выполнения, то и сама функция работу не прекращает. Одним из примеров строгой функции является head. А функция, имеющая определение g x = 1, строгой не является, поскольку, получая любой аргумент, она возвращает 1.

Интуитивно понятие строгости по отношению к чему-то означает, что это что-то не проверяется. Способ задания строгости функции может быть менее понятным, чем в предыдущих примерах. Например, head undefined действительно имеет тип undefined, но head (1 : undefined) уже не имеет тип undefined.

Профилирование с помощью GHC

Компилятор GHC может использоваться для получения статистики о ходе выполнения программы, позволяющей получить более глубокое представление о том, на что тратятся вычислительные мощности. В данном разделе основное внимание будет сосредоточено на двух видах профилирования: *профилирование потребления времени* (time profiling) дает информацию о времени, затраченном на каждую из функций в системе, а *профилирование потребления памяти* (memory profiling) позволяет отслеживать значения, требующие наибольшие объемы памяти. Вывод системы профилирования в GHC приписывает время и память к так называемым *центрам затрат* (cost centers), поэтому информация и суммарные данные всегда связаны с ними. По умолчанию центры затрат назначаются функциям, но с помощью аннотаций могут быть добавлены и дополнительные центры.

Для профилирования создавать код в виде библиотеки больше нельзя: нужно создавать полностью исполняемый код, который может быть запущен из командной строки. Поэтому сначала давайте посмотрим на изменения, которые следует внести в Cabal-файл, чтобы включить в него исполняемые файлы.

Каждый Haskell-проект может иметь несколько исполняемых файлов, идентифицируемых по имени, и содержать ссылку на файл, включающий точку входа в приложение в виде функции `main` типа `IO()`. Но пока нам важно знать только то, что тип `IO` позволяет выполнять некие сопутствующие действия вроде вывода на экран; углубленное внимание этому типу будет уделено в главе 9. Каждый из этих исполняемых файлов определяется в Cabal-файле как новый блок: имя указывается после ключевого слова `executable` в начале блока, а файл, содержащий функцию входа, указывается в свойстве `main-is`. Важно заметить, что свойство `main-is` требует ссылки на сам файл (включая расширение), а не на имя модуля, подобно другим свойствам. Дополнительные модули определяются в свойстве `other-modules`, а зависимости указываются в свойстве `build-depends` как библиотечные блоки.

Например, следующее объявление включает исполняемые файлы `profiling-example`, где главная функция определена в файле Main.hs и где используется модуль `Chapter5.Annotations`. Единственная зависимость относится к пакету `base`:

```
executable profiling-example
  build-depends:   base >= 4
```

```
  hs-source-dirs:  src
  main-is:         Main.hs
  other-modules:   Chapter5.Annotations
```

Чтобы проверить работоспособность объявления, создайте файл Main.hs. Модуль, в котором определяется точка входа, должен обязательно называться `Main` и содержать функцию `main :: IO ()`, следовательно, и файл должен быть назван соответствующим образом. В данном примере используется исполняемый файл, который с помощью функции `putStrLn` просто выводит слово `"Hello!"`.

```
module Main where

main :: IO ()
main = putStrLn "Hello!"
```

Затем в папке проекта нужно выполнить команды `cabal configure` и `cabal build`. Одна из строк на стадии сборки должна иметь вид `Linking dist/build/profiling-example/profiling-example` — тем самым мы показываем, где будут находиться исполняемые файлы при компиляции. Теперь можно запустить пример на консоли:

```
$ ./dist/build/profiling-example/profiling-example
Hello!
```

Чтобы включить профилирование, нужно соответствующим образом оповестить компилятор. В частности, нужно добавить три флага: флаг `-prof` включает профилирование, флаг `-fprof-auto` обеспечивает автоматический выбор привязок в качестве центров затрат, а флаг `-rtsopts` включает передачу ключей времени выполнения исполняемому файлу. Эти флаги указываются в свойстве `ghc-options` внутри блока исполняемых файлов:

```
executable profiling-example
  -- другие свойства
  ghc-options: -prof -fprof-auto -rtsopts
```

ПРИМЕЧАНИЕ

В GHC доступно множество других флагов. Их полный список можно получить по адресу http://www.haskell.org/ghc/docs/latest/html/users_guide/flag-reference.html. Один из наиболее полезных флагов — -Wall, предписывающий GHC сигнализировать обо всевозможных предупреждениях в коде. Предупреждения не являются ошибками компиляции, но дают весьма полезную информацию для диагностики кода, который может вести себя не так, как ожидалось.

Как обычно, для Cabal-файлов в EclipseFP включен графический редактор, позволяющий указывать исполняемые файлы и вносить изменения в большинство наиболее распространенных свойств в блоках исполняемых файлов. В частности, на вкладке Executables (Исполняемые файлы) показаны все приложения, упомянутые в файле проекта (рис. 5.6). Для добавления нового исполняемого файла нужно просто щелкнуть на кнопке в виде знака «плюс» и указать имя добавляемого файла. Роль каждого модуля изменяется в столбце Modules (Модули), у флагов компилятора на вкладке тоже имеется свое место, где их можно изменить.

Если ваш исполняемый файл был откомпилирован с поддержкой ключей времени выполнения, эти ключи можно указать в командной строке между ключевыми

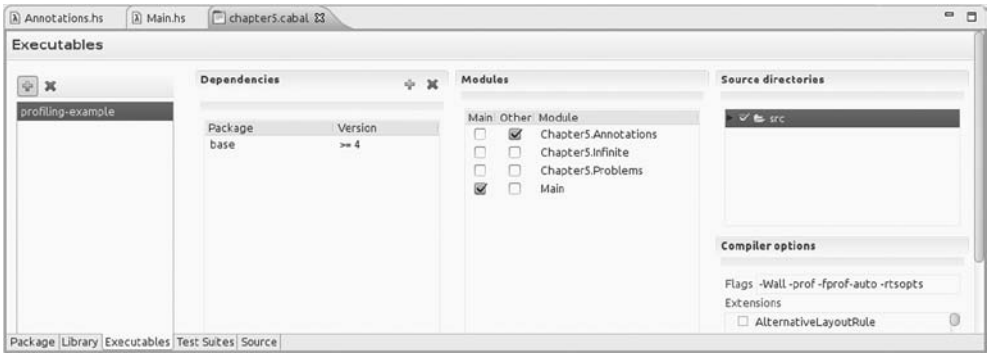


Рис. 5.6. Вкладка Executables в Cabal-редакторе EclipseFP

словами `+RTS` и `-RTS`. Профилирование потребления времени и памяти указывается с помощью ключа `-p`, поэтому для вызова нашего предыдущего примера со сбором информации о времени нужно запустить следующую команду:

```
$ ./dist/build/profiling-example/profiling-example +RTS -p -RTS
Hello!
```

ЗАПУСК ИСПОЛНЯЕМЫХ ФАЙЛОВ В ECLIPSEFP

При работе в среде EclipseFP у вас по сравнению с командной строкой есть более простой способ запуска своих приложений. Нужно просто щелкнуть правой кнопкой мыши на корневом каталоге проекта на вкладке Projects (Проекты), затем перейти к пункту Run As (Запустить как) и щелкнуть на пункте Run Haskell Application (Запустить приложение Haskell).

Зачастую для исполняемого файла нужно указать аргументы. Одним из примеров может послужить профилирование. Чтобы запустить исполняемый файл с аргументами, нужно щелкнуть правой кнопкой мыши на корневом каталоге проекта, затем, как и раньше, выбрать пункт Run As (Запустить как), но в конце выбрать пункт Run Configurations (Запустить настройки). После этого будет выведен список параметров запуска: нужно просто выбрать нужный параметр для вашего проекта из тех, что находятся под заголовком Haskell application (Haskell-приложение), после чего можно будет настроить все его свойства.

На первый взгляд в выполнении ничего не изменилось. Но если присмотреться к папке проекта, можно заметить, что в ней появился новый файл profiling-example.prof. Поскольку время прогона программы практически нулевое, никаких интересных данных, касающихся профиля ее выполнения, сгенерировано не будет. Поэтому давайте лучше выполним профилирование программы, вычисляющей факториал числа 100 000 и выводящей результат на экран. Файл Main.hs нужно привести к следующему виду:

```
module Main where

main :: IO ()
main = putStrLn $ show result

result :: Integer
result = foldr (*) 1 [1 .. 100000]
```

Теперь выполните команду `cabal build` и запустите исполняемый файл в режиме профилирования. В файле profiling-example.prof должно появиться примерно следующее:

```
        total time  =        5.68 secs   (5680 ticks @ 1000 us, 1 processor)
        total alloc = 9,981,322,816 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

result      Main     94.1   99.7
main        Main      5.9    0.3

                                                 individual   inherited
COST CENTRE MODULE                no. entries %time %alloc %time %alloc

MAIN        MAIN                  42      0    0.0    0.0   100.0  100.0
 main       Main                  85      0    1.2    0.0     1.2    0.0
 CAF        Main                  83      0    0.0    0.0    98.8  100.0
  result    Main                  86      1   94.1   99.7    94.1   99.7
  main      Main                  84      1    4.7    0.3     4.7    0.3
 CAF        GHC.IO.Encoding       79      0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal       77      0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Handle.FD      71      0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv 66      0    0.0    0.0     0.0    0.0
 CAF        GHC.Show              63      0    0.0    0.0     0.0    0.0
```

Информация поделена на три части. Первая часть является сводными данными всего потребления программой времени и памяти. Вторая часть показывает центры затрат (в данном случае — функции), являющиеся основными потребителями ресурсов. Как видите, основную часть времени и памяти потребляет функция `result`, а на функцию `main` приходится их меньшая часть. Последней частью является более подробное представление, показывающее информацию о потреблении времени и памяти в дереве. В данном случае показано множество внешних функций, работающих с дескрипторами файлов и осуществляющих кодирование, но в более крупном приложении мы увидим меньше таких функций, поскольку они будут скрыты в более объемном стеке вызовов.

Однако наиболее интересной частью является *профилирование куч* (heap profiling), позволяющее построить график, показывающий потребление памяти программой. Наиболее важные режимы профилирования куч — разбивка их по центрам затрат (указывается ключом времени выполнения `-h`) и разбивка по типу (указывается ключом `-hy`). Доступны и другие варианты группировки потребления: например, по модулям, по замыканиям и т. д., но они применяются редко. Если предыдущую программу запустить с любым из этих ключей, создается новый файл profiling-example.hp, содержащий необработанную информацию. Для преобразования ее в график есть два варианта.

- Воспользоваться утилитой `hp2ps`, поставляемой вместе с Haskell Platform. Запуск команды `hp2ps profiling-example.hp` приведет к созданию PostScript-файла, который затем можно просмотреть.
- Внутри EclipseFP можно непосредственно открыть файл с расширением hp, в результате его содержимое будет показано в виде графика. Кроме того, можно управлять тем, сколько хвостовых элементов должно быть сгруппировано.

На рис. 5.7 показан результат выполнения в командной строке следующей команды и дальнейшей обработки полученного файла в EclipseFP:

```
./dist/build/profiling-example/profiling-example +RTS -h -RTS
```

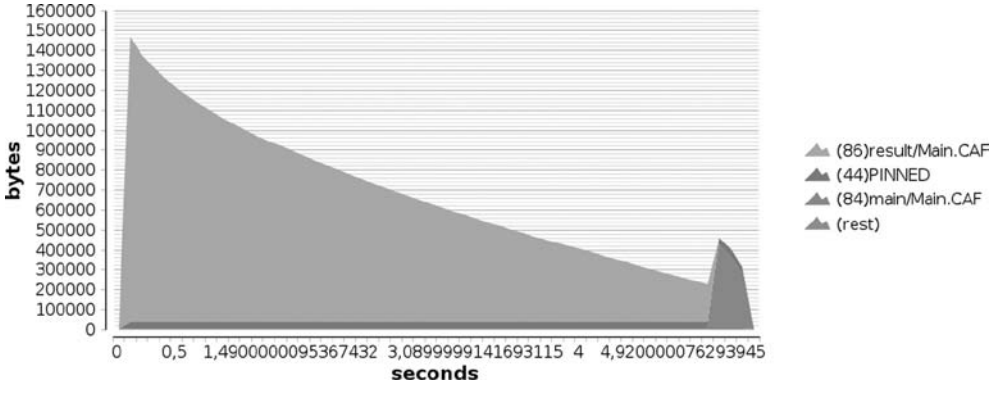


Рис. 5.7. Графическое представление профилирования куч по центрам затрат (при запуске с ключом -h)

Здесь видно, что в первую десятую долю секунды происходит существенное увеличение объема потребляемой памяти, показанное светло-серым цветом, что является отражением массового создания переходников, о которых говорилось в разделе «Проблемы стратегии ленивых вычислений». Затем по мере вычисления переходников объем потребляемой памяти падает. Второй пик, показанный на рисунке более темным оттенком в крайней правой позиции в конце графика, подчеркивает повышение объема потребляемой памяти в тот момент, когда происходит преобразование полученного числа в строку.

Разбивка по типам может проиллюстрировать расходование памяти в системе. Запустите приложение с ключом времени выполнения `-hy` и создайте график. Он должен быть похож на график, показанный на рис. 5.8.

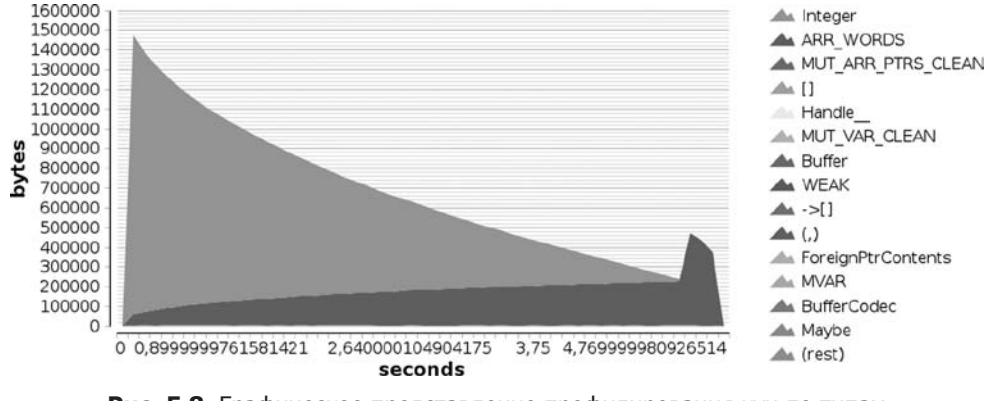


Рис. 5.8. Графическое представление профилирования куч по типам (при запуске с ключом -hy)

Здесь можно видеть, что сначала основной объем памяти используется элементами типа `Integer`, относящимися к переходникам, о которых уже говорилось. По мере продолжения выполнения программы все больше памяти потребляется типом `ARR_WORDS`. Сюда включена и память, расходуемая такими базовыми типами, как вычисленные целые числа (виден рост по мере того, как числа становятся больше) и строки.

Давайте получим профиль других версий, чтобы посмотреть, как получаемой при профилировании информацией подтверждаются наши исходные соображения о проблеме истощения памяти. Перепишите код, заменив функцию `foldr` функцией `foldl'`:

```
result :: Integer
result = foldl' (*) 1 [1 .. 100000]
```

График на рис. 5.9, полученный путем запуска новой версии с профилированием куч по типам, показывает, что теперь промежуточные переходники не создаются, поскольку на каждом этапе происходят принудительные вычисления, поэтому здесь не виден начальный пик значений типа `Integer`, а память потребляется только значениями типа `ARR_WORDS`.

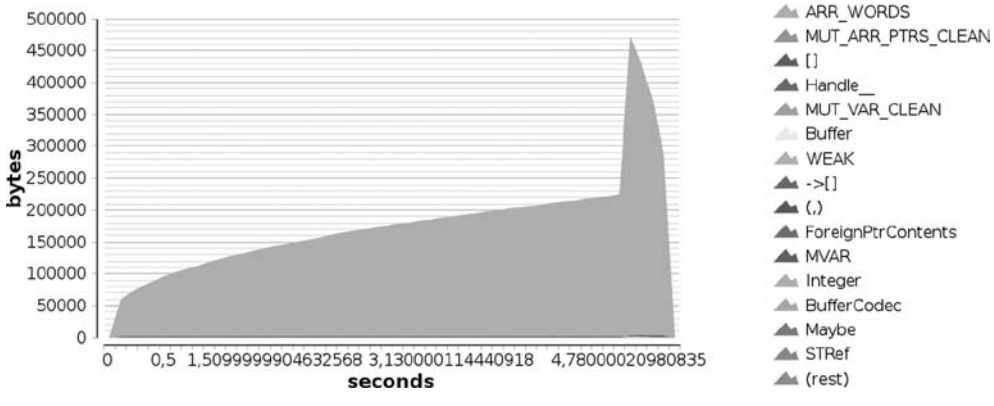


Рис. 5.9. Графическое представление версии foldl' с профилированием куч по типам (при запуске с ключом -hy)

Как видите, профилирование является отличным инструментом выявления проблем, связанных с потреблением как времени, так и памяти, поскольку оно помогает сфокусироваться именно на тех точках, в которых эти ресурсы тратятся понапрасну, не строя догадок об источниках утечек. Я советую профилировать большие программы, чтобы с помощью профайлера и выводимых им данных убедиться, что в ближайшем будущем ваше приложение будет демонстрировать высокую продуктивность.

Строгие аннотации

В этом разделе мы глубже погрузимся во внутреннее устройство GHC. При первом прочтении книги его можно пропустить, но чуть позже вам все же придется к нему вернуться, поскольку из содержащейся здесь информации можно извлечь немалую

пользу. В частности, материал этого раздела поможет вам осмыслить, сколько памяти и времени потребляется проблемным кодом, и даст вам знания, необходимые для улучшения вашего приложения в данных аспектах.

В общем, значение в Haskell, представленное в памяти, можно представлять себе в виде некоего заголовка, обозначающего его тип, и конструктора, используемого для его создания, после чего следуют ссылки на каждое из полей, составляющих это значение. Базовые типы, такие как целые числа или символы, отклоняются от данной схемы и представляются только заголовком и самим значением. В рамках указанной схемы клиент `Individual`, согласно определению предыдущих глав, будет соответствовать представлению, показанному на рис. 5.10.

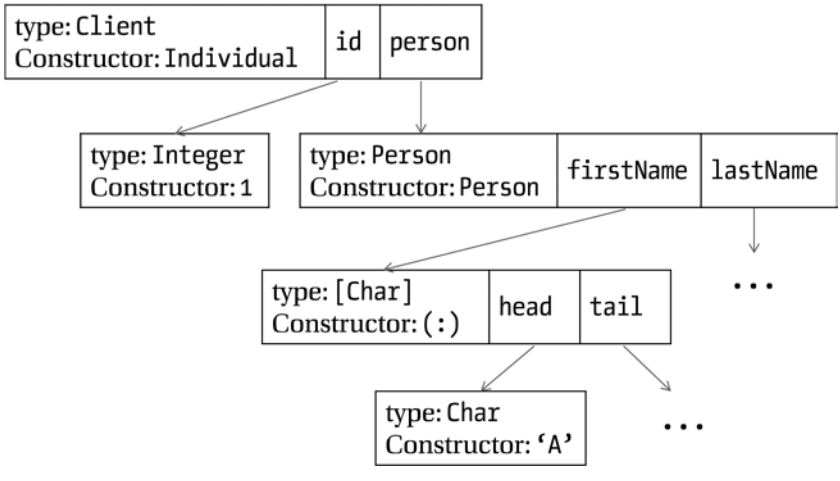


Рис. 5.10. Представление клиента Individual 1 (Person "Andrea" "Blacksmith") в памяти

Следует помнить, что перед полным вычислением Haskell-выражений эти выражения представлены переходниками. На рисунке представление в памяти показано на том этапе, когда выражение полностью вычислено. Если некоторые из его частей все еще подлежат вычислению, ссылки будут указывать на переходники, содержащие код, подлежащий исполнению.

Подобное представление имеет весьма гибкую форму, но при этом оно становится причиной потерь производительности. Прежде всего, вы можете создавать переходники для значений, о которых известно, что они будут использоваться в будущем или же могут понадобиться с целью повышения производительности для этого типа данных. Например, если реализуется новый вид списка, в котором хранится его длина, то в непосредственном хранении длины особого смысла нет; вместо этого используется принцип ленивых вычислений, поскольку на тот момент, когда понадобится запросить эту длину, произойдет длинная цепочка вычислений, от чего пострадает производительность.

В таком случае вам захочется, чтобы длина была *строгим полем* (strict field). Следуя такому же синтаксису, что и у образцов с восклицательным знаком в сопоставлениях, строгие поля объявляются путем написания знака ! перед самим типом

поля. В результате при каждом случае создания значения этого типа выражение, находящееся в строгой позиции, подвергнется принудительному вычислению точно так же, как и при явном указании на это с помощью функции `seq`. Возможная реализация списков с длиной имеет следующий вид:

```
data ListL a = ListL !Integer [a]
```

Для связывания значений с позициями полей в представлении значений в памяти также интенсивно используются ссылки. Это означает, что при каждом доступе к полю в значении необходимо переходить по ссылке. В этом опять проявляется высокая степень гибкости, позволяющая располагать потенциально весьма обширными структурами в памяти, но эта гибкость может стать лишней для доступа к небольшим полям, таким как одиночные целые числа, чье значение может быть непосредственно закодировано в том пространстве памяти, которое занимает ссылка (в объеме указателя для целевой архитектуры). Поле в такой ситуации называется *распакованным* (unpacked).

Распакованные поля являются специальным атрибутом компилятора GHC, их объявление имеет аннотацию `{-# UNPACK #-}`, которая ставится непосредственно перед объявлением поля. Например, с целью повышения эффективности можно принять решение распаковать идентификаторы всех конструкторов типа данных `Client`:

```
data Client = GovOrg     {-# UNPACK #-} !Int String
            | Company    {-# UNPACK #-} !Int String Person String
            | Individual {-# UNPACK #-} !Int Person
            deriving Show
```

Следует заметить, что распакованы могут быть не все поля: все зависит от типа поля. Подходят для распаковки такие базовые типы, как целые числа или символы. Другие типы данных могут использоваться, только если имеют единственный конструктор, а все их поля также являются распакованными, — это означает, что можно распаковать кортеж типов, которые сами по себе не подлежат распаковке, но нельзя распаковать список. Попытка распаковать поле `String` также приведет к выдаче предупреждения, поскольку на самом деле такое поле является списком значений типа `Char`.

Во многих случаях нужно подумать, какая именно реализация структур данных предпочтительнее конкретно для вашего приложения — более ленивая или более строгая. Более ленивая откладывает момент вычислений и позволяет вычислять только то, что конкретно нужно для программы, но при этом приходится идти на компромисс в плане использования больших объемов памяти и большей неопределенности насчет того, когда именно произойдут вычисления.

В некоторых пакетах, например в `containers`, предусмотрена как ленивая, так и строгая реализация структур данных. Например, оба варианта реализации проекций располагаются в разных модулях: `Data.Map.Lazy` и `Data.Map.Strict`. По умолчанию в модуле `Data.Map` используются ленивые версии. В этом случае разница, обозначенная в документации, состоит в том, что в строгой версии ключи и значения перед сохранением в проекции проходят принудительное вычисление, а в ленивой версии вычисления касаются только ключей.

ЕЩЕ ГЛУБЖЕ

Иногда нужно так вычислить выражение, чтобы в нем не осталось никаких переходников. Для таких случаев в Haskell Platform имеется пакет deepseq, который в модуле Control.DeepSeq предоставляет функции deepseq и ($!!). Эти функции подобны функциям seq и ($!) соответственно, но они берут на себя заботу еще и о принудительном вычислении подвыражений, не останавливаясь на уровне конструкторов.

Если нужно, чтобы ваши типы данных поддерживали глубокие вычисления (deep evaluation) с помощью пакета deepseq, их следует сделать экземплярами класса типов NFData. Реализация довольно проста, нужно лишь задать принудительное вычисление всех полей, после чего вернуть (). Например, в Client:

```
import Control.DeepSeq

instance NFData Client where
  rnf (GovOrg i n)                 = i `deepseq` n `deepseq` ()
  rnf (Company i n (Person f l) r) = i `deepseq` n `deepseq` f `deepseq` l
                                     `deepseq` r `deepseq` ()
  rnf (Individual i (Person f l))  = i `deepseq` f `deepseq` l `deepseq` ()
```

Предупреждения, данные по поводу принудительных вычислений с помощью функции seq, применимы и для deepseq, но они еще строже, поскольку в последнем случае объем принудительных вычислений еще больше.

Выводы

В данной главе мы с нескольких сторон рассмотрели вычислительную модель Haskell, в основе которой лежит *лень*:

- Мы узнали, что ленивые вычисления позволяют работать с кажущимися бесконечными или цикличными структурами и создавать образцы элегантного кода.
- Мы получили пространное описание модели ленивых вычислений и познакомились с особой ролью переходников, вычисление которых откладывается до возникновения необходимости в том или ином значении, но которые в то же время расширяют возможности совместного использования готовых результатов вычислений.
- Наряду с этим мы рассмотрели недостатки ленивых вычислений, наиболее важными из которых являются повышенный расход памяти и неопределенность момента вычисления переходника.
- Чтобы обойти эти проблемы, мы научились аннотировать код с помощью функции `seq` или использовать строгие аннотации как при сопоставлениях с образцом, так и в типах данных.
- Мы узнали, что GHC-профайлер является действенным средством определения фактов неоправданного потребления времени и памяти, познакомившись с основами его применения и с интерпретацией выдаваемых им результатов.

Часть II
АНАЛИЗ ДАННЫХ

Глава 6. Ознакомление с клиентами с помощью монад

Глава 7. Другие рекомендуемые монады

Глава 8. Использование нескольких ядер

Ознакомление с клиентами с помощью монад

6

Давайте вспомним, что на нас возложена обязанность по созданию магазина машин времени. Кроме красивого дизайна и интуитивно понятного пользовательского интерфейса хороший интернет-магазин должен подстраиваться под пользовательские нужды и предпочтения: он должен отслеживать действия клиентов и анализировать их поведение. С помощью этой информации могут разрабатываться мероприятия, которые улучшают работу магазина (например, могут объявляться скидки на товары или вестись целевая реклама) и в конечном счете увеличивают объем продаж. Для решения этих задач были разработаны многочисленные алгоритмы *анализ данных* (data mining). В частности, в данной главе основное внимание уделяется алгоритмам *кластеризации* (clustering), предназначенным для выявления групп клиентов со сходными потребностями. С помощью языка Haskell мы осуществим особую реализацию кластеризации, называемую алгоритмом *K-средних* (K-means).

Алгоритм K-средних легче освоить в понятиях множества векторов, которые являются объединением числовых значений, описывающих клиента, товар или покупку, причем это множество меняется с каждой итерацией. В императивном языке эти векторы моделировались бы как набор переменных, обновляемых внутри цикла. Решение, представленное в данной главе, начнется с базовой реализации, в которой будет отслеживаться вся информация. Мы введем идиому *линз* (lenses), которая ориентирована на работу со структурами данных и запросы к ним с использованием краткой формы записи. Далее мы увидим, как можно улучшить код, разбив его на множество базовых комбинаторов, связывающих различные части.

Анализ этих комбинаторов и их отношений с другими типами данных подводит нас к понятию *монады* (monad) — одной из центральных идиом в Haskell-коде. Мы рассмотрим определение монад и законы их функционирования, сравним их с другим распространенным классом типов — *функтором* (functor). В Haskell Platform доступны многие экземпляры класса `Monad`: в данной главе основное внимание уделено тем из них, которые имеют какое-то отношение к отслеживанию *состояния* (state).

Хотя сама идея монад не слишком сложна, они играют в Haskell очень важную роль. Именно поэтому эта и следующая главы посвящены углубленному изучению монад.

Анализ данных

Анализ данных — понятие довольно широкое, включающее в себя множество разных алгоритмов, в которых применяются статистика, машинное обучение и искусственный интеллект и которые направлены на выявление в данных разнообразных повторяющихся образцов. В этой и последующих главах интерес для нас будут представлять две конкретные задачи:

- Прежде всего, нужно будет выявить разные типы клиентов, проявляющих интерес к нашему магазину, на основе их пользовательской информации и истории покупок. К алгоритмам, пытающимся выявить те или иные группы, применяется понятие *кластеризация* (clustering), то есть речь ведется о выявлении *кластеров* (clusters) из тех элементов в множестве, у которых имеются общие свойства. Есть надежда, что на основе данной информации маркетологи смогут правильнее выстроить проведение своих кампаний по продаже товаров.
- Затем потребуется обнаружить покупательские склонности каждого типа клиентов. Это позволит разработать для них систему скидок (более подробно скидки рассматриваются в последней части книги). В этом отношении идея состоит в изучении *ассоциативных правил* (association rules) и их последующем применении с целью сделать правильные заключения.

ПРИМЕЧАНИЕ

Поскольку в магазине продаются машины времени, для наблюдения за тенденциями рынка мы могли бы просто отправиться на них в будущее. Но поскольку это нечестный прием, для улучшения ситуации на рынке приходится пользоваться текущими данными и текущей технологией.

Реализация K-средних

Алгоритм *K-средних* — один из простейших алгоритмов кластеризации некоего множества данных. В данном случае информация — множество точек в n-размерном пространстве, каждая из которых представляет особый наблюдаемый факт. Степень сходства между двумя фактами соответствует степени близости точек. Конкретной задачей алгоритма является деление всего множества точек на k разделов таким образом, чтобы совокупное расстояние между точками в каждом разделе было минимальным.

ПРИМЕЧАНИЕ

Число создаваемых разделов обычно представляется как k и должно быть задано в качестве входного значения алгоритма. Необходимость заранее указывать количество разделов является одним из недостатков алгоритма K-средних. Для наилучшего определения предоставляемого значения в литературе предлагаются различные методы. Неплохую подборку можно найти в статье википедии по адресу http://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set.

Например, на рис. 6.1 показано множество точек в двумерном пространстве. Алгоритм K-средних был применен к множеству точек, где k=3. Результат работы алгоритма (то есть три кластера точек) распознается на рисунке с помощью фигур,

использованных для рисования кластеров. Для кластера 1 были применены окружности, для кластера 2 — треугольники, для кластера 3 — крестики.

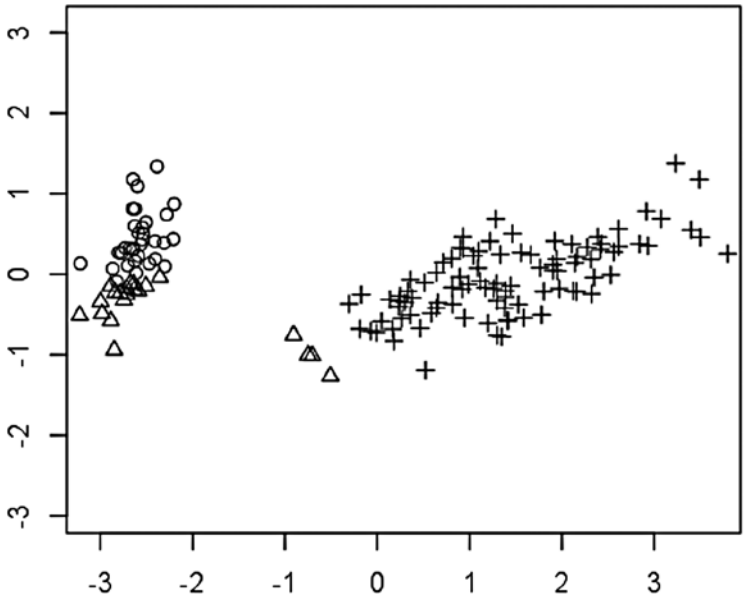


Рис. 6.1. Кластеры, полученные для заданного множества данных

В этом примере показаны основные характеристики алгоритма K-средних: он работает только в отношении векторов, к которым применимы понятия расстояния и приблизительности. В первом подходе мы используем для представления этой информации кортеж, но это ставит вопрос о том, что количество компонентов векторов должно быть постоянным во всех случаях применения алгоритма. Это не самое разумное допущение, поскольку если потребуется учесть какую-нибудь новую информацию (например, о том, что при кластеризации нужно также учитывать возраст), придется вносить изменения во все определения типа. Еще одной возможностью является использование списков, но в этом случае мы проиграем в безопасности, обеспечиваемой кортежами (у списков может быть различное число элементов). Лучше всего определить новый класс типов `Vector`, у которого будут экземпляры всех требуемых типов данных, поддерживаемых операцией `distance`:

```
class Vector v where
  distance :: v -> v -> Double
```

Затем нужно создать реализацию для числовых пар (a,b) и (c,d), используя эвклидову меру расстояния:

$$\sqrt{(c-a)^2+(d-b)^2}.$$

Одной из возможных реализаций для числовых пар с использованием этой меры расстояния может стать следующий код:

```
{-# LANGUAGE FlexibleInstances #-}

instance Vector (Double, Double) where
  distance (a,b) (c,d) = sqrt $ (c-a)*(c-a) + (d-b)*(d-b)
```

ПРИМЕЧАНИЕ

Для типов, имеющих форму в виде имени, за которым следует список различных переменных типа, в Haskell Report разрешаются только объявления экземпляров. Показанное объявление не следует этому правилу, поэтому компилятор предъявляет претензии. Но если включить расширение FlexibleInstances, GHC будет поддерживать такие объявления.

Нужно также указать, как осуществлять преобразование одного элемента наших данных в соответствующий вектор. Здесь снова рациональнее воспользоваться классом типов. Но на этот раз нужно указать два типа, входящие в класс типов: тип преобразуемых элементов и тип векторов, в которые они преобразуются. Это можно сделать с помощью *классов типов с несколькими параметрами* (multi-parameter type classes), для которых используется такой же синтаксис, как и для классов типов с одним параметром, и которые доступны в расширении `MultiParamTypeClasses`. Однако понятие класса типов с двумя и более параметрами отличается от понятия интерфейса в объектно-ориентированном языке, больше напоминая договор между двумя различными типами. Работа с такими классами типов может оказаться непростой: последствия их применения мы будем исследовать на протяжении всей книги. Неудивительно, что имя, которое дается элементам, переводимым в векторы, сопровождается ключевым словом `Vectorizable`. Вот как выглядят определение и простой пример тождественного преобразования между парами типа `Double`:

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Vector v => Vectorizable e v where
  toVector :: e -> v

instance Vectorizable (Double,Double) (Double,Double) where
  toVector = id
```

Описание кластера в алгоритме K-средних с использованием одного вектора для каждого кластера называется *центроидом* (centroid) кластера. Каждый элемент в множестве данных назначается кластеру, чей центроид является самым близким к точке данных. Теперь, когда мы знаем данный факт, у нас появляется начальная идея насчет того, как должен выглядеть тип K-средних:

```
kMeans :: (Vector v, Vectorizable e v) => Int  -- количество центроидов
                                       -> [e]  -- информация
                                       -> [v]  -- центроиды после сближения
```

Алгоритм K-средних весьма прост: на первом этапе генерируется k векторов, которые предназначены для использования в качестве исходных центроидов. Затем каждая точка назначается кластеру ближайшего центроида. Так создается первый раздел точек данных. Согласно алгоритму, после назначения всех точек вычисляются новые центроиды: обновленным центроидом каждого кластера становится средняя из всех точек в этом кластере. Эти новые центроиды составляют входные данные для новых назначений в системе кластер-точка и следующего обновления

центроидов и т. д. В какой-то момент кластеры приобретут стабильность: разделы и кластеры больше меняться не будут. Таким образом, процедура остановится и вернет в качестве окончательного результата центроиды.

Давайте проработаем каждый этап, а в завершение соберем все вместе в один большой алгоритм. Для создания исходных векторов есть несколько вариантов. Одна из возможностей заключается в генерации случайных векторов, еще одна, которая называется методом Форги (Forgy method), — в выборе k векторов в множестве данных. Лучше всего обобщить этот вариант и включить в функцию `kMeans` еще один аргумент для функции, генерирующей исходные значения. В таком случае тип функции приобретет следующий вид:

```
kMeans :: Vector v, Vectorizable e v => (Int -> [e] -> [v]) -> [e] -> [v]
```

На этапе назначения кластера нужно получить текущие центроиды и элементы множества и решить, какому центроиду будет соответствовать каждый элемент. Это делается на основе понятия приблизительности. Поскольку у вас имеется проекция ключа (кластера) на значения (точки), для хранения назначений стоит воспользоваться проекцией `Map`. Это означает, что нам нужно в класс типов `Vector` включить еще одно ограничение `Ord v`, поскольку ключи `Map` должны отвечать этому требованию.

Этот этап назначения кластера должен быть поделен на два шага. Первый — создание `Map` со всеми ключами, назначенными пустым спискам. Здесь неявно предполагается, что в любой точке алгоритма не может быть двух одинаковых ключей, но в более полной реализации об этом придется позаботиться. Второй — поэлементный обход и нахождении центроида, наиболее близкого к текущему элементу с последующим добавлением элемента к списку выбранного центроида. Сначала нужно с помощью функции `toVector` преобразовать элемент в вектор. В следующем фрагменте кода для создания исходных проекций и обновления каждого элемента в множестве данных используются свертки:

```
import Data.List
import qualified Data.Map as M

clusterAssignmentPhase :: (Vector v, Vectorizable e v) => [v] -> [e] -> M.Map v [e]
clusterAssignmentPhase centroids points =
  let initialMap = M.fromList $ zip centroids (repeat [])
   in foldr (\p m -> let chosenCentroid = minimumBy (\x y ->
                                          compare (distance x $ toVector p)
                                                  (distance y $ toVector p))
                                            centroids
                     in M.adjust (p:) chosenCentroid m)
            initialMap points
```

И наконец, для каждого кластера нужно вычислить новый центроид. Для этого требуется спроецировать элементы в кластере на вектор. Теперь становится понятно, что в типовом классе для `Vector` была пропущена еще одна функция — функция вычисления центроида из множества векторов. Давайте дополним класс типов и реализуем его для пар типа `Double`:

```
class (Default v, Ord v) => Vector v where
  distance :: v -> v -> Double
  centroid :: [v] -> v
```

```
instance Vector (Double, Double) where
  distance (a,b) (c,d) = sqrt $ (c-a)*(c-a) + (d-b)*(d-b)
  centroid lst = let (u,v) = foldr (\(a,b) (c,d) -> (a+c,b+d)) (0.0,0.0) lst
                     n = fromIntegral $ length lst
                 in (u / n, v / n)
```

С помощью этой функции вычисление новых центроидов становится тривиальным. В код закладывается следующий замысел: для каждого кластера (то есть придется воспользоваться функцией `fmap`) нужно преобразовать список связанных элементов в векторы (то есть внутри функции для применения к каждому элементу нужно иметь код `map toVector`), а затем получить центроид этого множества. И наконец, мы преобразуем проекцию в список элементов (*старый*, *новый*).

```
newCentroidPhase :: (Vector v, Vectorizable e v) => M.Map v [e] -> [(v,v)]
newCentroidPhase = M.toList . fmap (centroid . map toVector)
```

Хотя этот алгоритм известен как сходящийся, итерации можно принудительно остановить, когда количество изменений между очередными вычисленными центроидами станет меньше определенного порога. По этой причине в код включена функция, вычисляющая общее количество изменений и сравнивающая его с предопределенным значением:

```
shouldStop :: (Vector v) => [(v,v)] -> Double -> Bool
shouldStop centroids threshold = foldr (\(x,y) s -> s + distance x y) 0.0 centroids
< threshold
```

Располагая всеми этими частями, можно, наконец, завершить создание исходного кода для K-средних:

```
kMeans :: (Vector v, Vectorizable e v) => (Int -> [e] -> [v])  -- функция
инициализации
                                       -> Int       -- количество центроидов
                                       -> [e]       -- информация
                                       -> Double    -- порог
                                       -> [v]       -- готовые центроиды
kMeans i k points = kMeans' (i k points) points

kMeans' :: (Vector v, Vectorizable e v) => [v] -> [e] -> Double -> [v]
kMeans' centroids points threshold =
  let assignments     = clusterAssignmentPhase centroids points
      oldNewCentroids = newCentroidPhase assignments
      newCentroids    = map snd oldNewCentroids
   in if shouldStop oldNewCentroids threshold
      then newCentroids
      else kMeans' newCentroids points threshold
```

Чтобы протестировать `kMeans`, возьмем небольшую функцию, генерирующую k векторов (i, i), где i меняется от 1 до k:

```
initializeSimple :: Int -> [e] -> [(Double,Double)]
initializeSimple 0 _ = []
initializeSimple n v = (fromIntegral n, fromIntegral n) : initializeSimple (n-1) v
```

Используя этот фрагмент кода, можно запустить в интерпретаторе наш первый пример реализации алгоритма K-средних:

```
*Chapter6.KMeans> kMeans initializeSimple 2 ([(1,1),(1,2),(4,4),(4,5)]::
[(Double,Double)]) 0.001
[(1.0,1.5),(4.0,4.5)]
```

Чтобы проверить, удалось ли нам разобраться в том, как все части исходной реализации K-средних сочетаются друг с другом, давайте выполним упражнение 6.1, код в котором призван создать статистику запуска алгоритма.

УПРАЖНЕНИЕ 6.1. ПОДСЧЕТ КОЛИЧЕСТВА ЭТАПОВ

При профилировании итерационных алгоритмов очень часто смотрят на количество рекурсивных этапов, выполняемых до достижения порога. Усовершенствуйте предыдущую реализацию алгоритма K-средних, чтобы предоставить это значение в качестве дополнительных данных, выводимых функцией kMeans.

Линзы

Алгоритм K-средних обычно выражается более императивным способом, в котором центроиды и погрешность — это переменные, обновляемые на каждой итерации, пока порог остается больше погрешности. Одними из операций, в которых разница между более привычными языками и Haskell особенно очевидна, являются операции запроса структур данных и доступа к структурам данных. Эти операции должны выполняться с использованием либо сопоставления с образцом, либо записей, либо синтаксиса обновления записей, либо через вспомогательные функции, создаваемые автоматически компилятором.

Линзы (lenses) позволяют запрашивать и обновлять структуры данных, используя синтаксис, более приближенный к обычной точечной нотации, которую можно встретить в других языках. Эта нотация полностью определена в библиотеке и не является частью языка, что дает возможность еще раз оценить эффективность языка Haskell, позволяющего внутри самого языка выражать конструкции доступа к данным и обновления.

Концепция линз объединяет в структуре данных получателя (getter) и установщика (setter) конкретного поля. В этом отношении линза похожа на объект в Java или на свойство в C#. Кроме того, в библиотеку линз включен ряд комбинаторов, позволяющих составлять сочетания из нескольких линз (например, для доступа к более глубоким частям структуры) и предлагающих более понятный синтаксис (например, с использованием оператора `+=` для обновления числового поля путем прибавления к нему какого-нибудь значения).

В предыдущем абзаце речь шла не о какой-то конкретной библиотеке линз, а о библиотеке вообще. Этим подчеркивается тот факт, что в отличие от других частей Haskell Platform у Haskell-сообщества для решения данной задачи нет какой-то более предпочтительной или какой-то определенной библиотеки. Существует ряд известных пакетов линз, таких как `fclabels`, `data-accessor` и `data-lens`, однако в данной книге основное внимание уделено библиотеке `lens` Эдварда Кметта (Edward A. Kmett). В любом случае, хотя они различаются и в теоретических основах (в том, как линзы представлены внутренне и из чего они состоят), и в самой реализации, во внешнем интерфейсе больших различий эти пакеты не имеют.

После такого введения давайте сконцентрируемся на использовании линз в коде. Вспомним определения `Client` и `Person` из главы 3, поскольку нам придется с ними работать:

```
data Client i = GovOrg     i String
              | Company    i String Person String
              | Individual i Person

data Person = Person String String
```

Ранее в определениях использовался синтаксис записей, но здесь представлены необработанные варианты, поскольку после создания для них линз доступ посредством записей теряет смысл.

При использовании библиотеки `lens` генерация линз для каждого поля может вестись двумя способами. В обоих случаях кроме добавления модуля `lens` в качестве зависимости к вашему проекту потребуется импортировать модуль `Control.Lens`. Первый способ предусматривает написание линз вручную. Несмотря на предполагаемую в такой постановке вопроса сложность, на самом деле все просто: вам нужно лишь написать получателя и установщика линз, что можно сделать через сопоставление с образцом. Давайте сделаем это для типа данных `Person`:

```
firstName :: Simple Lens Person String
firstName = lens (\(Person f _) -> f)
                 (\(Person _ l) newF -> Person newF l)

lastName :: Simple Lens Person String
lastName = lens (\(Person _ l) -> l)
                (\(Person f _) newL -> Person f newL)
```

Это примеры простых линз, в которых тип структуры при изменении значения не меняется. Но бывает так, что нужно изменить тип. Предположим, к примеру, что имеется тип `Client Int` и нужно обновить идентификатор значением `Double`, после чего у клиента будет тип `Client Double`. То есть нам нужны полноценные линзы, которые получают в качестве дополнительных переменных типа различные типы внутренних переменных. Например:

```
{-# LANGUAGE LambdaCase #-}

identifier :: Lens (Client i) (Client j) i j
identifier = lens (\case (GovOrg i _)      -> i
                         (Company i _ _ _) -> i
                         (Individual i _)  -> i)
                  (\client newId -> case client of
                      GovOrg _ n      -> GovOrg newId n
                      Company _ n p r -> Company newId n p r
                      Individual _ p  -> Individual newId p)
```

В определении `data` линзы не должны отражать только поля. Линзу можно сгенерировать при наличии четко определенного способа получения и возвращения значений. Предположим, к примеру, что в именах для `Person` не содержится пробелов. Тогда можно создать линзу для полного имени: ее получение будет объединять имя и фамилию с пробелом между ними, а установка значения разобьет имя на две части с присвоением соответствующей части каждому полю.

```
fullName :: Simple Lens Person String
fullName = lens (\(Person f l) -> f ++ " " ++ l)
                (\_ newFullName -> case words newFullName of
                                     f:l:_ -> Person f l
                                     _     -> error "Incorrect name")
```

Но чаще всего требуется сгенерировать базовые линзы, которые просто получают или устанавливают значения поля в структуре, что является задачей с большим объемом шаблонного кода. Философия Haskell не допускает написания такого большого объема повторяющегося кода, поэтому создатели библиотеки включили в нее возможность автоматического создания линз. Чтобы воспользоваться этой возможностью, нужно написать объявление `data` с использованием синтаксиса записи, но с полями, перед именем которых ставится знак подчеркивания:

```
data Client i = GovOrg     { _identifier :: i, _name :: String }
              | Company    { _identifier :: i, _name :: String
                           , _person :: Person, _duty :: String }
              | Individual { _identifier :: i, _person :: Person }
              deriving Show

data Person = Person { _firstName :: String, _lastName :: String }
              deriving Show
```

Теперь нужно потребовать от библиотеки, чтобы она создала для вас линзы. Прежде всего, нужно включить расширение `TemplateHaskell`, позволяющее автоматически генерировать код. Затем в отношении каждого типа данных нужно вызвать функцию `makeLenses`. Обратите внимание на использование перед именем типа двух одинарных кавычек.

```
{-# LANGUAGE TemplateHaskell #-}

makeLenses ''Client
makeLenses ''Person
```

Вот и всё! Нужный нам код получен в фоновом режиме.

TEMPLATE HASKELL

Template Haskell — это название механизма метапрограммирования, включенного в GHC. Метапрограммированием называют технологию, позволяющую модифицировать код, который генерирует компилятор, причем новый код обычно генерируется в автоматическом режиме. В языке Lisp метапрограммирование встречается в виде макроса, выполняемого в ходе компиляции.

С метапрограммированием мы уже познакомились на примере механизма извлечения (deriving) встроенных классов типов. Template Haskell предлагает для компилятора GHC расширяемый интерфейс и позволяет авторам библиотек предоставлять собственные средства модификации кода, как это сделано в библиотеке lens. В Haskell имеется множество других библиотек, использующих Template Haskell: например, библиотека derive поддерживает автоматическое извлечение многих других классов типов, таких как NFData.

Template Haskell не является частью Haskell 2010 Report, поэтому нет ничего удивительного в том, что ваш код не будет легко переноситься на другие Haskell-компиляторы в неизменном виде. Однако GHC предоставляет ключ командной строки -ddump-splices. Полученный с его помощью код, сгенерированный Template Haskell, можно скопировать обратно в программу, что позволяет достичь полной совместимости.

Теперь, когда мы узнали, как создаются линзы, настало время воспользоваться ими. Одной из основных операций, осуществляемых с помощью линз, является запрос значения. Для этого применяются функция `view` или оператор `(^.)`:

```
*Chapter6.Lenses> let p = Person "John" "Smith" in (view firstName p, p^.lastName)
("John","Smith")
```

У линз есть очень хорошее свойство: для создания новых линз из них можно составлять композиции с помощью оператора `(.)`, что очень похоже на использование этого оператора для составления композиций из функций. В частности, код доступа к полям напоминает синтаксис языка C:

```
*Chapter6.Lenses> let client = Individual 3 (Person "John" "Smith")
*Chapter6.Lenses> view (person . lastName) client
"Smith"
*Chapter6.Lenses> client^.person.fullName
"John Smith"
```

Обновление осуществляется с помощью функции `set` или оператора `(.~)`. Как вы увидите, используемая здесь семантика совместима с чистыми языками, подобными Haskell: возвращается новая копия структуры данных с обновленным полем, то есть элемент на месте не обновляется.

```
*Chapter6.Lenses> set identifier 4 client
Individual {_identifier = 4, _person = Person {_firstName = "John", _lastName =
"Smith"}}
*Chapter6.Lenses> person.lastName .~ "Kox" $ client
Individual {_identifier = 3, _person = Person {_firstName = "John", _lastName = "Kox"}}
```

Хотя с точки зрения составления линзовых операций наличие обновляемой структуры в качестве последнего аргумента функции считается весьма удобным обстоятельством, иногда оно может усложнить чтение кода. Поэтому в библиотеку `lens` включен оператор `&`, который меняет очередность следования аргументов, позволяя использовать значение в начале выражения:

```
*Chapter6.Lenses> client & person.fullName .~ "Marianne Kox"
Individual {_identifier = 3, _person = Person {_firstName = "Marianne", _lastName =
"Kox"}}
```

Еще одним преимуществом библиотеки `lens` является включение в нее большого количества комбинаторов, которые напоминают типичные составные операторы обновления языка C или Java (такие как `+=` или `*=`). При указании имени нужно всегда следовать одной и той же схеме: имя оператора (он должен создавать сочетание текущего значения с новым значением), за которым следует знак тильды:

```
*Chapter6.Lenses> client & identifier +~ 2
Individual {_identifier = 5, _person = Person {_firstName = "John", _lastName =
"Smith"}}
```

Все эти операторы являются специальными экземплярами функции более общего назначения `over`, или ее инфиксной формы `(%~)`, которая получает функцию, применяемую к полю, указанному линзой:

```
*Chapter6.Lenses> client & over identifier (+2)
Individual {_identifier = 5, _person = Person {_firstName = "John", _lastName =
"Smith"}}
```

```
*Chapter6.Lenses> import Data.Char  -- для того, чтобы toUpper попала в область
видимости
*Chapter6.Lenses> client & person.fullName %~ (map toUpper)
Individual {_identifier = 3, _person = Person {_firstName = "JOHN", _lastName =
"SMITH"}}
```

В библиотеку включены линзы для множества различных типов. Например, имеется семейство линз от `_1` до `_9`, которые внедряются в каждый компонент кортежа при условии, что он имеет достаточную длину:

```
*Chapter6.Lenses> ("a","b") & set _1 "c"
("c","b")
*Chapter6.Lenses> ("a","b") & set _3 "c"
<interactive>:
    No instance for (Field3 ([Char], [Char]) b0 a0 [Char])
      arising from a use of `_3'
    Possible fix:
      add an instance declaration for
      (Field3 ([Char], [Char]) b0 a0 [Char])
    In the first argument of `set', namely `_3'
    In the second argument of `(&)', namely `set _3 "c"'
    In the expression: ("a", "b") & set _3 "c"
```

Иногда значение в указанном поле может быть недоступным. Это бывает, например, в линзах[1] для получения головы и хвоста списка. В таком случае у вас есть два варианта: использовать либо оператор `(^?)`, возвращающий значение, заключенное в оболочку `Maybe`, либо оператор `(^?!)`, который значение ни во что не заключает, но сигнализирует об ошибке, если элемент недоступен. Обновление выполняется с помощью тех же операторов, что и раньше:

```
*Chapter6.Lenses> "abc"^?_head
Just 'a'
*Chapter6.Lenses> "abc"^?!_tail
"bc"
*Chapter6.Lenses> "abc" & (_head .~ 'd')
"dbc"
*Chapter6.Lenses> "abc" & (_tail %~ map toUpper)
"aBC"
```

В Haskell наиболее распространен способ разбиения списка на голову и остальную часть списка (хвост), но можно также разбить его с конца. Библиотека `lens` предоставляет линзы для доступа как к последнему элементу в списке, так и к списку без последнего элемента, эти линзы называются `_last` и `_init`:

```
*Chapter6.Lenses> "abc"^?_init
Just "ab"
*Chapter6.Lenses> "abc" & (_last %~ toUpper)
"abC"
```

Ранее уже упоминалось, что в дистрибутив библиотеки включены также многие другие линзы: для списков, проекций, множеств. Если вы примете решение об использовании библиотеки `lens`, не забудьте проверить работу и этих линз.

---

[1] Технически такие линзы относятся не к Lens, а к Traversal.

В качестве последнего штриха нашего обзора линз мы рассмотрим линзу `traversed`. Эта линза позволяет проникать в список (или же, в более общем смысле, в любой экземпляр класса типов `Traversable`, куда также входят деревья и проекции) и обновлять каждый элемент, используя дополнительные линзы. Например, если имеется массив людей, можно сделать все буквы имен прописными, воспользовавшись следующей линзой:

```
*Chapter6.Lenses> let people = [Person "Jack" "Smith", Person "Marianne" "Branson"]
*Chapter6.Lenses> people & traversed.firstName %~ map toUpper
[ Person {_firstName = "JACK", _lastName = "Smith"}
, Person {_firstName = "MARIANNE", _lastName = "Branson"} ]
```

В упражнении 6.2 информация о библиотеке `lens` применяется к машинам времени. Имеет смысл выполнить данное упражнение, чтобы лучше понять, что такое линзы.

УПРАЖНЕНИЕ 6.2. ЛИНЗЫ МАШИН ВРЕМЕНИ

Сгенерируйте линзы для типа TimeMachine, созданного при изучении предыдущих глав, включив всю информацию, о которой упоминалось ранее, а также цену. Используя представленные в этом разделе операторы, создайте функцию, которая, располагая списком машин времени, повышает их цены на заданный процент.

Давайте воспользуемся линзами для реализации K-средних. Вместо различных аргументов для каждой части информации, которую нужно сохранить, создадим тип данных, хранящий все эти части. Линзы будут получены автоматически с помощью Template Haskell:

```
data KMeansState e v = KMeansState { _centroids :: [v], _points :: [e]
                                   , _err :: Double, _threshold :: Double
                                   , _steps :: Int }

makeLenses ''KMeansState
```

ПРИМЕЧАНИЕ

Перед их использованием в другом коде линзы нужно получить посредством Template Haskell. Поэтому нужно внимательно отнестись к написанию кода, который предшествует определению кода новых K-средних.

Как видите, погрешность сохраняется в поле, кроме того, соблюдается требование предыдущего упражнения по сохранению количества этапов. Новый алгоритм `kMeans'` рассматривается как серии изменений в этом состоянии: сначала создаются назначения, а затем обновляются центроиды, погрешность и количество этапов. Эти три последних шага реализованы с помощью линз. И наконец, алгоритм должен предполагать проверку условий остановки, сравнивая погрешность с порогом, что также отражено отдельным полем в типе данных состояния. Кроме того, должна быть изменена функция `kMeans`, чтобы возвращать только центроиды из полноценного состояния.

```
initializeState :: (Int -> [e] -> [v]) -> Int -> [e] -> Double -> KMeansState e v
initializeState i n pts t = KMeansState (i n pts) pts (1.0/0.0) t 0

kMeans :: (Vector v, Vectorizable e v) => (Int -> [e] -> [v]) -> Int -> [e] ->
Double -> [v]
```

```
kMeans i n pts t = view centroids $ kMeans' (initializeState i n pts t)

kMeans' :: (Vector v, Vectorizable e v) => KMeansState e v -> KMeansState e v
kMeans' state =
  let assignments = clusterAssignments state
      state1 = state  & centroids.traversed
                      %~ (\c -> centroid $ fmap toVector $ M.findWithDefault [] c
assignments)
      state2 = state1 & err .~ sum (zipWith distance (state^.centroids) (state1^.
centroids))
      state3 = state2 & steps            +~ 1
   in if state3^.err < state3^.threshold then state3 else kMeans' state3
```

Заметьте, что способ вычисления погрешности также изменился: вместо возвращения при обновлении центроидов пар (*старый центроид*, *новый центроид*) берутся центроиды в текущем и предыдущем состояниях, после чего выполняется агрегирование их расстояний с помощью функций `sum` и `zipWith`. В упражнении 6.3 от вас потребуется завершить эту реализацию с использованием линз, написав код для этапа назначения кластера.

УПРАЖНЕНИЕ 6.3. ЛИНЗЫ ДЛЯ K-СРЕДНИХ

Реализация алгоритма с использованием линз еще не завершена: не хватает функции clusterAssignments. Начиная с версии, показанной в предыдущем разделе, напишите нужные функции (которые теперь работают с полноценным состоянием), используя линзы.

Исследование монад

Одним из основных в философии Haskell является положение о многократном использовании кода. Поэтому при изучении языка и его библиотек полезно время от времени возвращаться назад и просматривать ранее написанный код, находя общие образцы кода, которые можно было бы абстрагировать. В данном разделе нам предстоит подумать над абстракциями, относящимися к значениям `Maybe` и обработке состояния. В обоих случаях используется структура одного и того же типа, что ведет к понятию *монады* (monad), которое является ключевым в данном разделе.

Отслеживание неполных данных

В предыдущем разделе мы сделали явное допущение о том, что у нас есть вся необходимая информация для удачного решения задачи с помощью алгоритма K-средних, следовательно, единственным необходимым преобразованием является конвертация информации в векторы. Однако с множеством данных реального мира такое случается довольно редко. Обычно для сбора всей информации, ее объединения и, возможно, устранения ряда несоответствий нужен этап предварительной обработки.

Скорее всего, необработанная информация будет поступать из какой-либо системы управления базами данных. Чуть позже мы увидим, как использовать постоянную библиотеку для обмена информацией с такими системами, а пока можно обойтись лишь некоторыми аспектами. В частности, в столбцах таблицы могут встречаться значения `NULL`, соответствующие отсутствию информации (или то, что она не была

записана). В качестве средства представления нулевого столбца типа τ в постоянной библиотеке выбран тип Maybe τ. Предположим, к примеру, что нужно вычислить среднее значение всех покупок, приобретенных конкретным клиентом. Возможный способ решения данной задачи может выглядеть следующим образом:

```
meanPurchase :: Integer -- идентификатор клиента
             -> Double  -- средний показатель покупок
meanPurchase clientId = let p = purchasesByClientId clientId
                        in foldr (+) 0.0 $ catMaybes $ map purchaseValue p

purchaseValue :: Integer -> Maybe Double
purchaseValue purchaseId =
  case numberItemsByPurchaseId purchaseId of
    Nothing -> Nothing
    Just n  -> case productIdByPurchaseId purchaseId of
               Nothing        -> Nothing
               Just productId -> case priceByProductId productId of
                                 Nothing    -> Nothing
                                 Just price -> Just $ (fromInteger n) * price
```

В этом примере используется функция catMaybes из модуля Data.Maybe. Эта функция отфильтровывает каждый имеющийся в списке элемент Nothing, что очень удобно при работе со списками значений Maybe.

ПРИМЕЧАНИЕ

В предыдущем и в следующих примерах этого раздела код обращения к базе данных вынесен за скобки как не имеющий отношения к рассматриваемому вопросу. Если есть желание испытать код в действии, достаточно добавить простое возвращаемое значение. Например, purchasesByClientId может вернуть [1,2,3], а numberItemsByPurchaseId, productIdByPurchaseId и priceByProductId могут вернуть лишь постоянное значение.

Вполне очевидно, что данный код нельзя назвать простым и легко поддерживаемым. Для Nothing или Just придется написать целый ворох проверок. Кроме того, в случае необходимости добавления среди всех прочих какого-либо нового запроса придется заново обозначить отступы во всем коде. Следовательно, нужно разработать комбинатор, позволяющий создавать более качественный и легко поддерживаемый код.

Основная идея заключается в следующем: комбинатор должен получить значение, заключенное в Maybe. Это значение будет принято к рассмотрению только в том случае, если оно конструируется с помощью Just, в противном случае возвращается Nothing. Если решите продолжить, то к элементу, заключенному в оболочку Just, нужно применить функцию, которая сама возвращает еще одно значение Maybe. И в конце концов в любом случае все нужно завершить значением Maybe, получаемым из приложения. Знание конкретных требований облегчает создание комбинатора:

```
thenDo :: Maybe a -> (a -> Maybe b) -> Maybe b
thenDo Nothing  _ = Nothing
thenDo (Just x) f = f x
```

Давайте теперь перепишем исходную функцию purchaseValue, используя этот комбинатор, который был применен инфиксным образом, чтобы сделать код понятнее. Новое решение имеет следующий вид:

```
purchaseValue :: Integer -> Maybe Double
purchaseValue purchaseId =
  numberItemsByPurchaseId purchaseId `thenDo` (\n ->
  productIdByPurchaseId purchaseId   `thenDo` (\productId ->
  priceByProductId productId         `thenDo` (\price ->
  Just $ fromInteger n * price       )))
```

Несомненно, в новом коде легче разобраться и его проще поддерживать. Кроме того, в нем низкоуровневая операция извлечения значений `Maybe` из оболочки скрыта в комбинаторе, что расширяет возможности его многократного использования.

ПРИМЕЧАНИЕ

Проанализируйте предыдущую функцию. Стиль написания аргумента функции в строке за пределами ее тела называется подвешиванием лямбд (hanging lambdas). Он часто применяется при использовании таких функциональных комбинаторов, как наш thenDo.

Возникает вполне резонный вопрос: зачем нужно создавать новый комбинатор `thenDo`? Сначала казалось, что назначение этой функции очень похоже на назначение функции `fmap` в функторе. Давайте напишем ее тип, специально приспособленный для `Maybe`:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

Проблема здесь в том, что результат в виде `Nothing` или `Just` не может зависеть от применяемой функции, он полностью определяется входным значением. При попытке использования функции с выходным значением `Maybe b` придется иметь дело со специализированным типом:

```
fmap :: (a -> Maybe b) -> Maybe a -> Maybe (Maybe b)
```

А значение, дважды заключенное в оболочку `Maybe`, противоречит требуемому результату. Однако возможен и прямо противоположный случай: функцию `fmap` можно выразить в понятиях нашего комбинатора `thenDo`:

```
fmap f x = x `thenDo` (\y-> Just $ f y)
```

Таким образом, только что определенный комбинатор несомненно более эффективен, чем `fmap`: он может использоваться для написания версии `fmap` для значений `Maybe`, поскольку `fmap` не может выражать поведение `thenDo`. Дополнительное упражнение 6.4 требует от вас провести проверку этого нового определения `fmap` на работоспособность.

УПРАЖНЕНИЕ 6.4. ПРОВЕРКА РАБОТОСПОСОБНОСТИ НАШЕЙ ВЕРСИИ FMAP

Используя эквациональное рассуждение, представленное в главе 3, удостоверьтесь в работоспособности данной реализации fmap. Проверьте, что функция fmap, определенная в данном разделе, работает точно так же, как экземпляр Functor для значений Maybe, проецирующий Just x на Just (f x) и Nothing на Nothing. Подсказка: разбейте проверку на случаи, зависящие от конструктора для значения Maybe.

Комбинаторы для состояния

Основываясь на успешном создании комбинатора для тех функций цепочки, которые могут завершиться неудачно при возвращении `Nothing`, можно подумать

и о том, чтобы сделать то же самое для реорганизации нашего кода реализации алгоритма K-средних. В частности, интересно было бы внести улучшение, позволяющее скрыть код управления состояниями, который имеется в последней версии.

Прежде всего, давайте подумаем о том, как представить функцию, которая манипулирует состоянием. Каждая из таких функций станет реальным строительным блоком, который впоследствии будет встроен в цепочку с помощью разрабатываемого комбинатора. Состояние, предшествующее выполнению функции, может рассматриваться как ее дополнительный аргумент. Тогда если в общем виде у нас есть функция типа `a -> b`, то функция, которая также получает состояние типа `s`, должна иметь тип `a -> s -> b`. Эта функция в конце ее выполнения должна быть также наделена состоянием, которое впоследствии может быть передано другой функции, ожидающей состояние. При условии, что функция возвращает значение типа `b`, мы можем добавить в пару к нему значение типа `s`. В окончательном виде функции, манипулирующие состоянием, имеют тип `a -> s -> (b,s)`. Используя предыдущие знания о работе со значениями `Maybe`, можно ожидать, что у нашего комбинатора будет тип, похожий на следующий:

```
thenDo :: (s -> (a,s)) -> (a -> s -> (b,s)) -> (s -> (b,s))
```

Поначалу это смотрится как-то неуклюже, поскольку более естественным представляется вариант `(a,s) -> (a -> s -> (b,s)) -> (b,s)`, то есть вариант со вставкой состояния непосредственно из начального вычисления во вторую функцию. Тем не менее больше подходит версия, которая генерирует функцию, все еще нуждающуюся в исходном состоянии, поскольку она позволяет создавать объединение с учетом состояния вычислений, для которого исходного состояния еще нет. Кроме того, это сделает более понятным образец кода, который должен определиться на основе этих примеров. Поскольку в коде будет часто использоваться сочетание `s -> (a,s)`, стоит ввести для него синоним типа:

```
type State s a = s -> (a, s)
```

Теперь параллель со случаем применения `Maybe` в типе комбинатора становится более очевидной:

```
thenDo :: State s a -> (a -> State s b) -> State s b
```

Реализация комбинатора очень проста: нужно просто применить состояние к первой функции, чтобы получить результат и новое состояние, которое передается второй функции. Чтобы проще было понимать Haskell-код, в него включена также сигнатура типа без синонимов:

```
thenDo :: State s a -> (a -> State s b) -> State s b
-- thenDo :: (s -> (a,s)) -> (a -> s -> (b,s)) -> s -> (b,s)
thenDo f g s = let (resultOfF, stateAfterF) = f s
               in g resultOfF stateAfterF
```

Интересно то, что благодаря композиции и раскаррированию (uncurrying) `thenDo` можно описать достаточно компактно:

```
thenDo f g = uncurry g . f
```

В версии, разработкой которой мы теперь займемся, будет записана только информация о центроидах, пределе погрешности и количестве этапов. Таким образом,

мы получим два отдельных набора информации: само состояние и векторы, для которых будет выполняться алгоритм. Следовательно, можно разделить информацию, передаваемую в программном потоке, и состояние. Кроме того, код будет без линзовых комбинаторов, что позволит сфокусироваться только на комбинаторах, относящихся к состоянию. В результате новое определение `KMeansState` приобретет следующий вид:

```
data KMeansState v = KMeansState { centroids :: [v]
                                 , threshold :: Double
                                 , steps :: Int }
```

Что касается значений `Maybe`, давайте перепишем код, используя комбинатор `thenDo`. Основная функция `kMeans` будет просто вызывать функцию `kMeans'`, которая задействует комбинатор с исходным состоянием. Результатом вычисления станет пара окончательных центроидов и последнее состояние: вам нужно вернуть только первый элемент, используя для этого `fst`.

```
newCentroids :: (Vector v, Vectorizable e v) => M.Map v [e] -> [v]
newCentroids = M.elems . fmap (centroid . map toVector)

clusterAssignments :: (Vector v, Vectorizable e v) => [v] -> [e] -> M.Map v [e]
clusterAssignments centrs points =
  let initialMap = M.fromList $ zip centrs (repeat [])
    in foldr (\p m -> let chosenCentroid = minimumBy (\x y -> compare (distance x $
toVector p)
(distance y $ toVector p))
centrs
in M.adjust (p:) chosenCentroid m)
initialMap points

kMeans' :: (Vector v, Vectorizable e v) => [e] -> State (KMeansState v) [v]
kMeans' points =
  (\s -> (centroids s,s))                          `thenDo` (\prevCentrs ->
  (\s -> (clusterAssignments prevCentrs points, s)) `thenDo` (\assignments ->
  (\s -> (newCentroids assignments, s))            `thenDo` (\newCentrs ->
  (\s -> ((), s { centroids = newCentrs }))        `thenDo` (\_ ->
  (\s -> ((), s { steps = steps s + 1 }))          `thenDo` (\_ ->
  (\s -> (threshold s, s))                         `thenDo` (\t ->
  (\s -> (sum $ zipWith distance prevCentrs newCentrs, s)) `thenDo` (\err ->
  if err < t then (\s -> (newCentrs, s)) else (kMeans' points) )))))))

initialState :: (Vector v, Vectorizable e v)
             => (Int -> [e] -> [v]) -> Int -> [e] -> Double -> KMeansState v
initialState i k pts t = KMeansState (i k pts) t 0

kMeans :: (Vector v, Vectorizable e v) => (Int -> [e] -> [v]) -> Int -> [e] ->
Double -> [v]
kMeans i k pts t = fst $ kMeans' pts (initialState i k pts t)
```

Итак, похоже, что внесение изменений с целью обработки состояний большей ясности по сравнению с предыдущим вариантом не внесло: везде приходится явным образом записывать аргумент `s`. Давайте тогда попробуем усовершенствовать наши комбинаторы, чтобы лучше подстроиться под ситуацию. Заметьте, в частности, что у нас есть три вида функций, манипулирующих состоянием: в первых состояние

остается таким же, вторые обращаются к конкретному компоненту состояния и возвращают результат, третьи обновляют состояние. Давайте напишем определения для каждой из них:

```
remain :: a -> (s -> (a,s))
remain x = \s -> (x,s)

access :: (s -> a) -> (s -> (a,s))
access f = \s -> (f s, s)

modify :: (s -> s) -> (s -> ((), s))
modify f = \s -> ((), f s)
```

ПРИМЕЧАНИЕ

В предыдущих фрагментах кода использовался блочный тип (). Это тип, у которого имеется только один элемент — пустой кортеж (). Его обычно применяют, чтобы возвратить в функции что-либо, для чего на самом деле нет подходящего значения.

Переписанная версия `kMeans'` имеет следующий вид:

```
kMeans' :: (Vector v, Vectorizable e v) => [e] -> State (KMeansState v) [v]
kMeans' points =
  access centroids                                  `thenDo` (\prevCentrs ->
  remain (clusterAssignments prevCentrs points)     `thenDo` (\assignments ->
  remain (newCentroids assignments)                 `thenDo` (\newCentrs ->
  modify (\s -> s { centroids = newCentrs })        `thenDo` (\_ ->
  modify (\s -> s { steps = steps s + 1 })          `thenDo` (\_ ->
  access threshold                                  `thenDo` (\t ->
  remain (sum $ zipWith distance prevCentrs newCentrs) `thenDo` (\err ->
  if err < t then remain newCentrs else kMeans' points )))))))
```

Анализ комбинаторов

В начале раздела мы тщательно исследовали способ работы с цепочками функций, предполагающий использование значений `Maybe`, — это привело нас к пониманию того факта, что нам мог бы быть полезен комбинатор, который мы назвали `thenDo`. Впоследствии подобная идея была успешно применена к значениям `State`. Следуя тем же путем, который мы использовали с функторами, можно спросить: а может ли эта схема быть абстрагирована в класс типов?

Ответ положительный: комбинатор `thenDo` в точности соответствует функции `(>>=)` (ее называют функцией привязки) класса типов `Monad`. Этот класс типов охватывает все типы, позволяющие объединять между собой вычисления определенного вида. Два примера мы уже встречали:

- Монада `Maybe` объединяет функции, выполнение которых может закончиться неудачей.
- Монада `State s` объединяет функции, отслеживающие внутреннее состояние типа `s`.

Однако возможности класса типов `Monad` простираются намного шире простой привязки:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  f >> g = f >>= (\_ -> g)

  fail :: String -> m a
  fail s = error s
```

ВНИМАНИЕ

Представленная в классе Monad функция return не имеет ничего общего с ключевым словом return в C или Java. Подобный выбор разработчиков данного класса типов признать удачным нельзя, поскольку он наводит на мысль об императивном программировании. Перед тем как продолжить, постарайтесь выбросить эту мысль из головы. Монады никоим образом не связаны с императивным программированием, состоянием или изменчивостью (хотя конкретные экземпляры класса типов Monad охватывают и эти варианты использования).

Роль функции `(>>=)` уже понятна, поэтому перейдем к `return`. Эта функция описывает порядок заключения в оболочку чистого значения с помощью монады. Обычно она также описывает самый простой элемент, который может быть получен (путем простого возврата значения) для каждого вида вычислений. Поначалу все это кажется крайне расплывчатым, поэтому давайте взглянем на реализацию наших монад `Maybe` и `State`:

- Конкретным типом реализации `return` монады `State s` является `a -> State s a` или же `a -> s -> (a,s)`, что эквивалентно. Единственной реализацией, которую можно себе представить, является возвращение переданного значения с неизмененным внутренним состоянием. Именно для этого и предназначен комбинатор `remain` из предыдущего раздела. Это также соответствует замыслу «простейшего» вычисления состояния, которое вообще не вносит в состояние никаких изменений.
- Для `Maybe` тип возвращаемого значения имеет вид `a -> Maybe a`. Следовательно, есть две альтернативы: вернуть либо значение, заключенное в оболочку `Just`, либо `Nothing`. В определении `return` у нас уже есть значение для оболочки, поэтому разумнее иметь `return = Just`. Кроме того, если посмотреть на последний пример в разделе, где обсуждались неполные данные, можно увидеть, что на заключительном этапе использовалось выражение `Just:`, а теперь его можно заменить на `return`.

Следующей функцией в типовом классе является `(>>)`. В ее определении можно увидеть объединение двух вычислений, причем во втором из них возвращаемое значение первого вычисления не используется. Это может показаться странным, но с подобной ситуацией мы уже сталкивались: когда в нашей монаде `State s` изменялось состояние, возвращаемое этой операцией, значение не использовалось (оно всегда было представлено пустым кортежем `()`). Как и во многих других исходных реализациях, эта функция определена здесь, поскольку ожидается, что в некоторых экземплярах может быть дано более быстродействующее определение функции `(>>)`, чем в исходном определении.

И наконец, в типовом классе находится функция `fail`. Согласно имени, она позволяет определять особое поведение монады, когда какие-то из ее вычислений терпят

неудачу. Например, неудачное вычисление внутри монады Maybe должно, согласно нашему интуитивному представлению, привести к возвращению значения Nothing. Кроме этого прямого назначения, в данном разделе мы выясним назначение рассматриваемой функции в определении Monad.

Теперь у нас уже достаточно информации для использования в предыдущих примерах монады вместо собственного нестандартного комбинатора. Как это сделать, покажет упражнение 6.5.

УПРАЖНЕНИЕ 6.5. ИСПОЛЬЗОВАНИЕ МОНАД ДЛЯ НЕПОЛНЫХ ДАННЫХ И К-СРЕДНИХ

Все составные части экземпляра Monad для Maybe уже рассматривались. Напишите для него объявление instance. Затем перепишите функцию purchaseValue, воспользовавшись функциями (>>=) и return.

Еще один интересный факт, обнаруженный в нашем комбинаторе для Maybe, заключается в том, что при его использовании можно написать вполне работоспособную реализацию fmap. Давайте сначала подумаем о том, как будет выглядеть тип функции fmap, специализированной для State s:

```
fmap :: (a -> b) -> State s a -> State s b       -- с синонимами типов
fmap :: (a -> b) -> (s -> (a,s)) -> (s -> (b,s)) -- без синонимов типов
```

В реализации не должно быть ничего непонятного: нужно лишь применить функцию к возвращаемому значению, оставив состояние таким, каким оно было:

```
instance Functor (State s) where
  fmap f gWithState = \s -> let (gResult, gState) in g s in (f gResult, gState)
```

В предыдущем разделе мы видели, что ее можно было бы также определить для Maybe, используя комбинаторы из этого раздела, которые, как мы сейчас уже знаем, являются функциями класса типов Monad.

```
fmap f g = g >>= (\x -> return $ f x)
```

Несомненно, это определение эквивалентно предыдущему определению. Плюс здесь в том, что эта реализация работает для любой монады, то есть для показанного ранее определения fmap каждый экземпляр Monad порождает экземпляр Functor. Эта функциональность включена в модуль Control.Monad базового пакета под именем liftM.

ПРИМЕЧАНИЕ

Если каждый экземпляр Monad является также экземпляром Functor, почему указанная связь не показана в объявлении этих классов? Так уж сложилось исторически, но можно ожидать, что скоро Functor станет суперклассом Monad. А пока вы можете убедиться в том, что для каждого вашего экземпляра Monad у вас есть экземпляр Functor: такой экземпляр может быть легко создан с помощью ранее упомянутой функции liftM.

Do-нотация

Понятие монады, взятое в Haskell Филом Уадлером (Phil Wadler) и другими разработчиками из математической *теории категорий* (category theory), встречается

в библиотеках Hackage повсеместно. Оказывается, экземплярами класса `Monad` являются многие вычислительные структуры. Учитывая успешность их применения, разработчики Haskell решили включить в язык специальный синтаксис для монад — так называемую do-нотацию[1].

Блок `do` начинается с ключевого слова `do`, за которым следует ряд выражений. В процессе компиляции эти выражения преобразуются в обычный код, использующий функции `(>>=)`, `(>>)` и `fail`. Поэтому легче всего разобраться в назначении этой нотации, выяснив, как можно было бы использовать эти функции и как они используются в do-нотации.

Сначала рассмотрим два вычисления `f` и `g`, причем во втором не требуется никаких входных данных из первого. Мы уже видели, что это соответствует выражению задания последовательности этих вычислений `f >> g`. В do-нотации это записывается следующим образом:

```
do f
   g
```

Однако возможна также ситуация, когда вторая функция использует результат первой функции. На этот случай имеется функция привязки: `f >>= g`. Однако обычно привязка выполняется иначе — при этом используется безымянная функция и выполняется присваивание имени результату вычисления `f`: `f >>= (\x -> g x)`. В do-нотации также вводится имя для значения результата, но с использованием синтаксиса `<-`. В частности, выражение `f >>= (\x -> g x)` записывается в следующем виде:

```
do x <- f
   g x
```

Поддерживается также ввод вычислений, которые не производятся в контексте монад. Например, может потребоваться вызов функции `(+)` для полученного перед этим числа. Однако предположим, мы сделали следующее:

```
do number1 <- obtainNumber1 -- допустимо любое значение Maybe, например Just 3
   number2 <- obtainNumber2 -- допустимо любое значение Maybe, например Just 5
   sum <- number1 + number2
   return $ sqrt sum
```

Тогда компилятор выразит недовольство, поскольку сложение не имеет требуемого возвращаемого типа, который должен быть `m a`, где `m` — это монада. Одним из решений является изменение предпоследней строки кода на `sum <- return $ number1 + number2`, но это приведет к ненужным издержкам. Лучше воспользоваться выражением `let`:

```
do number1 <- obtainNumber1 -- допустимо любое значение Maybe, например Just 3
   number2 <- obtainNumber2 -- допустимо любое значение Maybe, например Just 5
   let sum = number1 + number2
   return $ sqrt sum
```

[1] Включение этой специальной нотации является сильным стимулом для авторов библиотек выяснить, действительно ли их типы формируют монаду, чтобы пользователи обращались к их библиотеке с уже известным синтаксисом.

Обратите внимание на то, что после этой разновидности выражения `let` нам не нужно записывать `in`.

Ранее уже объяснялось, что непосредственно в объявлениях блоков `let` и `where`, а также в объявлениях функций можно использовать сопоставление с образцом. Такая же возможность доступна и при применении в блоке `do` синтаксиса `<-` или `let`. Если помните, при этом возникает риск несоответствия возвращаемого значения образцу. В таких случаях компилятор автоматически добавляет вызов `error` с соответствующим сообщением. В do-нотации поведение немного другое: вместо вызова `error` компилятор вызывает функцию `fail` монады. Например:

```
do True <- willThatHold -- местозаполнитель для функции, возвращающей булево
значение Maybe
   f 5
```

Этот код можно было бы преобразовать в версию с явной ветвью для тех значений, которые не равны `True`, даже если эта часть не появится в коде:

```
willThatHold >>= \x ->
  case x of
    True -> f 5
    _ -> fail "error"
```

Высокая эффективность блоков `do` обусловлена тем фактом, что они не ограничиваются всего двумя выражениями: синтаксис преобразуется также и для большего количества выражений. Например:

```
do x <- f
   g
   y <- h x
   return y
```

Согласитесь, этот код намного легче читать, чем соответствующую ему трансляцию, которая выглядит так:

```
f >>= (\x -> g >> (h x -> (\y -> return y)))
```

В do-нотации пример с `Maybe` становится более понятным:

```
purchaseValueWithDo :: Integer -> Maybe Double
purchaseValueWithDo purchaseId = do n <- numberItemsByPurchaseId purchaseId
                                    productId <- productIdByPurchaseId purchaseId
                                    price     <- priceByProductId productId
                                    return $ fromInteger n * price
```

Для реализации алгоритма K-средних нужно отказаться от использования нашего доморощенного типа данных и обратиться к реализации `State` из модуля `Control.Monad.State` пакета `mtl`. Этот пакет (из библиотеки Monad Transformers Library) является одной из основных библиотек, наряду с библиотеками `base` и `container`, образующих Haskell Platform. В данном пакете содержатся экземпляры множества различных монад и полезных функций для всех этих монад.

В частности, в него включены эквиваленты уже написанных комбинаторов `access` и `modify`. Вместо использования функции для получения части состояния в этой реализации предоставляется доступ к полному состоянию посредством функции

get. Опираясь на тот факт, что State является также функтором, access можно написать путем подтягивания (lifting) функции доступа к результату функции get:

```
access :: (s -> a) -> State s a
access f = fmap f get
```

Поскольку получение только части состояния таким способом требуется очень часто, для решения этой задачи в пакет mtl включена функция gets.

Эта конкретная реализация позволяет также полностью изменять внутреннее состояние посредством функции put. Есть также возможность для его обновления с помощью функции modify. Учитывая все это, реализация kMeans' приобретает следующий вид:

```
kMeans' :: (Vector v, Vectorizable e v) => [e] -> State (KMeansState v) [v]
kMeans' points = do prevCentrs <- fmap centroids get
                    let assignments = clusterAssignments prevCentrs points
                        newCentrs = newCentroids assignments

                    modify (\s -> s { centroids = newCentrs })
                    modify (\s -> s { steps = steps s + 1 })
                    t <- fmap threshold get
                    let err = sum $ zipWith distance prevCentrs newCentrs
                    if err < t then return newCentrs else kMeans' points
```

Последнее замечание состоит в том, что при использовании типа данных State из пакета mtl имеется несколько вариантов предоставления исходного состояния и соответственно, выполнения всех вычислений. Все это можно свести к трем типам, каждый из которых возвращает разный набор информации.

```
runState :: State s a -> s -> (a,s) -- возврат значения и финального состояния
evalState :: State s a -> s -> a    -- возврат только значения
execState :: State s a -> s -> s    -- возврат только финального состояния
```

Для K-средних интерес заключается только в возвращении значения, поэтому следует воспользоваться вторым вариантом:

```
kMeans :: (Vector v, Vectorizable e v) => Int -> [e] -> Double -> [v]
kMeans n pts t = evalState (kMeans' pts) (initializeState n t)
```

Монадические законы

Следует помнить, что не все определения (>>=) и return будут создавать настоящую монаду. Как и в случае с функторами, класс типов Monad регулирует поведение своих экземпляров рядом законов. Соблюдение этих законов не проверяется компилятором, но их все же нужно соблюдать, если вы не хотите, чтобы пользователь или компилятор столкнулся с непонятными ошибками в коде. Если при первом прочтении со всеми тонкостями разобраться не удастся, волноваться не стоит: эта информация пригодится только при разработке новых монад, для их использования она вообще не нужна.

Первые два закона относятся к операции привязки с помощью функции return:

- Выражение return a >>= f должно быть эквивалентно выражению f a, или в donотации выражение do { x <- return a; f x} должно быть эквивалентно мини-

мальному выражению `do { f a }`. То есть ничего не изменится, если вычислить выражение, посредством функции `return` заключенное в оболочку монады или без нее.

- Выражение `x >>= return` должно быть эквивалентно выражению `x`, или в do-нотации выражение `do { y <- m; return y }` должно быть эквивалентно выражению `do { m }`. Это означает, что функция `return` при привязке к другим вычислениям просто извлекает значение из оболочки и снова заключает значение в оболочку.

ПРИМЕЧАНИЕ

Для хорошего стиля программирования на языке Haskell второй закон имеет весьма важное значение. Следует помнить, что вычислять значения внутри монады с его немедленным возвращением не требуется, нужно просто включить вычисление значения в качестве выражения.

Следующий закон гласит об ассоциативности операции `(>>=)`, хотя его лучше выразить с помощью do-нотации. В частности, рассмотрим следующий код, в котором один блок `do` следует за другим блоком `do`:

```
do x <- m
   do y <- f x
      g y
```

Закон утверждает, что в этом коде сначала выполняется одна часть вычислений, вложенных в блок `do`, затем вторая часть:

```
do y <- do x <- m
           f x
      g y
```

Это означает, что порядок вложения блоков `do` может быть произвольным — результат не изменится. Это похоже на тот факт, что выражение `(1 + 2) + 3` эквивалентно выражению `1 + (2 + 3)`. Данное обстоятельство позволяет просто записать этот код следующим образом:

```
do x <- m
   y <- f x
   g y
```

И последний закон требует, чтобы определение функции `fmap`, которое было дано на основе монады, было действительно функцией `fmap` ее экземпляра `Functor`. То есть выражение `fmap f g` должно быть эквивалентно выражению `g >>= (\x -> return $ f x)`.

ПОВСЕМЕСТНОЕ ИСПОЛЬЗОВАНИЕ МОНАД

Если просмотреть доступную в Интернете информацию по Haskell, можно увидеть огромное количество руководств, посвященных монадам. Похоже, это свидетельствует о том, что в монадах довольно трудно разобраться, но этого и не нужно.

Чаще всего речь идет не о разработке, а об использовании монад. Вы уже познакомились с монадами Maybe и State, далее в книге будут встречаться и другие экземпляры этого класса типов. Если вы разберетесь в том, как использовать каждый из них, можно будет считать, что вы готовы к реальному программированию на Haskell и идете по пути полного понимания концепции монад.

Различные виды состояний

Важность освоения наиболее часто используемых экземпляров монад уже отмечалась. В данном разделе мы рассмотрим монады, имеющие отношение к сохранению или использованию внутреннего состояния. Две такие монады, `Reader` и `Writer`, могут рассматриваться как ограниченные версии монады `State`. Однако у них есть собственные конкретные варианты применения, поэтому интересно будет узнать, в каких сценариях можно использовать каждую из них. Затем мы перенесем свое внимание на монаду `ST`, имеющую весьма специфическое назначение: она позволяет применять изменяемые ссылки (как переменные в императивном языке), но управляемым образом, не поступаясь чистотой языка.

Состояние и линзы

Перед тем как углубиться в изучение других монад, хотелось бы выделить особый инструмент библиотеки `lens` — это специальные комбинаторы для использования линз внутри монады `State`. Благодаря этим комбинаторам код напоминает более последовательный стиль программирования, сохраняя при этом полную чистоту языка.

Вместо использования `get` с последующим применением линзы с помощью `view` можно напрямую обратиться к части структуры данных с помощью функции `use`. Эта функция уже дает результат в монаде `State`, поэтому для получения значения вызов дополнительной функции `fmap` или `return` не нужен.

Вспомним, что при использовании для линз функций обновления вам всегда приходилось создавать структуру, применяемую путем явного указания либо `$`, либо `&`. Но внутри монады `State` всегда нужно брать в расчет специальное значение — внутреннее состояние. Для каждой функции обновления, заканчивающейся тильдой (например: `.~`, `%~` или `+~`), в библиотеке `lens` имеется соответствующая функция, изменяющая внутреннее состояние, которая заканчивается знаком равенства (`.=`, `%=` или `+=` соответственно).

Если для хранения состояния используется тип данных с несколькими полями, причем для него имеются линзы, можно задействовать синтаксис, очень похожий на синтаксис языка C для изменения состояния. Например, в K-средних требуется следующее состояние для отслеживания центроидов, порогов и количества этапов:

```
data KMeansState v = KMeansState { _centroids :: [v]
                                 , _threshold :: Double
                                 , _steps :: Int }
```

В следующем коде показано, как переписать реализацию `kMeans'` с использованием функции `use` для получения информации или ее временного хранения, а также функций `(.=)` и `(+=)` для обновления центроидов и прохождения этапов на каждой итерации:

```
kMeans' :: (Vector v, Vectorizable e v) => [e] -> State (KMeansS\tate v) [v]
kMeans' points = do prevCentrs <- use centroids
                    let assignments = clusterAssignments prevCentrs points
                        newCentrs = newCentroids assignments
```

```
                centroids .= newCentrs
                steps += 1
                let err = sum $ zipWith distance prevCentrs newCentrs
                t <- use threshold
                if err < t then return newCentrs else kMeans' points
```

Теперь, имея представление о `State`, можно также немного остановиться на такой функциональной возможности `lens`, как *масштабирование* (zooming). При масштабировании линза берется в качестве источника входных данных, после чего в вычислениях используется информация о внутреннем состоянии, содержащаяся в этой линзе. В некотором смысле это напоминает фокусирование на некоторое время внимания на небольшой части структуры. Предположим, что у вас есть очень простая функция, которая будет увеличивать на какое-то число все идентификаторы списка `Clients`, а также обновлять имена элементов, переводя их в верхний регистр. Рассмотрим такое объявление состояния:

```
data ExampleState = ExampleState { _increment :: Int, _clients :: [Client Int] }
                  deriving Show
makeLenses ''ExampleState
```

Тогда следующая функция реализует упомянутую функциональность масштабирования в отношении каждого клиента:

```
zoomExample :: State ExampleState ()
zoomExample = do n <- use increment
                 zoom (clients.traversed) $ do
                   identifier += n
                    person.fullName %= map toUpper
```

Вот как выглядит пример применения этой функции к списку клиентов в интерпретаторе:

```
*Chapter6.StateLenses> :{
*Chapter6.StateLenses| let client1 = Individual 4 (Person "John" "Smith")
*Chapter6.StateLenses|     client2 = Individual 3 (Person "Albert" "Einstein")
*Chapter6.StateLenses| in execState zoomExample (ExampleState 2 [client1, client2])
*Chapter6.StateLenses| :}
ExampleState { _increment = 2, _clients = [
  Individual { _identifier = 6
             , _person = Person {_firstName = "JOHN", _lastName = "SMITH"}}
, Individual { _identifier = 5
             , _person = Person {_firstName = "ALBERT", _lastName = "EINSTEIN"}}]}
```

Монады Reader, Writer и RWS

Во многих случаях глобальное состояние при выполнении кода не меняется, а содержит пакет значений, принятых в качестве констант. Например, в алгоритме K-средних константы для конкретного запуска — это количество создаваемых кластеров, информация, согласно которой выполняется алгоритм, или предел погрешности. Поэтому стоит рассматривать эти значения не так, как все остальное состояние: поскольку мы не собираемся их изменять, давайте потребуем от компилятора Haskell гарантировать их неизменность.

Если помните, в главе 4 в качестве функтора было представлено выражение `(->) r`. Если рассматривать контекст как дополнительный скрытый аргумент функций, то его можно также считать монадой, представляющей именно вычисления, касающиеся дополнительного контекста, который не должен изменяться. Давайте попробуем написать соответствующий экземпляр `Monad`, чтобы попрактиковаться и сконцентрироваться на идеях монадических структур. Наипростейшей функцией, которая должна иметь тип `a -> (r -> a)`, является `return`. Итак, для нее есть только один вариант:

```
return x = \r -> x
```

Типы могут также помочь нам в написании для этой монады реализации `(>>=)`:

```
(>>=) :: (r -> a) -> (a -> r -> b) -> (r -> b)
```

Как вы знаете, результат `(>>=)` является функцией, получающей контекст. Используя этот контекст, можно задействовать первую функцию для получения значения типа `a`. Затем можно просто передать это значение вместе с контекстом второй функции, чтобы получить окончательный результат типа `b`:

```
f >>= g = \r -> g (f r) r
```

Теперь у нас есть весь код, необходимый для получения искомого экземпляра `Monad`, и его можно объединить в объявлении `instance`:

```
instance Monad ((->) r) where
  f >>= g = \r -> g (f r) r
  return x = \r -> x
```

Как и монада `State`, эта монада входит в библиотеку `mtl` под именем `Reader` (читатель), поскольку контекст может быть только прочитан, но не записан. Однако помимо монадической структуры требуется также инструмент получения контекста. Библиотека предоставляет для этого две разные функции:

- Функция `ask` извлекает весь контекст, подобно функции `get` монады `State` из библиотеки `mtl`.
- Функция `asks` применяет функцию к контексту и возвращает результат. Эта функция, которая хорошо подходит для запроса конкретного поля структуры, очень похожа на функцию доступа (ее мы разрабатывали для нашей самодельной монады `State`) и на функцию `gets` из `mtl`.

Типичным примером использования функции `Reader` является обработка параметров приложения. Обычно эти параметры, которые считываются в начале работы приложения из какого-нибудь конфигурационного файла, во время выполнения приложения не меняются. Явное включение аргумента `Settings` в каждую отдельную функцию приложения стало бы настоящей рутиной, поэтому заключение его в оболочку монады `Reader` можно считать весьма элегантным решением.

```
data Settings e v = Settings { i :: Int -> [e] -> [v], k :: Int, th :: Double, user
:: Person }

kMeansMain :: (Vector v, Vectorizable e v) => [e] -> Reader (Settings e v) [v]
kMeansMain points = do i' <- asks i
                       k' <- asks k
                       t' <- asks th
                       return $ kMeans i' k' points t'
```

Как и в случае с монадой `State`, нужна также функция выполнения монады, которой будет передаваться контекст. В данном случае она называется `runReader`, поскольку получает в качестве аргумента только исходное неизменяемое состояние.

Даже при том, что основная идея монады `Reader` заключается в описании некоторого неизменяемого контекста, реализация `mtl` также предоставляет вариант выполнения части кода с контекстом только для данного подвычисления. Для этого используется функция `local`, предоставляющая ей функцию изменения текущего состояния и выполняемое вычисление. Во внутреннем блоке вызовы `ask` или `asks` ссылаются на измененный контекст, который будет возвращен в исходное состояние после завершения вызова функции `local`. Например, может понадобиться сравнить выполнение K-средних при увеличении количества кластеров на единицу. Чтобы использовать предыдущий контекст `Settings`, его требуется изменить с учетом расширившегося множества кластеров:

```
compareClusters :: (Vector v, Vectorizable e v) => [e] -> Reader (Settings e v)
([v], [v])
compareClusters points = do c1 <- kMeansMain points
                            c2 <- local (\s -> s { k = k s + 1 })
                                        (kMeansMain points)
                            return (c1, c2)
```

Мы рассмотрели функции, которые *потребляют* состояние, но не изменяют его. Обратная сторона медали включает в себя функции, которые *генерируют* некое состояние, но никогда его не используют. Эта ситуация напоминает библиотеку протоколирования: вы постоянно добавляете сообщения в журнал, но никогда не оглядываетесь на предыдущие сообщения, вас интересует только растущий журнал. Для этого нужно воспользоваться монадой `Writer` (писатель), которая, конечно же, есть в библиотеке `mtl`.

Ключевое решение, принятое разработчиками для данной конкретной реализации монады `Writer`, заключается в том, что всякий раз, когда нужно добавить к выходному состоянию какое-нибудь новое значение, способ его объединения с предыдущим состоянием задается экземпляром `Monoid`. Вот два примера, которые помогут вам лучше разобраться с этим фактом:

- В случае создания журнала, состоящего из строк, монадическая структура — это то, что относится к списочному типу: нейтральным элементом является пустой список, а операцией объединения двух строк — их конкатенация. То есть чтобы создать журнал, в качестве типа аргумента для `Writer` нужно выбрать `String`.
- Другим случаем, когда информация может рассматриваться в качестве выходного параметра, является подсчет количества итераций для алгоритма K-средних. В этом случае при выполнении какого-то количества итераций это количество нужно добавить к текущему значению. Следовательно, монадической структурой является целочисленное значение с суммой. Следует помнить, что поскольку числа обычно имеют две монадические структуры (одна для сумм, другая для произведений), для использования сложения в качестве операции нужно заключить значения в оболочку из нового типа `Sum`.

Чтобы изменить выходное состояние в соответствии с новым значением (которое будет получено в сочетании с предыдущим значением), нужно использовать функцию `tell` с этим новым значением в качестве аргумента:

```
accessDatabase :: Writer String ()
accessDatabase = do tell "Start database access"
                    info <- readInformation
                    computeValue info
                    tell "Finish database access"
```

Поскольку исходное значение для выходной информации должно быть взято в качестве нейтрального элемента соответствующего моноида, какой-либо дополнительный аргумент для запуска монады `Writer`, использующей функцию `runWriter`, которая возвращает кортеж с результатом вычислений и выходной информацией, не нужен.

Монада `Writer` является примером монады, объявление экземпляров которой нужно изучать. Именно это наряду несколькими приемами написания правильных типов потребуется от вас при выполнении упражнения 6.6.

УПРАЖНЕНИЕ 6.6. ВНУТРЕННЕЕ УСТРОЙСТВО МОНАДЫ WRITER

Монада Writer является одной из монад, относящихся к типу (a,m) для любого экземпляра m класса Monoid. Но напрямую написать instance Monoid m => Monad (a,m) вы не можете, поскольку параметр типа a не должен записываться в определении, иначе это не будет соответствовать виду. Следовательно, для объявления нужно воспользоваться функцией newtype:

```
newtype MyWriter m a = MyWriter (a,m)
```

Теперь объявление можно написать начиная с instance Monoid m => MyWriter m. Кроме того, предоставьте определение функции tell. Не забудьте сначала записать особые типы для функций return и (>>=) — это намного упростит задачу.

Haskell старается тщательно ограничивать объем вычислительной мощности, выделяемой каждой функции, давая компилятору возможность выявлять больше разновидностей ошибок, чем в других языках. Эта философия может быть перенесена в контекст или в состояние конкретной функции: к информации, которая должна быть неизменной, нужно предоставлять доступ только по чтению, к выводимым, но не запрашиваемым данным — только по записи, а к предназначенному для работы внутреннему состоянию — по чтению и записи. Похоже, что во многих случаях вам потребуется сочетание монад `Reader`, `Writer` и `State`.

Порядок сочетания монад будет темой следующей главы, но для данного конкретного случая разработчики библиотеки `mtl` создали монаду `RWS` (этот акроним из первых букв каждой из монад `Reader`, `Writer` и `State`, составляющих функциональность новой монады), находящуюся в модуле `Control.Monad.RWS`. Конкретное значение этой монады получает три параметра типа: один для контекста, второй для вывода и третий для изменяемого состояния (соответственно один только для чтения, второй только для записи, третий для чтения и записи). Операторы, необходимые для доступа к каждому компоненту, остаются теми же: `ask` и `asks` получают значение `Reader`, `tell` включает новое значение в монаду `Writer`, а `get`, `put` и `modify` используются для запроса и обновления значения `State`.

Используя монаду `RWS`, можно создать нашу финальную версию алгоритма K-средних, которая хранит порог в виде контекста, запоминает количество итера-

ций с помощью монады Writer и задействует центроиды в качестве внутреннего обновляемого состояния. Обратите внимание на необходимость заключения целочисленных значений в оболочку Sum newtype, чтобы сообщить компилятору, какую из моноидных структур мы хотим использовать для целых чисел.

```
kMeans' :: (Vector v, Vectorizable e v) => [e] -> RWS Double (Sum Int) [v] ()
kMeans' points = do prevCentrs <- get
                    let assignments = clusterAssignments prevCentrs points
                        newCentrs = newCentroids assignments
                    put newCentrs
                    tell (Sum 1)
                    t <- ask
                    let err = sum $ zipWith distance prevCentrs newCentrs
                    unless (err < t) $ kMeans' points

kMeans :: (Vector v, Vectorizable e v)
       => (Int -> [e] -> [v]) -> Int -> [e] -> Double -> ([v], Sum Int)
kMeans i n pts t = execRWS (kMeans' pts) t (i n pts)
```

Как видите, RWS предлагает весьма элегантный способ разработки функций, явным образом выделяя назначение каждого фрагмента информации. Особая польза от применения этой монады проявляется при портировании алгоритмов, которые были ранее разработаны на императивных языках, поскольку позволяет сохранить чистоту кода в процессе портирования.

Изменяемые ссылки с монадой ST

Мы уже видели, как удачные сочетания дополнительных аргументов для функций и комбинаторов позволяют упростить описания вычислений состояния. Еще эти абстракции могут быть превращены в монады, позволяющие воспользоваться do-нотацией, которая делает код более понятным. Но кроме всего этого, Haskell предлагает настоящие изменяемые переменные в том же смысле, в каком они понимаются в C или Java, — и все благодаря монаде ST.

ВНИМАНИЕ

Возможно, после прочтения данного раздела вы начнете повсеместно использовать монаду ST в своем коде. Выяснить, как она работает, действительно интересно, поскольку это может привести к более эффективной реализации некоторых алгоритмов и дать представление обо всем диапазоне возможностей Haskell Platform. Тем не менее использование данной монады удачным решением не считается, поэтому сначала нужно испробовать другие решения, касающиеся организации создаваемого вами кода.

У вас может возникнуть вопрос: не нарушит ли использование монады ST чистоту языка? Ответ отрицательный. Причина в том, что способ реализации ST не допускает попадания изменяемых переменных вовне в большей степени, чем вы сами того захотите. То есть когда монада ST используется в каком-нибудь конкретном месте, вы можете создавать новые изменяемые переменные и изменять их столько, сколько вам заблагорассудится. Но в конце этих вычислений все изменяемые переменные окажутся уничтоженными, а единственным, что представляет для нас интерес, будет возвращаемое значение. Таким образом, для внешнего мира все

по-прежнему останется неизменным. Кроме того, система времени выполнения Haskell отделяет друг от друга изменяемые переменные разных экземпляров `ST`, гарантируя тем самым, что изменяемые переменные из разных миров не будут влиять друг на друга.

Давайте представим акторов в пьесе `ST`. Первым, разумеется, является сама монада `ST` из модуля `Control.Monad.ST`, которой передаются два аргумента, но только второй из них важен для практического применения — это тип значения, возвращаемого после вычислений (как и в других образцах монад состояния). Первый аргумент внутренне используется компилятором для присваивания уникального идентификатора, который не дает влиять друг на друга различным вычислениям. Первый аргумент указывается особым способом — с использованием так называемого *квантификатора* (quantifier) типа. Мы не будем сейчас углубляться в тему квантификаторов, пока достаточно знать, что, к примеру, вычисления в рамках монады `ST` возвращают целочисленное значение, объявляемое следующим образом:

```
{-# LANGUAGE RankNTypes #-}
import Control.Monad.ST

intExample :: forall s. ST s Integer
```

После объявления вычислений при их выполнении данное значение используется просто в качестве аргумента функции `runST`.

При вычислениях в рамках монады `ST` можно создавать изменяемые переменные типа `STRef a` из модуля `Data.STRef`, где `a` является типом значений, которые будут удерживаться в ячейке. Все определения и функции, имеющие отношение к `STRefs`, находятся в модуле `Data.STRef` пакета `base`. Каждая новая переменная должна создаваться через вызов функции `newSTRef`, которая использует исходное значение для переменной (неинициализируемые переменные не поддерживаются). Результирующим значением данной конкретной изменяемой переменной является идентификатор, который может потребоваться позже для обращения к ней и изменения ее содержимого.

Значение переменной может быть запрошено с помощью функции `readSTRef`, которой для работы нужен лишь идентификатор переменной. Для обновления переменной, как и в случае с монадой `State`, есть два способа:

- Можно напрямую указать новое значение, используя функцию `writeSTRef`.
- Можно указать функцию, которая будет заменять текущее значение ячейки `STRef` новым значением. Для этого можно задействовать функцию `modifySTRef`. Однако поскольку функция `modifySTRef` выполняется как ленивая, ее использование настоятельно не рекомендуется, потому что это может привести к утечкам памяти, подобным тем, которые рассматривались в предыдущей главе. Вместо нее следует применять строгую функцию `modifySTRef'`.

Например, следующий код вычисляет с помощью монады `ST` длину списка. Код начинается с создания новой изменяемой переменной с начальным значением 0. Затем осуществляется обход списка с обновлением значения на каждой итерации. Обратите внимание на то, что код передает специальный идентификатор для из-

меняемой переменной, чтобы она стала доступна в определенной нами функции `traverseList`.

```
listLength :: [a] -> Integer
listLength list = runST $ do l <- newSTRef 0
                             traverseList list l
                             readSTRef l
                  where traverseList [] _     = return ()
                        traverseList (_:xs) l = do modifySTRef' l (+1)
                                                   traverseList xs l
```

ПРИМЕЧАНИЕ

Напрямую использовать для списка функцию map или fold нельзя, поскольку в монадическом контексте типы таких функций этого не позволяют. В следующей главе мы познакомимся с монадическими аналогами этих функций, такими как mapM, foldM и forM.

УПРАЖНЕНИЕ 6.7. РЕАЛИЗАЦИЯ АЛГОРТИМА K-СРЕДНИХ С ПОМОЩЬЮ ST

Реализуйте алгоритм K-средних, используя монаду ST. В частности, вам нужно создать одну ячейку STRef для хранения центроидов, которые будут обновляться, и еще одну такую ячейку для хранения количества итераций.

Выводы

В данной главе мы наконец-то почувствовали, что такое *монада*:

- Мы изучили несколько реализаций алгоритма кластеризации K-средних, начиная с реализации, созданной собственноручно, затем перейдя к реализации, улучшенной с помощью комбинаторов, и закончив версиями с монадами `State` и `RWS`.
- Сбор информации из неполных данных привел нас к определению комбинаторов для работы со значениями `Maybe` более простым способом.
- Из сходства кода, использующего значения `Maybe` и `State`, мы вывели понятие монады как средства объединения вычислений, обладающего рядом специальных характеристик, таких как возможность неудачи при вычислениях и наличие внутреннего состояния.
- Монады являются одними из наиболее важных конструкций в Haskell, они имеют собственный синтаксис, называемый do-нотацией, который мы изучили достаточно глубоко. Это наиболее востребованный стиль написания монадического кода.
- Мы рассмотрели также некоторые другие монады: монада `Reader` содержит контекст только для чтения; монада `Writer` выводит значение только для записи, которое объединяется с помощью моноидной структуры; монада `RWS` сочетает в себе функциональность трех монад, `Reader`, `Writer` и `State`; монада `ST` реализует управляемые изменяемые переменные.
- Помимо монад в данной главе мы изучили *линзы*, которые обеспечивают запрос и обновление структур данных наиболее простым и эффективным образом.

Другие рекомендуемые монады

7

Предыдущая глава была введением в концепцию монад. В частности, для задач обработки ошибок используется монада `Maybe`, для констант — `Reader`, для протоколирования — `Writer`. Кроме того, с точки зрения монад было представлено состояние, что позволило прояснить вопрос о том, как монадические функции формируют базу, объединяющую код для всех этих вариантов применения.

Списочная монада может представить несколько результатов выполнения функции. Это нам пригодится для моделирования маршрутов между «годами» в машинах времени. Но еще нам нужно будет позаботиться о том, чтобы код не входил в бесконечные циклы. Этой проблемы можно избежать путем тщательного продумывания и использования монады `Logic`. Можно заметить, что списки играют в Haskell-программировании весьма важную роль: многие замыслы и алгоритмы зависят от их весьма специфичной структуры. В этой главе мы также проясним порядок взаимодействия между списочными и другими монадами.

Еще одной темой этой главы будет анализ данных. Мы реализуем априорный алгоритм, который выводит ассоциативные правила из множества данных. Вкратце, *ассоциативные правила* (association rules) выражают отношения между переменными: например, «большинство людей, покупающих машины времени, покупают также путеводители». Выразив априорный алгоритм через *списочную монаду* (list monad), мы получим новый класс типов `MonadPlus`, вводящий в разнообразие данных возможность выбора.

До сих пор мы рассматривали примеры, в каждом из которых использовалась всего одна монада. Как правило, именно такие примеры чаще всего встречаются в Haskell-программировании. Тем не менее более сложные приложения иногда требуют эффективного применения сразу нескольких монад, например `Maybe` для обработки ошибок и `Writer` для протоколирования информации о процессе. Наиболее распространенным средством объединения нескольких монад являются *монадические преобразователи* (monad transformers), и именно они будут одной из тем данной главы.

Возвращение более одного значения

В предыдущей главе мы узнали, как монада `Maybe` моделирует функции, выполнение которых может закончиться неудачей. Монадические функции поддержива-

ют программные потоки успешных значений и останавливают их при первом же обнаружении неудачи. Если посмотреть на списки с нескольких сторон, то можно прийти к выводу, что функция, возвращающая список, на самом деле возвращает для одних и тех же аргументов несколько возможных значений: пустой список моделирует неудачу, одноэлементный список — детерминированные вычисления, возвращающие только одно значение, а более длинный список — несколько возможных результатов, или маршрутов выполнения, что обычно называется моделированием *недетерминированного* (nondeterministic) поведения.

ВНИМАНИЕ

Когда речь идет о Haskell-списках, понятие «недетерминированность» используется несколько в ином смысле, чем при разговоре о других видах программных систем компьютерного мира. Это понятие было выбрано потому, что при имитации недетерминированных вычислений возможно несколько результатов, которые могут быть представлены с помощью списка. Однако вычисления внутри списочной монады являются полностью детерминированными и чистыми, они не добавляют никой неопределенности ни в способ вычисления, ни в способ упорядочения результатов.

Списочная монада

Поскольку тип `Maybe` формирует монаду, вполне логичен вопрос, формируют ли монаду списки. Чтобы проверить, так ли это, нужно открыть GHCi, вызвать функцию `return`, которая, как мы знаем, должна быть реализована любым монадическим типом, и заставить эту функцию вернуть список. Если выражение выполнится успешно, значит тип `[]` (вспомним, что так называется списочный тип) является монадой. Давайте проделаем это с помощью простого примера:

```
Prelude> return 1 :: [Integer]
[1]
```

По результату можно увидеть не только то, что списки формируют монады, но и то, что экземпляр `Monad` для списков не противоречит предыдущим рассуждениям: детерминированные вычисления только с одним значением моделируются как одноэлементный список. Второй функцией, представленной во всех экземплярах монад, является `(>>=)` — оператор привязки. Давайте сначала рассмотрим тип оператора привязки при создании экземпляра для данного конкретного случая:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Для проверки поведения оператора привязки мы применим к простому числовому списку функцию, которая при получении числа будет формировать в списке его удвоенное и утроенное значения:

```
Prelude> [1,2,3] >>= \x -> [2*x,3*x]
[2,3,4,6,6,9]
```

Как видите, функция применяется к каждому элементу списка, и все результаты собираются в новый список. Вообще-то ранее нам уже встречалась функция с таким поведением, которое можно выразить как «применение плюс сборка» — это

сочетание обхода списка, возвращающего список списков, и выведения результата на один уровень с помощью функции `concat`:

```
Prelude> map (\x -> [2*x,3*x]) [1,2,3]
[[2,3],[4,6],[6,9]]
Prelude> concat $ map (\x -> [2*x,3*x]) [1,2,3]
[2,3,4,6,6,9]
```

Но лучше всего разобраться в том, почему списочная монада, как было отмечено, моделирует недетерминизм, — это применить код с do-нотацией. Например, следующий фрагмент кода получает значения из двух списков, а затем вычисляет произведение каждой пары. Внимательно изучите результат:

```
Prelude> :{
Prelude| do x <- [1,2,3]
Prelude|    y <- [7,8,9]
Prelude|    return $ x * y
Prelude| :}
[7,8,9,14,16,18,21,24,27]
```

Код берет все возможные значения из первого списка и все возможные значения из второго списка, после чего возвращает результат применения функции умножения к каждой паре. В этом примере показано, как монадическая нотация может скрывать в себе ряд неожиданностей: ведь от синтаксиса `<-` можно было и не ожидать, что каждое значение в списке будет браться по очереди. Упражнение 7.1 поможет нам в полной мере разобраться в поведении этого кода на примере сломанных машин времени.

УПРАЖНЕНИЕ 7.1. СЛОМАННЫЕ МАШИНЫ ВРЕМЕНИ

Бывает, что машины времени ломаются. Представьте, к примеру, что у вас есть машина времени, которая произвольно перепрыгивает между предыдущим годом, тремя годами в будущем и пятью годами в будущем. Учтите, что для перемещения на четыре года в будущее вам придется осуществить прыжок на пять лет в будущее, а затем вернуться на один год назад.

Представьте, что у вас есть такая сломанная машина. Напишите функцию brokenThreeJumps, которая, начиная с какого-нибудь года, возвращает все возможные годы, в которые вы на вашей сломанной машине времени можете попасть после трех прыжков. Воспользуйтесь списочной монадой и do-нотацией.

Затем разработайте функцию brokenJumps, обобщающую предыдущую функцию и возвращающую все доступные годы после указанного количества прыжков.

Новый взгляд на монады

Из уже рассмотренного материала можно легко получить объявление экземпляра монад для списков. Если помните, монада состоит из функции `return`, в которую заключается единственное значение, и функции привязки (`>>=`), формирующей в монаде программные потоки нескольких вычислений. Соответствующий экземпляр для списка имеет следующий вид:

```
instance Monad [] where
  return a = [a]
  x >>= f  = concat $ map f x
```

Но есть и еще один возможный способ определения монады, и списки являются вполне подходящим примером этого другого представления. Вместо функции (>>=) определяется функция fmap, а также дополнительно определяется функция join, которая для каждой монады m должна иметь тип m (m a) -> m a. В случае списков функция join сводится к наличию типа [[a]] -> [a]. И здесь снова можно найти старого друга — функцию concat, выводящую списки на один уровень.

Определение монады в понятиях этих функций является эквивалентом их объявления с помощью функции return и привязки. В частности, у каждой монады имеется осмысленное определение для fmap и join. Вы уже знаете определение fmap для Maybe. А вот как выглядит соответствующий код реализации join:

```
join :: Maybe (Maybe a) -> Maybe a
join Nothing         = Nothing
join (Just Nothing)  = Nothing
join (Just (Just x)) = Just x
```

Поскольку речь идет об эквивалентности определений, нужно рассказать о том, как определить (>>=) в понятиях fmap и join, и наоборот. Начнем с оператора привязки. Вот предлагаемое определение:

```
x >>= f = join $ fmap f x
```

В качестве упражнения, позволяющего убедиться в справедливости этого правила, можно применить левую часть к списочной монаде и проверить, будет ли результат таким же, как в нашем определении функции (>>=). Эквациональное рассуждение в случае со списком похоже на такой пример:

```
join $ fmap f x = concat $ map f x = x >>= f
```

Следовательно, эквивалентность соблюдается по крайней мере для списка. Теперь давайте перейдем к обратным эквивалентностям:

```
fmap f x = x >>= (\y -> return $ f y)
join x   = x >>= id  -- id является функцией эквивалентности, id x = x
```

Хорошим упражнением, позволяющим освежить ваши знания об эквациональном рассуждении, станет проверка соблюдения обратных определений для списочных монад, которая уже проводилась в предыдущем коде.

ПРИМЕЧАНИЕ

Мы только что познакомились еще с одним способом получения определения fmap с помощью монадических функций помимо использования функции liftM в соответствии с объяснениями, данными в предыдущей главе. Каждый тип, принадлежащий типовому классу Monad, должен быть обитателем класса типов Functor. Как уже объяснялось, это не отражается на определении класса типов Monad по историческим причинам, но ситуация вскоре будет исправлена.

Неудачи и альтернативы

В монадах реализуется понятие неудач, поддерживаемое в типовом классе функцией fail. Но в некоторых случаях кроме неудач поддерживается также понятие нескольких вариантов удачного выполнения, как это происходит в случае с функ-

цией `[]`. Для подобных случаев в стандартную библиотеку включен класс типов `MonadPlus`, который расширяет класс `Monad` двумя операциями. Этот класс типов находится в модуле `Control.Monad`, содержащем множество других полезных функций, с которыми мы позже познакомимся. Рассмотрим для справки полное объявление класса типов:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Функция `mzero` относится к функциям, поддерживающим неудачу. В отношении этой функции нужно сделать одно важное замечание: она возвращает значение, являющееся параметрическим по отношению к каждому возможному элементу, заключенному в монаду, без какого-либо ввода извне. В отношении списков это означает, что возвращен может быть единственный вид значений — пустой список, поскольку только он может быть создан таким образом. Это можно проверить, запрашивая значение через интерпретатор:

```
*Chapter7.MonadPlus> import Control.Monad
*Chapter7.MonadPlus Control.Monad> mzero :: [Integer]
[]
```

Успешные результаты от нескольких ветвей выполнения могут быть объединены с помощью еще одного оператора, `mplus`. Поскольку списки моделируют несколько успешно полученных значений в качестве своих элементов, объединение нескольких частей должно быть подобно объединению двух списков. Давайте, к примеру, определим два различных способа возможной поломки машины времени: один будет заключаться в произвольном путешествии между предыдущим и последующим годами, второй — в произвольном путешествии между 1024 годом и двумя годами в будущем:

```
broken1 :: Integer -> [Integer]
broken1 n = [n-1, n+1]

broken2 :: Integer -> [Integer]
broken2 n = [1024, n+2]
```

У нас может не быть сведений о характере повреждения нашей машины времени. Поэтому стоит объединить все возможные годы, в которые на такой машине можно путешествовать, указав их либо в `broken1`, либо в `broken2`. Это альтернативное поведение и будет выявляться функцией `mplus` в `MonadPlus`:

```
*Chapter7.MonadPlus Control.Monad> broken1 73 `mplus` broken2 73
[72,74,1024,75]
```

Наиболее важной функцией, применяемой при неудачном поведении монад, является `guard`. Примеры ее применения в реализации априорного алгоритма будут показаны далее, а сейчас давайте познакомимся с ее типом и определением, которое занимает совсем мало места, но последствия его применения весьма существенны:

```
guard :: MonadPlus m => Bool -> m ()  -- для списка — Bool -> [()]
guard True  = return ()               -- для списка — [()]
guard False = mzero                   -- для списка — []
```

Заметьте, что функция реально возвращает некоторое значение, но оно нас не интересует, поэтому функция `guard` никогда не используется для получения ее значения в сочетании с оператором `<-`.

Чтобы понять поведение функции, давайте сконцентрируем внимание на следующем фрагменте кода:

```
do guard condition
   return 1
```

Как было показано в предыдущей главе, этот код транслируется компилятором в форму без do-нотации:

```
guard condition >>= (\_ -> return 1)
```

Забегая на шаг вперед, отметим, что после замены монадических функций специальными определениями для списков код будет читаться следующим образом:

```
concat $ map (\_ -> [1]) (guard condition)
```

Первым возможным случаем является удовлетворение условия. Тогда `guard condition` становится эквивалентом `[()]`. Этапы вычисления в данном случае будут следующими:

```
concat $ map (\_ -> [1]) [()] = concat [[1]] = [1]
```

Однако когда условие получит значение `False`, `guard condition` станет эквивалентом пустого списка. Это означает, что при выполнении операции обхода элементов списка полученный список также окажется пустым. Таким образом, `guard` обеспечивает отсутствие возвращаемого значения при невыполнении условия.

Создание нового класса типов `MonadPlus` и включение его в основной пакет распространения не имело бы смысла, если бы списки были только того типа, для которого существует экземпляр. Но это не так, потому что в него также включен экземпляр для `Maybe`, что очень удобно в использовании. До сих пор мы видели, как блок `do` с несколькими значениями `Maybe` возвращает только `Just`, если все его промежуточные вычисления также вернули значение `Just`. Это пригодилось при написании кода, обращающегося к базе данных и пытающегося получить конечный результат путем объединения различных частей информации: если какое-либо из обращений было неудачным, неудачным оказывался весь процесс. В предыдущей главе был показан следующий код:

```
do n         <- numberItemsByPurchaseId purchaseId
   productId <- productIdByPurchaseId purchaseId
   price     <- priceByProductId productId
   return $ fromInteger n * price
```

Иногда поведение, которое нужно получить, не является рядом обращений, каждое из которых должно быть проведено корректно. Для получения информации наряду с одним основным способом часто приходится пользоваться запасным. Неудача всего процесса может быть только в случае неудачи обоих способов. И опять эта возможность изменения маршрута, когда одна операция `Maybe` терпит неудачу, является примером дизъюнктивного поведения, выявляемого `MonadPlus`. Посмотрим, к примеру, что вернет интерпретатор при моделировании первого неудачного вычисления, при котором используется `Nothing`:

```
*Chapter7.MonadPlus Control.Monad> Nothing `mplus` Just 5
Just 5
```

Код переходит ко второму варианту, результат которого и возвращается. В случае корректности обоих вычислений экземпляр Monad для Maybe, как показано в следующем примере, отдает приоритет тому, что находится слева:

```
*Chapter7.MonadPlus Control.Monad> Just "first" `mplus` Just "second"
Just "first"
```

При чтении приведенного представления MonadPlus вспоминается другой класс типов — Monoid. В некотором смысле функция mzero немного похожа на функцию mempty в Monoid, а функция mplus — на функцию mappend. Однако путать эти два класса типов не следует. Последний из них работает с определенным типом данных, а первый предназначен для применения к тем типам данных, которые допускают переменную типа. Это означает, что конкретный тип данных без переменных типа [Integer] может быть сделан экземпляром Monoid, но не может быть сделан экземпляром MonadPlus. В то же время такой полиморфный тип, как [a], может быть сделан экземпляром MonadPlus, но никогда не может быть сделан экземпляром Monoid. В понятиях видов можно сказать, что класс типов Monoid относится к типам вида *, а MonadPlus — к типам вида * -> *.

Однако вследствие этой связи .Control.Monad предлагает две функции, похожие на моноидальные. Одной из таких функций является msum, заменяемая функцией mconcat, которая развертывает список моноидальных значений с помощью внутренней моноидальной операции до уровня MonadPlus. Повторим еще раз, разница в видах может прослеживаться в типах этих функций: функция mconcat имеет тип Monoid m => [m] -> m, а функция msum типизируется как MonadPlus m => [m a] -> m a. Вот два примера их применения, и вам нужно хорошенько их осмыслить:

```
*Chapter7.MonadPlus Control.Monad> mconcat [[1],[2,3],[],[1,2]]
[1,2,3,1,2]
*Chapter7.MonadPlus Control.Monad> msum [Nothing, Just 1, Just 2, Nothing]
Just 1
```

Еще одной характерной для MonadPlus функцией является mfilter. В данном случае обобщению подверглась функция filter. В частности, mfilter возвращает все значения, удовлетворяющие условию, но объединенные с помощью операции mplus из MonadPlus. Как воспользоваться функцией msum для реализации сходной функциональности, а именно для поиска элемента в списке, можно понять, выполнив упражнение 7.2.

УПРАЖНЕНИЕ 7.2. ВЕДЕНИЕ ПОИСКА С ПОМОЩЬЮ MONADPLUS

Напишите функцию find_, имеющую тип (a -> Bool) -> [a] -> Maybe a, которая возвращает первый элемент списка, удовлетворяющий заданному условию. Сделайте это с помощью функции msum, представленной в данном разделе, и поведения экземпляра Maybe класса MonadPlus.

Вывод ассоциативных правил

В предыдущей главе основное внимание уделялось обнаружению схожих видов клиентов с целью развертывания более эффективных рекламных кампаний, на-

целенных на одну или несколько из выявленных групп. Теперь цель заключается в анализе данных, чтобы предложить покупателям новый товар при их путешествии по интернет-магазину. Одним из простых вариантов решения этой задачи является использование ассоциативных правил.

При выводе ассоциативных правил выдвигается предположение, что данные состоят из серий *транзакций* (transactions), каждая из которых представляет собой множество *элементов* (items). Считается, что любая транзакция поступает из строки или из множества связанных строк обычной базы данных. Например, можно начать со следующей части информации (компания Wormhole Inc. купила путеводитель с идентификатором 3):

```
the company Wormhole Inc. bought the travel guide with identifier 3
```

Из этой части информации можно получить такую транзакцию:

```
{ company, name "Wormhole Inc.", purchased travel guide, purchased product 3 }
```

Эти элементы генерируются из множества возможных элементов. Поскольку информация обычно представляется в Haskell в виде дерева значений, сначала нужно преобразовать такие значения, получив одноуровневое представление, составленное из транзакций.

Наша цель заключается в выводе из этих транзакций *ассоциативных правил*. Эти правила образуются двумя множествами элементов, предпосылкой (antecedent) `A` и следствием (consequent) `C`, и представляются формулой `A => C`. Значение такого правила состоит в том, что зачастую, когда транзакция содержит элементы в `A`, она также содержит элементы в `C`. Вот возможный пример:

```
{ individual, purchased time machine } => { purchased travel guide }
```

В нем сообщается, что обычно, когда частное лицо покупает машину времени, оно также покупает и путеводитель, поэтому в отношении доступных путеводителей можно выстроить верное предположение.

Чтобы завершить это вступление, введем две единицы измерения, связанные с множеством элементов и правил. *Поддержка* (support) множества элементов определяется как доля транзакций, содержащих упомянутое множество элементов. Например, если ваши транзакции имеют вид `[{a, b}, {a, c}, {b, c}]`, поддержка `{b}` вычисляется путем деления количества транзакций, в которых содержится `b` (в данном случае — 2), на общее количество транзакций (в данном случае — 3). Следовательно, *поддержка* (`{b}`) = 2/3 = 0,66.

Достоверность (confidence) правила `A => C` определяется как *поддержка* (`A` и `C`) / *поддержка* (`A`). То есть эта единица измерения определяется как доля транзакций, поддерживающих правило, путем вычисления отношения транзакций, отвечающих всему правилу, к транзакциям, соответствующим предпосылке.

Выравнивание значений в транзакции

Как объяснялось ранее, ассоциативные правила можно вывести для выровненных (плоских) множеств элементов, но не для сложных значений. В данном разделе

представлены функции, призванные превращать вторые в первые. Прежде всего давайте вспомним определение клиентов и введем типы данных для товаров:

```
-- Клиенты
data Client = GovOrg     { clientName :: String }
            | Company    { clientName :: String, person :: Person, duty :: String }
            | Individual { person :: Person }
            deriving (Show, Eq, Ord)
data ClientKind = KindGovOrg | KindCompany | KindIndividual deriving (Show, Eq,
Ord)
data Person = Person { firstName :: String, lastName :: String, gender :: Gender }
            deriving (Show, Eq, Ord)
data Gender = Male | Female | UnknownGender deriving (Show, Eq, Ord)

-- Товары
data Product = Product { productId :: Integer, productType :: ProductType }
             deriving (Show, Eq, Ord)
data ProductType = TimeMachine | TravelGuide | Tool | Trip deriving (Show, Eq, Ord)
```

Тогда покупка в базе данных является простым объединением клиента и множества товаров:

```
data Purchase = Purchase { client :: Client, products :: [Product] } deriving
(Show, Eq, Ord)
```

Элементы в транзакции отражают полезную информацию о покупке. Это, в частности, вид и идентификатор приобретенного товара, вид клиента (правительственная организация, компания или индивидуальный покупатель). В случае с компаниями также представляет интерес должность лица, ответственного за покупку, а в случае с индивидуальным покупателем — его пол, если данная информация доступна.

```
data PurchaseInfo = InfoClientKind           ClientKind
                  | InfoClientDuty           String
                  | InfoClientGender         Gender
                  | InfoPurchasedProduct     Integer
                  | InfoPurchasedProductType ProductType
                  deriving (Show, Eq, Ord)
```

Чтобы сделать код понятнее и безопаснее с точки зрения типов, давайте с помощью `newtype` создадим новый тип, пометив эти множества, используемые как транзакции:

```
import Data.Set (Set)
import qualified Data.Set as S

newtype Transaction = Transaction (Set PurchaseInfo) deriving (Eq, Ord)
```

В качестве примера рассмотрим код преобразования информации о товаре и покупке в реальную транзакцию. Убедитесь в том, что основная часть работы представляет собой простое перемещение некоторых полей в значения типа `PurchaseInfo`. В упражнении 7.3 вам потребуется написать функцию `clientToPurchaseInfo`, которой нет в определении `purchaseToTransaction`.

```
productsToPurchaseInfo :: [Product] -> Set PurchaseInfo
productsToPurchaseInfo = foldr
  (\(Product i t) pinfos -> S.insert (InfoPurchasedProduct i) $
```

```
                          S.insert (InfoPurchasedProductType t) pinfos)
  S.empty

purchaseToTransaction :: Purchase -> Transaction
purchaseToTransaction (Purchase c p) =
  Transaction $ clientToPurchaseInfo c `S.union` productsToPurchaseInfo p
```

УПРАЖНЕНИЕ 7.3. ПРЕОБРАЗОВАНИЕ ИНФОРМАЦИИ О КЛИЕНТАХ В ЭЛЕМЕНТЫ

Напишите отсутствующую функцию clientToPurchaseInfo. У нее должен быть тип Client -> Set PurchaseInfo, и она должна отвечать требованиям по получению элементов из информации о клиентах (см. определение типа данных PurchaseInfo). Вот как выглядит пример, показывающий возвращаемое значение для тестирования:

```
*> clientToPurchaseInfo (Company "1984 Inc." (Person "George" "Orwell" Male)
"Director")
fromList [InfoClientKind KindCompany,InfoClientDuty "Director"]
```

Как видите, из простого типа Client должно быть получено все множество элементов.

Априорный алгоритм

Одним из простейших алгоритмов вывода ассоциативных правил является априорный алгоритм, предложенный в 1994 году Аргавалом (Agrawal) и Срикантом (Srikant). Априорный алгоритм работает в два этапа:

- Сначала он генерирует множества элементов, отвечающие условиям минимальной поддержки, — они называются *часто встречающимися множествами* (frequent sets).
- Затем из этих множеств он выводит ассоциативные правила с минимальной достоверностью.

Результатом становится множество ассоциативных правил, о которых известно, что они удовлетворяют некоему минимальному значению по обоим измерениям. Путем изменения таких значений можно экспериментировать, увеличивая или уменьшая количество правил. Создание слишком большого количества правил может привести к ненужной избыточности: правила справедливы для конкретного множества данных, их нельзя распространить и применить к новым сценариям. Таким образом, создание меньшего количества правил повышает достоверность при их дальнейшем применении.

Способ выполнения первого этапа выводится из наблюдения: если имеется множество элементов I с поддержкой s, то любое подмножество I будет иметь поддержку, равную или более высокую, чем s (поскольку все транзакции, составляющие I, имеют такую поддержку и будут учитываться для любого из своих подмножеств). Это означает, что подмножества с n элементами можно многократно использовать для создания более крупных множеств с $n+1$ элементами путем отбора тех из них, у которых имеется пересечение в $n-1$ элементов, и их объединения с последующей проверкой, реализуется ли ими требуемая поддержка. Из этого ясно, что нет смысла объединять множества элементов, имеющих поддержку ниже определенного минимума, поскольку объединение никогда не будет удовлетворять условию.

ПРИМЕЧАНИЕ

Для того чтобы генерировать из имеющихся множеств еще меньшее количество множеств элементов, уменьшая тем самым количество проверок, в реальном алгоритме реализуется ряд весьма интересных трюков.

Давайте введем тип данных для ассоциативных правил и множества элементов этого этапа. Если помните, они называются «часто встречающимися множествами». Рассмотрим следующий код:

```
newtype FrequentSet = FrequentSet (Set PurchaseInfo) deriving (Eq, Ord)

data AssocRule = AssocRule (Set PurchaseInfo) (Set PurchaseInfo) deriving (Eq, Ord)
instance Show AssocRule where
  show (AssocRule a b) = show a ++ " => " ++ show b
```

Наряду с определениями типов данных вам понадобятся вспомогательные функции, вычисляющие достоверность и степень поддержки соответственно для наиболее часто встречающихся множеств и для ассоциативных правил. Можно заметить, что этот код не обеспечивает высокую производительность, поскольку для списков в нем неоднократно вызывается функция вычисления длины. Обдумывание мер по повышению его быстродействия может стать хорошим упражнением.

```
setSupport :: [Transaction] -> FrequentSet -> Double
setSupport trans (FrequentSet sElts) =
  let total = length trans
      supp  = length (filter (\(Transaction tElts) -> sElts `S.isSubsetOf` tElts)
trans)
   in fromIntegral supp / fromIntegral total

ruleConfidence :: [Transaction] -> AssocRule -> Double
ruleConfidence trans (AssocRule a b) =
  setSupport trans (FrequentSet $ a `S.union` b) / setSupport trans (FrequentSet a)
```

Сначала нужно сгенерировать исходные множества из одного часто встречающегося элемента. В следующем примере кода вместо списочных функций или формирователей списков в качестве монады использован список. И пусть вас не смущает перевод из множества в список с помощью `S.toList`. Также пусть вас не смущает заключение единственного множества в оболочку `FrequentSet`. Это всего лишь шаблон, и он не является в этом коде главным:

```
import Control.Monad

generateL1 :: Double -> [Transaction] -> [FrequentSet]
generateL1 minSupport transactions =
  noDups $ do Transaction t <- transactions
              e <- S.toList t
              let fs = FrequentSet $ S.singleton e
              guard $ setSupport transactions fs > minSupport
              return fs

-- noDups удаляет в списке дубликаты
noDups :: Ord a => [a] -> [a]
noDups = S.toList . S.fromList
```

Для нас важно, как в коде используется оператор <- из do-нотации, — это похоже на синтаксис формирователей списков (что, как вы увидите далее, совсем не случайно), а также функция guard, с которой мы познакомились в предыдущем разделе.

В этом коде используется также специальное определение функции noDups, удаляющей дубликаты. В главе 4 уже рассматривалась имеющаяся для этой цели в Prelude базовая функция nub, которая работает довольно медленно, а также отмечалось, что преобразование в множество и обратно в список является более производительным способом решения задачи.

Следующий шаг в алгоритме касается способа генерирования часто встречающихся множеств уровня k+1 из часто встречающихся множеств уровня k. Как уже ранее отмечалось, это может быть сделано в два этапа: сначала путем проверки, использует ли совместно каждая пара множеств уровня k элементы k–1, а затем путем проверки, будет ли объединенное множество удовлетворять минимальным условиям поддержки. Перед тем как рассматривать код, давайте прикинем, что мы собираемся сделать с помощью этой функции: нам нужно создать все больше и больше уровней из предыдущих, пока не будет найдено пустое часто встречающееся множество. Нам уже знаком такой образец генерирования элементов до достижения определенного условия — это «развертка».

Поскольку нам уже известно, что для генерирования списка нужно воспользоваться функцией unfoldr, настало время подключить информацию, характерную для данного случая. Как вы знаете, функция unfoldr задействует некое начальное значение, которое при создании новых элементов изменяется в программном потоке. Какие части информации будут содержаться в этом значении? Вам нужны как минимум часто встречающиеся множества предыдущего уровня. Но чтобы иметь возможность узнать, используют ли совместно два часто встречающихся множества k–1 элементов, также будет полезно пройти и тот уровень, с которым вы работаете в данный момент (в ином случае вам придется его повторно рассчитать из количества элементов в множествах). Следовательно, начальное значение будет иметь тип (Int, [FrequentSet]).

На каждом уровне алгоритм будет производить список часто встречающихся множеств, поэтому тип элемента в unfoldr должен быть [FrequentSet]. Из-за этого возникает проблема: в результате развертки вместо простого списка [FrequentSet] получится тип [[FrequenSet]]. Самый простой способ решения данной проблемы заключается во включении вызова функции concat, которая произведет выравнивание списка до одноуровневого. В заключение определяется новая функция generateNextLk (см. далее). Обратите внимание, как для удаления из списка часто встречающихся множеств, не отвечающих условиям, используется функция guard:

```
generateNextLk :: Double -> [Transaction] -> (Int, [FrequentSet])
                         -> Maybe ([FrequentSet], (Int, [FrequentSet]))
generateNextLk _ _ (_, []) = Nothing
generateNextLk minSupport transactions (k, lk) =
  let lk1 = noDups $ do FrequentSet a <- lk
                        FrequentSet b <- lk
                        guard $ S.size (a `S.intersection` b) == k - 1
```

```
                        let fs = FrequentSet $ a `S.union` b
                        guard $ setSupport transactions fs > minSupport
                        return fs
  in Just (lk1, (k+1, lk1))
```

На данный момент у нас уже есть все ингредиенты для генерирования часто встречающихся множеств всех уровней. Нужны только лишь повторяющиеся вызовы функции `generateNextLk`, осуществляющей обработку предыдущих часто встречающихся множеств и номера уровня. С этим отлично справится функция `unfoldr`. Таким образом, если предположить, что `l1` — это исходное часто встречающееся множество, возвращенное функцией `generateL1`, то возможным средством генерирования часто встречающихся множеств является следующее выражение (оно станет частью полноценной реализации априорного алгоритма):

```
concat $ unfoldr (generateNextLk minSupport transactions) (1, l1)
```

Завершающим шагом является генерирование ассоциативных правил с минимальной достоверностью. На данный момент априорный алгоритм работает следующим образом: берется часто встречающееся множество `I` из предыдущих шагов. Затем множество разбивается на два изолированных множества `A` и `B` таким образом, чтобы множество `A` не было пустым. Каждое разделение будет приводить к правилу `A => B`, которое должно быть проверено на минимальную достоверность. Это должно быть сделано для каждого возможного разделения каждого часто встречающегося множества, которое должно быть получено с помощью функции `powerset`, — для этого отлично подойдет списочная монада. Данный Haskell-код является прямым переводом этого описания:

```
generateAssocRules :: Double -> [Transaction] -> [FrequentSet] -> [AssocRule]
generateAssocRules minConfidence transactions sets =
  do FrequentSet fs <- sets
     subset@(_:_) <- powerset $ S.toList fs
     let ssubset = S.fromList subset
         rule = AssocRule ssubset (fs `S.difference` ssubset)
     guard $ ruleConfidence transactions rule > minConfidence
     return rule

powerset :: [a] -> [[a]]
powerset [] = [[]]
powerset (x:xs) = powerset xs ++ map (x:) (powerset xs)
```

Осталось только склеить все эти функции в работоспособную функцию `apriori`, которая будет получать множество данных наряду со значениями минимальной поддержки и достоверности и возвращать ассоциативные правила, отвечающие этим требованиям. Мы реализовали прямой стиль передачи результата каждой функции следующей функции, но для каждого этапа в коде можно было бы использовать оператор `let` или бесточечную форму.

```
apriori :: Double -> Double -> [Transaction] -> [AssocRule]
apriori minSupport minConfidence transactions =
  generateAssocRules minConfidence transactions
    $ concat $ unfoldr (generateNextLk minSupport transactions)
                       (1, generateL1 minSupport transactions)
```

Задачи поиска

Многие задачи в компьютерной науке могут решаться значительно лучше, если на них смотреть как на *задачи поиска* (search problems). В задаче поиска неизвестно, как прийти к ответу напрямую, но есть возможность исследовать пространство возможных решений, пока не будет найден правильный ответ. Обычно способ исследования пространства напоминает обход дерева: из возможного решения можно создать множество других возможных решений, которые подвергаются проверке. Таким образом, поиск нужного значения похож на перемещение по дереву.

Пути в графе

Графы могут стать хорошим подспорьем для решения задач с использованием данного подхода, и в этом разделе представлен конкретный пример. В предыдущих главах мы познакомились с идеей машин времени, на которых можно путешествовать только в определенную точку. Задача, которую нужно теперь решить, заключается в поиске всех возможных вариантов перемещений между двумя годами. Как и в предыдущих примерах, модель будет представлена графом, заданным списком ребер (`start`, `end`). Предположим также, что здесь отсутствуют петли (то есть нет ребер (`y,y`) с одним и тем же годом `y` в обоих компонентах). Сначала давайте подумаем о сигнатуре функции. Нужно искать список путей, каждый из которых представлен списком лет между узлом `start` и узлом `end`. То есть нам нужны граф и два узла, а результатом будет список списков узлов:

```
paths :: [(Int,Int)] -> Int -> Int -> [[Int]]
paths edges start end = undefined -- сюда помещается код
```

Следует помнить, что в реальной жизни можно будет воспользоваться более производительным модулем `Data.Graph`, который был представлен в главе 4. Однако для упрощения при моделировании графа в данном разделе мы задействуем простой список пар.

ПРИМЕЧАНИЕ

Функция undefined модуля Prelude зачастую используется как местозаполнитель для еще не написанного кода. Попытка выполнения этой функции приведет к ошибке, но она весьма удобна для разработки общей структуры кода, когда с помощью функции undefined временно заполняются пустоты, которые позже будут заполнены нужным кодом.

Как и в любом другом алгоритме поиска, здесь есть несколько прорабатываемых вариантов. В общем случае нужно взять все узлы (`o,t`), чьим началом является ребро `start` (то есть `o` совпадает с аргументом `start`), и найти пути между целью этого ребра и тем годом, в котором нужно закончить маршрут. Эту логику нетрудно закодировать путем объединения списочной монады и функции `guard`. Как можно видеть, следующий код является практически дословной транслитерацией этой логики:

```
import Control.Monad

paths :: [(Int,Int)] -> Int -> Int -> [[Int]]
```

```
paths edges start end =
  do (e_start, e_end) <- edges
     guard $ e_start == start
     subpath <- paths edges e_end end
     return $ start:subpath
```

Однако этот код еще нельзя признать пригодным, поскольку нам нужно справиться со случаем, когда мы достигаем целевого года нашего путешествия. В этом случае к этим путям нужно добавить еще один путь, состоящий всего из одного года — того, который одновременно является и начальным (ребро `start`), и конечным (ребро `end`). Для этого можно было бы воспользоваться конструктором `(:)`, чтобы добавить элемент в начало `e_paths`. Однако в следующем коде мы отдаем предпочтение функциям `return` и `mplus`, чтобы посмотреть, как в этом сценарии задействовать функцию `MonadPlus`:

```
paths :: [(Int,Int)] -> Int -> Int -> [[Int]]
paths edges start end =
  let e_paths = do (e_start, e_end) <- edges
                   guard $ e_start == start
                   subpath <- paths edges e_end end
                   return $ start:subpath
   in if start == end
         then return [end] `mplus` e_paths
         else e_paths
```

Предположим, у нас есть граф, подобный следующему:

```
graph1 :: [(Int, Int)]
graph1 = [(2013,501),(2013,1004),(501,2558),(1004,2558)]
```

Если взять этот граф, то в GHCi легко можно получить все пути от 2013 до 2558 года:

```
*Chapter7.Graph> paths graph1 2013 2558
[[2013,501,2558],[2013,1004,2558]]
```

Монада Logic

Если исследовать пространство поиска простейшим способом, циклы могут стать проблемой. Возьмем, к примеру, другой граф, где присутствует цикл между годами 501 и 1004:

```
graph2 :: [(Int, Int)]
graph2 = [(2013,501),(501,2558),(501,1004),(1004,501),(2013,2558)]
```

Список путей между 2013 и 2558 годом является бесконечным (между годами 501 и 1004 всегда найдется еще один цикл). Но как показывает следующий пример, получить конечное множество путей все же возможно:

```
*Chapter7.Graph> take 3 $ paths graph2 2013 2558
[[2013,501,2558],[2013,501,1004,501,2558],[2013,501,1004,501,1004,501,2558]]
```

Проблема здесь состоит в том, что путь `[2013,2558]` никогда не будет найден, потому что поисковой стратегии, воплощенной в списочной монаде, сначала пона-

добится найти все подпути, проходящие через 501 год (поскольку он был прежде найден в списке ребер). Но список таких подпутей бесконечен, поэтому вычисление никогда не достигнет точки, в которой рассматриваются ребра (2013,2558).

Для получения более качественного контроля над поиском введем монаду Logic, находящуюся в модуле Control.Monad.Logic Hackage-пакета logict. По умолчанию эта монада ведет себя как список, поэтому подойдет почти весь код из предыдущего примера, придется только изменить сигнатуру типа функции:

```
import Control.Monad.Logic

pathsL :: [(Int,Int)] -> Int -> Int -> Logic [Int]
pathsL edges start end =
  let e_paths = do (e_start, e_end) <- choices edges
                   guard $ e_start == start
                   subpath <- pathsL edges e_end end
                   return $ start:subpath
   in if start == end then return [end] `mplus` e_paths else e_paths

choices :: [a] -> Logic a
choices = msum . map return
```

Единственным важным добавлением является вызов вновь определенной функции choices[1]. Эта функция позволяет при вычислениях в рамках монады Logic превратить список в множество благоприятных исходов. Это делается путем возвращения каждого элемента в списке (через map return) с последующим объединением этих удачных случаев вместе с помощью функции msum.

На этом этапе определение, использующее монаду Logic, страдает от той же проблемы, что и определение, реализованное посредством списков. Если взять первые три решения с помощью функции observeMany из модуля Control.Monad.List, можно наблюдать такое же поведение с проходом сначала всех путей, следующих через год 501:

```
*Chapter7.Graph> import Control.Monad.Logic
*Chapter7.Graph Control.Monad.Logic> observeMany 3 $ pathsL graph2 2013 2558
[[2013,501,2558],[2013,501,1004,501,2558],[2013,501,1004,501,1004,501,2558]]
```

Для решения проблемы в Logic вводится множество операторов, являющихся справедливыми: то есть множество операторов, не отдающих в вычислениях приоритета никакой из возможных ветвей. Вы можете увидеть разницу между обычной функцией объединения результатов mplus, которая сначала берет все, что исходит из первого аргумента, а затем все, что исходит из второго аргумента, и функцией interleave, которая, когда ей нужен еще один элемент, берет по одному элементу из каждого списка:

```
*Chapter7.Graph Control.Monad.Logic> [1,2] `mplus` [3,4]
[1,2,3,4]
*Chapter7.Graph Control.Monad.Logic> [1,2] `interleave` [3,4]
[1,3,2,4]
```

[1] Определение функции choices взято из статьи «Adventures in Three Monads» Эдварда Янга (Edward Z. Yang).

Есть также справедливая замена оператору (>>=), имеющая вид (>>-). Однако поскольку у оператора привязки есть свои особенности, с ним больше нельзя использовать do-нотацию. Соответственно нам нужно очистить функцию pathsL, записав ее с помощью монадических примитивов, как требуется в упражнении 7.4.

УПРАЖНЕНИЕ 7.4. ОЧИЩЕНИЕ С ИСПОЛЬЗОВАНИЕМ МОНАДИЧЕСКОЙ НОТАЦИИ

Базируясь на правилах, рассмотренных в главе 6, перепишите функцию pathsL, используя вместо do-нотации функции (>>=) и return. Помните, что guard является обычной функцией без специального синтаксиса, поэтому в отличие от любых других функций с ней ничего делать не нужно.

Замена оператора (>>=) оператором (>>-) дает следующую версию алгоритма поиска путей:

```
pathsLFair :: [(Int,Int)] -> Int -> Int -> Logic [Int]
pathsLFair edges start end =
  let e_paths = choices edges >>- \(e_start, e_end) ->
                guard (e_start == start) >>
                pathsLFair edges e_end end >>- \subpath ->
                return $ start:subpath
   in if start == end then return [end] `interleave` e_paths else e_paths
```

Теперь проблема устранена: ребра обходятся в справедливом порядке, и в результате мы получаем путь [2013,2558]:

```
*Chapter7.Graph Control.Monad.Logic> observeMany 3 $ pathsLFair graph2 2013 2558
[[2013,501,2558],[2013,2558],[2013,501,1004,501,2558]]
```

Данный пример дает лишь поверхностную картину возможностей пакета logict. Этот пакет предлагает намного больше комбинаторов, позволяющих программисту создавать код в стиле логического программирования с возвратами, переходами, условиями и т. п., поэтому для решения других задач поиска стоит заглянуть в документацию этого пакета. Монада Logic является также примером того, как монады могут использоваться для внедрения в язык Haskell вычислительных парадигм других языков без утраты характерных особенностей языка Haskell.

Возрождение монад и списков

До сих пор мы знакомились с примерами списков, рассматриваемых в качестве монад, которые использовались в модели возвращения нескольких значений. В данном разделе мы рассмотрим два других подхода, в которых монады могут иметь отношение к спискам: сначала мы поговорим о функциях, похожих на обычные списочные функции (например, проекции и свертки), но работающие в монадической среде, а затем узнаем, как синтаксис формирователей списков может быть расширен для использования с любой монадой.

Объединение значений в монаде

Иногда при вычислении всех элементов списка расчеты требуется выполнять не чистой функцией, а внутри монады. Для таких случаев в модуле Control.Monad

содержится ряд функций с окончанием M, которые предлагают ту же функциональность, что и обычные списочные функции. В данном разделе мы получим высокоуровневое представление о таких функциях и познакомимся с примерами их применения.

Первой из них будет функция mapM, имеющая тип Monad m => (a -> m b) -> [a] -> m [b]. Функция mapM применяет монадическую функцию к каждому элементу списка, выстраивая в цепочку обработку каждого элемента внутри монады. Скажем, к примеру, что у нас есть очень простая функция, ставящая одну строку впереди другой. Однако на месте дополнительного аргумента мы хотим использовать некий глобальный контекст, смоделированный с помощью монады Reader:

```
addPrefix :: String -> Reader String String
addPrefix s = ask >>= \p -> return $  p ++ s
```

Теперь при наличии списка строк, к которым мы собираемся добавить префикс из первой строки, мы могли бы интуитивно воспользоваться функцией map. Но map не решит задачу. Давайте попробуем определить причину этого, посмотрев на тип выражения, которое мы собираемся выполнить:

```
*Chapter7.UnderAMonad> :t map addPrefix
map addPrefix :: [String] -> [Reader String String]
```

Функция map не является той функцией, которая добавит префикс ко всем строкам в списке. Точнее, возвращаемое значение будет списком вычислений, причем каждому вычислению нужен контекст, в котором оно выполняется. Если же использовать функцию mapM, все эти вычисления будут выполняться в той же самой монаде Reader, а значит, иметь общий контекст. Получается, что функция mapM является именно той функцией, которая нам нужна:

```
addPrefixL :: [String] -> Reader String [String]
addPrefixL = mapM addPrefix
```

Проверить, что функция работает так, как ожидалось, можно, протестировав ее в интерпретаторе:

```
*Chapter7.UnderAMonad> runReader (addPrefixL ["one","two"]) "**-"
["**-one","**-two"]
```

Иногда нас могут интересовать не результаты вычислений, а некие другие эффекты, привносимые функциями в элементы списка. Скажем, к примеру, нужно запротоколировать серии данных с помощью монады Writer. В таком случае не важно, что будет возвращено, а важен факт того, что строки добавляются в журнал вычислений. Для таких случаев есть версия функции mapM, которая называется mapM_ и не возвращает значения. Вот как могла бы выглядеть использующая ее функция протоколирования:

```
logInformation :: [String] -> Writer String ()
logInformation = mapM_ (\s -> tell (s ++ "\n"))
```

Здесь для проверки ожидаемого характера работы функции тоже можно воспользоваться интерпретатором:

```
*Chapter7.UnderAMonad> runWriter $ logInformation ["one","two"]
((),"one\ntwo\n")
```

Модуль `Control.Monad` предоставляет также функции `forM` и `forM_`, которые, по сути, являются теми же функциями `mapM` и `mapM_`, но с аргументами, следующими в обратном порядке. То есть список идет первым, а затем следует функция. Используя функцию `form`, можно написать код, во многом похожий на цикл for-each в Java или C#. Предыдущая функция может быть переписана следующим образом:

```
logInformation infos = forM_ infos $ \s ->
                         tell (s ++ "\n")
```

Функции `mapM` и `forM` на самом деле являются специализированными вариантами функции более общего назначения `sequence`, которая выполняет все действия в контексте тех же монад. Тип `sequence` выглядит следующим образом: `Monad m => [m a] -> m [a]`. Советы для написания определений `sequence` и `mapM` можно найти в упражнении 7.5.

УПРАЖНЕНИЕ 7.5. ОПРЕДЕЛЕНИЕ ФУНКЦИЙ SEQUENCE И MAPM

Попробуйте написать определение функции sequence. Сделайте это в do-нотации и используйте сопоставление с образцом. Как и в ситуации с любыми другими списочными функциями, вам нужно рассмотреть случаи пустого списка и списка с головой и хвостом. Следует помнить, что x <- v «извлекает» значение, заключенное в монаде, из v :: m a в привязку x :: a.

Для функции mapM можно подсказать, что ее следует написать в виде сочетания двух других функций. Одна из функций, которую нужно использовать, уже была продемонстрирована на примере добавления общего префикса из совместно используемого контекста к списку строк.

Следует знать, что поведение `sequence` (и, следовательно, `mapM` и `forM`) большей частью определяется той монадой, в которой происходят вычисления. Например, если устанавливаемая последовательность является списком значений `Maybe`, поведение заключается в возвращении значения `Just`, если только все элементы в списке являются значениями `Just`, и значения `Nothing`, если какой-нибудь из элементов в списке имеет значение `Nothing`. Такое поведение проистекает из способа работы функции `(>>=)` для `Maybe`.

Таким же образом модуль, предоставляя «обновленные» проекции, предоставляет также свертки, работающие под общей оболочкой монады. Функцией, решающей эту задачу, является `foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a`. Обратите внимание на отсутствие правой и левой версий свертки: `foldM` всегда выполняет левую свертку (если нужно выполнить правую свертку, перед применением функции достаточно изменить порядок следования элементов списка на обратный). И опять инструмент протоколирования предоставляет нам пример, в котором могла бы использоваться эта функция. В данном случае мы познакомимся с версией факториала, который кроме вычисления результата также заботится о количестве этапов свертки, необходимых для вычислений. В коде вместо do-нотации используется функция `(>>)`, поскольку в данном случае это позволяет получить более лаконичный код:

```
factorialSteps :: Integer -> Writer (Sum Integer) Integer
factorialSteps n = foldM (\f x -> tell (Sum 1) >> return (f*x)) 1 [1 .. n]
```

И последней функцией на нашем маршруте является функция `filterM`, которая отфильтровывает некоторые значения в списке на основе монадического предиката. В Haskell-сообществе распространено использование функции `filterM` и списочной

монады для предоставления очень краткого определения функции, возвращающей совокупность всех подмножеств списка:

```
powerset :: [a] -> [[a]]
powerset = filterM (\_ -> [False,True])
```

Эта функция работает благодаря недетерминированной природе вычислений в списочной монаде. В качестве признака необходимости включения или исключения элемента фильтруемого списка функция `powerset` использует не значения `True` или `False`, а выражение, которое возвращает оба варианта (это можно заметить, вместо `[False,True]` написав `return False `mplus` return True`). Варианты для каждого из элементов создают множество возможных состояний — по одному для каждого подмножества исходного списка.

Помимо `mapM`, `forM`, `sequence`, `foldM` и `filterM` есть и другие функции, например `zipWithM` и `replicateM`, которые ведут себя так же, как их аналоги из разряда списочных функций. Ознакомьтесь с документацией по этим функциям, находящейся в хранилище Hackage.

ФУНКЦИИ LIFTM И AP

В предыдущей главе мы видели, что в каждой монаде есть функция liftM, имеющая тип Monad m => (a -> b) -> m a -> m b. Эта функция позволяет превратить любую обычную функцию в функцию, работающую в монаде (что обычно называется подтягиванием). Как мы знаем, это делает из каждой монады функтор. Все получается неплохо, пока у подтягиваемой функции не окажется более одного аргумента. Для такого случая модуль Control.Monad предлагает функции liftM2, liftM3 и т. д. Эти функции конвертируют в монадическую форму функции с двумя, тремя или большим количеством аргументов.

Однако похоже, что семейство функций, по одной на каждое количество аргументов, не слишком согласуется с присущей Haskell-коду элегантностью. Это позволяет нам ожидать решения, способного работать с любым количеством аргументов.

И такое решение существует! Оно называется функцией ap и имеет тип Monad m => m (a -> b) -> m a -> m b. Это небольшое изменение в сигнатуре позволяет составлять цепочку. Скажем, к примеру, что нам нужно подтянуть функцию compare. Сначала вся функция заключается в оболочку монады для соответствия типу первого аргумента. Это делается следующим образом:

```
Prelude> :t return compare
return compare :: (Monad m, Ord a) => m (a -> a -> Ordering)
```

Теперь для передачи первого аргумента мы используем функцию ap. Затем мы возвращаем еще одну функцию, ожидая на один аргумент меньше. При наличии монад о функции ap можно думать как о замене функции ($). Итак, если x имеет тип m a, то:

```
return compare `ap` x :: (Monad m, Ord a) => m (a -> Ordering)
```

И наконец, можно опять использовать функцию ap для передачи последнего аргумента и получения конечного результата.

Как правило, любой вызов в форме liftMn f x1 x2 ... xn можно заменить возвращением f `ap` x1 `ap` x2 `ap` ... `ap` xn. Возможность применения такого приема сыграет весомую роль еще в одном важном типовом Haskell-классе, который называется Applicative и будет представлен в главе 10.

Формирователи монад

Схожесть синтаксиса формирователей списков и do-нотации не случайна. Действительно, в давние времена становления Haskell, в далеком 1992 году, Филип

Уодлер (Philip Wadler) предложил общий синтаксис, подходящий для любой монады. Тем не менее решено было сохранить отдельную do-нотацию для произвольных монад, а синтаксис формирователей использовать только для списков. Однако в 2011 году в компилятор GHC был введен переработанный механизм формирователей монад (monad comprehensions), который включается с помощью расширения `MonadComprehensions`.

В синтаксисе формирователей монад лучше всего разобраться, сравнив его с do-нотацией, которая нам уже хорошо знакома. Первое правило перевода гласит, что генераторы в синтаксисе формирователей соответствуют привязке в do-нотации. Или, проще говоря, `e <- m` сохраняется как есть в обоих вариантах синтаксиса. Выражение в начале формирователя преобразуется в вызов функции `return` в do-нотации. Используя эти два правила, можно преобразовать пример с `Maybe` из предыдущей главы:

```
purchaseValueWithDo :: Integer -> Maybe Double
purchaseValueWithDo purchaseId = do n  <- numberItemsByPurchaseId purchaseId
                                    productId <- productIdByPurchaseId purchaseId
                                    price     <- priceByProductId productId
                                    return $ fromInteger n * price
```

Этот пример в синтаксисе формирователей имеет следующий вид:

```
{-# LANGUAGE MonadComprehensions #-}

purchaseValueWithDo :: Integer -> Maybe Double
purchaseValueWithDo purchaseId =
  [ fromInteger n * price | n         <- numberItemsByPurchaseId purchaseId
                          , productId <- productIdByPurchaseId purchaseId
                          , price     <- priceByProductId productId ]
```

В то же время мы уже видели, что формирователи имеют более богатый синтаксис, чем простые генераторы. Формирователи монад могут также использоваться с этим другим синтаксисом, но во многих случаях для этого от монады потребуется реализация более строгого класса типов, чем простой класс типов `Monad`. Одним из примеров могут служить предохранители: они преобразуются в вызовы функции `guard`, следовательно, выражение должно быть заключено в экземпляр `MonadPlus`. Например:

```
[ price | price <- priceByProductId productId, price > 5.0 ]
```

Этот фрагмент должен быть трансформирован следующим образом:

```
do price <- priceByProductId productId
   guard price > 5.0
   return price
```

Групповые и параллельные формирователи также требуют монаду, которая создает экземпляры классов типов `MonadGroup` (предлагая `mgroupWith`) и `MonadZip` (предлагая `mzip`) соответственно. Но эти экземпляры предоставляются в Haskell Platform только для списочных монад, а пакеты сторонних разработчиков реализуют такие классы типов только для типов данных, которые напоминают подобную структуру (такую как таблицы баз данных), поэтому вдаваться в подробности мы пока не будем.

Одним из сложных для усвоения аспектов, касающихся формирователей списков, являются преобразования. Обычно такие выражения, как `[ g x1 ... xn | x1 <- m1, ..., xn <- mn, then f by e ]`, преобразуются в:

```
do (x1, ..., xn) <- g (\(x1, ..., xn) -> e)
                      (do x1 <- m1
                          ...
                          xn <- mn
                          return (x1, ..., xn))
   return g x1 ... xn
```

Этот код довольно замысловат, но он может помочь понять механизм выполнения сложных выражений. Например, в главе 3 можно найти следующее выражение:

```
[x*y | x <- [-1,1,-2], y <- [1,2,3], then sortWith by x]
```

Используя правила преобразования, можно получить эквивалент этого выражения:

```
do (x, y) <- sortWith (\(x, y) -> x)
                      (do x <- [-1,1,2], y <- [1,2,3], return (x,y))
   return $ x*y
```

Эти преобразования (даже при том, что по внешнему виду они являются просто другим способом выражения тех же самых вещей и в силу этого повторяются) способны помочь пролить свет на то, чем на самом деле являются монады, а также на их связь с другими вычислительными структурами. Например:

```
for (x <- List(-1,1,2) ; y <- List(1,2,3)  if x < y) yield x*y
```

Если вам известен язык Scala, то вы должны быть в курсе, что такое выражение, как это, транслируется компилятором в следующий код (нужно учитывать, что в Scala используется объектно-ориентированная нотация, поэтому функции `map` и `flatMap` записываются как методы списков, а безымянные функции вводятся с помощью оператора `=>`, а не `->`):

```
List(-1,1,2).flatMap(x => List(1,2,3).withFilter(y => x < y).map(y => x*y))
```

Из всего этого можно понять, что идея здесь та же, что и в формирователях монад в Haskell. Основное различие языков состоит в том, что в Scala решено было моделировать монадоподобные структуры с помощью функции `flatMap`, что является эквивалентом объединяющего комбинатора, представленного в начале этой главы. Но как мы видели ранее, оба определения эквивалентны.

Идеи монад могут встречаться и в других неожиданных местах. Например, в последних версиях C# появился SQL-подобный синтаксис. Трансляция этого синтаксиса в стандартный вызов функции введена в справочник по языку. Одним из важных методов, привлекаемых в такой трансляции, является `SelectMany`, чья сигнатура имеет следующий вид:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>
  (this IEnumerable<TSource> source, Func<TSource, IEnumerable<TResult>> selector)
```

При всей многословности этого синтаксиса в нем нетрудно разобраться. Параметры типа для параметрических типов указываются между символами `<` и `>`. Кроме того, все параметры типа в сигнатуре метода должны быть заданы в явном виде (именно

поэтому после имени метода указано <TSource, TResult>). Для типа функции специального синтаксиса не существует, она просто называется Func. Поэтому чтобы транслировать эту сигнатуру в язык Haskell, она должна иметь следующий вид:

```
selectMany :: IEnumerable TSource -> (TSource -> IEnumerable TResult) ->
IEnumerable TResult
```

Но ведь это просто сигнатура типа использующейся в монаде функции (>>=)! Отсюда видно, что к монадам можно также прийти от базового синтаксиса языка SQL. Конечно, этот SQL-подобный синтаксис теперь используется в целях, о которых изначально никто не задумывался, например в задачах управления, но все это имеет смысл, когда рассматривается их связь с монадами.

Эти последние примеры из других языков программирования призваны помочь вам отнести одну из основных Haskell-концепций, каковой является монада, к категории, уже знакомой вам по предыдущему программистскому опыту.

Объединение монад

В этой и предыдущей главах мы весьма тщательно подходили к выбору примеров, чтобы в них использовалась только одна монада (это касается даже монады RWS, которая может рассматриваться как союз Reader, Writer и State в одной монаде). Однако понятно, что для выстраивания архитектуры вашего кода вам в некоторых случаях может понадобиться объединить несколько монад. Существует множество алгоритмов, которые лучше всего объяснить как объединение монады State, используемой для хранения ряда внутренних данных, с возможностью неудачи, предлагаемой монадой Maybe. В данном разделе показано, как объединять монады, при этом особое внимание уделено монадическим преобразователям.

В некоторых случаях можно составить собственный вариант объединения монад. Давайте для примера возьмем функцию, которая будет получать все возможные пути между двумя точками графа (как представленная ранее функция paths), но при этом управлять маршрутом как выходными данными монады Writer:

```
import Control.Monad
import Control.Monad.Writer

pathsWriter :: [(Int,Int)] -> Int -> Int -> [[Int]]
pathsWriter edges start end = map execWriter (pathsWriter' edges start end)

pathsWriter' :: [(Int,Int)] -> Int -> Int -> [Writer [Int] ()]
pathsWriter' edges start end =
  let e_paths = do (e_start, e_end) <- edges
            guard $ e_start == start
subpath <- pathsWriter' edges e_end end
return $ do tell [start]
subpath
in if start == end then tell [start] : e_paths else e_paths
```

Обратите внимание на две особенности. Во-первых, в коде задействован списочный экземпляр Monoid для использования в монаде Writer. Во-вторых, нам необходимо управлять изменением в монаде явным образом, и в данном коде это хорошо видно,

потому что в нем находится вложенный блок `do`. В данном случае этот блок просто создает новые вычисления в рамках монады `Writer`, задавая перед выполнением этих вычислений начальный год для подпути. И наконец, нам также нужно самостоятельно решить, как выполнять указанные вычисления: в данном случае решение состоит в том, чтобы применить к каждому элементу, возвращенному функцией `pathsWriter'`, выражение `map execWriter`.

Получившийся код неплохо читается, а место, в котором используется каждая монада, в нем четко выделено (первая часть блока `do` использует списочную монаду для составления нескольких возможных путей, а часть `Writer` ограничивается вложенным блоком). Однако если код окажется более сложным, может понадобиться вручную вызвать `execWriter` в середине кода и передать получившееся состояние от данного вычисления в следующий блок `Writer`. Очевидно, что лучшим решением была бы композиция из монад.

Монадические преобразователи

Наиболее предпочтительным в Haskell способом объединения эффекта применения нескольких монад в новой монаде является использование *монадических преобразователей* (monad transformers). Для объединения вычислительных эффектов (неудача, состояние и т. д.) предлагались (и продолжают предлагаться) также другие подходы, но наиболее востребованным вариантом остаются преобразователи. У пакетов `transformers` и `mtl` (сокращение от Monad Transformer Library) имеются сотни зависящих от них пакетов. Если говорить кратко, монадический преобразователь получает некую *базовую монаду* (base monad) и преобразует ее в *новую монаду* (new monad) с рядом дополнительных вычислительных эффектов. Например, в предыдущем коде использовался тип `[Writer [Int] ()]`, который можно увидеть в списочной монаде `[()]` (добавляющей недетерминизм), преобразуемой для организации дополнительного выхода (в форме монады `Writer [Int]`). Формально монадический преобразователь — это тип данных со следующей структурой:

```
MonadT e1 ... en m a
```

Давайте поэлементно рассмотрим общий тип преобразователя, чтобы понять базовую структуру:

- `MonadT` — это имя монадического преобразователя. Оно обычно совпадает с именем монады, эффект применения которой добавляет преобразователь. Например, монада `StateT` добавляет к другой монаде функциональность состояния, `ListT` — недетерминизм, `ReaderT` — контекст только для чтения и т. д.
- `e1 ... en` — это дополнительные переменные типа, в которых может нуждаться монадический преобразователь. В большинстве случаев такая переменная не нужна (например, для преобразователя `MaybeT` или `ListT`) или нужна только одна переменная (например, преобразователям состояния нужен тип внутреннего состояния), но в некоторых случаях (например, для преобразователя `RWST`) переменных нужно больше. Так, преобразователю `RWST` нужно по одной переменной для каждого контекста (только для чтения и только для записи), а также переменная для состояния.

- В позиции переменной типа `m` указывается монада преобразуемая, или заключаемая в оболочку (оба понятия используются как взаимозаменяемые).
- Результатом `MonadT` $e_1 \ldots e_n$ `m` должна стать монада как таковая, следовательно, она получает дополнительный параметр `a`, чтобы придать ей, как и ожидалось, вид `* -> *`.

В силу такой структуры монада, полученная в результате преобразования, во многом похожа на луковицу с внутренней базовой монадой в сердцевине и несколькими слоями, добавляющими базовой монаде дополнительную функциональность. Недостатком такой структуры является то, что вычисления в самом внутреннем слое должны «всплывать» до тех пор, пока не достигнут внешнего слоя. Эта функциональность предоставляется только одной функцией класса типов `MonadTrans` (это класс всех монадических преобразователей), которая называется `lift`:

```
class MonadTrans t where
lift :: Monad m => m a -> t m a
```

Например, если нужно транслировать предыдущий код в код, использующий монадические преобразователи, то в обращение к аргументу `edges` требуется включить функцию `lift`, чтобы можно было обратиться к do-нотации. Благодаря этому дополнительному вызову функции тип изменяется на тот, который монада фактически использует в примере:

```
lift edges :: WriterT [Int] [] ()
```

Таким образом, мы можем переписать пример, применяя монадические преобразователи. Обратите внимание на исчезновение вложенного блока `do`. Как видите, для `edges` нам понадобилась функция `lift`, а для `tell` — нет, поскольку функция `tell` находится на внешнем слое монады. Кроме того, код был изменен, чтобы вместо более конкретного конструктора списков `(:)` воспользоваться функцией `mplus` класса `MonadPlus`:

```
import Control.Monad.Writer

pathsWriterT' :: [(Int,Int)] -> Int -> Int -> WriterT [Int] [] ()
pathsWriterT' edges start end =
let e_paths = do (e_start, e_end) <- lift edges
guard $ e_start == start
tell [start]
pathsWriterT' edges e_end end
in if start == end then tell [start] `mplus` e_paths else e_paths
```

Как и в случаях с другими монадами, каждый монадический преобразователь имеет соответствующую функцию для инициирования вычислений с учетом дополнительной информации, которая может понадобиться такой монаде. В данном примере вместо функции `execWriter` в коде используется функция `execWriterT`, которая возвращает предназначенный только для записи вывод (в данном случае — список), заключенный в оболочку внутренней монады.

```
pathsWriterT :: [(Int,Int)] -> Int -> Int -> [[Int]]
pathsWriterT edges start end = execWriterT (pathsWriterT' edges start end)
```

Нужно запомнить одно очень важное обстоятельство: монадические преобразователи не обладают *коммутативностью*. То есть вычислительный эффект результирующей монады зависит от всего остального, что есть в преобразованиях. Например, монада `StateT s []` может представлять такие недетерминированные вычисления, где каждый из путей имеет разный результат и разное внутреннее состояние. Однако монада `ListT (State s)` реализует такие вычисления, которые могут возвращать несколько результатов, но состояние используется всеми ветвями совместно.

Одно из правил гласит, что эти эффекты интуитивно помещаются в стек монад в порядке, обратном тому, в котором они появляются в луковице преобразователя. Это соответствует нашему прежнему описанию: монада `StateT s []` сначала добавляет состояние ко всему остальному, а затем результаты с состоянием заключаются в оболочку недетерминизма; в то время как монада `ListT (State s)` сначала добавляет к результату недетерминизм, а затем состояние — ко всему остальному.

Рассмотрим следующий пример, объединяющий монады `Reader` и `Writer`, причем первая монада будет внешним слоем (и поэтому для использования функции `tell`, работающей в слое `Writer`, понадобится функция `lift`):

```
readerWriterExample :: ReaderT Int (Writer String) Int
readerWriterExample = do x <- ask
lift . tell $ show x
return $ x + 1
```

При выполнении функции может быть отслежен упомянутый факт помещения эффектов в стек монад в обратном порядке: внешний вызов соответствует функции `runWriter` (относящейся к внутренней монаде), а внутренний вызов (который «снимает» один слой) делается для функции `runReaderT`:

```
*Chapter7.CombiningMonads> runWriter (runReaderT readerWriterExample 3)
(4,"3")
```

Упражнение 7.6 на примере сложного стека монад поможет вам понять, как работает подтягивание, и выработать интуитивное понимание порядка помещения эффектов в стек.

УПРАЖНЕНИЕ 7.6. ДВА СОСТОЯНИЯ СРАЗУ

Напишите функцию, которая вычисляет факториал числа. Но вместо обычной реализации используйте реализацию, основанную на двух состояниях: одно для хранения убывающего счетчика, другое для хранения факториала. Один из подходов предполагает хранение состояния в виде кортежа, но для данного упражнения требуется, чтобы вы реализовали состояние с помощью монады StateT Integer (State Integer). То есть вам следует воспользоваться функцией lift для обращения к одному из внутренних состояний. Завершающий код для запуска вычислений в рамках монады должен быть похож на execState (execStateT factorial x) 1.

Чтобы покончить с формальностями, взгляните на табл. 7.1. В ней показаны способы внутреннего представления каждой из наиболее часто встречающихся монад, а также то, как монадический преобразователь вносит изменения в базовую монаду `m`. Эта информация станет ключом для понимания предыдущего примера со списком и состоянием.

Таблица 7.1. Наиболее часто используемые монады и связанные с ними преобразователи

Монада	Преобразователь	Внутреннее представление	Преобразование	Описание
`Identity`	`IdentityT`	`a`	`m a`	Представляет отсутствие преобразования
`[]`	`ListT`	`[a]`	`m [a]`	Используется для представления недетерминизма. Как уже отмечалось, для более качественного управления стратегией поиска следует использовать монады Logic и LogicT из пакета logict
`Maybe`	`MaybeT`	`Maybe a`	`m (Maybe a)`	Вычисления, которые могут завершиться неудачей. Этот преобразователь находится не в mtl, а в отдельном пакете MaybeT
`Reader r`	`ReaderT r`	`r -> a`	`r -> m a`	Доступен контекст только для чтения типа r
`Writer w`	`WriterT w`	`(a, w)`	`m (a, w)`	Результатом вычислений является значение только для записи типа w, где тип w должен быть типом Monoid
`State s`	`StateT s`	`s -> (a, s)`	`s -> m (a, s)`	Внутреннее состояние типа s, хранящееся внутри

Чтобы воспользоваться этой таблицей, представьте луковицу монад, которую вы хотели бы описать. Примените эти правила в рекурсивном порядке, заменяя переменную типа `m` в каждом случае внутренней монадой. Давайте сделаем это для описания разницы между `StateT s Maybe` и `MaybeT (State s)`:

- `StateT s Maybe a = s -> Maybe (a, s)`, следовательно, неудача влияет как на результат, так и на состояние: либо все вычисления терпят неудачу, либо они проходят удачно.
- `MaybeT (State s) a = (State s) (Maybe a) = s -> (Maybe a, s)`, то есть неудача влияет только на результат, но не на сохраняемое внутреннее состояние.

Вы, конечно же, заметили включение в таблицу не упомянутой ранее монады `Identity` (которая доступна в модуле `Control.Monad.Identity`). Эта монада не добавляет никаких эффектов к вычислениям: функция `return` возвращает идентичный результат, а единственным функциональным вариантом применения монады является привязка. Эта монада предназначена для того, чтобы быть внутренней монадой в луковице, обеспечивая привязку всего стека монад только с помощью монадических преобразователей. Например, в пакете `mtl` монада `Reader r` фактически имеет определение `ReaderT r Identity`.

Монадические классы

До сих пор говорилось, что функциональные возможности преобразователей доступны в обоих пакетах — в `transformers` и в `mtl`. Однако пакет `mtl` построен строго

в соответствии с иерархией классов типов, что исключает необходимость в большей части вызовов функции `lift`. Таким образом, работа с преобразователями оказывается не сложнее использования монад, создаваемых с нуля.

Возьмем в качестве примера монаду `Reader`. Базовыми операциями при использовании этой монады являются `ask`, `local` и `reader`, которые представляют собой функции, подлежащие обобщению (то есть другие функции, например `asks`, могут быть написаны с их использованием). В `Control.Monad.Reader` можно найти класс типов `MonadReader`, который состоит из этих функций:

```
class Monad m => MonadReader r m | m -> r where
ask :: m r
local :: (r -> r) -> m a -> m a
reader :: (r -> a) -> m a
```

ПРИМЕЧАНИЕ

В определении класса типов вы можете увидеть дополнительную часть информации m -> r. Это называется функциональной зависимостью и подробно рассматривается в главе 13.

Кроме того, в `mtl` определены экземпляры `MonadReader` для каждой комбинации монадических преобразователей, куда в качестве слоя включен преобразователь `ReaderT`. При его наличии вам не нужно явным образом записывать функцию `lift` для использования функций `ask`, `local` и всех остальных функций обработки контекста. Подобная комбинация класса типов и нескольких экземпляров определена и для остальных монадических преобразователей: `MonadWriter`, `MonadState`, `MonadLogic` (последний находится в пакете `logict`). Класс типов соответствует функциональности в списках, а `Maybe` получают через `MonadPlus`.

Например, если вы работаете с преобразователями, используя пакет `mtl`, то последний пример можно переписать без явного вызова функции `lift` для обращения к слою `Writer`:

```
readerWriterExample :: ReaderT Int (Writer String) Int
readerWriterExample = do x <- ask
tell $ show x
return $ x + 1
```

Помимо сокращения количества вызовов функции `lift` эти классы типов помогают задавать сигнатуры типов, охватывающие не только конкретную структуру монадических альтернатив, но и любую из структур, поддерживающих какую-либо функциональность. Приведем пример. Предположим, мы решили включить предыдущую функцию `readerWriterExample` в собственную библиотеку полезных функций и делаем это, используя указанную ранее сигнатуру типа. При этом функция строго соответствует данному конкретному примеру: мы сможем запускать функцию, только используя в точности такую же комбинацию монад, которая зафиксирована в сигнатуре. Однако создаваемый код должен запускаться в любой луковице, если она включает в себя функциональность `ReaderT` и `WriterT`. То есть нужно стремиться к обобщению, которое можно получить, используя монадические классы, а не непосредственный стек монад:

```
readerWriterExample :: (MonadReader Int m, MonadWriter String m) => m Int
readerWriterExample = do x <- ask
```

```
tell $ show x
return $ x + 1
```

Здесь наиболее важная часть выделена полужирным шрифтом: информация о монадической функциональности, используемой внутри функции, закодирована в ограничениях классов типов. В данном примере `m` представляет монаду, поддерживающую обе функциональности — и `Reader`, и `Writer`.

Чтобы показать, что этот код на самом деле универсальный, в представленном далее сеансе работы интерпретатора запускается аналогичная монадическая функция, использующая обе монады, и `ReaderT Int (Writer String)`, и `RWS Int String Int` (здесь та часть, которая относится к состоянию, не задействована, но это по-прежнему монада, поддерживающая операции из `MonadReader` и `MonadWriter`):

```
*Chapter7.CombiningMonads> runWriter (runReaderT readerWriterExample 3)
(4,"3")
*Chapter7.CombiningMonads> runRWS readerWriterExample 3 0
(4,0,"3")
```

Таким образом, если вместо непосредственного использования монады (например, `Reader`, `Writer` и т. д.) вам нужно экспонировать функцию, производящую монадическое значение, при программировании имеет смысл применить соответствующий класс типов. Это лучше в плане многократного использования кода, поскольку потребители вашей библиотеки смогут задействовать любую преобразованную монаду, в которую встроена соответствующая функциональность.

СРАВНЕНИЕ ПАКЕТОВ TRANSFORMERS И MTL

Как утверждалось в начале этого раздела, в Haskell Platform есть два разных пакета, поддерживающих монадические преобразования. Пакет transformers является основным: он предоставляет все монадические преобразователи и класс типов MonadTrans. Пакет mtl поддерживает эту функциональность, а также монадические классы. Если вы экспортируете функции, типы которых включают монады, вам нужно использовать монадические классы, а следовательно, пакет mtl.

Примеры в данном разделе работают только с mtl. Если вместо этого пакета вы решите использовать пакет transformers, вместо Control.Monad.Something нужно импортировать Control.Monad.Trans.Something.

Упражнение 7.6 требует создания функции `paths`, которая использует несколько монадических преобразователей. Как только что было объяснено, при программировании этой функции нужно задействовать соответствующий класс типов.

УПРАЖНЕНИЕ 7.7. СОЗДАНИЕ ФУНКЦИИ PATHS НА ОСНОВЕ МОНАДИЧЕСКИХ ПРЕОБРАЗОВАТЕЛЕЙ

Напишите новую версию функции pathsWriter, в которой граф сохраняется как контекст только для чтения. Это означает, что вам нужно воспользоваться функциональностью как из MonadReader (для обработки графа), так и из MonadWriter (для обработки путей), охватывающей базовую списочную монаду. Чтобы проверить универсальность воспользуйтесь для предоставления требуемой функциональности двумя различными монадами: ReaderT r (WriterT w []) a и RWST r w s a.

В заключение хотелось бы подчеркнуть важность монадических преобразователей для Haskell-программиста. Создание стека монад и добавление или удаление слоев

в процессе разработки приложения требуется довольно часто. Но это не означает, что ваш код должен быть похож на мешанину, где каждая функция вызывает любую монадическую функциональность. Решение на базе пакета `mtl`, позволяющее абстрагировать применение классов типов, дает возможность точно указать, какие слои монад задействовать в каждой функции, что очень хорошо в плане сочетаемости и многократного использования кода.

Выводы

В данной главе мы еще глубже погрузились в тему монад, с которыми познакомились в предыдущей главе:

- Мы узнали, что такое *списочная монада*, моделирующая недетерминированные вычисления, которые могут возвращать более одного значения. Мы увидели, как эти монады могут использоваться для решения задач поиска.
- В тех случаях, когда списочная монада не обеспечивает достаточной управляемости поиска, с помощью монады `Logic` могут быть реализованы справедливые конъюнкция и дизъюнкция.
- Мы познакомились с важным расширением класса `Monad` — классом `MonadPlus`, позволяющим моделировать неудачу и возможность выбора. Теперь мы знаем, как с помощью имеющихся там функций `mzero` и `mplus` реализовать априорный алгоритм для вывода правил.
- Мы получили высокоуровневое представление о таких полезных монадических функциях, как `sequence`, `mapM` и т. д.
- Мы узнали, как с помощью GHC-расширения `MonadComprehensions` синтаксис *формирователей* может быть обобщен для работы с любой монадой. Это позволяет сравнить понятие монады из языка Haskell с другими понятиями из языков Scala и C#.
- И наконец, мы выяснили, как объединять монады с помощью *монадических преобразователей*.

Использование нескольких ядер

8

Одним из основных преимуществ чистоты, присущей языку Haskell, является возможность беспрепятственного выполнения кода в параллельном режиме. Отсутствие побочных эффектов означает, что все зависимости от данных обозначены в коде явным образом. Соответственно компилятор (как и вы сами) может планировать параллельное выполнение различных заданий, не имеющих между собой зависимостей.

Явно выразить те части вашего кода, которые могут получить преимущества от выполнения в параллельном режиме, позволяет монада `Par`. Поддерживаемая `Par` модель позволяет создавать код, используя как модель *фьючерсов*, так и подход, называемый *параллелизмом потоков данных*. Затем заботу о выполнении вашего кода с использованием параллельных программных потоков берет на себя диспетчер. Монада `Par` является весьма эффективной абстракцией, поскольку вам не нужно задумываться об управлении созданием и уничтожением программных потоков: достаточно дать библиотеке возможность делать свое дело.

Однако иногда ряд параллельных заданий нуждается в таком способе совместного использования ресурсов, который невозможно выразить с помощью монады `Par`. В магазине машин времени, к примеру, несколько клиентов могут покупать одни и те же товары, а при этом одновременно должно происходить обновление нескольких баз данных. В таких сценариях важно обеспечить правильность одновременного доступа к ресурсам. В качестве инструмента управления этим поведением Haskell предлагает так называемую *программную транзакционную память*, в которой используется идея транзакций, позаимствованная из систем управления базами данных.

И наконец, может потребоваться распределить вычисления между несколькими узлами, находящимися в разных местах сети. Для таких случаев подойдет пакет облачных решений для Haskell (Cloud Haskell). В этом пакете для обмена данными между узлами, в ходе которого можно отправлять и получать как данные, так и код, используется модель акторов (actor model).

Параллелизм, одновременность, распределенность

При применении по отношению к программированию понятий параллелизма, одновременности и распределенности всегда возникает некая путаница. Поэтому прежде всего следует внести ясность в эту терминологию.

В программной модели *одновременности* (concurrency) вычисления выполняются в рамках нескольких, по большей части независимых программных потоков. Система может либо смешивать вычисления, либо запускать их параллельно, но в любом случае возникает иллюзия, что все они выполняются асинхронно. Одним из основных примеров применения одновременности является веб-сервер: одновременно им обслуживаются запросы от многих клиентов, и с программистской точки зрения каждый из запросов не зависит от остальных и обрабатывается асинхронно.

В большинстве случаев таким программным потокам требуется доступ к некоторым общим ресурсам. И здесь нужно добиться того, чтобы одновременный доступ не ввел систему в некое несогласованное состояние. Для этого было разработано множество технологий программирования, включая блокировки, семафоры или каналы. Что касается языка Haskell, то модель *программной транзакционной памяти* (Software Transactional Memory, STM) вводит в код из баз данных понятие атомарных транзакций, что позволяет поддерживать оптимистичную одновременность с возможностью отката.

Что касается *параллелизма* (parallelism), то он относится к способу одновременного выполнения кода на более чем одном ядре компьютера. Для увеличения производительности одновременно выполняемые задания часто запускаются в параллельном режиме. Эта прибавка в скорости может быть также применена к заданиям, которые изначально не предназначались для одновременного выполнения, но чьи зависимости от данных позволяют выполнять части алгоритма независимо друг от друга. Вспомним алгоритм быстрой сортировки: на каждом этапе список делится на две части, каждая из которых сортируется отдельно. В этом случае вместо последовательной может производиться параллельная сортировка двух списков.

Для этого второго варианта Haskell является идеальной площадкой: чистые вычисления не могут мешать друг другу, а их зависимости от данных выражены предельно ясно. Представляемая далее монада `Par` следует именно этому замыслу и допускает параллелизм выполнения заданий.

Во многих случаях причиной путаницы между параллелизмом и одновременностью являются языки, допускающие побочные эффекты. В таких языках, как Java или C, любая часть кода может иметь доступ к глобальному состоянию и вносить в него изменения. Поэтому при любой степени параллелизма приходится уделять внимание доступу к общим ресурсам, что при программировании требует применения специальных приемов, учитывающих одновременность выполнения заданий. В данном контексте оба понятия по-настоящему отделить друг от друга невозможно.

Параллельное программирование обычно ассоциируют с выполнением заданий на разных ядрах (микропроцессорах или графических процессорах) одной и той же компьютерной системы. Однако вычисления могут быть также распределены между разными компьютерами, обменивающимися данными по сети. В этом случае речь идет о *распределенном программировании* (distributed programming). Поскольку все действующие лица в системе не зависят друг от друга, координирование заданий должно производиться не так, как при параллельном программировании в рамках одной системы. Кроме того, взаимодействие по сети накладывает огра-

ничения, связанные с отказами или большими задержками. По совокупности этих причин для распределенного программирования нужны другие технологии.

Среду распределенного программирования в Haskell предоставляет пакет `distributed-process`. Этот пакет является частью проекта облачных решений для Haskell, и на него большое влияние оказал язык программирования Erlang. В частности, для управления взаимодействием различных систем в сети в нем используется модель акторов.

РЕЗЮМЕ

«Одновременность — это о том, чтобы иметь дело одновременно с множеством вещей. А параллелизм — это о том, чтобы делать одновременно множество вещей». — Роб Пайк (Rob Pike). Как говорится, почувствуйте разницу.

Площадка для параллельного, одновременного и распределенного программирования в Haskell намного шире, чем показано в данной главе. Рассматриваемые здесь библиотеки могут использоваться множеством иных способов, к тому же в Haskell имеется множество других пакетов. Для параллельного программирования используется пакет `parallel`, инструменты которого реализуют стратегический подход. Параллелизм относится не только к процессорам: ускоритель формирует код, запускаемый на графическом процессоре. Существующей в Haskell пакет `base` предлагает в модуле `Control.Concurrent` низкоуровневую функциональность для реализации концепции одновременности, включая изменяемые адреса памяти (`MVar`) и семафоры.

Монада Par

Первым в этой главе мы рассмотрим пакет параллельного программирования `monad-par`. Функции в этой библиотеке связаны с монадой `Par` и используют переменные `IVar` для хранения результатов взаимодействия. Как мы вскоре увидим, с помощью этого пакета вычисления можно моделировать двумя путями: в виде фьючерсов или в виде потоков данных.

Фьючерсы

Начнем с очень простой задачи, раскладывающей несколько чисел на простые множители. Алгоритм факторизации одного числа весьма прост: мы пытаемся его разделить на возрастающие натуральные числа. Если в какой-то момент от деления получится нулевой остаток, значит, число является простым множителем. Таким образом, исходное число может быть поделено на этот простой множитель, и процесс может начаться сначала. В какой-то момент вы получите то же число, с которого начали, и это будет означать, что достигнут последний простой множитель. Haskell-код, реализующий этот подход, выглядит так:

```
findFactors :: Integer -> [Integer]
findFactors 1 = [1]
```

```
findFactors n = let oneFactor = findFactor n 2
                in oneFactor : (findFactors $ n `div` oneFactor)

findFactor :: Integer -> Integer -> Integer
findFactor n m | n == m         = n
               | n `mod` m == 0 = m
               | otherwise      = findFactor n (m + 1)
```

В каком-то месте программы вам будет предложено разложить на множители два разных числа. Код такой функции пишется просто:

```
findTwoFactors :: Integer -> Integer -> ([Integer],[Integer])
findTwoFactors x y = (findFactors x, findFactors y)
```

Однако эффективность этой функции не слишком высока. Даже если доступен не один, а несколько процессоров, вычисление простых множителей `x` и `y` будет производиться последовательно (допустим, что полное вычисление каждого выполняется за одно и то же время). Нужно, как показано на рис. 8.1, поставить себе цель вычислять `findFactors x` в то же самое время, когда вычисляется `findFactors y`.

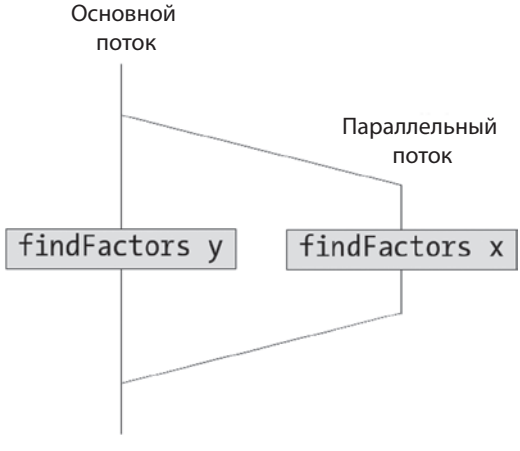


Рис. 8.1. Параллельные вычисления двух разложений на простые множители

С помощью пакета `monad-par` системе можно просто приказать выполнять оба вызова `findFactors` параллельно. В модуле `Control.Monad.Par` есть функция `spawnP`. Давайте присмотримся к ее типу:

```
spawnP :: NFData a => a -> Par (IVar a)
```

Функция `spawnP` предназначена для выполнения вычислений параллельно со всей остальной программой. В этой сигнатуре следует отметить три особенности. Прежде всего, она требует, чтобы тип выполняемых вычислений поддерживался классом типов `NFData`. Если помните, этот тип из пакета `deepseq` гарантирует, что вычисления выполнятся полностью. Функция `spawnP` накладывает это ограничение,

поскольку существует только один способ реально обеспечить параллельное выполнение кода. Если бы этого ограничения здесь не было, то из-за модели ленивых вычислений он был бы выполнен в любое другое время, что привело бы к потере преимуществ параллелизма. Поскольку функция `spawnP` полностью определяет, когда будут выполнены те или иные вычисления, параллельная модель `monad-par` называется *детерминированной*.

Вторая особенность заключается в том, что результат заключен в оболочку типа `Par`. Этот тип является монадой, которая поддерживает параллелизм. И наконец, вместо простого значения результатом выполнения функции `spawnP` является фьючерс `IVar`. *Фьючерс* (future) — это обещание того, что результат вычислений будет доступен по запросу. Для получения результата вычислений внутри `IVar` нужно вызвать функцию `get`. Эта функция тут же возвращает управление, если вычисления завершены, или блокирует выполнение до тех пор, пока не будет доступен результат. Точно такая же модель используется в Scala и в библиотеке параллельных вычислений языка C#.

Вы можете вызывать функции `spawnP` и `get` только внутри монады `Par`. С целью выполнения всех заданий функцию `runPar` нужно вызывать с полной трассой параллелизма. Версия `findTwoFactors`, создающая параллельное задание для вычисления разложения на множители `x` при сохранении факторизации `y` в текущем программном потоке, могла бы выглядеть так:

```
import Control.DeepSeq
import Control.Monad.Par

findTwoFactors :: Integer -> Integer -> ([Integer],[Integer])
findTwoFactors x y = runPar $ do
  factorsXVar <- spawnP $ findFactors x
  let factorsY = findFactors y
      _        = rnf factorsY
  factorsX <- get factorsXVar
  return (factorsX, factorsY)
```

Обратите внимание на вызов функции `rnf` из библиотеки `deepseq` для полного вычисления факторизации `y`.

Следование этим шагам не означает непосредственного создания параллельных заданий — для этого требуется сделать еще два шага. Сначала нужно скомпилировать программу как *многопоточную* (threaded run-time), что позволит GHC создать код, ориентированный на использование нескольких программных потоков. Чтобы добиться этого, добавьте в Cabal-файл флаг `-threaded`:

```
executable chapter8
  hs-source-dirs:  src
  main-is:         Main.hs
  build-depends:   base >= 4, monad-par, deepseq
  ghc-options:     -Wall -threaded
```

Кроме того, вам следует передать своей программе параметры, разрешающие использование нескольких ядер:

```
$ ./dist/build/chapter8/chapter8 +RTS -N2
```

Ключ `+RTS` показывает начало действия ключей, переданных Haskell-системе времени выполнения. В частности, ключ `-N2` указывает на то, что должны быть задействованы два процессора. Можно указать то количество процессоров, которое не превышает число процессоров, имеющихся в системе. Если хотите, можете указать ключ `-N` как таковой (без числа), позволив тем самым Haskell-системе времени выполнения самой принять решение о том, каким количеством процессоров следует воспользоваться.

Параллелизм и переменные IVar

Пакет `monad-par` предоставляет не только фьючерсы, но и обширную модель объявления параллельных вычислений. Вместо простого порождения параллельных вычислений вы указываете несколько этапов вычислений, которые совместно используют промежуточные результаты посредством переменных `IVar`. Эти переменные создаются с помощью ключевого слова `new`. Задания могут записываться в `IVar` с помощью функции `put`, а результат можно получить с помощью функции `get`. Следует заметить, что `IVar` является однократно записываемой переменной.

Рассмотрим пример создания письма для клиента с его счетом на товар. Это подразумевает выполнение четырех различных заданий: одно из них будет заключаться в поиске клиентской информации в базе данных, второе — в том же самом, но в отношении товара. Каждое из этих заданий будет контактировать с другими заданиями через соответствующие переменные `IVar`. Два других задания должны брать эту информацию и генерировать соответственно текст для письма и для конверта. Эта схема показана на рис. 8.2.

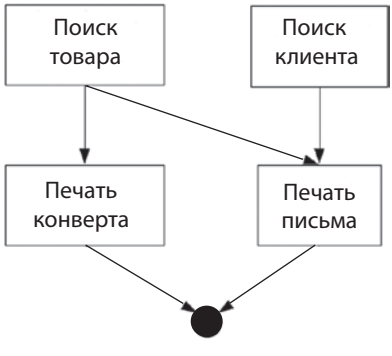


Рис. 8.2. Схема потоков данных при создании письма

Вычисления, формируемые таким способом, всегда следуют показанной схеме, где задания взаимодействуют посредством переменных `IVar`. Поэтому эта модель называется *программированием потоков данных* (dataflow programming). Основным преимуществом такого подхода является то, что все зависимости от данных выражены явным образом — через переменные `IVar`. Эту информацию библиотека `monad-par` использует для планирования параллельного выполнения заданий.

Схема потоков данных, показанная на рис. 8.2, реализована в следующем коде. Обратите внимание, что при использовании этой модели и переменных `IVar` вместо

функции spawnP применяется функция ветвления fork, которая ожидает вычислений типа Par ():

```
printTicket :: Int -> Int -> [(Int,String)] -> [(Int,String)] -> String
printTicket idC idP clients products = runPar $ do
  clientV  <- new
  productV <- new
  fork $ lookupPar clientV  idC clients
  fork $ lookupPar productV idP products
  envV    <- new
  letterV <- new
  fork $ printEnvelope clientV envV
  fork $ printLetter   clientV productV letterV
  envS    <- get envV
  letterS <- get letterV
  return $ envS ++ "\n\n" ++ letterS

lookupPar :: (Eq a, NFData b) => IVar (Maybe b) -> a -> [(a,b)] -> Par ()
lookupPar i _ []                    = put i Nothing
lookupPar i x ((k,v):r) | x == k    = put i $ Just v
                        | otherwise = lookupPar i x r

printEnvelope :: IVar (Maybe String) -> IVar String -> Par ()
printEnvelope clientV envV = do
  clientName <- get clientV
  case clientName of
    Nothing -> put envV "Unknown"
    Just n  -> put envV $ "To: " ++ n

printLetter :: IVar (Maybe String) -> IVar (Maybe String) -> IVar String -> Par ()
printLetter clientV productV letterV = do
  clientName  <- get clientV
  productName <- get productV
  case (clientName, productName) of
    (Nothing, Nothing) -> put letterV "Unknown"
    (Just n,  Nothing) -> put letterV $ n ++ " bought something"
    (Nothing, Just p)  -> put letterV $ "Someone bought " ++ p
    (Just n,  Just p)  -> put letterV $ n ++ " bought " ++ p
```

Одно из любопытных преимуществ отделения зависимостей потоков данных от реального параллельного выполнения заключается в том, что при этом могут применяться несколько стратегий планирования заданий. По умолчанию в monad-par используется планировщик под названием Direct. Доступны еще два планировщика: если вместо Control.Monad.Par импортировать Control.Monad.Par.Scheds.Spark или Control.Monad.Par.Scheds.Trace, то будет использован тот планировщик, который соответствует импортированному модулю.

Распараллеливание априорного алгоритма

Завершим мы этот раздел тем, что рассмотрим возможные способы усовершенствования априорного алгоритма для его выполнения в параллельном режиме. Код будет основан на реализации, предложенной в главе 7.

Если вы работаете со списками, то в пакет `monad-par` включена функция `parMap`. Ее назначение состоит в параллельном применении к каждому элементу списка некоей функции. Порождение задания для каждого элемента может показаться излишним, но планировщик учитывает количество ядер, доступных в системе. Чтобы применить функцию `parMap`, давайте сначала перепишем `generateL1`, чтобы из монадического стиля получить явные вызовы `map` и `concatMap`. Следующий код является полным эквивалентом соответствующего кода из главы 7:

```
generateL1 minSupport transactions =
  let c1 = noDups $ concatMap (\(Transaction t) -> map (FrequentSet . S.singleton)
$ S.toList t)
                              transactions
      l1NotFiltered = map (\fs -> (fs, setSupport transactions fs > minSupport)) c1
   in concatMap (\(fs,b) -> if b then [fs] else []) l1NotFiltered
```

Поскольку основное время в алгоритме тратится на вычисление поддержек, именно эта часть и была выбрана для параллельного выполнения. Прежде вычисления поддержек осуществлялись внутри функции `filter`, где также принималось решение о том, оставлять ли транзакцию в списке. Теперь эти две задачи разделены: поддержки множеств вычисляются в `l1NotFiltered`, а затем, в окончательной версии `concatMap`, принимается решение о том, нужно включать элемент или нет. После этого останется только заменить `map` на `parMap` и заключить все вычисления в оболочку `runPar`, чтобы воспользоваться параллелизмом потоков данных априорного алгоритма. Тогда будет получен следующий результат:

```
generateL1 minSupport transactions = runPar $ do
  let c1 = noDups $ concatMap (\(Transaction t) -> map (FrequentSet . S.singleton)
$ S.toList t)
                              transactions
  l1NotFiltered <- parMap (\fs -> (fs, setSupport transactions fs > minSupport)) c1
  return $ concatMap (\(fs,b) -> if b then [fs] else []) l1NotFiltered
```

ПРИМЕЧАНИЕ

Следует помнить, что при использовании библиотеки monad-par ваши типы данных должны быть экземплярами класса типов NFData.

В некоторых случаях эта стратегия может быть не самой лучшей для создания параллельных заданий. Вместо использования функции `parMap` можно делить список на половины до тех пор, пока не будет достигнута некая минимальная длина. Как только список станет достаточно мал, лучше выполнить его обход последовательно, поскольку создание параллельных заданий связано с издержками. Именно это и сделано в новой версии функции `generateNextLk`:

```
generateNextLk :: Double -> [Transaction] -> (Int, [FrequentSet])
               -> Maybe ([FrequentSet], (Int, [FrequentSet]))
generateNextLk _ _ (_, []) = Nothing
generateNextLk minSupport transactions (k, lk) =
  let ck1 = noDups $ [ FrequentSet $ a `S.union` b | FrequentSet a <- lk,
FrequentSet b <- lk
                                                   , S.size (a `S.intersection` b)
== k - 1 ]
```

```
      lk1 = runPar $ filterLk minSupport transactions ck1
   in Just (lk1, (k+1, lk1))

filterLk :: Double -> [Transaction] -> [FrequentSet] -> Par [FrequentSet]
filterLk minSupport transactions ck =
  let lengthCk = length ck
   in if lengthCk <= 5
      then return $ filter (\fs -> setSupport transactions fs > minSupport) ck
      else let (l,r) = splitAt (lengthCk `div` 2) ck
            in do lVar <- spawn $ filterLk minSupport transactions l
                  lFiltered <- get lVar
                  rVar <- spawn $ filterLk minSupport transactions r
                  rFiltered <- get rVar
                  return $ lFiltered ++ rFiltered
```

Как видите, использование библиотеки `monad-par` упрощает добавление параллелизма в текущий код. Эта библиотека нацелена на фьючерсы и программирование потоков данных. Но существуют и другие подходы. Например, в библиотеке `parallel` используется другая монада под названием `Eval`, которая помогает определить, как может быть обработана в параллельном режиме конкретная структура данных. Дополнительные сведения об этом и о других пакетах можно найти в Haskell-википедии (http://www.haskell.org/haskellwiki/Applications_and_libraries/Concurrency_and_parallelism).

РАСПАРАЛЛЕЛИВАНИЕ ЗАДАНИЙ С ПОБОЧНЫМИ ЭФФЕКТАМИ

О вычислениях с произвольными побочными эффектами мы пока еще не рассуждали. Однако на будущее имейте в виду, что пакет monad-par предоставляет для параллельных вычислений еще одну монаду, которая называется ParIO. Она доступна в модуле Control.Monad.Par.IO и допускает наличие побочных эффектов. Интерфейс у нее такой же, как в обычной монаде Par, за исключением выполнения вычислений через функцию runParIO.

Следует заметить, что реализация не гарантирует какой бы то ни было очередности выполнения заданий, и это выражается в побочном эффекте недетерминированного упорядочения.

С помощью библиотеки `monad-par` в параллельные могут быть превращены многие алгоритмы, работающие со списками или имеющие структуру типа «разделяй и властвуй». В упражнении 8.1 вам предлагается проделать это с еще одним представленным в данной книге алгоритмом анализа данных — с алгоритмом K-средних.

УПРАЖНЕНИЕ 8.1. ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ K-СРЕДНИХ

Напишите параллельную версию алгоритма K-средних, разработанного в главе 6. Для упрощения данной задачи можно посмотреть на первую реализацию, в которой монады не использовались. Следует помнить, что при применении таких функций, как parMap, нужно подумать о том, чтобы издержки от создания параллельных заданий не оказались больше получаемых от этого преимуществ.

Программная транзакционная память

В данном разделе рассматриваются проблемы, возникающие, когда несколько программных потоков взаимодействуют между собой и используют общие ресурсы.

То есть это именно та ситуация, когда имеет место *одновременность*. Хотя Haskell позволяет разрабатывать соответствующий инструментарий классическим способом, используя блокировки, семафоры и пр., в данном разделе мы увидим, что функциональный стиль программирования позволяет применить намного более мощную абстракцию, называемую программной транзакционной памятью.

Сначала нужно учесть тот факт, что код, поддерживающий концепцию одновременности, считается кодом с побочными эффектами. Когда несколько программных потоков выполняются асинхронно и совместно используют ресурсы, порядок, в котором это делается, влияет на наблюдаемый результат. В отличие от этого в чистом коде порядок, в котором вычисляются функции, совершенно не важен, поскольку результат всегда будет одним и тем же.

Больше узнать о том, как справляться с произвольными побочными эффектами, вы сможете в следующей главе. А пока вам достаточно знать, что в Haskell есть специальная монада `IO`, позволяющая использовать побочные эффекты. Единственным отличием кода, созданного при программировании с учетом и без учета побочных эффектов, является do-нотация.

Одновременное использование ресурсов

Давайте начнем путешествие по теме программирования концепции одновременности в Haskell с простого примера: имитации нескольких клиентов, покупающих товары в магазине. В первом приближении мы рассмотрим только те изменения, которые касаются денег, получаемых магазином машин времени. Код, создающий наши три программных потока, будет иметь следующий вид:

```
import Control.Concurrent
main :: IO ()
main = do v <- newMVar 10000
          forkIO $ updateMoney v
          forkIO $ updateMoney v
          forkIO $ updateMoney v
          _ <- getLine
          return ()

updateMoney :: MVar Integer -> IO ()
updateMoney v = do m <- takeMVar v
                   putStrLn $ "Updating value, which is " ++ show m
                   putMVar v (m + 500)  -- suppose a constant price
```

Первое, о чем нужно знать, — создание нового программного потока. Это делается с помощью функции `forkIO` из модуля `Control.Concurrent`. В качестве аргумента эта функция получает действие типа `IO ()`, после чего начинает выполнение кода в параллельном режиме.

ПРИМЕЧАНИЕ

Функция forkIO возвращает идентификатор потока, позволяющий приостанавливать и останавливать только что созданный программный поток. Однако функциональность модуля Control.Concurrent в этой книге не рассматривается.

Как видите, функция `main` создает три программных потока, выполняющих один и тот же код. Следующий вопрос заключается в том, как заставить эти потоки сотрудничать и совместно использовать ресурсы? Ведь по умолчанию обмениваться данными друг с другом они не могут. Ответ такой: посредством блока `MVar`, в котором может содержаться изменяемая переменная, доступная для чтения или обновления. Один из таких блоков создается до разветвления программных потоков с помощью функции `newMVar`, после чего он передается в качестве аргумента каждому из потоков. Таким образом программные потоки получают доступ к общему ресурсу в форме изменяемой переменной.

Каждый поток имеет возможность считывать значение переменной `MVar` с помощью функции `takeMVar` и записывать новое значение переменной с помощью функции `putMVar`. Этот тип пригоден для одновременного выполнения благодаря специальному поведению, проявляемому им при наличии нескольких программных потоков. Переменную `MVar` можно представить в качестве блока, который либо содержит какой-нибудь элемент, либо является пустым. При использовании функции `takeMVar` вы либо считываете хранящееся значение и опустошаете блок, либо блок остается заблокированным до тех пор, пока туда не будет помещен какой-нибудь элемент. И наоборот, функция `putMVar` либо записывает в блок новое значение, если блок пуст, либо пребывает в режиме ожидания. Кроме того, эти функции гарантируют пробуждение только одного заблокированного потока и обслуживание таких программных потоков в порядке «первым пришел, первым вышел». Это означает, что ни один из программных потоков не может подавить события остальных программных потоков.

Обратите внимание на то, что в конце кода находится вызов функции `getLine`. Назначение этой функции заключается в ожидании какого-либо пользовательского ввода. Необходимость в ней вызвана тем, что при завершении вычислений основным программным потоком любой другой поток, созданный функцией `forkIO`, завершает свое существование вместе с основным. Следовательно, чтобы увидеть эффект применения других потоков, нужно как-то заставить основной программный поток продолжить выполнение. Именно для этого и требуется ожидание пользовательского ввода.

Для реализации еще одного действия давайте добавим новый вид программного потока, который просто считывает текущее денежное значение и выводит его на экран. Поскольку здесь не нужно производить какие-либо вычисления, можно воспользоваться функцией `readMVar`, которая эквивалентна функции `readMVar`, а затем — функцией `putMVar` с тем же значением. Тогда это можно будет читать:

```
readMoney :: MVar Integer -> IO ()
readMoney v = do m <- readMVar v
                 putStrLn $ "The current value is " ++ show m
```

Чтобы стало еще интереснее, давайте добавим случайную задержку в диапазоне от 3 до 15 секунд. Следующая функция просто создает случайное число (разговор о случайных числах нам предстоит в следующей главе) с помощью функции `randomRIO`, а затем вызывает функцию `threadDelay`, приостанавливающую программный поток на время, равное указанному количеству микросекунд. Следует

учесть, что при использовании функции `randomRIO` нужно добавить зависимость от пакета `random`:

```
import System.Random

randomDelay :: IO ()
randomDelay = do r <- randomRIO (3, 15)
                 threadDelay (r * 1000000)
```

И наконец, можно написать функцию `forkDelay`, которая порождает `n` потоков со случайным предварительным временем ожидания:

```
import Control.Monad

forkDelay :: Int -> IO () -> IO ()
forkDelay n f = replicateM_ n $ forkIO (randomDelay >> f)
```

Затем можно реализовать пять новых обновлений и пять считываний:

```
main :: IO ()
main = do v <- newMVar 10000
          forkDelay 5 $ updateMoney v
          forkDelay 5 $ readMoney v
          _ <- getLine
          return ()
```

ПРИМЕЧАНИЕ

Ни одна из связанных с MVar функций не принуждает к вычислению тех данных, которые вставлены в блок. Это может стать источником проблем, поскольку цена за выполнение кодового фрагмента может быть заплачена значительно позже, в контексте других вычислений. Чтобы получить строгую версию MVar, нужно присмотреться к пакету strict-concurrency.

Атомарные транзакции

Перейдем к более сложному примеру. Основной замысел по-прежнему относится к клиенту, приобретающему конкретный товар, но в данном случае будет привлечен не один, а несколько ресурсов. Первым ресурсом, как и раньше, станут деньги, вырученные магазином, а вторым — текущий запас товара в магазине, который должен обновляться, отражая факт продажи единицы товара. Как и в предыдущем случае, мы добавим ряд дополнительных программных потоков, считывающих значения денег и запаса товара в магазине:

```
main :: IO ()
main = do v <- newMVar 10000
          s <- newMVar [("a",7)]
          forkDelay 5 $ updateMoneyAndStock "a" 1000 v s
          forkDelay 5 $ printMoneyAndStock v s
          _ <- getLine  -- для ожидания завершения
          return ()

updateMoneyAndStock :: Eq a => a -> Integer -> MVar Integer -> MVar [(a,Integer)]
-> IO ()
updateMoneyAndStock product price money stock =
```

```
  do s <- takeMVar stock
     let Just productNo = lookup product s
     if productNo > 0
       then do m <- takeMVar money
               let newS = map (\(k,v) -> if k == product then (k,v-1) else (k,v)) s
               putMVar money (m + price) >> putMVar stock newS
       else putMVar stock s

printMoneyAndStock :: Show a => MVar Integer -> MVar [(a,Integer)] -> IO ()
printMoneyAndStock money stock = do m <- readMVar money
                                    s <- readMVar stock
                                    putStrLn $ show m ++ "\n" ++ show s
```

Сначала может показаться, что код здесь очень непростой. Однако обновление товарного запаса и денег потребуется только в том случае, если для продажи будет достаточно товара. И если вы не сможете совершить покупку, вам не придется блокировать доступ к совместно используемой переменной `money`. Следовательно, нужно спланировать обе возможности и восстановить исходное значение переменной `stock`, если транзакция не удалась.

Кроме своей очевидной сложности, в коде есть и другие проблемы, связанные с несколькими одновременно выполняемыми сценариями. Может получиться так, что один из потоков `updateMoneyAndStock` получит доступ к переменной `stock`, а затем один из потоков `printMoneyAndStock` получит доступ к переменной `money`. В этот момент блокируется вся работа: обновляющий поток должен быть заблокирован, поскольку он не может завладеть переменной `money`, а поток, выводящий информацию, не может продолжить выполнение, поскольку ему отказано в доступе к переменной `stock`. Это типичный пример *мертвой блокировки* (deadlocking).

Еще одна проблема может проявиться в следующем случае, когда есть два обновляющих потока, *U*1 и *U*2, и один считывающий поток *R*. Вполне возможна ситуация, когда *U*1 обновляет значение переменной `money` и сразу же после этого *R* считывает значение этой переменной, получая сведения о деньгах после продажи единицы товара в потоке *U*1. Но когда *U*1 получает возможность возобновить выполнение, выполняется также весь поток *U*2. К этому времени переменная `stock` будет содержать изменения как от *U*1, так и от *U*2, и в результате *R* получит информацию о `stock`, которая не согласуется со значением, полученным от переменной `money`. В этом случае проблема состоит в том, что поток получает *несогласованную картину происходящего*.

Обе эти проблемы типичны для тех систем, у которых имеется множество агентов обновления и запроса данных, работающих одновременно. Наилучшим примером служат системы управления базами данных. Решение приходит в виде *транзакций*, где под транзакцией понимается вычисление, гарантированно запускаемое абсолютно независимо от других транзакций и имеющее согласованное представление данных. Транзакции создают иллюзию того, что все вычисления происходят внутри базы данных в атомарном блоке и целостность данных гарантируется.

В Haskell-программировании для реализации этой идеи служит пакет `stm`. Используя имеющуюся там библиотеку, можно определять блоки кода, запускаемые системой как атомарные единицы. В коде каждая транзакция преобразуется в вы-

числения внутри монады STM. Имя монады является акронимом от названия технологии (Software Transactional Memory — программная транзакционная память). Эта технология предназначена для реализации транзакций, используемых библиотекой. Рассмотрим в качестве примера версию обновляющего потока, в которой вместо MVars используется монада STM:

```
import Control.Concurrent.STM

updateMoneyAndStockStm :: Eq a => a -> Integer -> TVar Integer -> TVar
[(a,Integer)] -> STM ()
updateMoneyAndStockStm product price money stock =
  do s <- readTVar stock
     let Just productNo = lookup product s
     if productNo > 0
       then do m <- readTVar money
               let newS = map (\(k,v) -> if k == product then (k,v-1) else (k,v)) s
               writeTVar money (m + price) >> writeTVar stock newS
       else return ()
```

Когда вместо MVars используется stm, нужно задействовать переменные TVar. В отличие от ранее применяемых переменных переменные TVar могут считываться и записываться столько раз, сколько нужно. Следовательно, отпадает необходимость записывать обратно значение stock, если покупка не может быть произведена.

Вычисления в монаде STM производятся не напрямую, а через обязательный вызов функции atomically, которая преобразует транзакцию в монаду ввода-вывода IO. Например, чтобы пять раз с задержкой выполнить обновляющую транзакцию, нужно изменить функцию main:

```
main :: IO ()
main = do v <- newTVarIO 10000
          s <- newTVarIO [("a",7)]
          forkDelay 5 $ atomically $ updateMoneyAndStockStm "a" 1000 v s
          _ <- getLine  -- для ожидания завершения
          return ()
```

Наличие такой функции, как atomically, дает большое преимущество, заключающееся в возможности отделения тех частей кода, которые должны выполняться в виде транзакций, от тех частей, для которых это не нужно. Это имеет важное значение для достижения высокой производительности: гарантии «транзакционности» обходятся дорого, поэтому использовать транзакции нужно по минимуму.

Откат транзакций

При работе с базами данных зачастую возникают ситуации, в которых ваша текущая транзакция не может быть выполнена. Обычно такое случается при учете ограничений, которые должны применяться к вашим данным. Например, продажа товара из магазинных запасов может быть произведена, только если товар есть в наличии. При прерывании транзакции нужно, чтобы все вернулось на свои места, как будто не было никаких вычислений. Эта операция называется *откатом* (rollback).

Пакет `stm` обеспечивает не только гарантированную атомарность транзакций в Haskell, но и возможность отката всего, что сделано каким-то фрагментом кода. Чтобы указать на то, что транзакция не может быть продолжена, нужно использовать функцию `retry`. Рассмотрим, к примеру, сценарий, в котором клиент хочет рассчитаться картой: сначала нужно проверить, работает ли система обработки карт. В случае отрицательного ответа продолжение невозможно:

```
payByCard :: Eq a => a -> Integer -> TVar Integer -> TVar [(a,Integer)] -> STM ()
payByCard product price money stock =
  do working <- isCardSystemWorking
     if not working
     then retry
     else updateMoneyAndStockStm product price money stock

isCardSystemWorking :: STM Bool
isCardSystemWorking = ...  -- код проверки состояния системы работы с картами опущен
```

Код, использующий функцию `retry`, обладает особым поведением. Согласно первому описанию, транзакция выполняется снова и снова, пока, наконец, не будет найден сценарий, в котором она пройдет успешно. Конечно, это поведение избыточно. Внутри `stm` отслеживаются переменные `TVar`, воздействующие на транзакцию, и код выполняется снова, только если какая-либо из переменных изменяется. Отсутствие необходимости реализовывать эту функциональность самостоятельно делает код более модульным и простым в поддержке.

Еще одним свойством, на которое указывает предыдущий пример, является *композиционность* (compositionality) транзакций. Поскольку транзакция — это всего лишь значение монады `STM`, можно объединить несколько транзакций, создав более крупную транзакцию. В нашем примере механизмы проверки системы обработки карт, а также обновления денег и товарного запаса определены отдельно друг от друга, а затем объединены для создания более крупной транзакции `payByCard`.

Наряду с тем что `retry` является весьма эффективным средством, в некоторых случаях может быть удобнее не ждать, пока переменные изменятся и инварианты начнут вас устраивать. Для таких случаев `stm` предлагает комбинатор `orElse`. Чаще всего выражение `t1 `orElse` t2` ведет себя как `t1`. Но если в `t1` вызывается функция `retry`, эффект от `t1` откатывается и запускается `t2`. При успешном завершении `t2` никаких других действий не производится. Если в `t2` также вызывается функция `retry`, перезапускается *вся* транзакция `t1 `orElse` t2`.

В следующем примере комбинатор `orElse` используется для реализации поведения, предусматривающего попытку сначала рассчитаться по карте, а затем, если это не сработает, инициировать транзакцию для расчета наличностью:

```
pay :: Eq a => a -> Integer -> TVar Integer -> TVar [(a,Integer)] -> STM ()
pay product price money stock = payByCard product price money stock `orElse`
                                payByCash product price money stock

payByCash :: Eq a => a -> Integer -> TVar Integer -> TVar [(a,Integer)] -> STM ()
payByCash = ...  -- код запроса наличных опущен
```

В упражнении 8.2 можно воспользоваться нашими знаниями о транзакциях для системы машин времени.

УПРАЖНЕНИЕ 8.2. ПУТЕШЕСТВИЯ ВО ВРЕМЕНИ

Магазин машин времени предоставляет также сервис путешествий во времени. Но существует ряд ограничений, которые должны соблюдаться клиентами: одновременно могут путешествовать только *n* человек (поскольку компания владеет только n машинами времени), а в одном и том же году ни в коем случае не должны одновременно оказаться два человека.

Разработайте небольшое приложение, в котором клиенты моделируются в разных программных потоках, а ограничения всегда соблюдаются благодаря аккуратному использованию библиотеки stm. Подсказка: для сохранения тех лет, в которые попадают путешественники, воспользуйтесь переменной TVar, а для блокировки путешествий, не соответствующих правилам, — функцией retry.

Очереди производителей и потребителей

До сих пор наше основное внимание было сосредоточено на программных потоках, обменивающихся данными через общие переменные. Однако в мире одновременно выполняемых заданий существует множество других способов совместной работы с данными двух потоков. В этом разделе показано, как для реализации модели производителя-потребителя может использоваться очередь.

К примеру, один из способов построения архитектуры магазина предполагает наличие программных потоков для нескольких внешних и только одного внутреннего интерфейса. Внешние интерфейсы отвечают за запрос всей информации, которая может понадобиться для выполнения транзакции. Но они не отвечают за обработку заказов — эту ответственность берет на себя внутренний интерфейс.

Если бы для реализации данного решения можно было использовать только переменные `TVar`, для нас наступили бы трудные времена. Нам понадобилось бы конкретное количество допустимых внешних интерфейсов с возможностью обмена данными между ними, при этом внутренний интерфейс должен был бы постоянно отслеживать, нет ли в каких-либо из этих переменных новой информации. Более подходящим решением является отказ от переменных `TVar` и переход к *очереди* (queue).

Пакет `stm` предоставляет несколько видов очередей. Самым простым из видов является очередь `TQueue`, поместить в которую новый элемент можно с помощью функции `writeTQueue`. Эта очередь не налагает каких-либо ограничений на количество элементов, ожидающих в очереди (кроме очевидных ограничений, касающихся доступной в системе памяти), следовательно, функция `writeTQueue` никогда не будет блокировать программный поток. Обратная операция по извлечению из очереди первого элемента выполняется функцией `readTQueue`. Если очередь пуста, программный поток блокируется.

В данной модели внешний интерфейс действует как *производитель* (producer): он создает для очереди новые элементы, а внутренний интерфейс действует как *потребитель* (consumer): он получает из очереди информацию. Реализация всей картины с помощью очередей может выглядеть следующим образом:

```
import Control.Monad

main = do q <- newTQueueIO
  forkIO $ backend q                    -- создание 1 внутреннего интерфейса
  replicateM_ 10 $ forkIO (frontend q) -- создание 10 внешних интерфейсов
  _ <- getLine
  return ()

backend :: TQueue (String,Integer) -> IO ()
backend q = do
  m <- newTVarIO 10000
  s <- newTVarIO [("a",7)]
  forever $ atomically $ do (product,price) <- readTQueue q
                            pay product price m s

frontend :: TQueue (String,Integer) -> IO ()
frontend q = do (product,price) <- ...  -- получение информации о покупке от клиента
                atomically $ writeTQueue q (product,price)
```

Другие виды очередей можно классифицировать по двум измерениям. Эти два измерения характеризуются следующим образом:

- *Ограничена* (bounded) очередь по размеру или нет. Если очередь ограничена, для ее создания в качестве аргумента нужно указывать максимальное количество элементов. При вызове соответствующей функции `write`, если очередь заполнена, поток блокируется до тех пор, пока не появится дополнительное свободное место.
- Является очередь *закрываемой* (closable) или нет. Очередь, которая закрывается, не может больше получать элементы: когда такое происходит, каждый вызов функции `write` полностью игнорируется, а при каждом вызове функции `read` возвращается значение `Nothing`. Заметьте, что поведение закрытой и пустой очередей абсолютно разное.

Эти два измерения могут сочетаться четырьмя различными способами, как показано в табл. 8.1. В таблице также указан пакет, в котором можно найти каждую из очередей.

Таблица 8.1. Типы STM-очередей

	Неограниченная	**Ограниченная**
Незакрываемая	TQueue (пакет stm)	TBQueue (пакет stm)
Закрываемая	TMQueue (пакет stm-chans)	TBMQueue (пакет stm-chans)

Очереди могут пригодиться при разработке системы из упражнения 8.2. А в упражнении 8.3 от вас требуется воспользоваться очередью для более оперативного обслуживания клиентов в магазине.

УПРАЖНЕНИЕ 8.3. ВЫСТРАИВАНИЕ ПУТЕШЕСТВЕННИКОВ В ОЧЕРЕДЬ

В предыдущем упражнении все клиенты одновременно пытались получить доступ к конечному количеству машин времени. Это могло стать причиной возникновения проблемы равного доступа, поскольку stm не гарантирует пробуждения какого-то конкретного потока при использовании функции retry, если в режиме ожидания находится сразу несколько потоков.

Альтернативное решение строится на использовании очереди из клиентских запросов, по которым основной поток выделяет машины времени по мере их освобождения. Реализуете данное решение с помощью очереди TBQueue.

Облачные решения для Haskell

В этом, последнем, разделе мы познакомимся с миром облачных решений для Haskell. Этот мир представляет собой набор пакетов, предназначенных для решения вопросов *программирования одновременных и распределенных вычислений* с помощью *модели акторов* (actor model). В этом отношении большое влияние на облачные решения для Haskell оказал язык программирования Erlang. Кроме того, в облачных решениях предпринята попытка обеспечения согласованного представления независимо от используемого транспортного уровня.

При применении модели акторов программа строится вокруг *акторов* (которые в облачных решениях представлены типом `Process`). Каждый актор абсолютно независим от других: акторы вообще не используют какое-либо общее состояние — в этом они радикально отличаются от других подходов к решению проблемы одновременности. Тем не менее акторы могут обмениваться данными путем отправки и получения *сообщений* (messages). Соответственно, особую важность приобретают две операции: отправка сообщения другому актору и ожидание получения сообщения. Во втором случае доступно несколько вариантов: можно ожидать только в течение определенного времени, можно ожидать только сообщения определенного профиля и т. д.

Обмен данными посредством сообщений носит асинхронный характер, поэтому никаких гарантий насчет очередности получения сообщений не дается. Однако если сообщение получено при отсутствии активного ожидания его получения, оно помещается в *очередь сообщений* и становится доступным по следующему запросу. Это дает возможность разработки актора, который поочередно реагирует на сообщения, гарантируя, что ни одно из сообщений не будет потеряно.

Использование этой модели позволяет распределять акторов по различным компьютерам в сети, а не выполнять их все в рамках одного процесса. Если акторы смогут совместно использовать состояние, появится возможность распределить работу между различными компьютерами без лишних усилий или согласований между акторами. Поскольку какая-либо очередность сообщений не гарантируется, эта модель помогает создавать программы, устойчивые к задержкам в сети.

Поиск галактик

В этом разделе мы займемся имитацией пространственно-временного исследования. Система будет состоять из нескольких акторов: их основная часть займется поиском новых галактик. Кроме путешественников будет еще специалист, сообщающий путешественникам, когда приступать к исследованиям и когда возвращать результаты исследований. Это типичный пример *системы «главный–подчиненный»* (master-slave system).

При разработке модели акторов весьма полезно заранее определить виды сообщений, которыми смогут обмениваться акторы. В данном случае определяются два таких вида: один — где специалист предписывает путешественникам приступить к исследованиям, и другой — предназначенный для объявления путешественниками об открытии новой галактики. В облачных решениях для Haskell сообщения представляются простыми Haskell-значениями, поэтому давайте определим тип данных:

```
data GalaxyMessage = LookForGalaxy | GalaxyFound String
```

Кроме того, пакеты, которые будут применяться, требуют, чтобы каждый тип данных, используемый в качестве сообщения, был экземпляром как класса `Typeable`, так и класса `Binary`, что позволит запрашивать тип значения в процессе выполнения и проводить сериализацию значения в двоичную форму соответственно. Приятно то, что при наличии нескольких расширений и механизма извлечения весь необходимый код способен сгенерировать компилятор. Чтобы удовлетворить потребности облачных решений, нужно написать следующий код:

```
{-# LANGUAGE DeriveDataTypeable, DeriveGeneric #-}
import Data.Binary   (Binary)
import Data.Typeable (Typeable)
import GHC.Generics  (Generic)

data GalaxyMessage = LookForGalaxy | GalaxyFound String
                   deriving (Typeable, Generic)
instance Binary GalaxyMessage
```

ВНИМАНИЕ

Есть одна известная ошибка, не позволяющая этому коду работать при использовании любой версии пакета binary ниже версии 0.6.3.0. Если придется столкнуться с каким-либо странным поведением этого кода, попробуйте добавить в Cabal-файл ограничение версии binary >= 0.6.3.0.

Базовая функциональность облачных решений для Haskell реализована в пакете `distributed-process`. Этот пакет нужно добавить в качестве зависимости в каждый проект, в котором вы собираетесь использовать эту модель обмена данными. Чуть позже будет показано, что в качестве еще одной зависимости обычно добавляется транспортный уровень, который определяет способ обмена данными.

Самым простым актором является путешественник. Он будет ждать сообщение `LookForGalaxy`, выполнять работу и возвращать результаты. Ожидание сообщения реализуется с помощью функции `expect`, которая блокирует актора, если в очереди нет ни одного сообщения. Это делается с помощью следующего кода:

```
import Control.Distributed.Process

traveller :: Process ()
traveller = do LookForGalaxy <- expect
               -- Поиск галактик в пространстве-времени
               send ?? (GalaxyFound "Andromeda")
```

Но тут возникает проблема: в тот момент, когда нужно вернуть результат, еще не известно, с каким актором следует контактировать. Это показано в коде двумя

знаками вопроса в том месте, где должен быть помещен идентификатор актора. Обычным решением является изменение типа данных сообщения с целью включения такого идентификатора. Тогда можно будет написать работоспособный код:

```
data GalaxyMessage = LookForGalaxy ProcessId | GalaxyFound String

traveller :: Process ()
traveller = do LookForGalaxy master <- expect
               -- Поиск галактик в пространстве-времени
               send master (GalaxyFound "Andromeda")
```

Еще одним актором является `master`, отвечающий за выдачу заданий путешественникам. Здесь механизм облачных решений для Haskell предлагает нам сюрприз: можно отправить не только сообщение, но и выполняемую функцию. Для этого нужно последовательно пройти три шага:

- Нужно узнать систему, в которой создается новый актор. Каждая компьютерная система, задействованная в сети с актором, в облачных решениях для Haskell называется *узлом* (node). В некотором смысле узлы предоставляют материальную почву для создания акторов и управления ими.
- Нужно сообщить компилятору, что вы можете отправить конкретную функцию, чтобы компилятор сгенерировал специальный код. Это делается включением вызова функции `remotable` со списком функций, которые могут быть отправлены. В данном случае вызов будет читаться как `remotable ['traveller]`. Следует знать, что этот вызов должен располагаться после определения функции, присутствующей в списке, но перед кодом любого актора, пытающегося отправить функцию.
- И наконец, в узле с помощью функции `spawn` должен быть создан новый актор, возвращающий идентификатор. Однако вы не можете дать функции имя актора напрямую, его нужно заключить в создаваемое *замыкание* (closure). Для этого есть два способа: если актор `f` не получает никаких аргументов, осуществляется вызов `$(mkStaticClosure f)`, в противном случае — вызов `$(mkClosure f)`.

Для решения этой задачи все функции, `remotable`, `mkStaticClosure` и `mkClosure`, задействуют механизм Template Haskell. Поэтому при их использовании нужно включить в начало файла строку `{-# LANGUAGE TemplateHaskell #-}`.

Показанный далее код актора `master` использует несколько узлов и создает новый актор `traveler` в каждом из тех узлов, которому отправляет исходное сообщение `LookForGalaxy`. Следует заметить, что идентификатор актора `master` можно получить с помощью функции `getSelfPid`. Затем запускается бесконечный цикл ожидания сообщений от путешественников и для каждой найденной галактики с помощью функции `say` выводится журнальное сообщение:

```
import Control.Distributed.Process.Closure

master :: [NodeId] -> Process ()
master nodes =
  do myPid <- getSelfPid
     mapM_ (\node -> do pid <- spawn node $(mkStaticClosure 'traveller)
```

```
                    send pid (LookForGalaxy myPid))
        nodes
    forever $ do GalaxyFound g <- expect
                 say $ "Found galaxy: " ++ g
```

Последний шаг заключается в создании функции `main`, которая будет создавать узел. Здесь нужно решить, как должен вестись обмен данными, то есть выбрать *транспортный уровень* (transport layer) для ваших сообщений. Если у вас нет сложной сетевой схемы или вы просто хотите, чтобы все узлы были на одном компьютере, лучшим вариантом станет пакет `distributed-process-simplelocalnet`. Этот уровень использует для сообщений протокол TCP и обеспечивает автоматическое обнаружение других узлов сети через широковещательный протокол UDP.

Уровень `simplelocalnet` настраивается очень просто. Нужно лишь инициализировать его посредством функции `initializeBackend`, а затем вызвать функцию `startMaster` или `startSlave` в зависимости от той роли, которую нужно получить. Если вы решите сделать узел главным, транспортный уровень обнаружит все подчиненные узлы, доступные в сети, и передаст их актору в виде аргументов. В данном случае, чтобы решить, какое поведение требуется, в коде используются аргументы командной строки (которые могут быть получены посредством вызова функции `getArgs` модуля `System.Environment`):

```
import Control.Distributed.Process.Node (initRemoteTable)
import Control.Distributed.Process.Backend.SimpleLocalnet
import System.Environment

main :: IO ()
main = do args <- getArgs
    case args of
      ["master", host, port] -> do
        backend <- initializeBackend host port (Main.__remoteTable initRemoteTable)
        startMaster backend master
      ["traveller", host, port] -> do
        backend <- initializeBackend host port (Main.__remoteTable initRemoteTable)
        startSlave backend
      _ -> do putStrLn "Unknown parameters"
```

Изучив код, можно заметить, что этот транспортный уровень нуждается в указании хоста и порта, которые будут использоваться при обмене данными. После компиляции всего кода сеть акторов можно протестировать:

```
$ ./dist/build/chapter8-actors/chapter8-actors traveller localhost 8086
$ ./dist/build/chapter8-actors/chapter8-actors master localhost 8087
Sat Dec  7 08:23:18 UTC 2013 pid://localhost:8086:0:3: Found galaxy: Andromeda
```

ВНИМАНИЕ

При вызове функции startMaster внутренний интерфейс simplelocalnet пытается всего лишь найти узлы. Следовательно, до запуска главного узла нужно запустить все подчиненные узлы.

Иногда приложение нуждается в более чем одном виде сообщений. Если вам удалось определить все виды сообщений в единственном типе данных, можете вы-

звать функцию `expect`, а затем выполнить сопоставление с образцом в отношении полученного сообщения. Но в других случаях разные сообщения представляются с различными типами данных. Этот сценарий также поддерживается облачными решениями для Haskell.

Давайте, к примеру, добавим новое сообщение, которое мог бы отправить путешественник: он слишком близко подошел к пространственно-временному тоннелю и был им поглощен. Определение имеет следующий вид:

```
data WormHoleMessage = LostInWormHole
                     deriving (Typeable, Generic)
instance Binary WormHoleMessage
```

Теперь вместо постоянного поиска галактик путешественник случайно обнаружит это обстоятельство или заблудится. Сообщение другого вида будет отправлено в каждом случае:

```
import System.Random

traveller :: Process ()
traveller = do LookForGalaxy m <- expect
               b <- liftIO $ randomIO  -- генерирование случайного булева значения
               if b
           then send m (GalaxyFound "Andromeda")
else send m LostInWormHole
```

Главный актор выполнит такой же код, но должен будет изменить рабочий вызов уточненной версией: `receiveWait`. Эта функция получает список возможных образцов сообщений, каждый из которых представлен функцией `match`. Будет выбран первый же вариант, тип которого совпадает с аргументом функции `match`:

```
forever $ do receiveWait
[ match $ \(GalaxyFound g) -> say $ "Found galaxy: " ++ g
, match $ \LostInWormHole -> say "Lost in wormhole"
]
```

В пакете `distributed-process` определяются две другие функции, способные еще лучше уточнить получение сообщений. Вместо функции `receiveWait` можно вызвать функцию `receiveTimeout` для обозначения максимального количества времени ожидания актором.

Второй способ, который может быть более явным для захвата сообщений, состоит в замене функции `match` функцией `matchIf` или `matchUnknown`. В соответствии с предыдущими утверждениями на основе только типа полученного сообщения функция `match` решает, по какой ветви пойти. Этот подход может вызвать ряд проблем: в показанном примере при получении главным потоком сообщения `LookForGalaxy` будет выбрана первая ветвь (`LookForGalaxy` и `GalaxyFound` являются конструкторами одного и того же типа), но функция даст сбой из-за несоответствия образцу. Использование функции `matchIf` позволит включить дополнительное свойство, при котором нужное сообщение должно быть захвачено определенной ветвью. Обратный сценарий состоит в соответствии любому возможному сообщению независимо от его типа: он реализуется посредством функции `matchUnknown`.

С помощью модели акторов могут быть выражены многие системы одновременного выполнения заданий. Подойдет она и для системы путешествий, разработанной в упражнении 8.2. В упражнении 8.4 вам для управления такой системой потребуется воспользоваться пакетом `distributed-process`.

УПРАЖНЕНИЕ 8.4. РАСПРЕДЕЛЕНИЕ ПУТЕШЕСТВЕННИКОВ

На базе облачных решений для Haskell разработайте систему из упражнения 8.2 для групп путешественников, обменивающихся сообщениями с центральным главным актором. Отметьте, что вы можете неявно использовать механизм очередей, чтобы сделать реализацию проще и понятнее.

Поиск типизированных галактик

Обмен данными с помощью сообщений является весьма гибким механизмом, но у него есть два недостатка. Прежде всего, тип сообщения, которое может поступить, предварительно не указывается, следовательно, вам нужно выполнять динамические проверки в функции `expect` или `match`. Это влияет на производительность, а также не соответствует используемой в Haskell философии строгой типизации. Вторая проблема состоит в том, что обработка диалога, то есть потока сообщений между двумя конкретными акторами, требует большого объема шаблонного кода: постоянной и повсеместной передачи идентификаторов, идентификации каждого диалога, если с одним и тем же актором может общаться более одного корреспондента, и т. д.

Для решения обеих проблем пакет облачных решений для Haskell предлагает *типизированные каналы* (typed channels). Канал в этом пакете является однонаправленным и представляется двумя своими конечными точками. Выражение `SendPort t` определяет место, откуда актор может отправлять сообщения только типа `t`, а выражение `ReceivePort t` — место, откуда они могут считываться.

Кроме того, пакет облачных решений для Haskell позволяет отправить выражение `SendPort t` в качестве части сообщения. Таким образом, акторы *A* и *B* могут приступить к двунаправленному диалогу, воспользовавшись следующей последовательностью шагов:

- Актор *A* создает новый канал `ChA` и отправляет информацию `SendPort` посредством обычного сообщения.
- Актор *B* получает информацию об этом канале, создает новый канал `ChB` и отправляет по каналу `ChA` информацию `SendPort` о канале `ChB`.
- Актор *A* получает это сообщение. В результате у него теперь есть способ отправки и способ получения потока сообщений при его диалоге с актором *B*.

Разумеется, если требуется только однонаправленное общение, для обмена данными достаточно лишь сообщения с информацией `SendPort` от актора *A*.

Давайте изменим пример, касающийся разведки галактик, чтобы воспользоваться каналами. В предварительном сообщении `LookForGalaxy` больше не будет отправляться идентификатор актора, а вместо него мы отправим информацию `SendPort`.

Следует отметить, что этот порт определяет тип сообщений, которые будут через него поступать:

```
data GalaxyMessage = LookForGalaxy (SendPort GalaxyMessage) | GalaxyFound String
```

Изменения в реализации актора-путешественника касаются лишь замены функции `send` функцией `sendChan`:

```
traveller :: Process ()
traveller = do LookForGalaxy sendPort <- expect
               sendChan sendPort (GalaxyFound "Andromeda")
```

И наконец, главный актор нуждается в создании канала и отправки его после создания. Функция `expect` заменяется функцией `receiveChan`, которая получает сообщения от `ReceivePort`:

```
master :: [NodeId] -> Process ()
master =
  mapM_ $ \node -> do pid <- spawn node $(mkStaticClosure 'traveller)
                      (sendPort, rcvPort) <- newChan
                       send pid (LookForGalaxy sendPort)
                       GalaxyFound g <- receiveChan rcvPort
                       say $ "Found galaxy: " ++ g
```

Дополнительные возможности

Этот пример разведки галактик показал лишь самые базовые возможности облачных решений для Haskell. Отказоустойчивость, в основе которой лежат идеи языка Erlang, встроена в пакет `distributed-process`. Когда у какого-либо актора возникает критическая ошибка, его работа прекращается. Однако любой актор может учредить *монитор* (monitor), чтобы получать уведомления о подобных случаях. Например, главный актор может при таких условиях перезапустить нового подчиненного актора. Дополнительную документацию относительно данной возможности можно найти на веб-странице облачных решений для Haskell (http://haskell-distributed.github.io).

Единственным транспортным уровнем, использованным в этом разделе, был `simplelocalnet`. На момент написания книги также поддерживались сети Azure, а в разработке находились многие другие уровни. Все эти транспортные уровни основаны на пакете `network-transport`, который абстрагирует концепцию соединения от используемого протокола. Этот пакет может стать удачным выбором, если свойств соединений, о которых здесь рассказывалось, вам будет достаточно (без отправки дополнительного сообщения, отказоустойчивости и средств завершения передачи).

Хотя описанная в этом разделе конфигурация «главный–подчиненный» встречается довольно часто, существуют и другие образцы обмена данными, которые могут использоваться с облачными решениями для Haskell. Большой объем информации об этом и других образцах можно найти в серии публикаций в постах блога по адресу http://www.well-typed.com/blog/71.

Выводы

В данной главе мы изучали пакеты экосистемы Haskell, связанные с концепциями параллелизма, одновременности и распределенности. В частности:

- Монада `Par` реализует простой параллелизм посредством *фьючерсов*, позволяющих начать параллельное выполнение вычислений и чуть позже запросить результаты.
- Монады позволяют инициировать параллельные вычисления на базе схемы *потока данных*, в которой зависимости определяются через переменные `IVar`.
- Базовая *одновременность* может быть достигнута в GHC посредством функции `forkIO`, обеспечивающей создание новых программных потоков, и переменных `MVar`, предназначенных для совместного использования информации.
- Иногда возникает необходимость обеспечить длительный доступ к общим ресурсам в виде атомарных операций. В таких случаях посредством библиотеки `stm` можно использовать *транзакции*.
- Кроме простых переменных `TVar` библиотека `stm` предоставляет абстракции для нескольких типов *очередей*. Основное внимание в примерах было уделено использованию одной из них — неограниченной и незакрываемой очереди `TQueue`.
- В конце главы мы изучили основы *распределенного* программирования с использованием облачных решений для Haskell.

Часть III
ИСПОЛЬЗОВАНИЕ РЕСУРСОВ

Глава 9. Работа с файлами: ввод-вывод и библиотека conduit

Глава 10. Создание и синтаксический разбор текста

Глава 11. Безопасный доступ к базам данных

Глава 12. Веб-приложения

Работа с файлами: ввод-вывод и библиотека conduit

9

В первой и второй частях книги мы рассмотрели основы чистых вычислений, а также узнали, что чистые вычисления помогают обеспечить параллелизм и распределенность в коде. Однако иногда приходится покидать область чистых вычислений, чтобы оказаться в диком мире побочных эффектов. В первую очередь мы посмотрим, как в Haskell осуществляются обычные ввод и вывод данных. Для начала вводить информацию мы будем с консоли, а выводить на экран. Затем мы научимся использовать долговременные средства хранения данных, считывая файлы с диска и записывая их на диск.

Вычисления с побочными эффектами связаны с различными неприятностями: данные могут повреждаться, сетевое соединение разрываться. Haskell поддерживает механизм исключений, сигнализирующий о подобных ситуациях и позволяющий на них реагировать. В чистых вычислениях ошибочные сценарии мы обрабатывали с помощью типов `Maybe` и `MonadPlus`: к концу данной главы мы выясним, как связаны между собой различные способы обработки ошибок в Haskell.

Используемая в Haskell модель ленивых вычислений не лишена проблем, одна из которых заключается в присущей этой модели непредсказуемости ввода и вывода. Для решения этой проблемы Haskell-сообщество предлагает несколько библиотек потокового ввода-вывода. В частности, в данной главе мы рассмотрим библиотеку `conduit` и некоторые варианты ее применения, касающиеся обработки файлов и работы с сетью. После усвоения всей этой информации мы научимся сохранять данные о клиентах и товарах на диске, используя двоичный формат сериализации. Для этого предназначена также библиотека `cereal`.

Базовые ввод и вывод

Чтобы начать наше путешествие в мир побочных эффектов Haskell-программ, давайте организуем простое взаимодействие с консолью. Эти небольшие примеры помогут нам лучше понять концепции, связанные с побочными эффектами. Первый рассматриваемый фрагмент кода является исполняемой программой, которая просто выводит на консоль приветствие новичкам «Hello Beginning Haskell!»:

```
module Main where

main :: IO()
main = putStrLn "Hello Beginning Haskell!"
```

Второй пример чуть сложнее. В нем программа запрашивает место назначения и по заданному имени определяет, в какую точку во времени вы должны попасть. Здесь использован весьма простой алгоритм, причем важной частью кода является то, как информация поступает от пользователя и проходит по программе:

```
main = do putStrLn "Where do you want to travel?"
          place <- getLine
          let year = (length place) * 10
          putStrLn $ "You should travel to year " ++ show year
```

Если запустить программу, экран будет выглядеть так, как показано в следующем примере. Полужирным шрифтом выделена та строка, которая должна вводиться, обычный шрифт обозначает выводимые программой данные:

```
Where do you want to travel?
India
You should travel to year 50
```

БУФЕРИЗАЦИЯ

В некоторых системах, особенно в Windows, при выполнении данной программы может обнаружиться полное отсутствие какого-либо вывода. Это связано с буферизацией. Для повышения эффективности отправленная вами в файл (или на консоль) информация напрямую не записывается (или не выводится), а помещается в буфер, пока в нем не соберется определенный объем данных. Изменить способ буферизации можно с помощью функции hSetBuffering. Чаще всего она используется, чтобы заставить систему выдавать контент после каждого символа новой строки, при этом применяется следующий код:

```
import System.IO

-- включение построчной буферизации
main = do hSetBuffering stdout LineBuffering
-- продолжение кода
```

Для hSetBuffering доступно еще значение NoBuffering, которое заставляет систему полностью отказаться от буферизации. Однако при его использовании следует проявлять осторожность, поскольку из-за отсутствия буферизации при работе консольных функций в Windows могут возникать проблемы.

Интересно посмотреть на сигнатуру функций, занятых в примерах. Проще всего ее получить по запросу у интерпретатора с помощью его команды `:t`:

```
*Chapter9.BasicInputOutput> :t putStrLn
putStrLn :: String -> IO ()
*Chapter9.BasicInputOutput> :t getLine
getLine :: IO String
```

Использованные do-нотация и типы данных, заключенные в оболочке другого типа (в данном случае — `IO`), наводят на мысль, что на самом деле работа ведется внутри монады. В предыдущих главах было показано, что монада `State` представляет те вычисления, в которых осуществляется перенос состояния, монада `Maybe` — вычисления, которые могут потерпеть неудачу, и т. д. Вполне возможно, что монада `IO`,

вводимая в определенный фрагмент кода, ведет к появлению побочных эффектов. В показанном примере она служит для вывода информации и ее получения с консоли.

Между `IO` и остальными монадами, которые мы уже рассматривали, есть одно важное различие. Для любой из других монад существовал некий способ проверки получаемого значения (например, для `Maybe` использовалось сопоставление с образцом) или выполнения монады (с помощью функции, подобной `runState`). Таким образом, к контексту можно было вернуться и без соответствующей монады.

Однако у монады `IO` *нет такого аварийного люка*. Не существует операции, которая превращает значение `IO t` в простое значение `t`[1]. Именно так Haskell ставит барьер между вычислениями, которые приводят к побочным эффектам и поэтому должны работать внутри монады `IO`, и всеми другими вычислениями. Вам нужна уверенность в том, что функция без `IO` в своей сигнатуре чиста, свободна от побочных эффектов и обладает ссылочной прозрачностью. Больше того, каждый вызов `IO` в конкретном приложении должен исходить от базовой функции `main`, которая имеет тип `IO ()`.

ПРИМЕЧАНИЕ

Для компилятора Haskell чистое выражение должно рассматриваться не как выражение, приводящее к побочным эффектам и находящееся в монаде IO. Однако с программистской точки зрения IO не отличается от любой другой монады. Просто она вводит в свои вычисления побочные эффекты.

Имеющийся в Haskell модуль `Prelude` предлагает несколько функций для взаимодействия с консолью. Кроме ранее упомянутой функции `putStrLn`, выводящей строку текста, а затем начинающей новую строку, в нашем распоряжении есть функции `putStr` (вывод строки, но без завершающего символа новой строки) и `putChar` (вывод на экран одного символа). Для весьма распространенного случая, когда данные для вывода на экран еще не являются строкой, но могут быть в нее преобразованы с помощью функции `show`, можно воспользоваться функцией `print`. Рассмотрим небольшой пример, в котором у пользователя запрашиваются имя и фамилия, а система показывает значение типа `Client` (аналогично объявлению в главе 4):

```
main = do putStrLn "First name?"
          fName <- getLine
          putStrLn "Last name?"
          lName <- getLine
          putChar '>' >> putChar ' '
          print $ Person fName lName
```

Все эти функции вывода имеют в качестве возвращаемого типа `IO ()`. Вспомним, что `()` является «блочным типом», причем существует только одно значение этого

[1] Это утверждение не вполне соответствует действительности: существуют способы выполнения подобных преобразований. Но это считается крайне опасным, поскольку может нарушить ряд инвариантов, на которых строится Haskell-код. Основной областью применения этого небезопасного преобразования является взаимодействие с языками, которые не делают различий между чистыми функциями и функциями с побочными эффектами.

типа, которое по недоразумению также названо `()`. В Haskell-программировании оно обычно используется для идентификации тех монадических вычислений, у которых нет возвращаемого значения, но которые вам интересны с точки зрения влияния на контекст. Кроме функции `putStr` и ей подобных нотацию `()` в типе возвращаемого значения применяют функции `tell` (из монады `Writer`) и `put` (из монады `State`).

Обратными по отношению к предыдущим функциям являются функции, получающие информацию от пользователя. Наиболее востребована функция `getLine`, назначение которой заключается в сборе всего ввода до момента нажатия пользователем клавиши Enter. Для получения посимвольного ввода может использоваться функция `getChar`. И наконец, иногда возникает потребность в получении всего ввода вплоть до маркера конца файла, что весьма типично при передаче данных между процессами в Unix-подобной оболочке. Такую возможность предоставляет функция `getContents`.

HASKELINE

Если планируется создание приложения командной строки, то простые функции, входящие в модуль Prelude, не смогут предложить конечному пользователю должного комфорта. В частности, при вводе команд ему могут потребоваться возможность выбора из истории команд или механизм автозавершения.

Для решения подобных задач предназначена специально разработанная библиотека haskeline. Ядром этой библиотеки является предоставляющий соответствующие средства монадический преобразователь InputT. Предыдущий пример, запрашивающий персональные данные, может быть переписан следующим образом:

```
import System.Console.Haskeline

main = runInputT defaultSettings $ do
  fName <- getInputLine "First name? "
  lName <- getInputLine "Last name? "
  case (fName, lName) of
    (Just f, Just l) -> outputStrLn $ show (Person f l)
    (_     , _     ) -> outputStrLn "I cannot identify you"
```

Одним из отличий от стандартной библиотеки Prelude является то, что функции ввода данных возвращают свои значения в оболочке монады Maybe, предвидя тем самым те случаи, при которых ввод может завершиться раньше ожидаемого момента.

Поскольку `IO` является монадой, мы можем использовать многочисленные функции, с которыми познакомились в главах 6 и 7. Следует также напомнить, что монада является функтором, поэтому непосредственно к `IO` можно применять функцию `fmap`. Например, можно переработать следующий вариант применения переменной `s`, поскольку она служит только для передачи информации в функцию `upperS`:

```
import Data.Char

main = do s <- getLine
          let upperS = map toUpper s
          putStrLn upperS >> putStrLn upperS
Into a more concise form which directly generates upperS:
main = do upperS <- fmap (map toUpper) getLine
          putStrLn upperS >> putStrLn upperS
```

Еще одной возможностью является аккумулирование информации с помощью функции `foldM`. Следующий код перебирает весь список клиентов и для каждого из клиентов запрашивает у пользователя информацию о том, нужно ли включать данного клиента в особый список наиболее важных персон:

```
import Control.Monad (foldM)

createVIPList :: Show a => [Client a] -> IO [Client a]
createVIPList = foldM (\lst c -> do
         putStrLn $ "\nShould " ++ show c ++ "be included in the VIP list? "
         answer <- getLine
         case answer of
           'Y':_ -> return $ c:lst
           _     -> return lst) []
```

Интересно выяснить, что получится, если значения типа `IO` окажутся *внутри* определенного контейнера. Например, может потребоваться создать список действий и на основе пользовательского ввода выполнить одно из них. Именно это делается с помощью следующего кода:

```
main = do actionName <- getLine
          case lookup actionName listOfActions of
            Just action -> action  -- выполнить действие
            Nothing     -> putStrLn "Unknown action"

listOfActions :: [(String, IO ())]
listOfActions = [
  ("greet", do name <- getLine
               putStrLn $ "Hello " ++ name),
  ("sum"  , do putStrLn "First number: "
               n1 <- fmap read getLine
               putStrLn "Second number:"
               n2 <- fmap read getLine
               putStrLn $ show n1 ++ "+" ++ show n2 ++ "=" ++ show (n1+n2))]
```

Важно поразмышлять над тем, чем выполнение такого кода отличается от ожидаемого. В большинстве языков программирования вызов функции `lookup` инициирует вычисление функции `listOfActions`. Затем происходят все вызовы функции `getLine` или `putStrLn`, поскольку они присутствуют в теле функции `listOfActions`. Однако в Haskell-коде побочные эффекты не проявляются до тех пор, пока действие, имеющее тип `IO t` для некоторого значения `t`, не будет извлечено из оболочки и запрошено, а это происходит в строке, выделенной полужирным шрифтом и помеченной комментарием «выполнить действие». Важным обстоятельством является то, что значения типа `IO`, как и функции, являются в Haskell «равноправными гражданами» и могут группироваться, передаваться в качестве аргументов и возвращаться точно так же, как и любые другие значения.

Случайность

Давайте возьмем паузу в отношении изучения ввода и вывода и рассмотрим проблему случайности в Haskell-коде. Для этого мы имитируем поломку машины времени в середине путешествия. При возникновении поломки путешественник

во времени переживает одно из наиболее ярких потрясений в своей жизни: он оказывается в случайном месте случайного времени, не имея представления о том, что он обнаружит, выйдя из машины времени.

В следующем коде используется функция `randomRIO`, входящая в модуль `System.Random` пакета `random`, которая имитирует случайную прогулку из исходной точки, то есть серию случайных перемещений во времени, осуществляемых сломавшейся машиной времени. Функция `randomRIO` нуждается в нижней и верхней границах получаемого значения, которые в данном случае установлены равными 0 и 3000. Поскольку путешествие может быть бесконечным, код просто выводит 10 начальных прыжков:

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Monad.Loops
import System.Random

main = do (initial :: Int) <- fmap read getLine
          jumps <- unfoldrM (\_ -> do next <- randomRIO (0, 3000)
                                      if next == initial
                                        then return Nothing
                                        else return $ Just (next, next))
                            initial
          print $ take 10 jumps
```

ПРИМЕЧАНИЕ

В предыдущем коде был использован монадический аналог развертки — функция unfoldrM. Однако в известном модуле Control.Monad вы ее не найдете. Здесь нужно импортировать модуль Control.Monad.Loop из пакета monad-loops. Это очень интересный модуль, который нужно добавить к своему списку инструментов монадического программирования.

Теперь, когда вы знаете, как считывать данные с консоли и записывать их в консоль, а также как генерировать случайные числа, можно попробовать разработать небольшую консольную игру. Именно это и требуется сделать при выполнении упражнения 9.1.

УПРАЖНЕНИЕ 9.1. ВЫИГРАЙТЕ ПУТЕШЕСТВИЕ ВО ВРЕМЕНИ

Разработайте небольшую игру, в которой можно выиграть путешествие во времени на одной из машин, имеющихся в магазине. Игра может генерировать случайное число в диапазоне между тремя и семнадцатью. Затем пользователю предлагается пять попыток на угадывание числа. Если число оказывается угаданным, программа показывает игроку поздравительное сообщение. В случае неудачи программа выводит сообщение, предлагающее пользователю сделать повторную попытку.

При разработке игры нужно подумать о том, как придать коду модульность и универсальность: диапазон чисел или количество попыток, дающихся на отгадывание числа, могут быть встроены в аргументы.

Предыдущий код работает в функции `main`, имеющей тип `IO ()`. Вызов функции `randomRIO` происходит не в блоке `let`, который указывает на то, что эта функция также работает в контекстах `IO`. Убедиться в этом можно, запросив у интерпретатора тип этой функции:

```
*Chapter9.Randomness> import System.Random
*Chapter9.Randomness System.Random> :t randomRIO
randomRIO :: Random a => (a, a) -> IO a
```

Давайте немного подумаем о том, почему случайность нуждается для своей работы в контексте `IO`. Главная причина состоит в том, что функция `randomRIO` *нарушает свойство ссылочной прозрачности* чистого Haskell-кода: не каждый вызов функции `randomRIO` вернет одинаковый результат. Этот факт означает, что компилятор может не справиться с выполнением определенной оптимизации. Например, если в чистом коде имеется фрагмент вроде `g (f x) (f x)`, он может быть заменен фрагментом `let h = f x in g h h`, в котором для сокращения объема работы реализован на один вызов `f` меньше. Попробуем вместо этого написать такой код:

```
do x <- randomRIO (0, 10)
   y <- randomRIO (0, 10)
   return g x y
```

Однако этот код нельзя заменить следующим:

```
do z <- randomRIO (0, 10)
   return g z z
```

Причина в том, что два вызова функции `randomRIO` могут вернуть разные случайные значения. Многие другие полезные свойства Haskell-кода в присутствии побочных эффектов теряются, поэтому вы не можете прибегнуть к таким средствам, как эквациональные рассуждения. Помимо удобства поддержки кода, это еще один повод для того, чтобы не смешивать чистый код и код с монадой `IO`.

Есть еще одна причина для возвращения функцией `randomRIO` результата в `IO`. Любой вызов этой функции должен обновить значение *глобального генератора случайных чисел* (global random generator), которое хранится системой в памяти. Фактически можно создавать чистые функции, работающие со случайными значениями, при условии предоставления исходного генератора случайных чисел, значение которого имеет тип `StdGen`. Соответствующие чистые функции возвращают случайное значение и генератор для следующего значения. Например, предыдущий код может быть «очищен» для работы со значениями типа `StdGen`:

```
import Data.List

getJumps :: StdGen -> Int -> [Int]
getJumps gen initial = unfoldr (\g -> let (next, nextG) = randomR (0, 3000) g
                                      in if next == initial
                                         then Nothing
                                         else Just (next, nextG))
                               gen
```

Значение типа `StdGen` можно либо создать с фиксированным начальным значением посредством `mkStdGen`, либо получить глобальное значение через `getStdGen`. Таким образом, исходный код получения случайного числа, который также выводит результат, может иметь следующий вид:

```
main = do (initial :: Int) <- fmap read getLine
          gen <- getStdGen
          print $ take 10 $ getJumps gen initial
```

В этом примере содержится весьма ценный урок: зачастую функции с побочными эффектами можно разбить на несколько чистых функций. Есть также управляющая функция, работающая в `IO` и занимающаяся организацией программных потоков между другими компонентами. Так ваш код будет проще поддерживать.

Работа с файлами

Теперь, когда мы узнали, как справляться с побочными эффектами и как обмениваться данными с консолью, нужно сделать следующий шаг и научиться читать информацию из места ее долговременного хранения и записывать ее в это место. Иными словами, нужно научиться считывать и записывать файлы. В некотором роде сами файлы превращаются в объекты, способные выполнять определенные операции, такие как перемещения файлов из одного места в другое или удаление файла с диска. В этом разделе нам предстоит углубленное изучение функций, обеспечивающих такие возможности в Haskell Platform.

Чтение и запись

Модуль `Prelude` предоставляет функции для операций с файлами: либо для считывания целой строки из файла, либо для ее записи в файл. Это такие функции, как `writeFile` или `appendFile`, осуществляющие вывод данных, и функция `readFile`, осуществляющая их ввод. Одной из доступных возможностей является считывание списка клиентов и принятие в отношении каждого из них решения, выиграл он путешествие во времени в определенную точку или нет (для этой второй части кода используется функция `randomRIO`). Основным предположением является то, что в каждой строке файла содержатся данные о клиенте, следовательно, можно воспользоваться функцией `lines` из модуля `Data.String`, которая выделяет строку между границами, обозначенными символами новой строки.

```
import Data.String

main = do clients <- fmap lines $ readFile "clients.db"
          clientsAndWinners <- mapM (\c -> do (winner :: Bool) <- randomIO
                                              (year   :: Int ) <- randomRIO (0,
3000)
                                              return (c, winner, year))
                                    clients
          writeFile "clientsWinners.db" $ concatMap show clientsAndWinners
```

Однако выбор только этих операций отрицательно сказывается на производительности: считываемая из файла информация целиком хранится в памяти, а данные, предназначенные для записи в файл, должны быть собраны в строку до начала записи. Зачастую требуется более гибкий контроль: например, может понадобиться считать только строку или файл либо записать информацию на диск по ходу выполнения кода, не дожидаясь завершения всего процесса. В таком случае следует обратиться к модулю `System.IO`.

Как и во многих других языках программирования, процесс работы с файлом предусматривает предварительное открытие *дескриптора* (handle) этого файла,

выполнение любой необходимой операции с файлом и закрытие доступа к файлу. В дескрипторе отслеживается вся внутренняя информация, которая может понадобиться системе для работы с файлом.

Первый шаг реализует функция `openFile`. Аргументами этой функции служат путь к файлу (в документации показано, что этот аргумент имеет тип `FilePath`, но это всего лишь синоним типа `String`) и режим открытия (для чтения, для записи, для чтения и записи, для добавления). В результате возвращается дескриптор файла. Кроме открытия собственно файла можно воспользоваться любым предопределенным дескриптором, таким как `stdin`, `stdout` или `stderr`, которые проецируются на стандартные потоки ввода, вывода и ошибок (обычно на консоли).

Обратная операция *закрытия* файла осуществляется с помощью функции `hClose`. Как и в любом другом языке программирования, закрыть файл после завершения работы с ним очень важно, поскольку открытый дескриптор потребляет ресурсы машины.

Для *чтения или записи* можно воспользоваться обобщениями предыдущих консольных функций, работающими с любым дескриптором файла. Все они имеют префикс `h`, то есть мы можем воспользоваться функциями `hGetChar`, `hGetLine`, `hGetContents`, `hPutChar`, `hPutStr` и `hPutStrLn`. Кроме того, с помощью функции `hIsEOF` можно запросить у системы информацию о том, закончено ли чтение файла. Вооружившись данным арсеналом, можно создать более эффективную версию предыдущего примера построчного считывания файла:

```
import System.Environment
import System.IO

main = do (inFile:outFile:_) <- getArgs
          inHandle  <- openFile inFile  ReadMode
          outHandle <- openFile outFile WriteMode
          loop inHandle outHandle
          hClose inHandle
          hClose outHandle
       where loop inHandle outHandle = do
               isEof <- hIsEOF inHandle
               if not isEof
                  then do client <- hGetLine inHandle
                          (winner :: Bool) <- randomIO
                          (year   :: Int ) <- randomRIO (0, 3000)
                          hPutStrLn outHandle $ show (client, winner, year)
                          loop inHandle outHandle
                  else return ()
```

Можно заметить, что вместо конкретно заданных имен файлов для ввода и вывода информации код получает их через функцию `getArgs`. Эта функция, которая находится в модуле `System.Environment`, возвращает список параметров командной строки, заданных для исполняемой команды.

Поскольку открытие файла, работа с ним и последующее закрытие дескриптора — весьма распространенная задача, Platform предлагает специальную функцию `withFile`, которая берет на себя заботу о файле и ожидает только выполняемого

действия. Например, предыдущий код может быть переписан следующим образом:

```
main = do (inFile:outFile:_) <- getArgs
          withFile inFile  ReadMode  $ \inHandle ->
            withFile outFile WriteMode $ \outHandle ->
              loop inHandle outHandle
       where loop inhandle outHandle = do ...
```

Дополнительно функция `withFile` защищает от тех ошибок, которые можно допустить при обработке файла, гарантируя обязательное закрытие дескриптора файла. В следующем разделе показано, как справляться с возможными ошибками, которые могут возникать при работе с файлами (такими как файлы с поврежденными данными, файлы, не соответствующие определенной схеме, и т. д.).

Знания о считывании информации из файлов и ее записи в файлы, а также знания о ее продвижении в строку и из строки с помощью функций `show` и `read` позволяют нам сохранять информацию о клиентах и товарах на диске. В упражнении 9.2 предлагается классифицировать клиентов в нескольких файлах на основе их категорий.

УПРАЖНЕНИЕ 9.2. КЛАССИФИКАЦИЯ КЛИЕНТОВ

Давайте вспомним, что клиенты в нашем магазине могут быть физическими лицами, компаниями или правительственными организациями. В настоящий момент записи обо всех клиентах хранятся в магазине в одном файле. В каждой строке содержится информация о клиенте, которая закодирована с применением к этой информации функции show.

В данном упражнении нужно создать небольшую программу, которая считывает информацию об этих клиентах и генерирует еще три файла. В каждом из этих файлов должны содержаться данные обо всех клиентах одной из трех возможных категорий.

До сих пор код просто считывал данные из каждого файла, получая их в виде строки. Но в реальных ситуациях могут получить развитие два различных сценария. Первый из них относится к файлу, содержащему текст. В этом случае при преобразовании текстовых данных в байтовую последовательность могут применяться разные системы кодировки. Для изменения применительно к дескриптору файла текущей кодировки используется функция `hSetEncoding`. В модуль `System.IO` могут быть включены кодировки `latin1`, `utf8`, `utf16` или `utf32`, а также их версии с обратным и прямым порядком следования байтов. Если хочется получить гарантии того, что содержимое выходного файла для победителей будет записано в кодировке UTF-8, перед входом в цикл код нужно изменить:

```
withFile outFile WriteMode $ \outHandle -> do
  hSetEncoding outHandle utf8
  loop inHandle outHandle
```

Как бы то ни было, в следующей главе мы узнаем, что при работе с файлами использование значений типа `String` довольно редко является оптимальным вариантом. Вместо них нужно задействовать значения типа `ByteString` и `Text`. А в данном случае на первый план выходят проблемы, связанные с кодировкой, и вам непременно придется указывать, каким образом выполнять преобразование последовательности байтов в текстовые значения.

Работа с файлами

Перейдем к другому разряду операций, производимых с файлами. Перемещение, копирование или удаление затрагивает сами файлы, а не хранящиеся в них данные. Для этих операций в Haskell Platform включен пакет `directory`, в использовании которого нет ничего сложного.

Перемещением, копированием и удалением файлов в системе занимаются функции `renameFile`, `copyFile` и `removeFile`, входящие в модуль `System.Directory`. Здесь важно отметить, что ни одна из этих операций не поддерживает работу с папками — только с файлами.

Из-за невозможности работы упомянутых функций с папками пакет `directory` предлагает для этого другой набор функций. Перечислим наиболее важные:

- `getDirectoryContents` возвращает список всех элементов внутри папки.
- `createDirectory` создает в системе новую папку. Однако если она уже существует или отсутствует какая-либо из родительских папок, происходит ошибка. С обоими этими условиями успешно справляется функция `createDirectoryIfMissing`.
- `renameDirectory` позволяет переместить папку в системе. Следует заметить, что если путь, куда перемещается папка, уже существует, происходит ошибка.
- `removeDirectory` удаляет папку из системы. Обычно папка не может быть удалена, если у нее есть содержимое. Для таких случаев используется функция `removeDirectoryRecursive`, которая сначала удаляет каждый элемент, содержащийся в папке, а потом и саму папку.

Кроме кодировки еще одной серьезной проблемой при работе с файлами является формат файловых путей, который меняется в зависимости от базовой операционной системы. Например, Windows использует буквы дисководов, являющиеся префиксами пути, а в качестве разделителей пути допускает как символ `\`, так и символ `/`. Что же касается систем Unix и Mac OS X, то они позволяют применять в качестве разделителя только символ `/`. Поскольку Haskell-сообщество придает межсистемной совместимости особую важность, в Platform включена библиотека `filepath`, позволяющая абстрагироваться от этих проблем.

Самой важной в этой библиотеке является функция `(</>)`, которая объединяет два сегмента пути в один более крупный сегмент. Если, к примеру, нужно прочитать конфигурацию базы данных, находящуюся в файле `database.settings` внутри папки `config`, правильнее всего будет поступить следующим образом:

```
withFile ("config" </> "database.settings") ReadMode $ \handle -> ...
```

И наоборот, может понадобиться разбить конкретный путь к файлу в папке на местоположение файла и его имя. Это можно сделать с помощью функции `splitFileName`. Рассмотрим в качестве примера код, который получает имя файла входных данных из аргументов командной строки и записывает его в файл `example`, находящийся в той же папке:

```
import System.FilePath

main = do (file:_) <- getArgs
```

```
let (folder, _) = splitFileName file
withFile (folder </> "example") WriteMode $ \handle -> ...
```

Иногда может потребоваться не только разбить путь на папку и имя файла, но и получить список всех сегментов пути. В таком случае вместо данной функции можно воспользоваться функцией `splitDirectories`.

И наконец, в пакет `filepath` включены функции для работы с расширениями. Чтобы добавить расширение к существующему пути к файлу, можно воспользоваться функцией `(<.>)`. И наоборот, для создания кортежа из имени файла и всех связанных с ним расширений можно воспользоваться функцией `splitExtensions`. Пакет включает множество других небольших полезных функций вроде функции замены расширения, сброса последнего расширения и т. д. Если вашему приложению требуется работать с файловыми путями, то весьма полезно будет заглянуть в документацию по пакету `filepath`.

ПРИМЕЧАНИЕ

Хочется еще раз подчеркнуть: не стоит заниматься самодеятельностью для объединения, добавления расширений или решения любых других задач по работе с файловыми путями. Лучше воспользоваться пакетом filepath, обеспечив тем самым отсутствие ошибок в коде и возможность его функционирования в среде различных операционных систем.

Обработка ошибок

При вводе и выводе, а также при выполнении многих других операций с побочными эффектами, например при выводе на печать или обмене данными по сети, может случаться множество разнообразных ошибок. Во всех предыдущих примерах при попытке открыть не существующий в системе файл программа просто должна была аварийно завершаться. Знание способов обнаружения подобных условий возникновения ошибок и восстановления работы программы имеет весьма важное значение.

Однако перед тем как приступить к обработке ошибок в контексте монады `IO`, давайте рассмотрим порядок обработки ошибок в чистом коде, то есть вновь обратимся к теме, которая нам уже знакома. Тогда мы сможем понять, в чем разница между чистыми ошибками и исключениями. Последнее понятие относится к механизму сигнализации о ненормальных условиях в `IO`.

Чистые ошибки

До сих пор при невозможности выполнения операции наиболее распространенным способом справиться с этой ситуацией было возвращение значения `Maybe`. К примеру, это происходило, когда функция не могла применяться к некоторым конструкторам значений. Так, функция `head` неприменима к пустым спискам `[]`, а получение имени компании невозможно для индивидуального клиента. Таким образом, при возникновении какой-нибудь проблемы вызывающая функция не должна была получать в качестве результата значение `Nothing`.

К сожалению, значение `Maybe` не относится к точным средствам выяснения характера происшедшей ошибки. Можно объявить о том, какая именно операция не за-

вершилась успехом, но при этом не установить причину. А ведь во многих случаях актуальной является именно эта информация: неудачная транзакция в базе данных по причине недоступности подключения или по причине нарушения каких-либо ограничений — не одно и то же. Для таких сценариев больше подойдет тип `Either`, который объявляется довольно просто:

```
data Either a b = Left a | Right b
```

Например, если `x :: Either Int String`, то `x` может содержать либо целочисленное значение — в таком случае будет использоваться конструктор `Left`, либо строковое значение, заключенное в оболочку конструктора `Right`.

В соответствии с условиями при применении типа `Either` для обработки ошибок вариант `Right` используется, когда вычисление завершается успешно, а `Left` — в случае неудачи. Таким образом, если `r` является типом правильных результатов, а `e` указывает на возможные ошибки, то `Either e r` является обычным типом для применения в функциях. В качестве примера давайте определим версию функции `companyName`, сообщающую о конкретной ошибке при невозможности извлечения названия компании-клиента:

```
data CompanyNameError = GovOrgArgument | IndividualArgument

companyName :: Client i -> Either CompanyNameError String
companyName Company { clientName = n } = Right n
companyName GovOrg     { }             = Left GovOrgArgument
companyName Individual { }             = Left IndividualArgument
```

Теперь пользователь этой функции может провести сопоставление с образцом в отношении результата и в случае необходимости определить тип ошибки:

```
printCompanyName :: Client i-> IO ()
printCompanyName c = case companyName c of
  Right n -> putStrLn n
  Left GovOrgArgument     -> putStrLn "A government organization was given"
  Left IndividualArgument -> putStrLn "An individual was given"
```

Решение о том, что именно нужно использовать для указания на факт неудачного завершения операции, `Maybe` или `Either`, обычно дается нелегко. Особенно трудно выбрать нужный вариант при наличии библиотеки, стиль спецификации ошибок в которой отличается от стиля, выбранного в вашем приложении. К счастью, в пакете `errors` имеется модуль `Control.Error.Util`, в котором содержатся весьма полезные функции преобразования стилей. Сигнатуры этих функций, задействованных в преобразовании, имеют следующий вид:

```
hush :: Either a b -> Maybe b
note :: a -> Maybe b -> Either a b
```

Функция `hush` используется главным образом для того, чтобы забыть о какой-либо конкретной ошибке в значении `Either` и при неудачном вычислении просто вернуть `Nothing`. Следуя по другому направлению, нужно сообщить, какое значение ошибки возвратить, если значение `Maybe` оказывается значением `Nothing`. Имя функции служит напоминанием о ее назначении: вам нужно к возможному значению ошибки «добавить примечание» («add a note»).

Поскольку решение о том, что именно использовать, `Maybe` или `Either`, дается нелегко, да и к тому же может иметь различные варианты в разных местах приложения, можно подумать о том, чтобы абстрагироваться от способа обработки ошибок. И здесь в качестве инструмента пригодятся имеющиеся в Haskell классы типов.

В главе 7 мы изучили порядок использования класса типов `MonadPlus` для возвращения значений, объявляющих ошибочное состояние, и его операции `mplus`, служащей для объединения нескольких таких значений и возвращения тех из них, которые не были ошибками. При применении класса `MonadPlus` можно воспользоваться типом `Maybe` или списками для оповещения об ошибках соответственно с помощью значений `Nothing` и пустых списков. К сожалению, сделать `Either` экземпляром `MonadPlus` нельзя. Проблема в том, что операция `mempty` в этом типовом классе не должна иметь аргументов. Поэтому указать, какое значение должно быть заключено в оболочку конструктора `Left` для возвращения ошибки, не представляется возможным.

В пакет `mtl` включен обобщенный класс типов `MonadPlus`, для которого заданными экземплярами могут быть и `Maybe`, и `Either`: имя этого класса типов `MonadError`. Любой тип, поддерживающий этот класс типов, должен предоставлять две различные операции, как показано в его объявлении:

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

Первая операция отвечает за оповещение о неудаче. Как видите, она отвечает тому требованию, которому не отвечает `mempty`: получает дополнительный аргумент в виде возвращаемого значения ошибки. Например, функции `companyName` можно придать общую форму для работы как с `Maybe`, так и с `Either`:

```
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Error

companyName :: MonadError CompanyNameError m => Client i -> m String
companyName Company { clientName = n } = return n
companyName GovOrg     { }             = throwError GovOrgArgument
companyName Individual { }             = throwError IndividualArgument
```

В случае использования `MonadPlus` функция `mplus` служит средством восстановления после ошибки. Вообще-то выражение `x `mplus` y` описано в данном контексте как возвращающее значение `x`, если представляется успех, или, в противном случае, возвращающее значение `y`, если `x` представляет неудачу[1]. Этой операции также был придан общий вид: функция `catchError` имеет аналогичное назначение, но в случае неудачного завершения операции получает доступ к значению ошибки соответствующей функции `throwError`. Давайте, к примеру, создадим функцию `companyName` и в случае неудачного завершения операции вернем с помощью `MonadError` предопределенное пустое значение:

[1] Хотя в случае со списками, где a `mplus` b является объединением двух списков, эта аналогия, по большому счету, неприменима и соответственно возвращает все успешно полученные значения из обеих ветвей.

```
companyNameDef :: MonadError CompanyNameError m => Client i -> m String
companyNameDef c = companyName c `catchError` (\_ -> return "")
```

Наряду с `MonadError` в `mtl` и в `transformers` включен преобразователь монад `ErrorT`, который можно добавить к нашему комплекту. Понять его назначение лучше на примере преобразователя `MaybeT`, то есть думая о вычислениях, которые могут потерпеть неудачу, с добавлением тега, определяющего ошибку в соответствующем случае. Преобразователь назван `ErrorT`, а не `EitherT`, и этому есть причина: он требует, чтобы тип ошибки был экземпляром класса типов `Error`. Этот класс типов заставляет тип предоставлять способ создания сообщений об ошибках из строк, что можно увидеть из его объявления:

```
class Error a where
  noMsg  :: a
  strMsg :: String -> a
```

В необходимости этого дополнительного ограничения есть своя причина. Если нужно добиться полной совместимости с функцией `fail` внутри класса `Monad`, вам необходимо иметь возможность преобразования из строкового типа в тип ошибки. Причина в том, что `fail` принимает в качестве аргумента строку.

Во многих случаях эта функциональность вам не понадобится, поэтому ограничение, связанное с `Error`, можно отбросить. Соответствующий преобразователь монад включен в пакет `either`. В данном случае тип носит название `EitherT`, потому что он работает точно так же, как этого можно было бы ожидать от `Either` без последующего ограничения. А что еще лучше, стек монады, содержащей `EitherT`, является также экземпляром `MonadError`, поэтому вы можете по-прежнему использовать тот же самый интерфейс. Кроме того, пакет `errors` поддерживает применение `EitherT` вместо `ErrorT` и предоставляет функции преобразования между стеками, используя `MaybeT` и `EitherT`:

```
hushT :: Monad m => EitherT a m b -> MaybeT m b
noteT :: Monad m => a -> MaybeT m b -> EitherT a m b
```

ПРИМЕЧАНИЕ

Различия между применением ErrorT и EitherT небольшие и сводятся к вопросу о том, нужна вам совместимость с имеющейся в Monad функцией fail или нет. То есть выбор зависит от решения, нужно или нет при вызове функции fail внутри монады восстанавливать возвращаемую строку. В любом случае, чтобы отложить принятие решения, можно воспользоваться MonadError.

БЕЗОПАСНЫЕ ФУНКЦИИ

При разработке Haskell-библиотек исторически сложилось так, что некоторые функции в модуле Prelude не обеспечивают чистых способов обработки ошибок. Наиболее типичным примером может служить функция head. В случае ее применения к [] эта функция, вместо того чтобы возвращать какое-либо представление об ошибке, выдает исключение (которое, как показано в следующем разделе, может быть перехвачено только внутри монады IO).

Для решения этой проблемы предназначен пакет safe, который предоставляет множество безопасных версий широко распространенных функций. Например, у функции head есть версия headMay, возвращающая значение, заключенное в оболочку Maybe, что позволяет возвращать для пустых списков значение Nothing, а также версия headDef, которой передается дополнительный аргумент со значением, возвращаемым по умолчанию в том случае, если список окажется пуст.

Перехват исключений

Вопрос о том, как справиться с внешним миром, открывающим дверь для абсолютно новой категории ошибок, таких как несуществующие файлы или ненадежные соединения, уже рассматривался. Для такого рода событий Haskell предоставляет *механизм исключений* (exception mechanism). Между исключениями и уже рассмотренными чистыми ошибками есть два основных различия.

- Чистые ошибки могут возникать и перехватываться в любом месте, обычно путем простого сопоставления с образцом финального значения вычислений. В отличие от этого исключения могут обрабатываться только внутри контекста `IO` (но генерироваться они могут из любого места).
- При использовании типа `Either` нужно заранее указать любые возможные ошибки, которые могут случиться при выполнении фрагмента кода. В то же время имеющийся в Haskell механизм исключений является расширяемым. Это позволяет новым побочным эффектам вызывать ошибки по-новому, но одновременно это вредит анализу кода, поскольку нельзя сказать заранее, какое именно исключение может быть выброшено.

Точкой входа для любой работы с исключениями является модуль `Control.Exception`. В примерах кода, приводимых далее в этом разделе, подразумевается, что этот модуль уже включен в список импорта. Во многих случаях внутри тела функции и в объявлениях `let` нужно указывать типы исключений, поэтому во всех примерах подразумевается включение GHC-расширения `ScopedTypeVariables`.

Начнем с примера, в котором к исходной функции, записывающей список победителей из базы данных клиентов с помощью функций `readFile` и `writeFile`, добавляется обработка исключений:

```
import Control.Exception
import System.IO.Error
import System.Random
main = do clients <- fmap lines $ readFile "clients.db"
  clientsAndWinners <- mapM (\c -> do (winner :: Bool) <- randomIO
                                      (year   :: Int ) <- randomRIO (0, 3000)
                                      return (c, winner, year))
                            clients
     writeFile "clientsWinners.db" $ concatMap show clientsAndWinners
  `catch` (\(e :: IOException) -> if isDoesNotExistError e
                                  then putStrLn "File does not exist"
                                  else putStrLn $ "Other error: " ++ show e)
```

В первую очередь внимание нужно обратить на использование функции `catch`. Идея проста: сначала вы объявляете основной запускаемый код (`main`), а затем — обработчик для конкретного класса исключений. Второе, на что следует обратить внимание, — в коде явно указывается *тип исключений*, обрабатываемый с его помощью.

В данном конкретном случае интерес представляет тип `IOException`, описывающий исключения, связанные с вводом и выводом данных. В значении типа `IOException` кодируется дополнительная информация о виде возникшей проблемы: эту информацию можно запросить, воспользовавшись множеством функций из модуля

`System.IO.Error`. В данном примере нас интересуют несуществующие файлы, поэтому проверка производится с помощью функции `isDoesNotExistError`.

Динамичность и расширяемость механизма исключений в Haskell придает спецификации типов в сценариях обработки ошибок особую важность. В модуль `Control.Exception` также включен предопределенный набор исключений, выдаваемых функциями в Platform. Помимо прочих в этот набор включена функция `ArithException`, которая сигнализирует о наличии численной ошибки, например переполнения или деления на нуль, функция `ErrorCall`, которая позволяет обрабатывать вызовы функции `error`, и функция `PatternMatchFail`, которая запускается при неудачном сопоставлении с образцом конкретного значения.

В следующем коде у пользователя запрашиваются два целых числа и показывается частное от их деления друг на друга. При этом могут выдаваться два вида исключений: прежде всего, пользователь может ввести что-нибудь, что не является числом, — это ведет к вызову функции `error` внутри `read`. Другой возможной проблемой является деление на нуль. Здесь можно видеть, что у каждого исключения есть собственный обработчик:

```
main = do (n1 :: Int) <- fmap read getLine
          (n2 :: Int) <- fmap read getLine
       putStrLn $ show n1 ++ " / " ++ show n2 ++ " = " ++ show (n1 `div` n2)
     `catch` (\(_ :: ErrorCall) -> putStrLn "Error reading number")
     `catch` (\(e :: ArithException) -> case e of
          DivideByZero -> putStrLn "Division by zero"
          _            -> putStrLn $ "Other arithmetic error: " ++ show e)
```

Эти типы исключений отличаются от `IOException`: в то время как для получения вида возникшей проблемы значение `IOException` должно быть запрошено через функции специального назначения, эти типы определяются как простые алгебраические типы данных (ADT), что дает нам возможность воспользоваться сопоставлением с образцом для обнаружения источника проблем.

Модуль `Control.Exception` включает в себя множество других вариантов перехвата при обработке исключений. Одним из них является функция `catches`, которая получает список обработчиков для различных исключений. Например, с использованием этой функции показанный ранее код может быть написан без ряда вызовов `catch`:

```
main = do ...
  `catches` [ Handler (\(_ :: ErrorCall) -> putStrLn "Error reading number ")
            , Handler (\(e :: ArithException) -> case e of ...)]
```

Можно также воспользоваться функцией `handle`, которая просто выполняет операцию `catch` с аргументами, располагаемыми в обратном порядке. Наиболее часто она применяется при длинном исполняемом коде, но коротком коде обработки ошибок, поскольку она выставляет обработку исключений с очевидным упреждением. Третий способ написания того же самого кода вычисления частного имеет следующий вид:

```
main = handle (\(_ :: ErrorCall)      -> ...) $
       handle (\(e :: ArithException) -> ...) $
       do (n1 :: Int) <- fmap read getLine
          ...
```

Иногда исключение требуется трактовать точно так же, как ошибку при чистых вычислениях. Тем самым, если все остальные источники ошибок носят чистый характер, можно получить более понятный код. Обратимся к сценарию, в котором производится проверка на приемлемость ряда значений из базы данных. В данном случае исключения могут возникать в коде соединения с базой данных, но в проверке приемлемости будет использоваться значение `Maybe` или `Either`. Мост между двумя мирами перебрасывается через функцию `try`, возвращающую тип `Either`, который может содержать финальный результат в значении `Right` или же исключение в значении `Left`.

Для каждой функции обработки исключений есть соответствующая функция с окончанием `Just`: `catchJust`, `handleJust` и `tryJust`. Этим функциям передается дополнительный аргумент, который называется *фильтром исключений* (exception filter). С его помощью принимается решение о том, должен обработчик перехватывать конкретное исключение или перебрасывать его дальше. В качестве аргумента этим фильтрам передается значение исключения, а они должны вернуть значение `Maybe`: если возвращается значение `Nothing`, значит, исключение должно перебрасываться дальше, если результатом служит `Just e`, исключение обрабатывается кодом, соответствующим данной функции `catch`. Нам уже попадался подобный случай: из всех возможных значений `ArithException` реальный интерес заключается только в `DivisionByZero`. Следовательно, чтобы гарантировать, что любые другие исключения будут выброшены дальше правильно, можно использовать функцию `catchJust`:

```
main = catchJust (\e -> if e == DivideByZero then Just e else Nothing)
   (do (n1 :: Int) <- fmap read getLine
       (n2 :: Int) <- fmap read getLine
       putStrLn $ show n1 ++ " / " ++ show n2 ++ " = " ++ show (n1 `div` n2)
    `catch` (\(_ :: ErrorCall) -> putStrLn "Error reading number") )
   (\_ -> putStrLn "Division by zero")
```

ПРИМЕЧАНИЕ

Как видите, механизм исключений в Haskell намного более эффективен, чем аналогичные механизмы в других языках программирования. Кроме определения обработчиков по типу, такие функции, как catchJust, позволяют выполнять динамическую проверку необходимости перехвата конкретного исключения. Использование функции catchJust и подобных ей функций гарантирует обработку только тех исключений, с которыми вы знаете, как поступать, и перебрасывание остальных исключений последующим обработчикам.

Такие комбинаторы, как `catchJust`, позволяют четко обозначить то исключение, которое должен перехватить обработчик. Иногда у обработчика есть совершенно противоположное намерение: перехватывать каждое исключение, которое может генерироваться в коде. Решение принимается после изучения того, как различные типы исключений связаны друг с другом. В частности, Haskell-исключения формируют *иерархию*: каждое исключение типа `E` имеет родительское исключение типа `P`, указывая на тот факт, что исключения типа `E` являются подмножеством исключений типа `P`. На вершине этой иерархии находится тип `SomeException`. Если нужно добавить обработчик, перехватывающий в конце все возможные исключения, которые могут возникать в приложении, код должен иметь следующий вид:

```
main = do ...
  `catch` (\(e :: SomeException) -> ...) – обработчик неперехваченных исключений
```

Большинство языков программирования, в которых исключения используются в качестве механизма обработки ошибок, не только предоставляют способы их генерации и перехвата, но и гарантируют, что даже в случае неизвестного исключения будет выполнен определенный фрагмент кода. Обычно в этом случае выполняется фрагмент очищающего кода или кода освобождения ресурсов. Например, если открыт дескриптор файла, нужно обеспечить его закрытие, даже если при обработке возникло какое-то исключение. В Haskell эта функциональность предоставляется через функцию `finally`. Она может быть использована для создания более отказоустойчивой версии того кода, который записывает в файл победивших клиентов:

```
main = do (inFile:outFile:_) <- getArgs
          inHandle  <- openFile inFile  ReadMode
          outHandle <- openFile outFile WriteMode
          ( loop inHandle outHandle
            `finally` (do hClose inHandle
             hClose outHandle) )
where loop inHandle outHandle = ...
```

Однако этот код не вполне корректен. В частности, может случиться, что исключение будет выброшено *в момент* открытия любого из файлов. В данном случае использовать функцию `finally` нельзя, потому что вызов `hClose` без открытия дескриптора окажется некорректным. При работе с ресурсами очень часто используется трехэтапный процесс: вы получаете ресурс, выполняете какую-нибудь операцию, а потом освобождаете ресурс. Даже в случае возникновения исключения в процессе обработки данных ресурс нужно освободить, но запускать этот код в случае отказа в получении ресурса не нужно. Этот образец реализован в Haskell в явном виде с помощью функции `bracket`. Правильнее всего переписать предыдущий код следующим образом:

```
main = do (inFile:outFile:_) <- getArgs
  bracket (openFile inFile ReadMode) -- получение inHandle
          hClose                     -- освобождение inHandle
          (\inHandle -> bracket (openFile outFile WriteMode)
                        hClose
                        (\outHandle -> loop inHandle outHandle))
    where loop inHandle outHandle = ...
```

Новые знания об исключениях можно применить в упражнении 9.3, взяв за основу упражнение 9.2 и введя в него требуемые усовершенствования.

УПРАЖНЕНИЕ 9.3. УЛУЧШЕННАЯ КЛАССИФИКАЦИЯ КЛИЕНТОВ

Добавьте к коду, написанному при выполнении упражнения 9.2, обработку исключений.

Генерация исключений

Разобравшись с перехватом исключений, нужно научиться их *генерировать* (throw). Чтобы многократно использовать любой из предопределенных в `Control.Exception` типов исключений, следует просто вызвать функцию `throwIO`, если все это происходит внутри монады `IO`, или функцию `throw`, если нужно выбросить

исключение из чистого кода (но при этом следует помнить, что обработчик попрежнему должен быть внутри монады IO). В следующем простом примере многократно используется исключение типа NoMethodError:

```
main = do throw $ NoMethodError "I don't know what to do"
     `catch` (\(e :: SomeException) -> do putStr "An exception was thrown: "
                                          putStrLn $ show e)
```

Чаще всего приходится выдавать исключения нового нестандартного типа, описывающего те исключения, которые могут возникать в вашем коде. В целях использования типа в качестве исключения нужно создать для него экземпляры классов типов Show, Typeable и Exception. К счастью, имеющийся в Haskell механизм извлечения избавляет нас от написания всего шаблонного кода. В следующем примере тип исключений объявляется в системе аутентификации:

```
{-# LANGUAGE ScopedTypeVariables, DeriveDataTypeable #-}
import Data.Typeable

data AuthenticationException = UnknownUserName String
                             | PasswordMismatch String
                             | NotEnoughRights String
                             deriving (Show, Typeable)

instance Exception AuthenticationException
```

Теперь вашим новым типом исключений можно пользоваться так же, как и любым встроенным исключением:

```
main = do throw $ UnknownUserName "Alejandro"
          `catch` (\(e :: AuthenticationException) -> ...)
```

КЛАСС ТИПОВ TYPEABLE

В имеющемся в Haskell механизме исключений широко используется класс типов Typeable. Он служит для получения информации о типе значения во время выполнения программы (поскольку обычно Haskell после компиляции кода с целью повышения производительности всю информацию о типах стирает). Typeable используется также в модуле Data.Dynamic с целью создания интерфейса для динамических значений, которые во время выполнения программы можно привести к другим типам.

Как и в других случаях, заключив функциональность исключения в оболочку класса типов, мы упрощаем написание кода, поскольку даем возможность использовать его многократно несколькими стеками монад и множеством пользователей вашей библиотеки. В большинстве случаев для составления стека исключений вполне достаточно ранее представленного класса типов MonadError. Если же потребуется реальное обобщение функций и интерфейса Control.Exception, можно порекомендовать класс типов MonadCatch из пакета exceptions.

ВНИМАНИЕ

В чистом коде вообще не рекомендуется генерировать исключения. Вместо них нужно стремиться использовать другие виды механизмов прохождения ошибок, такие как Maybe или Either. В коде IO исключения могут быть представлены более компактным и понятным кодом (вроде обработчика исключения последнего шанса, который регистрирует все критические ошибки), но злоупотреблять этим все же не следует.

Потоковый ввод-вывод данных с помощью библиотеки conduit

О представленной в предыдущих разделах системе ввода-вывода обычно говорят как о классическом, или ленивом, вводе-выводе. Она стала частью Haskell начиная с первых версий. Однако механизм ее работы недостаточно хорошо приспособлен к характерным для Haskell ленивым вычислениям, из-за чего возникает так называемая *проблема потокового ввода-вывода данных* (streaming data problem). Поэтому был разработан ряд *потоковых библиотек* (stream libraries), весьма элегантно и эффективно решающих эту проблему.

Проблемы ленивого ввода-вывода

Рассмотрим следующий простой фрагмент кода ввода-вывода: вы открываете файл, получаете его содержимое, закрываете файл, а затем работаете с только что полученной информацией. На первый взгляд здесь не должно быть никаких проблем: `hGetContents` считывает всю информацию, `hClose` гарантирует закрытие дескриптора.

```
main = do h <- openFile "/some/text/file" ReadMode
          s <- hGetContents h
          hClose h
          print s
```

Однако если запустить этот код, на экране появится пустая строка:

```
$ ./dist/build/chapter9-stream/chapter9-stream
""
```

Стоит поменять местами операцию вывода на экран и операцию закрытия файла, все заработает.

Это простое изменение порядка выполнения операций высвечивает проблему во взаимодействии между операциями ввода-вывода и моментом вычисления каждого элемента в программе. Из-за ленивой природы вычислений, характерной для Haskell, значение `s` не вычисляется, пока в нем не возникнет необходимость, а это происходит только при вызове функции `print`. Однако в этот момент дескриптор уже закрыт, следовательно, `hGetContents` не в состоянии извлечь какую-либо информацию из файла. Здесь поведение нашей программы не соответствует нашим ожиданиям.

Одно из возможных решений сводится к вычислению `s` с использованием библиотеки `seq` или `deepseq`. Хотя это решение вполне работоспособно, при его применении возникают две проблемы. Первая из них касается эффективности, поскольку вся строка принудительно заносится в память, расходуя скудные ресурсы системы. К тому же, пытаясь пойти на хитрость и принудительно заносить в память только ту строку, которая вам нужна, вы столкнетесь с проблемами поддержки кода и возможности компоновки: какая именно функция должна отвечать за помещение в память заданного значения? Как заранее узнать, какие значения понадобятся во всем приложении?

Еще хуже оставлять дескриптор файла открытым на все время работы приложения. При этом возникнут явные проблемы, связанные с получением ресурсов без их освобождения. Ресурсы, относящиеся к файлам, могут обходиться недорого, но если речь идет о подключениях к сети или к базе данных, этот способ становится абсолютно неприемлемым.

Ленивые вычисления и монада `IO` могут преподнести еще больше сюрпризов. Предположим, что в ходе выполнения функции `hGetContents` в предыдущем коде возникает исключение. С этим исключением столкнется не тот блок кода, который генерирует строку, а тот, который использует вычисленное значение. Это означает, что в середине каких-нибудь чистых вычислений, использующих `s`, на их пути может возникнуть исключение, а способа его обработки без обращения к `IO` не будет. Более того, ваш код утратит свою предсказуемость, поскольку вы не сможете гарантировать, что некое значение не повлечет за собой запуск каких-нибудь вычислений в рамках монады `IO`.

Как видите, использование простой модели обработчиков, экспонируемых модулем `System.IO`, приводит к непредсказуемости момента освобождения ресурсов и генерации исключения. Понятно, что возможные решения вроде принудительного вычисления или оставления дескрипторов открытыми неэффективны. Все это и называется «проблемой потокового ввода-вывода данных».

Как уже упоминалось, для ее решения Haskell-сообщество предложило потоковые библиотеки. Эти библиотеки обычно предоставляют абстракцию данных, получаемых из ресурса, в форме *потока ввода-вывода* (stream). Кроме того, способ, посредством которого ресурс, генерирующий данные, обретается и освобождается, сделан намного более предсказуемым. Во многих случаях потоковые библиотеки приводят также к росту производительности, поскольку они способны гарантировать, что в память попадут только те данные, которые нужны для выполнения операции.

Согласно этому замыслу, в Hackage создано несколько библиотек. Начальный подход, предложенный Олегом Киселевым (Oleg Kiselyov), реализован в пакетах `iteratee` и `enumerator`. Впоследствии другие библиотеки, включая `pipes`, `io-streams` и `conduit`, предоставили более удобный и простой в использовании интерфейс. Все эти три библиотеки нашли довольно широкое применение.

Однако в этой книге мы сосредоточимся на библиотеке `conduit`, поскольку она задействована в других библиотеках, таких как библиотека `Persistent`, предназначенная для подключения к базам данных (см. далее). Кроме базовой библиотеки существует множество других библиотек, подключающих библиотеку `conduit` к различным источникам данных. В любом случае, концепции, использованные в библиотеке `conduit`, очень похожи на концепции любых других библиотек.

Знакомство с библиотекой conduit

В основе работы библиотеки `conduit` лежат потоки ввода-вывода данных, которые создаются, модифицируются и потребляются различными акторами. Например, если бы пример «выигравшие клиенты» моделировался с применением библиотеки `conduit`, то в нем был бы актор, предоставляющий поток строк из файла, еще один актор, модифицирующий этот поток с целью добавления информации о том,

выиграл каждый клиент или нет, и, наконец, последний актор преобразовывал бы этот поток в новый файл в системе.

Таким образом, есть три вида акторов, участвующих в обработке данных. Они определены в модуле `Data.Conduit` пакета `conduit`:

- Актор `Source` предоставляет поток потребляемых значений. Примером может служить получение текстовых данных из файла на диске, из сетевого подключения, а также получение каждого из элементов, имеющихся в списке.
- Поведение актора `Sink` прямо противоположно: потребление потока значений без создания какого-либо дальнейшего потока ввода-вывода. Дополнительно может производиться конечный результат. Акторы `Sink`, не производящие результата, записывают данные на диск или отправляют их по сети. Свертка списка также может рассматриваться как актор `Sink`: она потребляет поток ввода-вывода данных и возвращает одно значение, являющееся сверткой с некоторой функцией.
- Актор `Conduit` как потребляет, так и производит потоковые значения. Промежуточный шаг в «выигравших клиентах» является примером актора `Conduit`, так как представляет собой проекцию потока ввода-вывода.

Иногда речь заходит о производителе (`Producer`), каковым является любой актор, способный создать поток вывода значений (это либо `Source`, либо `Conduit`), и о потребителе (`Consumer`), каковым является любой актор, получающий поток ввода в качестве входных данных, и следовательно, состоящий из акторов `Sink` и `Conduit`.

Полная последовательность операций с данными на пути от `Source` к `Sink` устанавливается посредством *оператора соединения* (`$$`). Чтобы актор `Source` оказался в начале цепочки, а `Sink` — в ее конце, нужно воспользоваться *операторами слияния*: оператор (`$=`) соединяет `Source` и `Conduit`, производя `Source`; оператор (`=$=`) соединяет двух акторов `Conduit`; оператор (`=$`) соединяет `Conduit` и `Sink`, производя `Sink`. Следует напомнить, что знаки равенства можно рассматривать в качестве признаков того места, куда направлена последовательность операций. На рис. 9.1 показаны два возможных варианта соединения всей последовательности операций. Какому из способов отдать предпочтение, конкретных предписаний не существует, хотя на практике чаще применяется верхний вариант с оператором (`$$`) в начале.

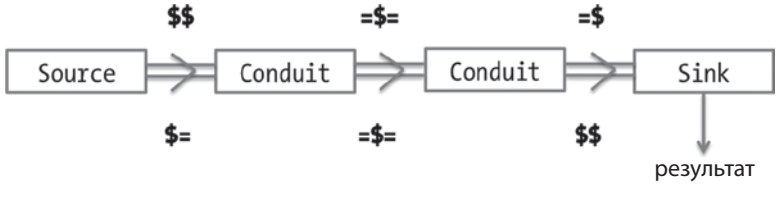


Рис. 9.1. Акторы из библиотеки conduit, связанные друг с другом для обработки потока ввода-вывода

Важно отметить, что каждый актор должен позаботиться о получении и освобождении своих ресурсов безопасным и предсказуемым образом. Например, актор

«записи в файл» `Sink` может открыть дескриптор при старте потока ввода данных и безопасным образом закрыть дескриптор, когда поток ввода закончится. Кроме того, данные могут запрашиваться у акторов `Source` и `Conduit` по мере их потребления другими акторами `Conduit` или `Sink`, что позволяет достичь более высокой производительности.

Давайте посмотрим на ряд примеров, использующих простейший вид потока ввода-вывода — список. Все функции, относящиеся к такому использованию списков, находятся в модуле `Data.Conduit.List`. Функция `sourceList` производит поток, который выдает по очереди каждый элемент списка. Функция `fold` потребляет список и производит результат в виде свертки всех элементов конкретной функцией. Давайте посмотрим на результат в интерпретаторе:

```
*Chapter9.Stream> import Data.Conduit
*Chapter9.Stream Data.Conduit> import qualified Data.Conduit.List as L
*Chapter9.Stream Data.Conduit L> L.sourceList [1,2,3,4,5] $$ L.fold (+) 0
15
```

В модуле `Data.Conduit.List` имеется множество других акторов для работы с потоками ввода-вывода, с интерфейсом, похожим на интерфейс списочных функций. Весьма полезным считается проецируемый актор `Conduit`, который применяет функцию к каждому элементу потока, производя новый поток с результатами. Также в отношении потока можно воспользоваться фильтром для исключения значений, не представляющих интереса. Давайте в качестве примера вычислим сумму квадратов всех нечетных чисел от одного до двадцати:

```
> L.sourceList [1 .. 20] $$ L.filter odd =$= L.map (\x -> x*x) =$ L.fold (+) 0
1330
```

Как и в случае со списками, развертка, обеспечиваемая в данном случае функцией `unfold`, представляет собой еще один способ генерирования потоков ввода-вывода. В следующем примере показано, как эта функция используется для генерирования бесконечного потока натуральных чисел, из которого с помощью функции `isolate` берутся только первые десять:

```
> L.unfold (\x -> Just (x,x+1)) 1 $= L.isolate 10 $$ L.consume
[1,2,3,4,5,6,7,8,9,10]
```

В примере также показаны случаи применения функции `consume`, которая преобразует поток ввода-вывода в простой список.

Операции, похожие на списочные, дают массу возможностей, но интересно узнать, как создавать собственных акторов `Conduit`. Первое наблюдение заключается в том, что для использования акторов `Source`, `Conduit` и `Sink` нам не нужно изучать различные технологии, поскольку все три вида представляют более общий тип `ConduitM i o m r`, где `i` — тип элементов потока ввода; `o` — тип элементов, производимых актором; `r` — тип результата выполнения актора. На основе данной информации можно увидеть, что, к примеру, тип `Source m o`, не получающий ввода и не производящий никакого результата, фактически является синонимом типа для `ConduitM () o m ()`.

На самом деле `ConduitM` является преобразователем монад. Аргумент `m`, который должен быть экземпляром класса типов `Monad`, позволяет акторам библиотеки `conduit` выполнять действия внутри монады, не ограничиваясь чистыми вычис-

лениями. Например, актор `Source`, считывающий данные из файла, будет иметь аргумент `m`, равный `IO`. Как воспользоваться этим аргументом, мы вскоре увидим.

В контексте `ConduitM` мы получаем доступ к четырем функциональным возможностям, используемым для создания потоков ввода-вывода:

- Функция `await` пытается получить следующий элемент в потоке ввода. Если ее работа завершается успешно, элемент возвращается в оболочке `Just`. Если у потока больше нет элементов, функция возвращает `Nothing`.
- Внутри `ConduitM` потоком ввода можно управлять: функция `leftover` позволяет возвращать некоторые элементы в поток ввода. В настоящий момент в документации не одобряется возвращение тех элементов, которые не были получены посредством вызова функции `await`.
- Функция `yield` служит для отправки значений в поток вывода.
- И наконец, возвращаемое значение программируемого преобразователя `ConduitM` указывается простым использованием имеющегося в монадах метода `return`.

В качестве примера давайте создадим простой актор `Conduit`, получающий поток ввода от `Client` и возвращающий каждую персону, появляющуюся в потоке, будь она отдельным клиентом или сотрудником компании:

```
people :: Monad m => Conduit (Client i) m Person
people = do client <- await
            case client of
              Nothing -> return ()
              Just c -> do case c of
                             Company { person = p }    -> yield p
                             Individual { person = p } -> yield p
                             _                         -> return ()
                           people
```

Обратите внимание на то, что коду не нужно возвращать в потоке вывода элемент для каждого элемента в потоке ввода: правительственные учреждения в поток вывода не попадают. Таким образом можно реализовывать фильтры. Проверить возможность этого кода исполнять роль обычного актора `Conduit` можно в интерпретаторе:

```
> L.sourceList [ GovOrg 1 "NASA", Individual 2 (Person "A" "S")] $$ people =$
L.consume
[Person {firstName = "A", lastName = "S"}]
```

Важно отметить, что если монада `m` заключается в оболочку `ConduitM`, результат прохождения данных через поток от `Source` к `Sink` будет находиться внутри этой монады `m`. Предположим, что нам нужно подсчитать количество правительственных организаций, являющихся клиентами. Организовать такой подсчет можно на основе простого счетчика, но в иллюстративных целях давайте сделаем это с помощью монады `State`. Поскольку `ConduitM` является преобразователем монад, нам нужно перед действиями `State` вставить вызовы функции `lift`.

```
import Control.Monad.State

countGovOrgs :: MonadState Int m => Sink (Client i) m Int
```

```
countGovOrgs = do client <- await
                  case client of
                  Nothing -> do n <- lift $ get
                                return n
                  Just c -> do case c of
                                  GovOrg { } -> lift $ modify (+1)
                                             _ -> return ()
                               countGovOrgs
```

При применении этого актора `Sink` к актору `Source` все полученное все же нужно выполнить в монаде, поддерживающей `MonadState`. В данном случае нас интересует только состояние, поэтому функцией, необходимой для получения результата, будет `execState`:

```
main = let clients = [ GovOrg 1 "Zas", Individual 2 (Person "Alejandro" "Serrano")]
           conduitGovOrgs = L.sourceList clients $$ countGovOrgs -- :: State Int ()
       in print $ execState conduitGovOrgs 0
```

В качестве еще одного примера применения библиотеки `conduit`, в котором мы сможем задействовать другие подчиненные монады, давайте реализуем промежуточный шаг «выигравших клиентов» в данной среде. По аналогии с использованием в коде функции `randomRIO` можно подумать о выводе на экран или о любом другом действии в монаде `IO`.

```
import Control.Monad.Trans

winners :: Conduit (Client i) IO (Client i, Bool, Int)
winners = do client <- await
             case client of
               Nothing -> return ()
               Just c  -> do (w :: Bool) <- lift $ randomIO
                             (y :: Int)  <- lift $ randomRIO (0, 3000)
                             yield (c, w, y)
                             winners
```

Выполнение упражнения 9.4 поможет лучше освоить идиому `conduit`.

УПРАЖНЕНИЕ 9.4. ВАРИАНТЫ ПРИМЕНЕНИЯ БИБЛИОТЕКИ CONDUIT

Переделайте списочные функции, представленные в главе 3, для работы с потоками ввода-вывода. В частности, напишите определение unfold актора Source для генерирования потока значений на основе генератора, определение функции map библиотеки conduit для применения функции ко всем элементам потока, определение функции filter для отбрасывания ряда элементов, а также определение fold актора Sink для вычисления свертки с применением двоичной операции ко всему потоку.

Доступ к файлам посредством библиотеки conduit

Проблема, подтолкнувшая нас к изучению библиотеки `conduit`, касалась не списков, а поиска более приемлемого способа доступа к файлам, повышения производительности и достижения более высокой степени предсказуемости. Настало время рассмотреть функции из модуля `Data.Conduit.Binary`, которые библиотека `conduit` предоставляет для решения этих задач.

Интерфейс довольно прост: функции `sourceFile` и `sourceHandle` генерируют поток из файла, а функции `sinkFile` и `sinkHandle` потребляют поток, записывая его в файл. Разница между двумя видами функций состоит в том, что те функции, чьи имена заканчиваются на `File`, обеспечивают открытие и закрытие дескриптора соответствующего файла, а те функции, чьи имена заканчиваются на `Handle`, должны снабжаться уже открытым дескриптором, и они *не* закрывают файл по завершении своей работы. Первая группа функций предоставляет все инструменты библиотеки `conduit`, а вторая группа позволяет легко организовать взаимодействие.

Любопытно, что эти потоки вместо значений типа `String` предлагают значения типа `ByteString`. Значения типа `ByteString` обеспечивают более эффективный способ обработки байтов данных. Более детально тип данных `ByteString`, который определен в пакете `bytestring`, рассматривается в следующей главе, там же представлены способы преобразования данных этого типа в данные типа `String`, но для нашего простого примера «выигравших клиентов», где к каждой строке должна быть добавлена дополнительная информация, достаточно просто знать, что функция `pack` преобразует данные типа `String` в данные типа `ByteString`. Соответствующий код выглядит следующим образом:

```
import qualified Data.ByteString.Char8 as BS
import qualified Data.Conduit.Binary as B
import Data.Monoid

winnersFile :: (Monad m, MonadIO m) => Conduit BS.ByteString m BS.ByteString
winnersFile = do client <- await
       case client of
         Nothing -> return ()
         Just c  -> do (w :: Bool) <- liftIO $ randomIO
                       (y :: Int ) <- liftIO $ randomRIO (0, 3000)
                       yield $ c <> BS.pack (" " ++ show w ++ " " ++ show y)
                       winnersFile
```

Как можно заметить, код `winnersFile` имеет прямое отношение не к монаде `IO`, а к типовому классу под названием `MonadIO`. Это класс всех стеков монад, поддерживающих внутри своих тел вызов действий монады `IO`. Особенность здесь в том, что посредством функции `liftIO` класс типов `MonadIO` добавляет вычисления для реализации подтягивания.

Располагая этой информацией, мы сможем привлекать всех акторов к работе по считыванию данных из файла и записи их в файл так же просто, как в следующем коде. Обратите внимание на функцию `lines`, которая делит файл на части, определяемые по наличию символов новой строки:

```
main = B.sourceFile "clients.db" $$ B.lines =$= winnersFile =$ B.sinkFile
"clientsWinners.db"
```

Однако компилятор такой код не примет. Здесь необходимо дополнительное обобщение, введенное ранее посредством ограничения класса `MonadIO`: операции, создающие акторов `Source` или `Sink` в модуле `Data.Conduit.Binary`, используют дополнительный преобразователь монад `ResourceT`. Для осуществления операции нужно добавить вызов функции `runResourceT`, которая выполнит действия в монаде:

```
main = runResourceT $
B.sourceFile "clients.db" $$ B.lines =$= winnersFile =$ B.sinkFile "clientsWinners.db"
```

Располагая данным кодом, можно гарантировать, что файлы будут открыты и закрыты тогда, когда это потребуется.

ПАКЕТ RESOURCET

В предыдущем примере было показано, как библиотека conduit использует преобразователь монад ResourceT для управления выделением и освобождением ресурсов. Этот преобразователь, который находится в пакете resourcet, может рассматриваться как обобщение функции bracket из модуля Control.Exception. В частности, можно воспользоваться его функцией allocate для получения ресурса наряду с действием освобождения, которое гарантированно произойдет, когда управление выйдет за пределы ResourceT.

Если бы была предоставлена только эта функциональность, то никаких преимуществ от использования ResourceT по сравнению с bracket мы бы не получили. Однако ResourceT позволяет также освобождать ресурсы явным образом: путем вызова функции release с идентификатором, возвращенным функцией allocate.

Этот пакет весьма полезен для реализации диспетчеров ограниченных ресурсов. Если вы намереваетесь задействовать пул ресурсов для их совместного использования, вам также нужно присмотреться к пакету pool-conduit, основой которого является пакет resourcet.

За пределами текстовых файлов

Как было упомянуто в начале главы, `IO` позволяет получать доступ к самым разным ресурсам и выполнять многие виды вычислений с побочными эффектами. Однако до сих пор наше внимание было нацелено на взаимодействие с консолью и обращение к текстовым файлам на диске. В данном разделе мы познакомимся с двумя примерами работы внутри библиотеки `conduit`, но относящимися теперь к сетевому взаимодействию и двоичной сериализации Haskell-данных.

Основы сетевого взаимодействия

Пакет `network-conduit` построен на основе библиотеки `conduit` с целью предоставления компактного и удобного интерфейса для сетевого взаимодействия. Основное упрощение достигается тем, что в этом пакете актор в сети рассматривается либо в качестве сервера, слушающего входящие соединения, либо в качестве клиента (подключенного к серверу), а также тем, что соединение всегда устанавливается по протоколу TCP. Сетевой интерфейс, предоставляемый низкоуровневыми пакетами, позволяет существенно расширить доступные варианты поведения, но на практике для большинства приложений достаточно этих двух режимов.

Как на сервере, так и на клиенте `network-conduit` кроме параметра соединения ожидает значения типа `Application`. Тип `Application m` является лишь синонимом типа `AppData m -> m ()`. Назначением этого значения должно быть получение акторов `Source` и `Sink`, предоставленных типом `AppData`, и связывание их для поддержания обмена информацией между сервером и клиентом. Обратите также внимание на параметризацию внутренней монады `m`, в которой будет выполняться тип `Application`.

Разрабатываемое сетевое приложение представляет собой еще один взгляд на «выигравших клиентов». В данном случае клиент отправляет свое имя, полученное с консоли, а сервер возвращает информацию о том, выиграл он или нет. Находящийся на сервере основной актор `Conduit` очень похож на использовавшийся в предыдущих упражнениях: единственным дополнением будет вызов функции `putStrLn` для вывода имени пользователя на экран:

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Monad.Trans
import qualified Data.ByteString.Char8 as BS
import Data.Conduit
import Data.Monoid
import System.Random

isWinner :: Conduit BS.ByteString IO BS.ByteString
isWinner = do client <- await
          case client of
            Nothing -> return ()
            Just c  -> do lift $ BS.putStrLn c
                          (w :: Bool) <- liftIO $ randomIO
                          (y :: Int ) <- liftIO $ randomRIO (0, 3000)
                          yield $ c <> BS.pack (" " ++ show w ++ " " ++ show y)
                          isWinner
```

Следующим шагом станет создание значения `Application`, которое установит контакт входного потока данных с сервером и выходного потока данных с каждым клиентом. Вдобавок можно получить доступ к значениям `appSource` и `appSink`:

```
import Data.Conduit.Network

serverApp :: Application IO
serverApp d = do appSource d $$ isWinner =$ appSink d
```

Последний шаг заключается в запуске сервера в точке входа. Одной из функций, решающих эту задачу, является `runTCPServer`. В качестве аргументов она получает порт, на котором будет осуществляться прослушивание, и вид принимаемого соединения. Возможность соединения через IPv4, IPv6 или с указанного адреса может быть ограничена. Но в данном случае приветствуется любой клиент:

```
import Network

main :: IO ()
main = withSocketsDo $ runTCPServer (serverSettings 8900 HostAny) serverApp
```

ПРИМЕЧАНИЕ

В Windows перед любым обменом данными по этому каналу нужно инициализировать сетевую подсистему. Это, как и в нашем примере, реализуется включением основного кода в вызов функции withSocketsDo из модуля Network пакета network. Сама по себе эта функция на других операционных системах не дает никакого эффекта, поэтому ее всегда нужно включать в код с целью сохранения максимальной совместимости между платформами.

На противоположной стороне, на которой находится клиент, все намного проще. После соединения, установленного функцией `runTCPClient`, он должен записать

имя, определенное командной строкой, в функцию `appSink`, которая отправит эту информацию на сервер. После возвращения информация просто выводится на экран. Полностью код выглядит так:

```
main :: IO ()
main = withSocketsDo $ do
  (name:_) <- getArgs
  runTCPClient (clientSettings 8900 (BS.pack "127.0.0.1")) (clientApp name)

clientApp :: String -> Application IO
clientApp name d = do (yield $ BS.pack name) $$ appSink d
                      appSource d $$ (do Just w <- await
                                         lift $ BS.putStrLn w)
```

Как видите, библиотека `conduit` позволяет рассматривать файлы на диске и сетевые соединения с помощью одних и тех же абстракций. Это существенно упрощает многократное использование функций преобразования данных между различными ресурсами, что в рассматриваемом примере делалось с помощью функции `isWinner`.

ПРИМЕЧАНИЕ

Пакет network для сетевого взаимодействия предоставляет более удобный интерфейс сокетов. Используя этот пакет, посредством функций sourceSocket и sinkSocket можно создавать сокеты, а также обращаться к данным через пакет network-conduit.

Двоичная сериализация

В Hackage имеются две библиотеки, поддерживающие автоматическую сериализацию Haskell-значений в двоичные данные. То есть они осуществляют преобразование значения в поток байтов, который позже может быть считан для восстановления исходного значения. Эти данные могут передаваться по сети, записываться на диск и вообще использоваться точно так же, как и любая другая закодированная в байтах информация.

Ранее упомянутыми библиотеками для двоичной сериализации являются `binary` и `cereal`. Обе они предоставляют практически одинаковый интерфейс: единственное различие заключается в том, что при использовании `binary` вы задействуете класс типов `Binary` из модуля `Data.Binary`, а при использовании `cereal` — класс типов `Serialize` из модуля `Data.Serialize`. Есть также соответствующие пакеты для сериализации через потоки ввода-вывода: `binary-conduit` и `cereal-conduit`. Разница, касающаяся понятий ленивых вычислений и производительности, имеет весьма важное значение и состоит в том, что `cereal` является строгим, а `binary` — ленивым. Это, к примеру, означает, что `binary` может справляться с бесконечными потоками ввода-вывода данных (например, поступающими через сетевое соединение), а `cereal` — нет.

Поскольку оба пакета так похожи друг на друга, в данном разделе основное внимание уделено библиотеке `binary` и пакету `binary-conduit`. Этот пакет сериализации мы уже использовали в предыдущей главе при изучении облачных решений для Haskell. Для возможности сериализации собственных типов данных в первую

очередь нужно создать экземпляр класса типов `Binary`. До выхода GHC 7.2.1 код для экземпляра приходилось создавать самостоятельно, но начиная с этой версии компилятор способен создать экземпляр за вас, поэтому в детали создания экземпляров `Binary` мы вдаваться не будем.

Предположим, нам нужно сериализовать значения типа `Person`, который был представлен в главах 2 и 3, используя возможность автоматического получения экземпляров `Binary`. Прежде всего, нужно в GHC добавить расширение `DeriveGeneric`. Затем нужно включить тип `Generic` в выражение типа `deriving`. Экземпляр `Generic` содержит информацию о конструкторах и аргументах, объявляющих определенный тип данных. Используя эту информацию, `binary` сможет создать полноценный экземпляр `Binary`, и нам останется только его объявить, но сделать это нужно будет без предоставления какой-либо реализации функций. Полностью код для `Person` имеет следующий вид:

```
{-# LANGUAGE DeriveGeneric#-}
import Data.Binary
import GHC.Generics

data Person = Person { firstName :: String, lastName :: String }
              deriving (Show, Read, Generic)

instance Binary Person
```

После получения экземпляра `Binary` вам открывается доступ к функциям `decode` и `encode`, которые занимаются соответственно прямым и обратным преобразованием значений типа `ByteString` в нужные значения. Эти функции находятся в пакете `binary-conduit`, позволяя создать либо `Source` или `Conduit` из `encode`, либо `Sink` или `Conduit` из `decode`. В следующем примере осуществляется сериализация на диск списка людей. Конкретный код, который вводит в игру пакет `binary`, выделен полужирным шрифтом:

```
import Data.Conduit
import qualified Data.Conduit.List as L
import qualified Data.Conduit.Binary as B
import qualified Data.Conduit.Serialization.Binary as S

main = runResourceT $ L.sourceList clients $$ S.conduitEncode =$ B.sinkFile
"people.db"
       where clients = [ Person "Alejandro" "Serrano", Person "The Doctor" "Who?" ]
```

Обратная операция также реализуется очень просто. Единственное отличие заключается в том, что вместо функции `conduitEncode` используется функция `conduitDecode`. Показанный далее пример в потоке ввода-вывода получает элементы типа `Person` и выводит их на экран:

```
import Control.Monad.Trans

main = runResourceT $
  B.sourceFile "people.db" $$ S.conduitDecode
                    =$ L.mapM_ (\(p :: Person) -> lift $ putStrLn $ Show p)
```

Очень важно, чтобы библиотеке `binary` было известно, какого рода значение подвергается десериализации либо по применению к нему других функций, либо путем

явной записи сигнатуры (подобно тому, как это сделано в примере). В той версии `binary`, которая использовалась при написании данной книги, если заменить `Person` на `String`, код все равно будет работать, но станет генерировать поток ввода-вывода из четырех строк вместо двух значений типа `Person`.

Разумеется, этот механизм сериализации не слишком эффективен, когда нужно запросить и преобразовать большой объем данных, поскольку в память должно быть помещено и в ней раскодировано все множество данных. Более приемлемым решением для хранения информации было бы использование системы управления базами данных. Подключению к базам данных в Haskell, а также сохранению и получению находящихся там данных посвящена глава 12.

Выводы

Данная глава посвящена созданию системы взаимодействия с «внешним миром» и изучению вопросов, связанных с побочными эффектами:

- Мы познакомились с монадой `IO` в качестве одной из монад, предлагающих контекст для *побочных эффектов* в вычислениях. Кроме того, мы выяснили, как в Haskell чистые вычисления отделяются от вычислений с побочными эффектами.
- Мы научились считывать данные с *консоли* и из *файлов*, а также записывать туда данные, используя «классический» ввод-вывод.
- Мы узнали еще об одном источнике побочных эффектов — *случайности*, предоставляемой пакетом `random`.
- Мы пересмотрели и расширили свои навыки обработки *ошибок* в чистом коде. В частности, мы познакомились с типом `Either`.
- Вычисления с побочными эффектами открывают дверь *исключениям*, которые предлагают расширяемый, но не отличающийся чистотой способ обработки ошибочных сценариев внутри монады `IO`.
- «Классический» ввод-вывод в условиях лени имеет ряд недостатков. Решение всех связанных с этим проблем, основанное на абстракции потоков ввода-вывода, заключается в использовании библиотеки `conduit`.
- И наконец, мы научились использовать библиотеку `conduit` в нескольких сценариях, таких как считывание данных из файлов и их запись в файлы, обмен данными по сети и сериализация Haskell-значений.

Создание и синтаксический разбор текста 10

В предыдущей главе мы открыли двери побочным эффектам и, в частности, обсудили сохранение данных Haskell-приложений на диске и передачу их по сети, но единственный представленный в ней механизм реальной сериализации имел двоичную природу и относился к пакету `binary`.

Однако во многих случаях предпочтение отдается не двоичному, а *текстовому представлению* данных. Хотя оно, как правило, больше по объему, данные, закодированные в виде текста, легче воспринимаются человеком, что существенно упрощает изучение и отладку программ. Более того, в виде текста данные кодируются во многих форматах обмена данными, таких как XML или JSON, в которых к обычной информации добавляются теги для ее структурирования.

Для работы с текстом в Haskell имеются весьма широкие возможности, на которых и строится материал данной главы. Как мы увидим, встроенный тип данных `String` является во многих случаях не самым лучшим выбором по сравнению с более эффективными альтернативными типами `ByteString` и `Text`. Кроме того, для создания на выходе текста не следует пользоваться простой конкатенацией строк — в программах должны использоваться *построители* (builders).

Существует и обратная операция, которая называется *синтаксическим разбором*, или *парсингом* (parsing), и заключается в декодировании информации из текстового представления. В Haskell синтаксический разбор выполняется с помощью комбинаторов, примером данного подхода могут служить функции библиотеки `attoparsec`. Интересно то, что для синтаксического разбора вводятся новые классы типов, наиболее важным из которых является `Applicative`.

Пять текстовых типов данных

Правильная трактовка текста является сложной задачей почти во всех языках программирования. За простым понятием текстовых данных всегда стоят разные тонкости, связанные с кодировкой и представлением в памяти. В данном разделе рассказывается, как обрабатывать текст в Haskell-коде наиболее подходящим и эффективным способом.

До сих пор мы работали с текстовыми значениями в рамках типа данных `String`, который является простым синонимом типа `[Char]`, то есть списка символов. Главным

преимуществом типа `String` является простота модели, позволяющая любой функции, способной работать со списками, работать также и с текстом. Поскольку списки являются неотъемлемой частью Haskell, это весьма важное свойство. К еще одному достоинству типа `String` относится поддержка кодировки Unicode. Возможно, вы еще в этом не убедились, но любой код, написанный с применением типа `String`, способен отлично работать с китайскими, греческими и кириллическими символами.

Недостаток типа `String` заключается в том, что за простоту применения приходится расплачиваться производительностью. Даже при том, что GHC проводит всестороннюю оптимизацию данных типа `String` по сравнению с простым представлением, используемым в других языках, избежать издержек как по времени, так и по расходованию памяти не удается. При работе с большими объемами текста (например, при генерировании вывода для веб-приложения) требуется более высокая производительность.

В этом плане две вещи вступают в противоречие, и это противоречие желательно сбалансировать:

- С одной стороны, хотелось бы видеть строку в качестве простого *потока байтов* (stream of bytes). Если необработанным байтам не придается никакого дополнительного значения, производительность можно существенно повысить, поскольку операционные системы обычно предоставляют специализированные операции для очень быстрого перемещения больших блоков байтов как в память, так и на диск.
- С другой стороны, у этих байтов есть *предполагаемое значение* (intended meaning). Чтобы восстановить это значение, нужно учесть используемую данными кодировку: одни и те же байты могут иметь совершенно разное значение, когда интерпретируются как набор только латинских символов или когда для их декодирования используется кодировка UTF-8. Таким образом, если в строке нужны реальные символы в той или иной кодировке, на выполнение операций над ними следует обратить особое внимание. Нужно нацелиться на получение типа данных, который позволяет достичь наивысшей производительности при условии правильности обработки данных.

Haskell Platform разделяет тип данных `String` на два других типа. Когда вам нужна очень быстрая реализация, а текст рассматривается только как необработанные байты, следует выбрать тип `ByteString` из пакета `bytestring`. В то же время тип `Text` из пакета `text` нацелен на описание строк в кодировке Unicode при сохранении довольно высокой производительности.

ПРИМЕЧАНИЕ

Тип данных String отлично подходит для проведения экспериментов и для того этапа разработки, на котором идет создание прототипов, поскольку использование типа String в Haskell существенно упрощено. Однако при написании кода для реальных приложений следует предпочесть тип ByteString или Text. Помимо прочего, этот выбор поможет решить проблемы кодировки.

Кроме этих различий, касающихся того, чему нужно уделить основное внимание, есть и другие аспекты, требующие рассмотрения. В обеих библиотеках для соответствующих данных предоставляются версии как для строгих, так и для ленивых

вычислений. Вспомним о едва заметном балансе строгости и лени: использование строгих версий `ByteString` или `Text` означает, что в конечном итоге вы можете столкнуться с необходимостью вычислять такие объемы текстовых данных, которые превысят ваши ожидания. В то же время лень может поставить перед вами ряд проблем, о которых шла речь в предыдущей главе.

В табл. 10.1 отражены эти два аспекта и показаны модули, которые нужно импортировать для поддержания каждого варианта функциональности. Обратите внимание на то, что для ленивых вычислений данных в кодировке Unicode можно использовать как тип данных `Text`, так и встроенный тип данных `[Char]`.

Таблица 10.1. Модули Haskell Platform для текстовых данных

	Строгие вычисления	**Ленивые вычисления**
Необработанный поток байтов	Data.ByteString	Data.ByteString.Lazy
Данные в кодировке Unicode	Data.Text	Data.Text.Lazy или [Char]

Не менее важным по сравнению с принятием решения об используемом типе данных является значение порядка преобразования данных между различными представлениями. К примеру, может понадобиться вставить некое значение типа `String`, полученное путем вызова функции `getLine`, в значение типа `Text`. Еще одним распространенным сценарием является получение данных из файла (или по сети) в виде потока байтов, который описывается как значение `ByteString`, и декодирование его в значение `Text`. Все возможные пути между двумя этими типами сведены в схему, показанную на рис. 10.1.

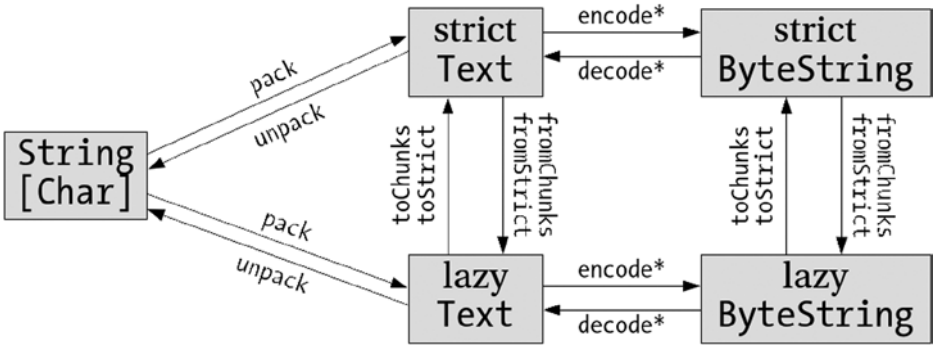


Рис. 10.1. Преобразования между типами текстовых данных

По поводу схемы на рис. 10.1 следует отметить два момента. Во-первых, преобразования между строгими и ленивыми типами могут быть реализованы посредством функций `toStrict` и `fromStrict`, а также `toChunks` и `fromChunks`. Все эти функции доступны в модулях, соответствующих ленивым версиям, то есть в `Data.Text.Lazy` и `Data.ByteString.Lazy`. Разница в том, что функции с окончанием `Strict` получают или производят строгие версии, в то время как функции с окончанием `Chunks` получают или производят *списки* из строгих значений. Эти две сигнатуры имеют следующий вид:

```
toStrict :: Data.ByteString.Lazy.ByteString -> Data.ByteString.ByteString
toChunks :: Data.ByteString.Lazy.ByteString -> [Data.ByteString.ByteString]
```

Это небольшое различие позволяет сохранить некоторую долю лени даже со строгими версиями, поскольку элементы в списке будут вычисляться только по мере необходимости.

Между `Text` и `ByteString` нет единого варианта преобразования, а в модулях `Data.Text.Encoding` и `Data.Text.Lazy.Encoding` доступно целое семейство вариантов. Каждая из функций содержит в своем имени ту кодировку, которая должна использоваться для придания смысла необработанному потоку байтов. Например, `encodeUtf8` преобразует текст в байтовое представление, используя для этого кодировку UTF-8, а `decodeLatin1` осуществляет операцию преобразования, но с кодировкой Latin-1.

ПРИМЕЧАНИЕ

В модулях Data.ByteString.Char8 и Data.ByteString.Lazy.Char8 определены функции непосредственного преобразования из String в ByteString. Но в них каждый символ рассматривается как восьмиразрядное значение, а любая дополнительная информация отбрасывается (большинство символов нуждается для кодирования в более чем одном байте). Таким образом, использование функций, имеющихся в этих модулях, может привести к потере информации, поэтому при работе с ними нужно проявлять особую осмотрительность.

Независимо от преобразований между типами внешний интерфейс `ByteString` и `Text` почти одинаков. Поэтому импорт соответствующих модулей зачастую осуществляется в подходящей форме. Функции также очень похожи на списочные функции, которые можно использовать с типами данных `String`. Основное различие состоит в том, что в значениях `Text` и `ByteString` не определяются конструкторы, поэтому для добавления одного символа с помощью функции `cons` нужно переключаться на функцию `(:)` и на сопоставление с образцом с помощью функций `head` и `tail`. В то же время, как показано в следующем примере, обычные функции, такие как `map`, `reverse`, `foldr` или `foldl`, свои имена сохраняют:

```
*Chapter10.Text> import qualified Data.Text as T
*Chapter10.Text T> :t T.pack "Hello Text type"
T.pack "Hello Text type" :: T.Text
*Chapter10.Text T> import Data.Char
*Chapter10.Text T Data.Char> T.map toUpper (T.pack "Hello Text type")
"HELLO TEXT TYPE"
*Chapter10.Text T Data.Char> T.intercalate (T.pack ",") (map T.pack
["apples","oranges"])
"apples,oranges"
*Chapter10.Text T Data.Char> T.length (T.pack "A Text example")
14
```

И `Text`, и `ByteString` являются экземплярами класса типов `Monoid`. Это весьма полезно, поскольку оператор `(<>)` эквивалентен объединению — довольно распространенной операции над данными этих типов:

```
*Chapter10.Text T Data.Char Data.Monoid> (T.pack "Hello ") <> (T.pack "Beginning
Haskell")
"Hello Beginning Haskell"
```

Однако при использовании `Text` или `ByteString` вместо `String` один недостаток все же проявляется: код для преобразования из литералов наполняется вызовами `T.pack`. Из-за популярности данных пакетов разработчики GHC ввели расширение, позволяющее записывать постоянные значения непосредственно в виде строковых литералов, делая ответственным за вставку соответствующих вызовов `pack` компилятор (во многих случаях при дальнейшей оптимизации литералы компилируются непосредственно в свое представление в типе данных `Text` или `ByteString`). Это расширение называется `OverloadedStrings`. В следующем примере оно включается в GHCi и используется для непосредственного создания значения типа `Text`:

```
*Chapter10.Text T Data.Char Data.Monoid> :set -XOverloadedStrings
*Chapter10.Text T Data.Char Data.Monoid> T.map toUpper "Hello Text type"
"HELLO TEXT TYPE"
```

Следует знать, что в некоторых случаях может понадобиться ввести явную сигнатуру типа, чтобы зафиксировать тип создаваемого текстового значения. Однако в большинстве случаев решение может принять реализованный в Haskell механизм вывода типа.

В предыдущей главе мы познакомились с функциями модуля `Data.Conduit.Binary`, создающими акторов `Source` и `Sink` для обработки файлов на диске, и функциями модуля `Data.Conduit.Network`, делающими то же самое для сетевых соединений. При этом типом потоков ввода и вывода был тип `ByteString`, поддерживающий применение необработанных байтов. Что же касается обработки данных типа `Text`, их нужно преобразовывать с учетом кодировки с помощью актора `Conduit` из модуля `Data.Conduit.Text`. Функции в этом модуле получают в качестве аргумента кодек, к примеру `utf8`. Окончательная и полностью корректная версия приложения «выигравших клиентов», в которой используется библиотека `conduit`, выглядит следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Conduit
import qualified Data.Conduit.Binary as B
import qualified Data.Conduit.List as L
import qualified Data.Conduit.Text as T
import Data.Text

main :: IO()
main = runResourceT $
  B.sourceFile "clients.db" $$ T.decode T.utf8 =$=
  T.lines =$= winnersFile =$= L.concatMap (\x -> [x, "\n"]) =$=
  T.encode T.utf8 =$ B.sinkFile "clientsWinners.db"

winnersFile :: (Monad m, MonadIO m) => Conduit Text m Text
winnersFile = ... -- то же, что в предыдущей главе, но для типа Text
```

ПРИМЕЧАНИЕ

Помимо библиотеки conduit к файлам можно обращаться напрямую как к данным типа Text или ByteString, используя для этого классический ввод-вывод. Функции для такого обращения, доступные в модулях Data.Text.IO и Data.ByteString.IO, имеют те же имена, что и их конкуренты для типа String.

Создание текста со скоростью ветра

До сих пор в примерах мы видели, как разными способами можно работать с фрагментами текстовых данных, но полноценного примера, в котором список клиентов сохраняется на диске, еще не было. В любом случае трудностей быть не должно: один из способов заключается просто в создании большого значения типа `Text` с последующим его сохранением с помощью библиотеки `conduit`.

Поскольку основное назначение данных — их последующее считывание другими приложениями, представление этих данных нужно как-то упорядочить. В частности, тип `Client` должен кодироваться следующим образом:

- Каждый клиент записывается в собственной строке. Таким образом, файл содержит список клиентов, отделенных друг от друга символами новой строки. Это позволит впоследствии быстро считывать информацию с помощью функции `lines`.
- Данные каждого клиента начинаются со слова `client`, а поля, составляющие информацию, заключаются в круглые скобки и представляют собой список с запятыми в качестве разделителей.
- В первом поле каждой строки указывается вид клиента: `gov` (правительственная организация), `com` (компания) или `ind` (индивидуальный клиент).
- Поскольку символы новой строки, запятой и круглых скобок используются для особых целей, внутри каждого представления перед каждым таким символом должен указываться экранирующий символ: `\n`, `\,`, `\(` и `\)`.
- Правила для внедренного типа данных `Person` будут такими же, но вместо слова `client` информация будет предваряться словом `person`.

Реализация с типом данных `Text` выглядит довольно просто:

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Conduit
import qualified Data.Conduit.Binary as B
import qualified Data.Conduit.List as L
import qualified Data.Conduit.Text as T
import Data.Monoid
import Data.Text

saveClients :: FilePath -> [Client Int] -> IO ()
saveClients fpath clients = runResourceT $
  L.sourceList clients $$ L.map clientToText
    =$= L.concatMap (\x -> [x, "\n"])  -- write '\n' between clients
    =$= T.encode T.utf8 =$ B.sinkFile fpath

clientToText :: Client Int -> Text
clientToText (GovOrg  i n)     =
  "client(gov," <> escapeString (show i) <> "," <> escapeString n <> ")"
clientToText (Company i n p d) =
  "client(com," <> escapeString (show i) <> "," <> escapeString n <> ","
    <> personToText p <> "," <> escapeString d <> ")"
clientToText (Individual i p)  =
  "client(ind," <> escapeString (show i) <> "," <> personToText p <> ")"
```

```
personToText :: Person -> Text
personToText (Person f l) = "person(" <> escapeString f <> "," <> escapeString l <>
")"

escapeString :: String -> Text
escapeString = replace "\n" "\\n" . replace ","  "\\," .
               replace "("  "\\(" . replace ")"  "\\(" . pack
```

Несмотря на простоту, этот код крайне неэффективен. При каждом объединении двух элементов должно создаваться новое значение типа `Text`, что вызывает издержки на выделение памяти, на копирование данных, а также на отслеживание значения и его освобождение по истечении в нем надобности. Кроме того, преобразование целочисленных значений в значения типа `Text` влечет за собой промежуточное преобразование в данные типа `String` посредством функции `show`.

Оба пакета, и `text`, и `bytestring`, предоставляют тип данных `Builder` (построитель), который может использоваться для эффективного генерирования больших текстовых значений. Трюк состоит в том, что сам построитель значением не является, но в нем закодирован алгоритм создания больших текстовых значений из составных частей. То есть вам нужно просто выполнить построитель, который в итоге сгенерирует требуемую строку. На последнем этапе выполнения можно проделать массу оптимизаций. Например, построитель знает, насколько длинным будет конечный результат, поэтому появляется возможность сразу выделить столько памяти, сколько потребуется.

Значения типа `Builder` состоят из трех основных блоков. Можно создать значение этого типа из одного символа, воспользовавшись функцией `singleton`, или построить его из более длинной строки, используя одну из функций, `fromString`, `fromText` или `fromLazyText`. Однако в следующем коде используется расширение `OverloadedStrings`, позволяющее создать значение типа `Builder` из символьной строки. И наконец, значения типа `Builder`, так же как и значения типа `Text`, объединяются с помощью `Monoid (<>)`. Следующий код полностью аналогичен предыдущему, но в нем вместо значений типа `Text` используются значения типа `Builder`:

```
import Data.Text.Lazy
import qualified Data.Text.Lazy.Builder as B
import qualified Data.Text.Lazy.Builder.Int as B

clientToText :: Client Int -> B.Builder
clientToText (GovOrg i n) =
  "client(gov," <> B.decimal i <> B.singleton ','
                <> B.fromText (escapeString n) <> B.singleton ')'
clientToText (Company i n p d) =
  "client(com," <> B.decimal i <> B.singleton ','
                <> B.fromText (escapeString n) <> B.singleton ','
                <> personToText p <> B.singleton ','
                <> B.fromText (escapeString d) <> B.singleton ')'
clientToText (Individual i p) =
  "client(ind," <> B.decimal i <> B.singleton ','
                <> personToText p <> B.singleton ')'

personToText :: Person -> B.Builder
```

```
personToText (Person f l) =
  "person(" <> B.fromText (escapeString f) <> B.singleton ','
            <> B.fromText (escapeString l) <> B.singleton ')'

escapeString :: String -> Text
escapeString = replace "\n" "\\n" . replace ","  "\\," .
               replace "("  "\\(" . replace ")"  "\\(" . pack
```

Можно заметить, что дополнительно в код был импортирован модуль `Data.Text.Lazy.Builder.Int`. Этот модуль обеспечивает доступ к комбинатору `decimal`, являющемуся весьма быстродействующим построителем для преобразования целочисленных значений в текст. Если вы отдаете предпочтение числам по основанию 16, то вместо него можно воспользоваться комбинатором `hexadecimal`.

Для получения конечного значения типа `Text` нужно просто вызвать для построителя функцию `toLazyText`. Исходя из ее имени вы получите не строгое, а *ленивое* значение типа `Text`. Следует помнить, что в случае необходимости для преобразования этого значения в строгое у вас всегда есть функция `toStrict`. Давайте рассмотрим пример создания в GHCi описания для клиента:

```
> B.toLazyText $ clientToText (Company 1 "Black Hole Inc." (Person "John" "Smith")
"Traveller")
"client(com,1,Black Hole Inc.,person(John,Smith),Traveller)"
```

Поскольку вместо ленивого типа данных в `conduit` используются данные типа `ByteString`, новая версия `saveClients`, применяющая построители, должна в определенном месте вызывать функцию `toStrict`. Это отражено в реализации:

```
saveClients fpath clients = runResourceT $
  L.sourceList clients $$ L.map clientToText =$= L.map (toStrict . B.toLazyText)
    =$= L.concatMap (\x -> [x, "\n"])  -- между клиентами записывается '\n'
    =$= T.encode T.utf8 =$ B.sinkFile fpath
```

Библиотека `bytestring` предоставляет построители как для строгих, так и для ленивых данных типа `ByteString` в модулях `Data.ByteString.Builder` и `Data.ByteString.Lazy.Builder` соответственно. Следует заметить, что назначение построителя типа `ByteString` заключается в производстве не текстового контента, а потока необработанных байтов. Поэтому он может использоваться для создания вашего собственного *двоичного* формата с помощью новых комбинаторов. Например, библиотека предоставляет такие функции, как `in16LE` или `word64BE`, которые непосредственно включают какое-нибудь значение указанного количества разрядов (8, 16, 32 или 64) с прямым или обратным порядком следования байтов непосредственно в виде необработанных данных.

ПРИМЕЧАНИЕ

Запомните, что при генерировании большого значения типа Text или длинного потока байтов нужно использовать построители. С их помощью при сохранении выразительности кода достигается намного более высокая производительность, чем при простой конкатенации строк.

Представленные здесь приемы, ориентированные на клиентов, могут использоваться для записи на диск информации о товарах и покупках. Руководством для решения этой задачи послужит упражнение 10.1.

УПРАЖНЕНИЕ 10.1. ПОСТРОЕНИЕ ТОВАРОВ И ПОКУПОК

Напишите построители для следующих типов данных, которые кодируют товары и покупки, сделанные в магазине машин времени:

```
data Product = Product { id :: Int, name :: String, price :: Double, descr :: String }
data Purchase = Purchase { client :: Client Int, products :: [Product] }
```

Вы можете использовать любое представление по своему усмотрению. Но при этом имейте в виду, что нужно корректно применять экранирующие символы, чтобы впоследствии другие приложения смогли прочитать файл.

Синтаксический разбор с использованием пакета attoparsec

При всей полезности эффективного генерирования выходной информации зачастую возникает необходимость обратного считывания значений в приложение. Следуя примеру с магазином, текстовый формат можно использовать для обмена информацией между различными приложениями: веб-интерфейсом, бухгалтерской программой, программой инвентаризации и т. д.

Наиболее эффективным инструментом решения проблемы распознавания образцов является *парсер* (parser). Обычно описание грамматики (то есть структуры, которой будут соответствовать строки) дается в отдельном файле, а не в файле с исходным кодом. Парсер преобразует это описание в исполняемый код на конкретном языке программирования. Наиболее известными примерами таких парсеров являются bison и yacc для C, JavaCC для Java и Happy для Haskell.

В данном разделе основное внимание уделяется не парсерам, а пакету `attoparsec`, который имеется в Hackage. При его использовании грамматика описывается с помощью набора комбинаторов *внутри* Haskell-кода. Преимущество такого подхода заключается в том, что при написании парсера вы можете опираться на модульность и многократно использовать функциональные возможности языка программирования Haskell. Например, можно объявить функцию, которой в качестве аргумента передается парсер, а она сгенерирует еще один парсер, такой же, как заданный в качестве аргумента, но выполняемый дважды.

ПРИМЕЧАНИЕ

В следующих примерах кода будет использоваться модуль Data.Attoparsec.Text, создающий парсеры, работающие со значениями типа Text. Если же вас интересует синтаксический разбор значений типа ByteString, нужно воспользоваться модулем Data.Attoparsec.

Простейшим примером может послужить парсер, которому для успешной работы нужно провести сопоставление с конкретной строкой. Соответствующий комбинатор, вполне естественно названный `string`, в качестве аргумента получает сопоставляемую строку. Давайте попробуем запустить в интерпретаторе парсер, успешно завершающий работу только в том случае, если в разных строках имеется подстрока `hello`:

```
*Chapter10.Parsing> :set -XOverloadedStrings
*Chapter10.Parsing> import Data.Attoparsec.Text
```

```
*Chapter10.Parsing Data.Attoparsec.Text> parse (string "hello") "hello"
Done "" "hello"
*Chapter10.Parsing Data.Attoparsec.Text> parse (string "hello") "hello world"
Done " world" "hello"
*Chapter10.Parsing Data.Attoparsec.Text> parse (string "hello") "bye bye"
Fail "bye bye" [] "Failed reading: takeWith"
*Chapter10.Parsing Data.Attoparsec.Text> parse (string "hello") "bye"
Partial _
*Chapter10.Parsing Data.Attoparsec.Text> parse (string "hello") "he"
Partial _
*Chapter10.Parsing Data.Attoparsec.Text> feed (parse (string "hello") "he") "llo"
Done "" "hello"
```

Как видите, функция, которая в конечном счете исполняет код парсера, называется `parse`. В качестве первого аргумента она получает описание декодируемого формата, а в качестве второго — сопоставляемую строку. Однако результат синтаксического разбора может оказаться неожиданным:

- При выполнении кода парсера *обрабатывается только часть строки*, а необработанная часть остается для дальнейшей обработки. Конструктор `Done`, предназначенный для результатов, заключает в оболочку как ту часть строки, которая не была задействована в процессе синтаксического разбора, так и часть, успешно прошедшую сопоставление. В данном примере синтаксический разбор строки `"hello"` не возвращает остаток строки, и после его применения к строке `"hello world"` часть строки `" world"` остается доступной.
- В некоторых случаях выполнение кода парсера приводит к сбою, который описывается конструктором результата `Fail`.
- Третьей возможной неожиданностью является возвращение части строки. Это побуждает нас учитывать еще одно важное свойство пакета `attoparsec`: он выполняет синтаксический разбор строк *пошагово*. В данном примере, когда вами предоставляется строка `"he"`, пакету `attoparsec` не хватает данных, чтобы завершить выполнение парсера, поэтому он предлагает вам функцию обратного вызова, которую можно использовать для дополнительного ввода. Это делается с помощью функции `feed`: в данном случае, если предоставить строку `"llo"`, синтаксический разбор выполняется успешно.

В `attoparsec` включен ряд встроенных парсеров для различных чисел, например `decimal`, `hexadecimal` и `signed` для целочисленных значений, `rational` для точных дробей и `double` для значений с плавающей точкой. Давайте попробуем ими воспользоваться:

```
*Chapter10.Parsing Data.Attoparsec.Text> parse decimal "78 cents"
Done " cents" 78
```

Как видите, возвращается не строка, а *фактическое целочисленное значение*, которое может быть выделено парсером из начала строки. Это еще одно важное свойство `attoparsec`: создаваемые парсеры в ходе своей работы генерируют Haskell-значения. Тип результата кодируется в переменной типа, относящейся к типу `Parser`, которая представляет основные блоки пакета `attoparsec`. Например, типом десятичного числа является `Integral a => Parser a`.

Строковые и числовые парсеры являются основными блоками, которые могут использоваться для построения более крупных блоков. Для этого понадобятся комбинаторы, объединяющие парсеры определенным образом. Первый из них, (<|>), представляет *дизъюнкцию* двух парсеров: если первый из парсеров не выдаст успешный результат некоторого ввода, произойдет попытка применения второго парсера. Предположим, что нужно провести сравнения либо с `hello`, либо с `bye`. Как это сделать, показывает следующий GHCi-вывод. Обратите внимание на то, что комбинатор (<|>) находится в модуле `Control.Applicative` (вы вскоре узнаете почему), а также на то, что в коде используется вариант функции синтаксического разбора `parseOnly`, возвращающий успешный или неуспешный результат работы парсера с помощью значения `Either`:

```
*> import Control.Applicative
*> parseOnly (string "hello" <|> string "bye") "hello"
Right "hello"
*> parseOnly (string "hello" <|> string "bye") "bye!"
Right "bye"
*> parseOnly (string "hello" <|> string "bye") "aloha" -- должна быть неудача
Left "Failed reading: takeWith"
```

ПРИМЕЧАНИЕ

Типы, получающиеся в результате работы соединенных таким образом парсеров, должны совпадать.

Следующим шагом является объединение нескольких парсеров в *последовательность*. Здесь нужно учесть одну небольшую характерную особенность: поскольку каждый парсер возвращает результат, `attoparsec` требует информации о том, как объединять результаты. Это делается посредством комбинаторов (<$>) и (<*>). Первый комбинатор указывается сразу же после функции, которая будет объединять результаты, а второй служит для перебора каждого из аргументов в том случае, если у функции их больше одного. Чтобы прояснить ситуацию, давайте определим парсер, получающий строку `hello` или `bye`, за которой следует число, и создающий значение типа `GreetingYear`:

```
{-# LANGUAGE OverloadedStrings #-}
import Control.Applicative
import Data.Text
import Data.Attoparsec.Text

data GreetingYear = GreetingYear Text Int

greetingYearParser :: Parser GreetingYear
greetingYearParser = GreetingYear <$> (string "hello" <|> string "bye") <*> decimal
```

Код вообще без синтаксического разбора надо было бы писать с применением вместо комбинатора (<$>) прикладной функции ($), а вместо комбинатора (<*>) простого пробела. Как видите, интерфейс комбинаторов для парсеров похож на обычный Haskell-код, что упрощает его использование. С помощью комбинаторов (<$>) и (<*>) вы *подтягиваете* свои обычные функции для работы со значениями, заключенными в оболочку `Parser`. Взгляните на типы этих двух операций:

```
(<$>) :: (a -> b)        -> Parser a -> Parser b
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

Первая из них является, по сути, вашей старой знакомой функцией `fmap`, заставляющей чистую функцию работать с элементами функтора. `Parser`, конечно же, является экземпляром класса `Functor`. Вспомните предыдущий пример:

```
GreetingYear                                        :: Text -> (Int -> GreetingYear)
(string "hello" <|> string "bye")                   :: Parser Text
GreetingYear <$> (string "hello" <|> string "bye") :: Parser (Int -> GreetingYear)
```

Функция `(<*>)` относится к разряду тех вспомогательных функций, которые помогают передать функции больше аргументов. Или же, с иной точки зрения, `(<*>)` внутри `Parser` извлекает функцию из оболочки и применяет один аргумент:

```
GreetingYear <$> (string "hello" <|> string "bye")              :: Parser (Int ->
GreetingYear)
decimal                                                         :: Parser Int
GreetingYear <$> (string "hello" <|> string "bye") <*> decimal :: Parser
GreetingYear
```

Благодаря каррированию вы можете поочередно применять комбинатор `(<*>)` к каждому аргументу функции с несколькими аргументами.

Иногда для создания пригодного к поддержке значения типа `Parser` одних лишь комбинаторов `(<$>)` и `(<*>)` может быть недостаточно. Предположим, к примеру, что вам нужно распознать точно такой же образец, как и показанный ранее, но с пробелом между словом приветствия и числом. Для этого у вас уже есть почти все строительные блоки, но как все же распознать отдельный символ? Это делается с помощью функции `char`. Теперь можно написать код:

```
greetingYearParserS :: Parser GreetingYear
greetingYearParserS = (\g _ y -> GreetingYear g y) <$>
                      (string "hello" <|> string "bye") <*> char ' ' <*> decimal
```

Обратите внимание на то, что весьма симпатичное применение типа `GreetingYear` было заменено абстракцией, единственным назначением которой является игнорирование некоторых из этих аргументов. Если бы вы изменили данный парсер (например, требуя дополнительный символ `!` после приветствия), то вам бы пришлось собрать новую абстракцию и позаботиться об игнорировании новых ненужных данных. Понятно, что возможности поддержки такого кода хуже. Модуль `Control.Applicative` предлагает версию `(<*>)`, а именно `(<*)`, которая описывает случай, когда часть ввода должна пройти синтаксический разбор, но не должна использоваться для построения более крупной структуры. С применением этой версии показанный ранее код приобретает следующий вид:

```
greetingYearParserS :: Parser GreetingYear
greetingYearParserS = GreetingYear <$> (string "hello" <|> string "bye") <* char '
' <*> decimal
```

Теперь настало время создать парсер для вывода в последнем разделе значений типа `Client`. Сначала понадобится реализовать синтаксический разбор имен с учетом ранее изложенных правил экранирования символов (вспомним, что такие символы, как `\`, `n`, `(` и `)`, должны заменяться парой из обратного слеша и само-

го символа). Давайте создадим парсер, возвращающий единственный символ: нам пригодится комбинатор `satisfy`, соответствующий любому символу, удовлетворяющему конкретному предикату, и функция `notInClass`, возвращающая значение `True` только в том случае, если элемент не входит в определенный набор. Кроме того, если встретятся символ \ и подобные комбинации, хотелось бы вернуть соответствующий одиночный символ, для которого используется функция `const`:

```
aChar :: Parser Char
aChar =     (const ',')  <$> (string "\\,") <|> (const '\n') <$> (string "\\n")
        <|> (const '(')  <$> (string "\\(") <|> (const ')')  <$> (string "\\)")
        <|> satisfy (notInClass ",\n()")
```

Замысел состоит в том, чтобы вызывать этот парсер повторно до тех пор, пока не будет найден какой-нибудь несопоставляемый ввод. Одним из способов получения такого результата могло бы стать создание нового значения типа `Parser`, которое стало бы рекурсивно вызывать само себя. При выполнении каждого шага происходило бы добавление символа к значению типа `String`, которое было бы возвращено позже. Но вам нужен также и базовый код, применяемый, если не будет найден соответствующий символ. Способ создания значения типа `Parser`, возвращающего какое-нибудь значение без получения ввода, требует такого фрагмента `pure :: a -> Parser a`. Со всеми этими составляющими код приобретет следующий вид:

```
aString :: Parser String
aString = ((:) <$> aChar <*> aString) <|> (pure "")
```

Этот образец весьма распространен в парсерах (если вы знаете теорию грамматики, данная операция называется «звезда Клини» и является одной из основных в этой теории). Поэтому `Control.Applicative` предлагает функцию `many`, которая просто повторно применяет парсер, пока не будет найден сопоставляемый ввод. Результатом станет список, состоящий из типов внутренних парсеров, в данном случае — `[Char]`.

```
aString = many aChar
```

Узнав, как выполняется синтаксический разбор строк с экранирующими символами и целых чисел, вы можете создать парсер для типов `Person` и `Client`. Показанный далее код несложен: единственным нововведением является прикладная функция `(<$)`, игнорирующая первый аргумент. Ее назначение аналогично назначению функции `(<*)`.

```
aPerson :: Parser Person
aPerson = Person <$ string "person(" <*> aString <* char ',' <*> aString <* char
')'

aClient :: Parser (Client Int)
aClient =     GovOrg     <$ string "client(gov," <*> decimal <* char ',' <*>
aString <* char ')'
          <|> Company    <$ string "client(com," <*> decimal <* char ',' <*>
aString <* char ','
                         <*> aPerson <* char ',' <*> aString <* char ')'
          <|> Individual <$ string "client(ind," <*> decimal <* char ',' <*>
aPerson <* char ')'
```

Проверить работу парсера по созданию выходных данных можно с помощью интерпретатора:

```
*> let b = clientToText $ Company 1 "Black Hole Inc." (Person "John" "Smith")
"Traveller"
*> let c = Data.Text.Lazy.toStrict $ Data.Text.Lazy.Builder.toLazyText b
*> parseOnly aClient c
Right (Company {clientId = 1, clientName = "Black Hole Inc.",
                person = Person {firstName = "John", lastName = "Smith"}, duty =
"Traveller"})
```

Чтобы завершить это краткое знакомство с пакетом `attoparsec`, давайте рассмотрим модуль `Data.Attoparsec.Combinator`. В нем можно найти комбинаторы парсеров, соответствующие многим другим типичным образцам. Например, там имеется функция `option`, помогающая в создании парсеров с дополнительным вводом. При вызове `option` используются два аргумента: вторым из них является парсер, пытающийся выполнить сопоставление вводимых данных, а если это не удается, вызов `option` ведет к возвращению значения, заданного в качестве первого аргумента.

Для сопоставления со списками элементов разными способами помогает множество других комбинаторов. Например, комбинатор `sepBy` выполняет синтаксический разбор списка с определенными разделителями между элементами. Он может подойти и к списку клиентов: `Parser` для данного случая должен осуществлять повторный вызов `aClient`, но при этом ожидать между элементами присутствия символов новой строки:

```
parseClients :: Parser [Client Int]
parseClients = sepBy aClient (char '\n')
```

Этот модуль также отличается наличием функций для синтаксического разбора списков, причем имена таких функций оканчиваются на единицу, например `many1`, `sepBy1` и т. д. Для успешного завершения работы функции в списке должен содержаться хотя бы один элемент.

В упражнении 10.2 от вас потребуется провести синтаксический разбор продуктов и покупок из упражнения 10.1. Это упражнение можно использовать для получения навыков работы с интерфейсом `attoparsec` и множеством предоставляемых им комбинаторов.

УПРАЖНЕНИЕ 10.2. СИНТАКСИЧЕСКИЙ РАЗБОР ТОВАРОВ И ПОКУПОК

Создайте парсеры для выходной информации, которая может быть сгенерирована функциями, созданными при выполнении упражнения 10.1. Подсказка: сначала прочитайте документацию по attoparsec, имеющуюся в Hackage, чтобы лучше разобраться во встроенных парсерах, предлагаемых этим пакетом.

По аналогии с тем, как `Builder` может работать с библиотекой `conduit`, `Parser` может использоваться для создания потока значений из текстового ввода. В данном случае парсер может стать либо актором `Sink`, возвращающим значения как тако-

вые, либо актором Conduit, предоставляющим информацию для дальнейшей обработки. Эта функциональность предоставляется модулем Data.Conduit.Attoparsec из пакета attoparsec-conduit.

При использовании этого пакета создание функций, загружающих всех клиентов в файл, сгенерированный построителем в последнем разделе, становится простым объединением трех акторов: Source считывает содержимое файла в тип ByteString, Conduit декодирует данные в значения типа Text, и, наконец, Sink выполняет синтаксический разбор этой информации.

```
loadClients :: FilePath -> IO [Client Int]
loadClients fPath = runResourceT $
  B.sourceFile fPath $$ T.decode T.utf8 =$ sinkParser parseClients
```

Представление новых классов типов

Тот факт, что (<$>) и (<*>) не находятся в каком-либо из конкретных модулей attoparsec, а располагаются в собственном модуле Control.Applicative, подсказывает, что эти комбинаторы могут пригодиться и в других сценариях. Было бы интересно выяснить, как подтягивание работает с другими типами данных. Предположим, к примеру, что эти комбинаторы также будут работать со значениями Maybe. Тогда могли бы существовать следующие типы:

```
(<$>) :: (a -> b)       -> Maybe a -> Maybe b
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

В сочетании эти функции предоставляют способ подтягивания функции для работы с возможно пустыми значениями, а также для возвращения значения Nothing, когда опущен какой-нибудь аргумент, и значения Just, если доступны все аргументы. Давайте подтвердим наши интуитивные предположения, запустив в интерпретаторе несколько примеров:

```
*Chapter10.TypeClasses> import Control.Applicative
*Chapter10.TypeClasses Control.Applicative> (+) <$> Just 2 <*> Just 3
Just 5
*Chapter10.TypeClasses Control.Applicative> (+) <$> Just 2 <*> Nothing
Nothing
```

Если и Parser и Maybe могут использоваться с одними и теми же функциями, это свидетельствует о том, что перед нами новый класс типов, а именно Applicative. Этот класс типов, как и ряд других связанных с ним классов типов, рассматривается в данном разделе. Как уже отмечалось, функция (<$>) очень похожа на функцию fmap, которая позволяет изучить отношения с классом типов Functor, и мы уже знаем, что Maybe также является экземпляром класса типов Applicative, следовательно, здесь также есть какая-то связь с классом Monad.

Класс Applicative

Позвольте представить вам базовый класс типов Applicative. Хотя функции (<*) и (*>) также являются частью этого класса типов, у них имеются предлагаемые по

умолчанию определения, которые мы рассмотрим позже. Основные же функции данного класса типов определены следующим образом:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

По поводу функции `(<*>)` у вас уже есть некоторые интуитивные предположения: она позволяет извлечь какую-нибудь функцию, находящуюся внутри контекста типа `f`, и применить к ней один аргумент. В этом и заключается главная сила класса типов `Applicative` — он способен проводить вычисления, когда все находится внутри одного контекста. В следующих примерах в оболочку `Maybe` заключены как функции, так и аргументы, а функция `(<*>)` позволяет выполнить приложение:

```
*Chapter10.TypeClasses Control.Applicative> Just (+) <*> Just 2 <*> Just 3
Just 5
*Chapter10.TypeClasses Control.Applicative> Just (+) <*> Just 2 <*> Nothing
Nothing
*Chapter10.TypeClasses Control.Applicative> Nothing <*> Just 2 <*> Just 3
Nothing
```

Как показано в третьем примере, и применяемая функция и аргументы могут иметь тип `Just` или `Nothing`. Интуитивно понятно, что, как было показано ранее, при отсутствии применяемой функции вернуть какой-либо результат невозможно.

Второй функцией в типовом классе является `pure`. Глядя на ее тип `a -> f a`, нетрудно догадаться, чем она занимается: она заключает чистое значение в контекст `f`. Например, при использовании `Maybe` чистота является эквивалентом `Just`. Таким образом, в предыдущем коде два первых примера можно переписать следующим образом:

```
*Chapter10.TypeClasses Control.Applicative> (pure (+) <*> pure 2 <*> pure 3) ::
Maybe Int
Just 5
*Chapter10.TypeClasses Control.Applicative> pure (+) <*> Just 2 <*> Nothing
Nothing
```

Сочетание заключения чистой функции в оболочку и ее последующего применения к нескольким аргументам при использовании класса `Applicative` встречается довольно часто. Поэтому функция `(<$>)` была создана с этим простым определением:

```
(<$>) :: (a -> b) -> f a -> f b
f <$> x = pure f <*> x
```

ЗАКОНЫ КЛАССА APPLICATIVE

Как и Monoid, Functor, Monad, а также многие другие классы типов, класс Applicative не только предоставляет ряд функций, но и требует соблюдения ряда законов. В данном случае должны соблюдаться следующие законы:

- pure id <*> x = x, или id <$> x = x
- (.) <$> x <*> y <*> z = x <*> (y <*> z)
- f <*> pure x = pure ($ x) <*> f
- f <$> pure x = pure f <*> pure x = pure (f x)

> Первые три закона заставляют значения класса Applicative хорошо работать с обычными функциями: функция тождественности внутри f так ею и остается, а сочетание и применение функций точно так же отражается и внутри таких данных, относящихся к классу Applicative. Последний закон предписывает чистым вычислениям не изменяться: ни когда они полностью выполняются снаружи, а затем подтягиваются, ни когда они подтягиваются частями.

Ранее мы видели, что `Parser` и `Maybe` являются экземплярами класса `Applicative`. Частью этой группы является и наш старый знакомый тип `[]`. Как отмечалось в главе 7, списки могут использоваться для моделирования недетерминизма, именно это и делает их экземпляр, относящийся к `Applicative`. Например, если имеется множество функций, готовых к применению, и множество данных, которые могут использоваться в качестве аргументов, функция `(<*>)` возвращает список всех попарных вариантов применения:

```
*> import Data.Char
*> [("Hello, " ++), \x -> "HEY, " ++ map toUpper x] <*> ["Alex", "John", "Paul"]
["Hello, Alex","Hello, John","Hello, Paul","HEY, ALEX","HEY, JOHN","HEY, PAUL"]
```

Функция `(<$>)` хорошо подойдет для описания вычислений, в которых фиксированная функция может получать несколько разных аргументов. Давайте, к примеру, создадим несколько поздравлений для нескольких людей:

```
*> (++) <$> ["Hello, ", "Good bye, "] <*> ["Alex", "John", "Paul"]
["Hello, Alex","Hello, John","Hello, Paul","Good bye, Alex","Good bye, John","Good
bye, Paul"]
```

Классы Functor, Applicative и Monad

В предыдущем разделе при использовании функции `(<$>)` мы отметили ряд весьма интересных обстоятельств. Сигнатура `(<$>)` идентична сигнатуре `fmap` класса типов `Functor`:

```
(<$>) :: Applicative f => (a -> b) -> f a -> f b
fmap  :: Functor     f => (a -> b) -> f a -> f b
```

Кроме того, функция `(<*>)` очень похожа на функцию `(=<<)` (которая является просто функцией `(>>=)`, но с обратным порядком следования аргументов) класса типов `Monad`:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(=<<) :: Monad       f => (a -> f b) -> f a -> f b
```

И конечно же, как мы видели, типы данных `Maybe` и `[]`, являющиеся монадами, являются также экземплярами класса типов `Applicative`. В данном разделе мы выясним взаимоотношения этих весьма важных классов типов.

Прежде всего, класс типов `Applicative` заставляет каждый тип `Applicative` быть также типом `Functor`. Кроме того, законы класса типов `Applicative` применяются таким образом, что определение `fmap` должно быть полностью идентично `(<$>)`. Однако как отмечалось применительно к парсерам, тип `Applicative` строже, чем `Functor`. Причина в том, что с `Functor` внутрь соответствующего контекста можно

перемещать функции лишь с одним аргументом. Если же требуется применить функцию `f` с двумя аргументами, то однократное использование функции `fmap` с целью избавиться от одного аргумента позволяет получить следующий тип:

```
fmap     :: (a -> b -> c) -> f a -> f (b -> c)
fmap f   :: f a -> f (b -> c)
fmap f x :: f (b -> c)
```

С одним значением `Functor` сделать что-либо большее невозможно, поскольку получить функцию внутри контекста `f` нельзя. Однако если бы функция `f` имела тип `Applicative`, то для дальнейшего предоставления аргументов можно было бы использовать функцию `(<*>)`. Переход от `Functor` к `Applicative` дает весьма ценную возможность применения функций с любым количеством аргументов.

Классы типов `Monad` и `Applicative` связаны самым непосредственным образом. Если вкратце, то каждое значение типа `Monad` является также значением типа `Applicative`, а это значит, что каждая монада, о которой мы узнали к этому моменту, может также использоваться с интерфейсом `Applicative`. Это можно проверить, если вспомнить функцию `ap`, представленную в главе 7, в качестве средства обобщения функции `liftM` для работы с любым количеством аргументов:

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

Эта сигнатура полностью идентична сигнатуре функции `(<*>)` в `Applicative` и работает точно так же. В главе 7 также обсуждалось, как вместо сигнатуры `liftMn f` $x_1 \ldots x_n$ можно было бы использовать более расширяемый вариант:

`return f `ap`` $x_1$ `` `ap` ... `ap`` x_n

Здесь все следует тому же образцу, который использовался в определении функции `(<$>)`, в понятиях функций `pure` и `(<*>)`. В частности, здесь показано, что если тип является экземпляром класса типов `Monad`, можно определить чистую функцию в его экземпляре `Applicative`, сделав ее идентичной функции `return`.

ПРИМЕЧАНИЕ

Тот факт, что Applicative не выступает в роли суперкласса для Monad, является исторически сложившимся обстоятельством: концепция Monad уже широко использовалась, когда родилась концепция Applicative. Чтобы сохранить совместимость, в определении Monad не производилось никаких изменений с целью включения в состав Functor или Applicative как суперклассов. Однако вскоре положение дел изменится, и в классах типов станет прослеживаться полная иерархия: сначала Functor, затем Applicative и, наконец, Monad.

Конечно же, поскольку классы `Applicative` и `Monad` не идентичны и `Applicative` находится в иерархии классов на более высоком месте, должно быть что-нибудь, что можно сделать с данными типа `Monad`, но чего нельзя сделать с данными типа `Applicative`. Ответ можно найти путем изучения разных сигнатур `(<*>)` и `(=<<)`, упоминавшихся в начале раздела. В них можно найти различие: функция, предоставляемая в качестве первого аргумента функции `(<*>)`, должна иметь тип `f (a -> b)`, в то время как в `(=<<)` этот тип описывается как `a -> f b`.

Смысл здесь в том, что использование монад в качестве аргумента функции может изменить функцию, выполняемую на следующем шаге (поскольку возвращается

f b), в то время как в данных типа Applicative аргумент функции может повлиять на текущий выполняемый шаг, но не на оставшиеся шаги. Чтобы прояснить ситуацию, рассмотрим функцию, которая в зависимости от того, является первый аргумент единицей или нет, завершает работу либо прибавлением единицы, либо удваиванием числа:

```
\x -> if x == 1 then Just (+ 1) else Just (* 2)
```

Если применить один аргумент, используя функцию (<*>), получится следующий тип:

```
(\x -> if x == 1 then pure (+ 1) else  pure (* 2)) <$> Just 1
  :: (Num a) => Maybe (Maybe (a -> a))
```

И здесь класс типов Applicative делу не поможет ввиду отсутствия способа извлечения из оболочки этих двух уровней Maybe с помощью либо (<$>), либо (<*>). Однако в главе 7 при описании монад была представлена функция объединения монад с требуемой сигнатурой Monad m => m (m a) -> m a. Таким образом, если вы используете монаду, можно «сплавить» два уровня в один и продолжить выполнение.

Это не обязательно означает, что интерфейс монад всегда лучше, чем интерфейс Applicative. Тот факт, что при использовании монад на порядок операций можно повлиять посредством значений, вычисляемых в ходе выполнения, существенно усложняет анализ внутренней ситуации. Однако в библиотеке синтаксического разбора вам ничто не мешает тщательно проинспектировать структуру вычислений с целью применения разнообразных вариантов оптимизации, существенно повышающих эффективность синтаксического разбора. Поэтому для библиотеки attoparsec был выбран интерфейс класса типов Applicative.

НАПИСАНИЕ БОЛЕЕ ПОНЯТНОГО МОНАДИЧЕСКОГО КОДА

В этой главе и в главе 7 было показано множество разных способов выражения одних и тех же концепций в классах Functor, Applicative и Monad:

```
fmap f x = f <$> x = do y <- x; return (f x) = liftM f x
f <$> x <*> y = return f `ap` x `ap` y
pure x = return x
```

Эти равенства способны помочь в написании более удобочитаемого кода, особенно в отношении монад. Рассмотрим, к примеру, следующий код:

```
do name <- getLine
   let upperName = map toUpper name
   putStrLn upperName
```

В предыдущих главах мы ратовали за удаление промежуточных шагов let с функцией fmap, но с помощью функции (<$>) этот код может быть записан так, чтобы еще более походить на обычное применение функции:

```
do upperName <- map toUpper <$> getLine
   putStrLn upperName
```

С помощью функций класса Monad этот код может быть записан еще короче:

```
putStrLn =<< (map toUpper <$> getLine)
```

Всегда при записи кода важно найти оптимальный баланс между удобством чтения и краткостью.

Класс Alternative

Функции (<$>) и (<*>) уже рассматривались, но при использовании данных типа Parser применялась еще одна важная функция — (<|>). Она является частью класса типов Alternative, который выглядит так:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  some  :: f a -> f [a]
  some v = (:) <$> v <*> many v
  many  :: f a -> f [a]
  many v = some v <|> pure []
```

Как видите, класс типов Alternative очень похож на MonadPlus. Он дает возможность обозначить неудачу посредством функции empty и сделать выбор посредством функции (<|>), что во многом напоминает функции mempty и mplus. Например, его можно использовать для возвращения одной из нескольких возможностей Maybe, а именно первой, которая задается с помощью Just:

```
*Chapter10.TypeClasses> import Control.Applicative
*Chapter10.TypeClasses Control.Applicative> Nothing <|> Just 2 <|> Just 3
Just 2
```

ПРИМЕЧАНИЕ

Разумеется, все, что относится к MonadPlus, относится также и к Monad, и к Alternative.

Функции some и many предназначены для использования в средах, похожих на среду парсера. Если имеется вычисление v, хранящее или возвращающее элементы типа t, то можно использовать функции many и some для его повторного выполнения и получения вместо него списка элементов t. Отличие состоит в том, что при вызове функции many можно получить пустой список, в то время как some заставляет вернуть хотя бы один элемент. Эти функции не применяются вне парсеров, поскольку во многих случаях рекурсивное определение не позволяет им завершиться.

Класс Traversable

Теперь, когда нам было представлено множество новых классов типов, настал благоприятный момент для знакомства с последним из «структурных» классов типов, примерами которого могут быть контейнер и контекст. Имя этого класса типов Traversable, а его определение выглядит следующим образом:

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f
    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
```

```
    mapM f = unwrapMonad . traverse (WrapMonad . f)
    sequence :: Monad m => t (m a) -> m (t a)
    sequence = mapM id
```

Вид у него довольно устрашающий, поэтому давайте сначала сделаем ряд упрощений. Если присмотреться, в типовом классе `Traversable` определяются две группы функций, одна (содержащая функции `traverse` и `sequenceA`) для класса `Applicative`, вторая для класса `Monad` (с функциями `mapM` и `sequence`), с одной и той же сигнатурой, исключая ограничения класса типов. Таким образом, нам нужно лишь выстроить свои интуитивные предположения относительно одной из двух групп.

Одним из примеров типа `Traversable` является `[]`. Типы для `sequence` и `sequenceA` в данном случае могли бы быть следующими:

```
sequenceA :: Applicative f => [f a] -> f [a]
sequence  :: Monad       m => [m a] -> m [a]
```

Функции подсказывают, как перейти от списка вычислений типа `f a` к отдельно взятому вычислению со списком в его основе. Предположим, что нам нужно сделать `f` или `m` равным значению `Maybe`, относящимся сразу и к `Applicative`, и к `Monad`. В таком случае конкретный тип будет иметь следующий вид:

```
sequenceA, sequence :: [Maybe a] -> Maybe [a]
```

Если считать, что `Maybe` привносит некий эффект, тогда то, что делают функции `sequence` и `sequenceA`, заключается в выносе всех подобных эффектов «за пределы списка». Поскольку эффект, вызываемый `Maybe`, заключается в возможности неудачного завершения работы, мы вправе ожидать, что выполнение любой из таких функций окажется успешным, только когда каждый элемент списка будет относиться к `Just`, в противном случае неудача должна выноситься «за пределы списка», а это может быть сделано только путем возвращения значения `Nothing`. С помощью интерпретатора можно проверить, что так оно и происходит:

```
*Chapter10.TypeClasses> import Data.Traversable
*Chapter10.TypeClasses Data.Traversable> sequenceA [Just 2, Just 3, Just 4]
Just [2,3,4]
*Chapter10.TypeClasses Data.Traversable> sequenceA [Just 2, Nothing, Just 4]
Nothing
```

Более конкретная версия `sequence` в `Control.Monad` (только для списков) уже использовалась в главе 7 точно для такой же цели: выполнения всех действий в списке монадических значений и возвращения результатов этих действий.

Функции `sequence` и `sequenceA` так же часто упоминаются, как функции, «меняющие местами два функтора». Если посмотреть на их тип, можно увидеть, что вначале идет `t (f a)`, а в конце — `f (t a)`. Порядок следования `t` и `f` поменялся на обратный, то есть произошло именно то, что означает «поменять местами». Кроме того, поскольку элемент `t` относится к типовому классу `Traversable`, он должен также относиться к типовому классу `Functor`, а поскольку элемент `f` относится к типовому классу `Applicative`, он должен также относиться к типовому классу `Functor`. Это означает, что имеет место некий вид структур (подобный спискам и данным типа `Maybe`), который способен вполне естественным образом попадать «внутрь» любого возможного эффекта.

Давайте рассмотрим другой набор функций: `traverse` и `mapM`. В данном случае важно вспомнить о порядке использования функции `mapM`, находящейся в модуле `Control.Monad`. Обычно способ восприятия таких функции заключается в том, чтобы рассматривать их как «функцию `fmap` с дополнительными эффектами». Первым аргументом является функция, возвращающая некоторое значение и попутно реализующая ряд дополнительных вычислительных эффектов. Эта функция применяется к каждому элементу, содержащемуся в значении типа `Traversable`, и все эффекты объединяются и выносятся за пределы возвращаемой структуры.

Давайте еще раз посмотрим на особый случай `Traversable t` в виде значения `[]` и `Applicative f` в виде значения `Maybe`. Конкретная сигнатура для `traverse` и `mapM` приобретает следующий вид:

```
traverse, mapM :: (a -> Maybe b) -> [a] -> Maybe [b]
```

В данном случае первый аргумент применяется к каждому элементу списка. К сожалению, поскольку значение `Maybe` нужно вернуть в конце, единственное, что можно сделать, — это вернуть `Just` только в том случае, если каждое применение этого первого аргумента возвращает `Just`.

Одним из важных свойств всех этих функций является то, что они при работе с данными этого типа следуют структуре `Traversable`. Например, если `traverse` применяется к списку из пяти элементов, получающийся в результате список также будет иметь пять элементов плюс к этому какой-нибудь дополнительный эффект. Это весьма важное обстоятельство, поскольку оно означает, что функции могут изменять только содержащиеся в списке значения, но не образец, которому они следуют. Это приближает такие функции к `fmap` и расширяет возможности их применения. Например, в главе 6 шла речь о линзах `traversed`, позволяющих попадать вовнутрь контейнеров и извлекать или изменять значения каждого из содержащихся в них элементов. Теперь можно понять, почему этот класс типов обладает таким именем (`Traversable` означает «преодолимый») и почему это именно тот класс типов, который нам нужен.

Согласно имеющейся в Hackage документации, касающейся `Traversable`, обход структуры совершается слева направо. Эта небольшая деталь не играла особой роли ни для класса типов `Functor` (поскольку там не возникало никаких дополнительных эффектов), ни для класса типов `Foldable` (поскольку используемый в основе моноид обладает свойствами ассоциативности). Что же касается класса `Traversable`, то выполнение действий в том или ином порядке может приводить к разным результатам. Подумайте о случае, когда `Applicative f` или `Monad m` является монадой `IO` или `Writer`.

Реализация `Traversable` для конкретного контейнера кажется непростой задачей, но на практике она решается довольно легко. Возьмем в качестве примера класс типов `BinaryTree2`, который был представлен в главе 4:

```
data BinaryTree2 a = Node2 a (BinaryTree2 a) (BinaryTree2 a)
                   | Leaf2
                   deriving Show
```

В упражнениях 4.8 и 4.9 нам предлагалось реализовать его экземпляры `Functor` и `Foldable`. Экземпляры `Functor` обычно следовали образцу применения соответ-

ствующих функций к каждому значению, имеющемуся в структуре, с последующим рекурсивным вызовом функции `fmap` для каждой подчиненной структуры. Экземпляры `Foldable` выполняют очень похожую работу, но с применением между ними моноидных операций. То есть экземпляры выглядели примерно так:

```
instance Functor BinaryTree2 where
  fmap f (Node2 x l r) = Node2 (f x) (fmap f l) (fmap f r)
  fmap _ Leaf2         = Leaf2

instance Foldable BinaryTree2 where
  foldMap f (Node2 x l r) = (f x) <> (foldMap f l) <> (foldMap f r)
  foldMap _ Leaf2         = mempty
```

Наилучшим способом создания экземпляра `Traversable` является применение функции `traverse`, которая самостоятельно создает минимальное полноценное определение. Функция `traverse` была ранее представлена как «функция `fmap` с эффектами». Единственное, что нужно будет сделать, — это последовать той же самой структуре, что и в случае `fmap`, но гарантировав при этом, что в ходе создания используются данные типа `Applicative`.

```
instance Traversable BinaryTree2 where
  traverse f (Node2 x l r) = Node2 <$> f x <*> traverse f l <*> traverse f r
  traverse _ Leaf2         = pure Leaf2
```

Поскольку эти экземпляры всегда следуют одной и той же структуре, у GHC есть возможность их автоматического вывода. Это делается путем включения в исходный файл пакета расширений:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}
```

Кроме того, нужно изменить часть `deriving` в определении `BinaryTree2`:

```
deriving (Show, Functor, Foldable, Traversable)
```

Не нужно излишеств: используйте формат JSON

Основное внимание в этой главе уделяется созданию и синтаксическому разбору текста. Обычно это утомительное и не гарантирующее отсутствия ошибок занятие. При каждом добавлении новых данных приходится изменять обе части, сохраняя их синхронность, чтобы каждый вывод, сгенерированный с помощью построителя, можно было обработать с помощью парсера.

Вместо этого можно было бы отдать предпочтение использованию библиотеки, помогающей обмениваться информацией в виде текста. Если это возможно, также можно выбрать формат, общепринятый во многих языках программирования и на многих платформах, поскольку в будущем может понадобиться взаимодействие между инструментальными средствами. Все эти возможности предлагает формат JSON: он имеет практически универсальную поддержку среди языков и платформ, а также простое текстовое представление, упрощающее его обработку как людьми, так и компьютерами. Кроме того, JSON является стандартом де-факто

для обмена данными в Интернете, поэтому его использование открывает двери к созданию веб-приложения на языке Haskell.

Тип данных `Client` может быть представлен в формате JSON следующим образом:

```
{ "type": "company", "id": 1, "name": "Black Hole Inc."
, "person": { "first": "John", "last": "Smith" }, "duty": "Traveller" }
```

В этом небольшом фрагменте показаны практически все доступные способы создания JSON-значения. Прежде всего, у вас есть базовые типы, такие как числа, булевы значения и строки, которые представляются их литералами. Из этих базовых значений можно создавать либо массивы (которые здесь не показаны, но записываются с помощью того же синтаксиса, что и Haskell-списки), либо объекты, являющиеся проекциями ключ–значение. Вдобавок у нас также есть специальное значение `null`. В данном случае полное значение является объектом с четырьмя ключами и значениями.

К тому же Haskell-сообщество предлагает простую, но эффективную библиотеку для чтения и записи значений в формате JSON. Эта библиотека называется `aeson`, а все варианты ее использования вращаются вокруг вполне компактного типа данных:

```
data Value = Object Object
           | Array  Array
           | String Text
           | Number Number
           | Bool   Bool
           | Null
```

Как уже отмечалось, каждый из конструкторов соответствует типу того значения, которое можно найти в формате JSON.

Давайте начнем с написания функций для преобразования JSON-представлений и значений типа `Client`. Большинство вариантов преобразования в JSON-значение сводится к заключению данных в оболочку соответствующего конструктора. В случае создания значений `Object` это проще всего сделать с помощью функции `object`, которой в качестве аргумента передается список пар. Каждая из этих пар определяется как *ключ* `.=` *значение*. В результате преобразование в формат JSON приобретает следующий вид:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Aeson
import Data.Text

clientToJSON :: Client Integer -> Value
clientToJSON (GovOrg i n) =
object [ "type"   .= String "govorg"
       , "id"     .= Number (fromInteger i)
       , "name"   .= String (pack n) ]
clientToJSON (Company i n p d) =
object [ "type"   .= String "company"
       , "id"     .= Number (fromInteger i)
       , "name"   .= String (pack n)
       , "person" .= personToJSON p
       , "duty"   .= String (pack d) ]
```

```
clientToJSON (Individual i p) =
object [ "type"   .= String "individual"
       , "id"     .= Number (fromInteger i)
       , "person" .= personToJSON p ]

personToJSON :: Person -> Value
personToJSON (Person f l) = object [ "first" .= String (pack f)
                                   , "last"  .= String (pack l) ]
```

Теперь если нужно заново создать значение типа `Person` из JSON-значения, следует проверить наличие каждого ключа. Это можно сделать с помощью функций `HashMap` (похожих на соответствующие функции в `Map`), использующих тип `Value` из библиотеки `aeson`. Поскольку функции `HashMap` определяются в пакете `unordered-containers`, то для выполнения примера этот пакет нужно включить в ваш проект.

```
import qualified Data.HashMap.Strict as M

jsonToPerson :: Value -> Maybe Person
jsonToPerson (Object o) = do String f <- M.lookup "first" o
                              String l <- M.lookup "last" o
                              return $ Person (unpack f) (unpack l)
jsonToPerson _          = Nothing
```

Для доступа к полям в JSON-значении существует несколько подходов. Как мы уже видели в главе 6, для подобной навигации внутри объекта хорошо подходят линзы. Пакет `lens-aeson` предоставляет множество функций, призванных упростить использование значений типа `Value` из библиотеки `aeson`. В данном конкретном случае можно задействовать функцию `key`, предоставляющую доступ в объекте к названному полю. Поскольку ключ может быть недоступен, используется не функция (`^.`), а ее функция-партнер (`^?`), которая может вернуть значение `Nothing`. Предыдущий код может быть приведен к следующему виду:

```
import Control.Lens ((^?))
import Control.Lens.Aeson

jsonToPerson :: Value -> Maybe Person
jsonToPerson j = do String f <- j ^? key "first"
                    String l <- j ^? key "last"
                    return $ Person (unpack f) (unpack l)
```

Другим весьма интересным компонентом модуля `Control.Lens.Aeson` является функция `nth`, которая по заданному индексу получает элемент из соответствующей позиции, как будто JSON-значение трактуется как массив. Для остальных элементарных JSON-типов `lens-aeson` предоставляет следующие комбинаторы: `_Number` (с двумя его вариантами `_Double` и `_Integer` для автоматического преобразования в один конкретный числовой тип), `_String` и `_Bool`.

В любом случае непосредственное сопоставление со значением типа `Value` не является в библиотеке `aeson` рекомендуемым способом декодирования информации формата JSON. Вместо этого нужно воспользоваться *JSON-парсером*. Интерфейс этого парсера похож на тот, что использовался в `attoparsec`, в частности, в нем также задействованы функции из класса типов `Applicative`. Основное отличие заключается

в наличии множества базовых блоков для парсеров: в библиотеке `aeson` применяется функция `(.:)`, извлекающая значение по заданному ключу. Обычный способ преобразования из формата JSON в тип `Person` мог бы выглядеть так:

```
import Data.Aeson.Types
import Control.Applicative

jsonToPerson :: Value -> Parser Person
jsonToPerson (Object o) = Person <$> o .: "first" <*> o .: "last"
jsonToPerson _          = Control.Applicative.empty
```

Функции для прямого и обратного преобразования формата JSON обычно определяются не сами по себе, а как части классов типов `ToJSON` и `FromJSON`, предоставляемых библиотекой `aeson`. Давайте добавим их к коду для типа `Person`:

```
instance ToJSON Person where
  toJSON = personToJSON

instance FromJSON Person where
  parseJSON = jsonToPerson
```

Серьезным достоинством классов `ToJSON` и `FromJSON` является то, что в библиотеке `aeson` имеется ряд предопределенных экземпляров для многих типов. Например, при написании их для `Person` можно также провести сериализацию в JSON-списки людей, объектов, содержащих данные типа `Person`, и т. д. В частности, давайте воспользуемся только что определенным экземпляром чтобы задать парсера для `Client` и соответствующих экземпляров `ToJSON` и `FromJSON`. Обратите внимание на то, что `FromJSON` позволяет проявлять более высокую степень обобщения в отношении типов, принимаемых в качестве идентификаторов:

```
{-# LANGUAGE FlexibleInstances #-}
jsonToClient :: FromJSON i => Value -> Parser (Client i)
jsonToClient (Object o) =
  case M.lookup "type" o of
   Just (String "govorg")   -> GovOrg <$> o .:  "id"       <*> o .: "name"
   Just (String "company") -> Company <$> o .: "id"        <*> o .: "name"
                                      <*> o .: "person" <*> o .: "duty"
   Just (String "individual")  -> Individual <$> o .: "id" <*> o .: "person"
                          _ -> Control.Applicative.empty
jsonToClient _ = Control.Applicative.empty

instance ToJSON (Client Integer) where
  toJSON = clientToJSON

instance FromJSON i => FromJSON (Client i) where
  parseJSON = jsonToClient
```

Используя интерпретатор, можно посмотреть, что преобразование в формат JSON выполняется правильно:

```
*> toJSON $ Company (1 :: Integer) "Black Hole Inc." (Person "John" "Smith")
"Traveller"
Object fromList [("name",String "Black Hole Inc."),("duty",String "Traveller")
                ,("person",Object fromList [("last",String "Smith")
                                           ,("first",String "John")])
                ,("id",Number 1),("type",String "company")]
```

Для проведения операции преобразования нужно вызвать функцию `fromJSON`, которая запустит парсер `Parser` с заданным значением `Value` и произведет результат `Result`, в качестве которого может выступить либо `Success`, либо `Error`. Обратите внимание на то, что в коде явно указана сигнатура, чтобы сообщить интерпретатору, какой экземпляр `FromJSON` ему нужно использовать.

```
*> :{
*| fromJSON $ toJSON $
*| Company (1 :: Integer) "Black Hole Inc." (Person "John" "Smith") "Traveller"
*| :: Result (Client Integer)
*| :}
Success (Company { clientId = 1, clientName = "Black Hole Inc."
                 , person = Person {firstName = "John", lastName = "Smith"}
                 , duty = "Traveller" })
```

Теперь, когда мы знаем, как выполнять прямые и обратные преобразования значений типа `Value` из библиотеки `aeson`, следующим шагом должно стать преобразование этих значений типа `Value` в настоящие текстовые представления. Для генерирования текста нужно лишь воспользоваться функцией `encode`, возвращающей значения типа `ByteString`. Для преобразования текста в формат JSON имеется сразу несколько возможностей:

- Функция `decode` получает значения типа `ByteString`, выполняет синтаксический разбор текста для выдачи значения типа `Value` из библиотеки `aeson`, а затем производит преобразование этого значения в какой-нибудь другой тип с помощью функции `FromJSON`. Результат заключается в оболочку `Maybe`, чтобы можно было сигнализировать о неудаче.
- Функция `eitherDecode` решает ту же задачу, но в случае неудачи в конструкторе `Left`, являющемся составной частью `Either`, возвращает сообщение об ошибке.
- Два шага можно разделить. Сначала библиотека `aeson` предоставляет функцию `json`, а парсер `Parser` из библиотеки `attoparsec` выполняет синтаксический разбор из типа `ByteString` в тип `Value`. Далее для генерирования конечного значения можно вызвать функцию `fromJSON`.

Любопытной особенностью является то, что в библиотеке `aeson` кодирование и декодирование реализовано для ленивых значений типа `ByteString`, поэтому может понадобиться преобразование в строгий вариант. Например, с библиотекой `conduit` это можно сделать так:

```
import Data.Conduit
import qualified Data.Conduit.Binary as B
import qualified Data.Conduit.List as L
import qualified Data.ByteString.Lazy as LB

saveClients :: FilePath -> [Client Integer] -> IO ()
saveClients fPath clients = runResourceT $
  yield (toJSON clients) $$ L.map (LB.toStrict . encode) =$ B.sinkFile fPath
```

В упражнении 10.3 от вас требуется воспользоваться форматом JSON для сохранения информации о товарах и продажах в файле и загрузки ее из файла:

УПРАЖНЕНИЕ 10.3. ФОРМАТ JSON ДЛЯ ТОВАРОВ И ПРОДАЖ

Напишите экземпляры ToJSON и FromJSON типов данных Product и Purchase, представленных в упражнении 10.1. Затем, используя библиотеки aeson и conduit, создайте приложение, которое считывает список данных типа Product, представленных в формате JSON, и показывает среднюю цену.

Экземпляры `ToJSON` и `FromJSON` всегда следуют одному и тому же образцу: `ToJSON` записывает каждое из полей конструктора парами ключ–значение в формате JSON, а дополнительное поле служит для уведомления конструктора в случае, если имеется более одного типа данных (подобно `"type"` в примере `Client`). `FromJSON` просто пытается прочитать каждый из этих ключей. Это именно тот случай, когда Haskell позволяет избегать применения шаблонного кода. Если же вам не требуется какое-нибудь особенное поведение, библиотека `aeson` может написать экземпляры за вас.

Для этого нужно включить в GHC расширение `DeriveGeneric` и добавить `Generic` к списку экземпляров в инструкции `deriving`. Экземпляр `Generic` содержит информацию о конструкторах и полях в типе данных. Затем останется только написать пустые экземпляры `ToJSON` и `FromJSON`, после чего `aeson` сделает все остальное, используя эту информацию из экземпляра `Generic`. Вот как это выглядит в случае с `Client`:

```
{-# LANGUAGE DeriveGeneric #-}
import GHC.Generics

data Client i = ... deriving (Show, Generic)
data Person = ... deriving (Show, Read, Generic)

instance ToJSON i   => ToJSON   (Client i)
instance FromJSON i => FromJSON (Client i)
instance ToJSON Person
instance FromJSON Person
```

Эти автоматически извлеченные экземпляры будут использовать в качестве ключей имена полей, объявленные в конструкторах. Поэтому они работают только для объявлений данных с синтаксисом записей. В том случае, если у типа имеется более одного конструктора, ключ, предназначенный для того, чтобы их различать, называется тегом (`tag`).

СИНТАКСИЧЕСКИЙ РАЗБОР АРГУМЕНТОВ КОМАНДНОЙ СТРОКИ

Когда вам попадается некая операция, в том или ином виде требующая синтаксического разбора, практически всегда к ней найдется интерфейс Applicative. Если не считать декодирования формата JSON с помощью библиотеки aeson, есть еще один весьма полезный пакет, который называется optparse-applicative и помогает описать и организовать множество аргументов командной строки приложения.

Вот как, к примеру, могут быть определены аргументы приложения, от которых ожидается одно имя файла, содержащего список клиентов, и сведения о том, хранится эта информация в формате JSON или нет:

```
import Options.Applicative

data Args = Args String Bool -- тип данных, сордержащий аргументы

args :: Parser Args -- чтение аргументов
```

```
args = Args <$> strOption (long "file" <> help "Database of clients to load")
<*> switch (long "json" <> help "Whether the database uses JSON")

argsInfo :: ParserInfo Args -- определение аргументов
                            -- и вспомогательный текст
argsInfo = info args fullDesc

main :: IO ()
main = do Args fPath json <- execParser argsInfo
...
```

Большим преимуществом класса типов Applicative является то, он предлагает общий интерфейс для решения тех задач, которые на первый взгляд никак не связаны друг с другом. Этот класс типов стал применяться весьма широко и теперь рассматривается в Haskell как один из базовых классов типов наряду с классами Functor и Monad.

Выводы

В данной главе мы изучили предлагаемые Haskell средства для работы с текстом в различных ситуациях:

- Сначала мы рассмотрели различия между типами данных `String`, `ByteString` и `Text`, а также раскрыли назначение каждого из них.
- Пакеты `text` и `bytestring` предоставляют тип `Builder`, позволяющий эффективно генерировать текстовые данные.
- Мы познакомились с предлагаемой пакетом `attoparsec` обратной операцией: синтаксическим разбором. Эта библиотека использует комбинаторные подходы для генерирования модульных и удобочитаемых парсеров.
- Мы также познакомились с классом типов `Applicative`, который по своей эффективности занимает место между классами `Functor` и `Monad`. Кроме того, мы изучили классы типов `Alternative` и `Traversable`.
- И наконец, мы обсудили прямое и обратное преобразование формата JSON с использованием библиотеки `aeson`.

Безопасный доступ к базам данных

11

В предыдущих двух главах мы узнали, как осуществляется доступ к файловой системе для чтения и сохранения информации на долгосрочной основе. Мы познакомились со всеми инструментами для работы с нестандартным форматом файлов или с древовидным представлением информации, например в формате JSON.

Однако в большинстве корпоративных приложений с файлами не работают. В них для хранения информации приложения и содержания ее в безопасности используются *системы управления базами данных* (database management system), или СУБД. Больше того, многие СУБД предоставляют такие функциональные возможности, как транзакции и протоколирование, восстановление и репликация, которые могут оказаться весьма полезными в плане поддержки, защиты от ошибок и масштабирования. В данной главе мы, как говорится, с высоты птичьего полета окинем взглядом многочисленные средства доступа к базам данных в экосистеме Haskell, и в частности те, которые позволяют получать доступ к реляционным СУБД с помощью языка SQL.

Также в этой главе описываются две специальные библиотеки: *Persistent* и *Esqueleto*. Эти библиотеки не привязаны к какой-то конкретной СУБД, а предоставляют некий уровень абстракции, надстраиваемый над любой выбранной вами СУБД. Основное преимущество библиотеки Persistent состоит в том, что она позволяет осуществлять взаимодействие с базой данных посредством определенных пользователем абстрактных Haskell-типов, которые преобразуются в формат базы данных и обратно без какого-либо дополнительного вмешательства (подобно системам объектно-реляционного проецирования в объектно-ориентированном мире). Тем самым обеспечивается высокая степень безопасности типов при работе с кодом баз данных. Библиотека Esqueleto, которая применяется вместе с библиотекой Persistent, призвана учесть особенности SQL-ориентированных СУБД.

Средства доступа к базам данных

Зная, что в Haskell имеется строгая экосистема доступа к библиотекам баз данных, вполне логично заглянуть в Hackage, чтобы увидеть множество таких библиотек под заголовком «Database» (на момент написания книги их там было более 150). В общих чертах, эти библиотеки могут быть разбиты на три категории:

- Естественные реализации баз данных в Haskell. Примером библиотеки данной категории может быть библиотека `acid-state`, позволяющая сохранять Haskell-типы непосредственно на диске. Основным недостатком такого подхода являются проблемы взаимодействия с другими системами и языками.
- Библиотеки для доступа к конкретным СУБД, такие как `sqlite-simple`, `mysql-simple`, `pgsql-simple` (для некоторых наиболее известных реляционных систем) или `MongoDB`. В этих библиотеках пользователям соответствующих СУБД нетрудно разобраться, но они привязаны к конкретным системам, что затрудняет перенос.
- Библиотеки, реализующие уровни абстракции поверх некоторых СУБД и предоставляющие общий интерфейс для нескольких систем (зачастую наряду со специальными соединителями для каждой из них). Диапазон охватываемых СУБД для каждой библиотеки разный: некоторые из них фокусируются только на работе с SQL-ориентированными реляционными СУБД, в то время как другие предназначены для работы с СУБД, не относящимися к реляционным. Главное преимущество таких библиотек состоит в том, что при смене СУБД почти весь код доступа к базе данных остается прежним. Этот подход очень хорошо вписывается в Haskell-философию многократного использования кода.

Абстрагирование нескольких СУБД

В Hackage можно найти группу пакетов, сфокусированных на работе в мире SQL, и среди пользователей СУБД эта группа имеет наибольший спрос. Относящиеся к этой группе библиотеки `HDBC` (сокращение от Haskell Database Connectivity) и `hsql` скрывают конкретные детали каждой СУБД, показывая вместо этого SQL-модель (таблицы, строки, столбцы) как таковую, без каких-либо дальнейших абстракций. Нужно лишь подключить тот пакет, который соответствует конкретной выбранной СУБД (например, `hsql-mysql` или `HDBC-postgresql`). Это раскроет перед разработчиком все возможности этих систем, позволяя использовать готовые инструкции и управлять рамками транзакций. Однако при выборе любой из таких библиотек приходится писать SQL-инструкции вручную, что зачастую становится источником ошибок, а кроме того, приходится продвигать результаты в требуемый для вашего приложения тип данных (что обусловливается применением конкретных типов данных для SQL-значений). Из-за этих проблем указанные пакеты редко используются напрямую, а чаще становятся зависимостями для более высокоуровневых пакетов.

СОВЕТ

Самыми интересными для изучения являются библиотеки, предоставляющие общий уровень, располагаемый поверх различных систем управления базами данных. Изучив только одну библиотеку, вы получаете доступ к широкому диапазону баз данных.

На этом более высоком уровне можно найти библиотеку `HaskellDB`. Идея, заложенная в нее разработчиками, заключалась в предоставлении абстракции реляционной алгебры поверх SQL-ориентированной базы данных. Таким образом, ваша работа сводится к таким базовым операциями, как проекции или ограничения, которые позволяют описать все запросы к базе данных. Больше того, схема таблиц также

внедряется в Haskell-код, а следовательно, безопасность всех ваших запросов гарантируется (вы не сможете получить доступ к несуществующим столбцам, не сможете сравнивать столбцы различных типов данных). Основной недостаток библиотеки `HaskellDB` заключается в том, что экспонируемая ею абстракция менее известна по сравнению с SQL-запросами, поэтому для ее использования вам придется заново учиться выражать некоторые концепции.

Знакомство с библиотеками Persistent и Esqueleto

Одной из выбранных для этой главы библиотек является библиотека Persistent, которая доступна в Hackage в пакете `persistent`. Этот пакет поддерживает как реляционные, так и нереляционные СУБД, упрощая тем самым переход между этими двумя мирами в том случае, если это потребуется для улучшения масштабируемости приложения. При использовании библиотеки Persistent вы продолжаете работать с Haskell-типами (поэтому продвижение из вашего типа `Client` и обратно не требуется), причем весь связующий код библиотека генерирует автоматически (с помощью механизма Template Haskell). Это обеспечивает реальную проверку типов в инструкциях базы данных и предотвращение широкого спектра связанных с этим ошибок применения.

Читатели, имеющие опыт объектно-ориентированного программирования, могут распознать в данном описании концепцию системы объектно-реляционного проецирования (Object-Relational Mapper, ORM), которая также берет на себя связывание класса с заданной таблицей базы данных. Однако будучи реализованной на функциональном языке, воплощающем чистоту, библиотека Persistent не обеспечивает автоматического сохранения любых изменений, вносимых в значение, в котором сохраняется информация одной из табличных строк. То есть вам придется явно запрашивать операции вставки, обновления и удаления.

Один из недостатков Persistent заключается в том, что эта библиотека поддерживает общность этих операций только для каждой из баз данных, подключение к которым она обеспечивает. Поскольку сюда входят как реляционные, так и нереляционные СУБД, общая основа иногда имеет ограничения. Например, невозможно легко выразить объединение двух таблиц (в SQL-смысле), поскольку в такой СУБД, как MongoDB, подобное понятие отсутствует. Поэтому были разработаны библиотеки, расширяющие возможности Persistent в одном направлении, что ограничило круг применяемых баз данных. Именно здесь в игру вступает библиотека Esqueleto, предметно-ориентированный язык которой служит для выражения SQL-запросов внутри Haskell-кода, использующего модели библиотеки Persistent.

Богатая функциональность самой библиотеки Persistent касается трех областей:

- описание *схемы* базы данных позволяет использовать ее для работы с обычными Haskell-типами;
- создание и адаптация схемы существующей базы данных к новой схеме, а также добавление или удаление столбцов — функциональность так называемого *переноса* (migration);
- фактическая работа с информацией в базе данных (вставка, обновление, удаление и выполнение запросов).

Соединение

СУБД представляет собой программное обеспечение для фактической работы по сохранению информации в базе данных и извлечению этой информации. Порядок работы предельно прост: вы отправляете инструкцию в СУБД, а она проделывает определенную работу, возвращая некие значения или информацию об успешном проведении операции. Однако перед отправкой инструкций нужно установить начальное соединение с СУБД.

Если заглянуть в имеющийся в Hackage раздел «Database», то можно увидеть, что наряду с базовым пакетом `persistent` существует множество других пакетов, имя которых состоит из префикса `persistent-` и имени СУБД (например, `persistent-sqlite`). Каждый из таких пакетов предоставляет для конкретной системы *внутренний интерфейс* (backend), то есть код, необходимый для взаимодействия с ней и придания смысла возвращаемым ею результатам.

ВНИМАНИЕ

В Hackage имеются и другие пакеты, имена которых начинаются с префикса persistent- (например, persistent-map). К библиотеке Persistent имеют отношение только те пакеты, которые находятся в разделе «Database».

Каждый внутренний интерфейс предлагает два способа взаимодействия с соответствующей системой: через отдельное соединение или путем создания пула соединений (множества открытых соединений, многократно используемых в различных операциях для увеличения объема общих ресурсов и повышения тем самым производительности). Для первого варианта у каждого внутреннего интерфейса имеется функция `with`*`СУБД`*`Conn`, которая генерирует соединение с заданными параметрами, необходимыми для каждой конкретной системы. Это соединение может указываться в качестве аргумента функции `runSqlPersistM`, выполняющей множество действий в базе данных. Например, далее показан код, необходимый для вставки данных типа `Client` (в которых содержится имя, фамилия, адрес, страна и возраст) в базу данных SQLite, сохраненную в файле example.db. Обратите внимание на то, что на данный момент код компилироваться не будет, поскольку мы еще не предоставили определений для сущностей базы данных. Как это делается, мы узнаем чуть позже.

```
{-# LANGUAGE OverloadedStrings #-}
import Database.Persist.Sqlite

exampleConn = withSqliteConn "example.db" $ \conn ->
  flip runSqlPersistM conn $ do
    spain     <- insert $ Country "Spain"
    _client1  <- insert $ Client "Alejandro" "Serrano" "Home Town, 1" spain 25
    return ()
```

ПРИМЕЧАНИЕ

Функция withSqliteConn ожидает значение типа Text с путем к базе данных. В показанном примере для написания этого значения в коде с использованием синтаксиса строкового литерала применяется GHC-расширение OverloadedStrings.

Поскольку открытие соединения и выполнение некоего действия в базе данных — весьма распространенная операция, многие внутренние интерфейсы для подобных простых случаев предоставляют специальную весьма удобную функцию `run`*СУБД*. Например, с использованием функции `runSqlite` показанный код можно сделать более компактным. Если действовать таким образом, не придется самостоятельно передавать значение `conn` через несколько функций, в чем можно убедиться, изучив следующий пример. Во всех остальных аспектах эти два фрагмента кода полностью взаимозаменяемы.

```
exampleConn = runSqlite "example.db" $ do
  spain     <- insert $ Country "Spain"
  _client1  <- insert $ Client "Alejandro" "Serrano" "Home Town, 1" spain 25
  return ()
```

Создание пула обычно ничуть не сложнее создания одиночного соединения: вместо функции `with`*СУБД*`Conn` используется соответствующая функция `with`*СУБД*`Pool`, которая получает на входе информацию для установления контакта с базой данных и максимальное количество открытых соединений. Затем применяется функция `runSqlPersistMPool` с выполняемыми действиями. Далее показан такой же пример для SQLite, но с пулом из 10 соединений (особого смысла в применении пула в данном небольшом примере нет, скорее всего, этот пул мог бы быть полезен для совместного использования разными действиями). Обратите внимание, насколько этот код похож на код примера с единственным соединением:

```
examplePool = withSqlitePool "example.db" 10 $ \pool ->
  flip runSqlPersistMPool pool $ do
    spain     <- insert $ Country "Spain"
    _client1  <- insert $ Client "Alejandro" "Serrano" "Home Town, 1" spain 25
    return ()
```

Среди СУБД наблюдается большое разнообразие в методах соединения. В табл. 11.1 дано краткое пояснение параметров для наиболее распространенных систем.

Таблица 11.1. Параметры соединений для внутренних интерфейсов библиотеки Persistent

СУБД	Одиночное соединение	Пул соединений	Параметры
Sqlite	withSqliteConn	withSqlitePool	Путь к файлу: или «:memory:» для создания временной базы данных в памяти
PostgreSQL	withPostgresqlConn	withPostgreSqlPool	Строка подключения вида «host=h port=p user=u password= p dbname=db»
MySQL	withMySQLConn	withMySQLPool	Значение типа ConnectionInfo. Предоставляется значение defaultConnectionInfo, с помощью которого происходит подключение к базе данных test в localhost, чьи поля connectHost, connectUser и т. д. могут затем быть изменены в соответствии с вашими потребностями

Таблица 11.1 (продолжение)

СУБД	Одиночное соединение	Пул соединений	Параметры
MongoDB	withMongoDBConn	withMongoDBPool	Пять различных параметров для базы данных, хоста, порта аутентификации и времени ожидания. Заметьте, что версия для создания пула нуждается в двух дополнительных параметрах, касающихся количества полос и количества соединений, приходящихся на каждую полосу

Схемы и переносы

Как уже отмечалось в начале этой главы, для обеспечения безопасности типов библиотека Persistent нуждается в описании схемы той базы данных, с которой вы будете работать. Кроме того, ей нужна информация о проецировании этой схемы на имеющиеся Haskell-типы, которые будут использоваться для представления данных. Чтобы собрать все это вместе, в Persistent задействованы довольно сложные типы для кодирования всех инвариантов и связей. Со многими технологиями, применяемыми в этих типах, мы еще не знакомы. Всю необходимую информацию можно найти в главе 13.

Однако есть и хорошая новость: обо всем этом вам беспокоиться не придется, потому что Persistent поставляется с пакетом `persistent-template`, который способен из единственного описания схемы базы данных сгенерировать весь необходимый код (Haskell-типы, экземпляры некоторых классов типов, связующий код для продвижения). В оставшейся части главы предполагается, что помимо внутреннего интерфейса SQLite в вашем проекта в качестве зависимостей перечислены оба пакета, `persistent` и `persistent-template`.

ПРИМЕЧАНИЕ

Для генерирования всего этого кода пакет persistent-template использует механизм Template Haskell, с которым мы познакомились в главе 6. Следует напомнить, что этот механизм является средством метапрограммирования компилятора GHC, а это означает, что во время компиляции любая функция способна сгенерировать новый код или манипулировать уже существующим кодом.

Описание сущностей

Начнем с весьма простого примера для нашего магазина машин времени: описания базы данных, содержащей информацию о клиентах. Чтобы ничего не усложнять, предположим, что нашими клиентами всегда выступают физические лица, информация о которых состоит из имени, фамилии, адреса и возраста. Следующий код определяет сущность в базе данных, предназначенную для хранения этой информации. Позже это определение будет использовано для взаимодействия с нужными таблицами или документами в базе данных.

```
{-# LANGUAGE TemplateHaskell, QuasiQuotes, TypeFamilies, EmptyDataDecls,
             FlexibleContexts, GADTs, OverloadedStrings #-}

import Database.Persist.TH

mkPersist sqlSettings [persistLowerCase|
Client
  firstName String
  lastName  String
  address   String
  age       Int
  deriving Show
|]
```

За несколькими необходимыми расширениями языка, перечисленными в директиве транслятора `{-# LANGUAGE #-}`, следует вызов функции `mkPersist`. Этой функции нужны параметры для генерирования кода и описание базы данных. В данном случае код сообщает системе генерирования кода о необходимости следования соглашениям стандарта SQL, для чего в качестве первого аргумента передается функция `sqlSettings`[1]. Последний аргумент указывается не с помощью обычного синтаксиса Haskell, а с использованием *квазицитирования* (quasiquotation), то есть кода между символами `[,` `|` и `|]`, который будет обрабатываться не парсером компилятора Haskell, а специальным читателем (в данном случае — функцией `persistLowerCase`). Таким образом, нам важно узнать, как именно требуется выразить схему базы данных для этой системы квазицитирования.

ВНИМАНИЕ

Важно понимать, что внутри системы квазицитирования вы пишете уже не Haskell- код, а некий код, соответствующий специальному синтаксису библиотеки Persistent. Необходимость изучения дополнительного (хотя и небольшого по объему) языка некоторыми специалистами считается недостатком пакета persistent-template.

Как видите, для каждой сущности, которую нужно сохранить, есть собственный блок. В данном случае сущность у нас одна, а именно `Client`. Затем можно увидеть все поля, составляющие эту сущность: сначала имя, а затем тип. И наконец, можно включить инструкцию `deriving`, что, собственно, и делается в любом определении любого другого Haskell-типа. Это базовая конструкция, о которой по мере чтения этой главы мы будем узнавать все больше и больше.

СОВЕТ

Если при квазицитировании сразу же после имени сущности написать json (например, Client json), Persistent будет автоматически извлекать экземпляры ToJSON и FromJSON библиотеки aeson. Это пригодится в следующей главе, где рассказывается о создании веб-приложений с помощью Haskell.

[1] Это ни в коей мере не означает, что данное определение нельзя использовать для нереляционных СУБД. Это всего лишь выбор, сделанный нами для представления информации (как выбор целых чисел для хранения первичных ключей), и при таком выборе базы данных на основе SQL подходят больше остальных.

Поскольку в данном коде функция `persistLowerCase` используется как система квазицитирования, каждая сущность проецируется на таблицу, а каждое поле — на столбец, имя которого составляется преобразованием из значения с символами в смешанном регистре (где каждое слово в идентификаторе начинается с прописной буквы) в значение с символами подчеркивания (разные слова в идентификаторе отделяются друг от друга символами подчеркивания). В рассматриваемом примере данные будут сохраняться в таблице `client` со столбцами `first_name`, `last_name`, `address` и `age`. Чтобы сохранять имена в смешанном регистре, вместо функции `persistLowerCase` следует задействовать функцию `persistUpperCase`.

Кроме связующего кода для контактирования с базой данных функция `mkPersist` сгенерирует конструкторы типов данных, представляющие каждое из полей сущности. Имена этих конструкторов создаются путем соединения имени сущности с именем каждого из полей в смешанном регистре. В данном случае получаются имена `ClientFirstName`, `ClientLastName` и т. д. Это, в частности, означает, что в разных сущностях могут быть одинаковые имена полей (например, можно использовать имя поля как для сущности индивидуального клиента, так и для сущности клиента, представляющего компанию), поскольку их можно обособить за счет префикса. В этом заключается их отличие от Haskell-записей, у которых не может быть одинаковых имен полей.

Как вы, наверное, заметили, до сих пор не было дано описания *идентификатора* (identifier), то есть не было показано, каким может быть первичный ключ таблицы. Первичный ключ — это часть информации, которая уникальным образом определяет строку таблицы и которая обычно используется в SQL-совместимых средах. Идея с идентификатором заинтересовала также разработчиков библиотеки Persistent, поэтому в схеме для каждой сущности всегда имеется неявное определение поля `id`. Вследствие этого кроме всех явным образом объявленных полей у вас также будет поле `ClientId`, представляющее уникальный идентификатор каждой строки.

Помимо решения задач повышения производительности это поле служит для ссылок из одной сущности на другую или для внедрения одной сущности в другую. Давайте, к примеру, добавим информацию о стране пребывания каждого клиента. Сделать это можно простым добавлением еще одного поля `String` для названия страны. Но, как гласит теория баз данных, признать эту идею удачной нельзя: повторяющаяся информация приведет к неоправданным расходам памяти, но что еще хуже — может привести к проблемам, когда одна и та же страна будет не везде написана одинаково (например, `"United States"`, `"US"` или `"USA"`). Правильным решением было бы создание новой сущности, хранящей информацию о странах, и ссылка на нее в сущности `Client` (на жаргоне баз данных это называется «нормализацией»). С решением данной задачи достойно справится поле идентификатора. Новая схема объявления будет выглядеть следующим образом:

```
mkPersist sqlSettings [persistLowerCase|
Country
  name String
  deriving Show
Client
  firstName String
```

```
  lastName  String
  address   String
  country   CountryId
  age       Int
  deriving Show
|]
```

При таком определении схемы вставка, показанная в предыдущем разделе, является вполне корректным Haskell-кодом. Обратите внимание на то, что библиотека Persistent создала тип данных для каждой сущности. Эти типы данных могут считаться записью, единственный конструктор которой использует общее с определяемой сущностью имя. Таким образом, в примере оказываются сгенерированными конструкторы `Country` и `Client`. Также следует заметить, что вставка возвращает идентификатор новой сущности базы данных, а код пользуется этим для ссылки в данных клиента на страну `"Spain"` (Испания).

ОСОБАЯ РОЛЬ ИДЕНТИФИКАТОРОВ

В большинстве баз данных столбец идентификатора является всего лишь еще одним столбцом с особым свойством уникальности строк и, как правило, возможностью автоприращения. Однако вы видели, что в библиотеке Persistent такие поля рассматриваются в особом порядке. Почему так происходит?

Основная причина в том, что при наличии для каждого идентификатора каждой сущности специальных типов данных невозможно было бы смешивать идентификаторы разных сущностей. В нашем примере тип ClientId не совпадал бы с типом CountryId, следовательно, значение последнего идентификатора невозможно передать, если запрашивается элемент первого.

Вторая причина заключается в том, что, скрывая детали представления идентификатора, Persistent делает еще один шаг в направлении поддержания различных баз данных: представление может выбираться в зависимости от конечной СУБД. Например, у MongoDB есть специальный тип ObjectId, разработанный для идентификации объектов. Это отличается от того пути, по которому пошли базы данных на основе SQL.

О том, что Persistent всегда генерирует специальное поле идентификатора для каждой сущности, мы уже рассказывали. Но в других случаях существуют иные *ограничения уникальности* (uniqueness constraints) — множество полей, которые уникальным образом идентифицируют значение в базе данных. Поскольку общие значения этих полей должны появляться в базе данных только один раз, СУБД должна не допускать их повторения. Таким образом, база данных может защитить данные от целого класса некорректных значений. Кроме того, дополнительная работа по вставке и обновлению зачастую компенсируется выигрышем в производительности, который достигается за счет уникальности.

На ограничения уникальности, объявленные внутри сущности, ссылаются по имени, которое должно начинаться с прописной буквы (в следующем далее примере все ограничения имеют префикс `Unique`, но это не является обязательным требованием), а затем перечисляются поля, включенные в ограничение. Например, можно считать, что комбинация имени и фамилии, адреса и страны уникальным образом идентифицирует клиента в базе данных, поэтому стоит добавить к ним ограничения уникальности. Кроме того, страны также идентифицируются уникальным образом по их названиям (двух стран с одинаковым названием не бывает). Все это реализовано в следующем коде (некоторые сущности и поля опущены):

```
mkPersist sqlSettings [persistLowerCase|
Country
  ...
  UniqueCountryName name
  deriving Show
Client
  ...
  UniqueClient firstName lastName address country
  deriving Show
|]
```

Для анализа данных весьма полезно сохранять возраст каждого клиента. Но не все клиенты придут в восторг от необходимости предоставления такого рода информации, поэтому лучше сделать ее необязательной. В SQL это соответствует пометке столбца как *допускающего значение null* (nullable), а в Haskell это соответствует заключению значения в оболочку `Maybe`. Несомненно, это последнее и было выбрано в Persistent для признака необязательности значения в поле: если значение `age` нужно сделать необязательным, его объявление нужно поменять с `age Int` на `age Int Maybe`.

Кроме возраста было бы полезно сохранять информацию о половой принадлежности клиента. В начале книги уже рассматривался вопрос о том, что использование булева значения не является верным выбором для представления пола в Haskell, вместо него лучше воспользоваться более удобным типом данных. Тем самым будет повышен уровень безопасности типов и исключены ошибки, возникающие из-за неоднозначных вариантов проецирования (как определить, какому полу соответствует значение `True` — мужскому или женскому?).

Те же самые аргументы относятся и к Persistent: когда нужно сохранить значение в поле при наличии множества значений, не используйте булевы или целочисленные типы. Мы уже видели, как решить подобную проблему, когда множество значений велико и к тому же может меняться со временем (как происходит со странами). Для небольшого фиксированного списка значений наилучшим выбором будет Haskell-перечисление и использование этого типа в объявлении сущности (в нашем примере это создание типа данных `Gender` и объявление поля `gender` типа `Gender` в сущность `Client`). Единственным камнем преткновения на этом пути является необходимость включения дополнительного кода с целью обеспечить доступность нового типа данных для библиотеки Persistent, причем по причине ограничений механизма Template Haskell это нужно будет сделать в отдельном файле. В данном случае начните с написания этого кода в новом модуле `Chapter11.Gender`:

```
{-# LANGUAGE TemplateHaskell #-}
module Chapter11.Gender where

import Database.Persist.TH
data Gender = Male | Female
    deriving (Show, Read, Eq)
derivePersistField "Gender"
```

Здесь важной частью является вызов `derivePersistField`. Теперь в файл с объявлением схемы нужно импортировать этот новый модуль и добавить к нему строку `gender Gender Maybe` (также неплохо будет сделать необязательной информацию

о принадлежности к определенном полу). В результате значение `Client` должно выглядеть так:

```
Client "Alejandro" "Serrano" "Home Town, 1" spain (Just 25) (Just Male)
```

Приведенная информация является основой языка схем, но вам доступны также многие другие возможности. Поэтому прежде, чем объявлять свои сущности, настоятельно рекомендуем вам изучить документацию по библиотеке Persistent (https://github.com/yesodweb/yesod/wiki/Persistent-entity-syntax) или по платформе Yesod (http://www.yesodweb.com/book/persistent).

Создание базы данных

Теперь давайте рассмотрим обратный сценарий: предположим, что вы пытаетесь запустить код добавления страны и клиента в совершенно пустую базу данных. Например:

```
runSqlite "example.db" $ do
  spain     <- insert $ Country "Spain"
  _client1  <- insert $ Client "Alejandro" "Serrano" "Home Town, 1" spain (Just 25)
(Just Male)
  return ()
```

Если выполнить этот код либо посредством исполняемого файла, либо в GHCi, будет получено сообщение об ошибке следующего вида:

```
chapter11: user error (SQLite3 returned ErrorError while attempting to perform
prepare "INSERT INTO \"country\"(\"name\") VALUES(?)": no such table: country)
```

Об источнике данной ошибки догадаться нетрудно: вами объявлено, как нужно проецировать сущности на базу данных, но таблиц, на которые ссылается этот код, пока нет. Приятной новостью будет уже упомянутая функциональность библиотеки Persistent, касающаяся возможности автоматического создания или адаптации базы данных под ожидаемый схемой формат. Этот процесс адаптации называется *переносом* базы данных.

Перенос в Persistent состоит из двух этапов. Прежде всего, нужно сообщить библиотеке о необходимости генерирования кода переноса базы данных. Для решения этой задачи библиотека опять прибегает к помощи механизма Template Haskell, и вам придется объявить схему еще раз. Поскольку использование одной и той же схемы для создания проекций и переноса — дело весьма распространенное, можно воспользоваться весьма удобной функцией `share`, чтобы передать код квазицитирования обеим функциям. Проще говоря, это означает, что вам достаточно изменить вызов `mkPersist`:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
```

Следующим шагом станет выполнение переноса путем вызова функции `runMigration` с одним аргументом, в качестве которого используется имя, заданное `mkMigrate`. Например, с помощью следующего кода можно сгенерировать таблицу в базе данных Sqlite:

```
runSqlite "example.db" $ runMigration migrateAll
```

Если разместить эту строку кода перед двумя вставками в базу данных, вы увидите на экране журнал переноса. Эта журнальная запись может выглядеть так, как показано в следующем фрагменте. Как видите, в мире SQL все поля и ограничения в сущностях естественным образом превращаются в таблицы:

```
Migrating: CREATE TABLE "country"("id" INTEGER PRIMARY KEY,"name" VARCHAR NOT
NULL,CONSTRAINT "unique_country_name" UNIQUE ("name"))
Migrating: CREATE TABLE "client"("id" INTEGER PRIMARY KEY,"first_name" VARCHAR NOT
NULL,"last_name" VARCHAR NOT NULL,"address" VARCHAR NOT NULL,"country" INTEGER
NOT NULL REFERENCES "country","age" INTEGER NULL,"gender" VARCHAR NULL,CONSTRAINT
"unique_client" UNIQUE ("first_name","last_name","address","country"))
```

Важно отметить, что переносы могут только добавлять и удалять столбцы: код не может узнать о замене одного имени поля другим. Но даже простой случай добавления нового поля вызывает вопрос: какое значение должно быть записано для тех строк, которые уже были в таблице? Решение приходит в форме нового атрибута `default`, который может быть назначен полю и определяет записываемое значение в случае переноса.

ВНИМАНИЕ

Атрибут default применяется только к переносам. Он не оказывает никакого влияния на реальный Haskell-код, а также не позволяет оставлять значение для поля неопределенным.

В качестве примера давайте добавим к сущности `Country` поле `canWeSend`, выражающее возможность поставки машин времени и аксессуаров в определенную страну. Поскольку таблицы уже созданы и к ним добавлены значения, для процедуры переноса нужна информация о значениях, записываемых в это поле. Если решить, что наилучшим вариантом будет `True` (доставка может быть осуществлена во все страны, уже имеющиеся в базе данных), определение схемы должно быть изменено так:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
Country
  name      String
  canWeSend Bool default=True
  UniqueCountryName name
  deriving Show
Client
  ...
|]
```

В следующей врезке рассказывается, как работать с унаследованной базой данных, имеющей имена таблиц и столбцов, не совпадающие с именами наших сущностей и полей. Затем выполнение упражнения 11.1 приведет нас к определению сущности, которая будет использоваться для хранения информации о товарах в оставшейся части главы.

ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕЙ БАЗЫ ДАННЫХ

Как выполнить перенос для создания новой или адаптации существующей базы данных на основе той схемы таблиц, которая ожидалась библиотекой Persistent, мы уже видели. Но зачастую возникают ситуации, требующие доступа к существующим базам данных.

В таких случаях для изменения имен таблиц, на которые проецируются сущности и поля, можно указать атрибут sql. Кроме того, для сущности можно указать атрибут id, чтобы объявить

столбец в таблице, содержащий идентификатор. Следующий код является примером для сущности Country:

```
Country sql=my_country_table id=the_country_identifier
  name      String
  canWeSend Bool default=True sql=send_there_is_possible
  UniqueCountryName name
  deriving Show
```

Именем таблицы станет my_country_table, а столбец the_country_identifier будет считаться столбцом идентификатора.

УПРАЖНЕНИЕ 11.1. ОПРЕДЕЛЕНИЕ ТОВАРОВ

Добавьте в базу данных определение сущности Product. Эта сущность должна включать название и описание каждого товара, его цену в виде значения типа Double, а также количество имеющихся в наличии единиц товара.

Следует учесть, что библиотека Persistent не позволяет использовать в полях значения типа Integer: вместо них применяют тип Int64, представляющий целочисленные значения, помещающиеся в 64 разряда.

И наконец, давайте добавим простое определение для покупок в магазине: покупка будет представлена коллекцией, состоящей из клиента, отдельного товара, в отношении которого клиент может сделать заказ на более чем один экземпляр, и общей уплаченной суммы (здесь имеется неявно выраженное предположение, что каждая покупка состоит только из одного типа товара). Сущность `Purchase` в этом случае будет связана с сущностями `Client` и `Product`, поэтому, как показано в приведенном далее коде, нужно воспользоваться их идентификаторами:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
...
Client
  ...
Product
  ...
Purchase
  client  ClientId
  product ProductId
  number  Int
  amount  Double
  deriving Show
|]
```

Запросы

Выполнение запросов является одной из основных операций в базе данных. Твердые знания различных вариантов выбора данных из имеющихся источников обеспечивают достижение нужной приложениям производительности. В данном разделе показано, что Persistent предоставляет несколько возможностей для запроса данных, а Esqueleto еще больше расширяет эти возможности с помощью SQL-операторов, таких как операторы объединения.

ПРИМЕЧАНИЕ

Во всех остальных примерах данной главы рассматриваются только инструкции. При желании выполнить любую из этих инструкций ее нужно заключить в оболочку функции runSqlite (или функции, соответствующей версии для других баз данных), как это сделано в предыдущих примерах переноса. Сигнатуры в коде также не показаны, поскольку они зависят от свойств системы типов, с которыми мы познакомимся позже (см. главу 13). Однако компилятор GHC способен корректно строить сигнатуры, поэтому вы по-прежнему можете копировать и вставлять примеры по мере их появления в книге.

Запросы по идентификатору или по уникальным полям

Поскольку каждую сущность, имеющуюся в базе данных, Persistent снабжает идентификатором, простейшим способом выполнить запрос к базе данных является, разумеется, запрос значения с заданным идентификатором. Это делается с помощью функции `get`, которая возвращает значение, заключенное в оболочку `Maybe`, поскольку может случиться так, что соответствующего значения для заданного идентификатора не окажется. Например, в сущности `Purchase` мы определили поле `client` для хранения значения `ClientId`; затем этот идентификатор может быть использован для извлечения всей информации о клиенте. Эту задачу решает следующий код:

```
getPurchaseClient p = get (purchaseClient p)  -- возвращает Maybe Client
```

В случае, если у вас имеется идентификатор, но он не получен из предыдущего запроса, вследствие чего заключен в оболочку соответствующего конструктора для идентификатора, вы все равно можете произвести поиск объекта, используя конструктор `Key`. Хотя этого лучше избегать, подобный вариант может понадобиться, к примеру, если идентификатор извлекается из маршрута в веб-приложении (то есть из URL-адреса, подобного `/clients/5`, где 5 — это идентификатор клиента). Поскольку способ трактовки идентификатора меняется в зависимости от фактически используемой СУБД, этот рабочий прием зависит от применяемой базы данных: значение идентификатора нужно создать с тем типом, который подходит для вашей конкретной базы данных. Например, в SQLite используются целые числа, поэтому код для запроса `Client` по идентификатору будет иметь следующий вид:

```
getClientById n = get $ Key (PersistInt64 $ fromIntegral n) -- возвращает Maybe
Client
```

Другой способ идентификации значения — ограничения уникальности — мы уже рассматривали. Для каждого из этих ограничений Persistent генерирует конструктор, который в качестве аргументов получает значение для всех полей в его определении. Затем с помощью функции `getBy` можно выполнить запрос к базе данных, используя эту комбинацию данных. Например, ранее в отношении сущности `Client` были определены ограничения уникальности `UniqueClient`, поэтому можно выполнить запрос с одним из них:

```
getClientByInfo fName lName addr cnName = do
    cn <- getBy $ UniqueCountryName cnName
    case cn of
      Just (Entity cId _) ->
```

```
    do cl <- getBy $ UniqueClient fName lName addr cId
       case cl of
         Just (Entity _ client) -> return $ Just client
         Nothing                -> return Nothing
Nothing -> return Nothing
```

Важно отметить, что результат выполнения функции `getBy` является не просто запрашиваемым значением, а некой комбинацией внутри типа данных `Entity`, состоящей из идентификаторов, связанных со значением, и самого значения. В показанном ранее примере можно найти два разных варианта использования `getBy`: один для нахождения идентификатора страны по ее названию (обратите внимание, что образец `Entity` соответствует только тому идентификатору, который действительно привязан к переменной), второй для нахождения клиента с заданными персональными атрибутами и принадлежащего этой стране.

Выбор нескольких сущностей

Вся мощь баз данных проявляется при выполнении запросов не только через идентификаторы, но и через другие поля внутри сущностей. Кроме того, языки запросов обычно весьма выразительны. Например, можно запросить всех клиентов из США, купивших как минимум два товара в течение последних нескольких лет; данная информация позволила бы найти оптимальный подход к клиентам в этой части света.

При таких запросах обычно не гарантируется возвращение одного результата или полное отсутствие результатов: требованиям запроса может отвечать целое множество значений. От Persistent можно потребовать двух разных вариантов представления этого множества: либо в виде типа `Source` из библиотеки `conduit`, либо в виде обычного списка. Можно также выбрать возвращение только первого значения множества, заключенного в оболочку `Maybe`. Затем можно получить назад значение `Nothing`, если результатом запроса будет пустое множество.

Можно также потребовать от Persistent показать все множество полей каждого значения, соответствующего запросу (то есть всех полей, принадлежащих сущности), или вернуть только их идентификаторы. В первом случае информация из полей заключается в оболочку конструктора `Entity` (как при использовании функции `getBy`), который содержит как сущность, так и ее идентификатор. Как показано в табл. 11.2, пять из шести возможных комбинаций имеют в Persistent соответствующую функцию.

Таблица 11.2. Доступные в Persistent функции для выполнения запросов

	В виде типа Source	**В виде списка**	**Только первый результат в Maybe**
Идентификатор и значение	selectSource	selectList	selectFirst
Только идентификатор	selectKeys	selectKeysList	Функция недоступна

Все функции, перечисленные в табл. 11.2, имеют одинаковые аргументы и используются одинаковым образом, однако в примерах будет использоваться функция

`selectList`, поскольку это одна из тех функций, вывод которой достаточно простой, но в то же время позволяющий понять, что происходит в базе данных.

Первым из двух аргументов является список *фильтров* (filters). Каждый фильтр ограничивает значение одним или несколькими полями для удовлетворения определенного условия. Например, вам может потребоваться, чтобы возраст (`age`) клиента (`Client`) был больше 18 лет. Эти фильтры пишутся с помощью одного из конструкторов, которые Persistent предлагает для каждого из полей сущности, и набора встроенных операторов. Фильтр возраста, к примеру, представляется как `ClientAge >. Just 18` (`Just` здесь нужно потому, что поле `age` было определено как необязательное, поэтому его значения заключаются в оболочку `Maybe`).

Кроме простого равенства и операторов упорядочения (`==.`), (`!=.`)[1], (`>.`), (`>=.`), (`<.`) и (`<=.`), Persistent поддерживает оператор для поля, имеющего значение из множества требуемых значений, а именно оператор (`<-.`), который читается как «в», а также его отрицание (`/<-.`), которое читается как «не в». В следующем примере с помощью этого оператора выполняется запрос информации обо всех клиентах (`Client`) из Испании (`Spain`) или Германии (`Germany`) старше 18 лет. Обратите внимание на то, что, прежде чем создать сам запрос, необходимо получить идентификаторы этих стран:

```
getAdultsOfSpainAndGermany = do  -- возвращает [Entity Client]
  es <- getBy $ UniqueCountryName "Spain"
  de <- getBy $ UniqueCountryName "Germany"
  let countries = map entityKey (catMaybes [es, de])
  selectList [ ClientCountry <-.countries, ClientAge >=. Just 18 ] []
```

Если нужно лишь узнать количество клиентов, удовлетворяющих этим условиям, можно просто применить к возвращаемому списку функцию `length`. Но этот вариант не отличается эффективностью, поскольку из базы данных придется извлечь большой объем данных, которые в дальнейшем просто отбрасываются. Вместо этого нужно воспользоваться функцией `count`, возвращающей только то количество значений, которое удовлетворяет условиям запроса. Например:

```
countAdultsOfSpainAndGermany = do
  sp <- getBy $ UniqueCountryName "Spain"
  de <- getBy $ UniqueCountryName "Germany"
  let countries = map entityKey (catMaybes [es, de])
  count [ ClientCountry <-. countries, ClientAge >=. Just 18 ]
```

Иногда уже упомянутых операторов не хватает или же они неудобны для выражения альтернативных запросов. В таких случаях можно воспользоваться оператором «или», то есть оператором (`||.`), предназначенным для слияния двух списков фильтров в единый запрос. Предположим, к примеру, что нужно получить клиентов из Испании и США, достигших возраста, разрешающего приобретение алкоголя, или более старшего возраста. Но этот возраст в обеих странах не одинаков: в Испании это 18 лет, в то время как в США — 21 год. Таким образом, в запросе этот факт должен быть отражен: как показано в коде, нужно получить клиентов из Испании старше 18 лет или из США старше 21 года:

---

[1] Оператор «не равно» записывается не как (/=.), а как (!=.), что напоминает синтаксис языка C. Причина в том, что оператор (/=.) используется в запросах на обновление.

```
getAdultsOfSpainAndUS = do  -- возвращает [Entity Client]
  Just (Entity spId _) <- getBy $ UniqueCountryName "Spain"
  Just (Entity usId _) <- getBy $ UniqueCountryName "United States of America"
  selectList (   [ ClientCountry ==. spId, ClientAge >=. Just 18 ]
             ||. [ ClientCountry ==. usId, ClientAge >=. Just 21 ] )
             []
```

Как видите, в качестве второго аргумента `selectList` указан пустой список. Этот список представляет *параметры* (options) запроса, которые не влияют на результаты, а задают способ их представления. Один из типичных параметров подразумевает сортировку по одному или нескольким полям. В показанном примере, чтобы вернуть клиентов в порядке от самого старшего к самому младшему, список параметров должен быть таким: `[ Desc ClientAge ]`. Как видите, для указания порядка следования используется один из конструкторов, `Asc` или `Desc`, а также конструктор соответствующего поля.

Другие параметры позволяют извлечь только одно подмножество результатов. Если все множество результатов считать упорядоченным списком, то параметр `OffsetBy` позволяет отказаться от нескольких элементов в его начале и вернуть только те, которые расположены от заданного места и до конца; а параметр `LimitTo` ограничит количество возвращаемых значений, уменьшив его до заданного, игнорируя при этом все следующие элементы. Эти параметры весьма полезны для разбиения информации на страницы в веб-приложении: например, в магазине можно показывать по десять товарных позиций на странице. Тогда на странице n (если счет ведется с 1) будут показаны десять товарных позиций начиная с элемента $(n-1)\times 10$ (результат, получаемый из Persistent, имеет индексацию, начинающуюся с нуля). Соответствующий запрос для получения информации об этих товарах будет иметь следующий вид:

```
getProductsPage n = selectList [ ] [ Asc ProductPrice, LimitTo 10, OffsetBy
((n-1)*10) ]
                    -- вазвращает [Entity Product]
```

Создание SQL-запросов с помощью Esqueleto

Давайте, используя лишь функции `get`, `getBy`, `selectList` и `count` (ранее не представленную версию, возвращающую количество результатов), попробуем получить названия стран всех тех клиентов, которые купили более трех товаров. Показанная далее реализация выполняет это за несколько шагов: сначала она получает всех возможных покупателей, затем подсчитывает количество покупок каждого из них, отфильтровывая тех, у кого более трех покупок, и, наконец, получает названия стран для тех, кто остался после фильтрации. Обратите внимание на сложность кода и на несколько обращений этого кода к базе данных:

```
getCountriesWithBigBuyers = do  -- возвращает [Country]
  buyers <- selectKeysList [ ] [ ]
  buyersPurchases <- mapM (\b -> count [ PurchaseClient ==. b ] >>= \c -> return
(b,c)) buyers
  let buyersPurchases' = filter (\(_,c) -> c > 3) buyersPurchases
  mapM (\(b,_) -> do Just cl <- get b
                     Just cn <- get $ clientCountry cl
                     return cn)
       buyersPurchases'
```

Если вы знакомы с реляционными базами данных, то знаете о механизме *объединения* (joins) результатов выполнения запросов, который мог бы помочь при решении данной проблемы. Однако библиотека Persistent поддерживает также и нереляционные базы данных, что делает возможность объединения недоступной для нас.

ПРИМЕЧАНИЕ

На самом деле Persistent поддерживает отправку к базе данных необработанного запроса. Но в таком запросе отсутствуют проверка типов и автоматическое продвижение, то есть утрачиваются все преимущества от использования библиотеки.

В соответствии с ранее предоставленными возможностями решение этой проблемы заключается в использовании еще одной библиотеки под названием Esqueleto. Пакет `esqueleto` поддерживает написание SQL-запросов с проверкой типов. Кроме того, в нем многократно используются определения схемы библиотеки Persistent, что делает из этих двух библиотек великолепный тандем. Первое, что нужно сделать для применения Esqueleto, — это изменить инструкцию импорта `Database.Persist` на `Database.Esqueleto`, поскольку вторая библиотека подключает первую. Однако следует заметить, что вам по-прежнему нужен импорт какого-то конкретного внутреннего интерфейса, например модуля `Database.Persist.Sqlite`.

Как и в предыдущем случае, результат можно вернуть в двух различных структурах данных. Функция `select` показывает запрошенные значения в виде списка, а функция `selectSource` заключает их в оболочку `Source` из пакета `conduit`. Обе эти функции получают в качестве аргумента тип `SqlQuery`, поэтому вам нужно сконцентрироваться на том, как строить значения этого типа. Кроме того, есть варианты `selectDistinct` и `selectDistinctSource`, которые также позволяют избавиться от дублирующихся значений.

Сначала нужно выбрать, к какой сущности будет запрос. Для этого используется ключевое слово `from`. Это очень интересная часть библиотеки, поскольку в качестве аргумента передается функция и в зависимости от типа функции запрашивается подмножество сущностей. Например, если нужно сделать запрос о клиентах, нужно передать функцию, чей уникальный аргумент относится к типу `Client`. В большинстве случаев этот тип выводится компилятором на основе последующих фильтров (например, если фильтром является `ClientAge`, компилятор сделает вывод, что ваш запрос относится к `Client`), но в некоторых случаях могут понадобиться дополнительные разъяснения. Если нужно выполнить запрос, в котором задействуется более одной сущности, эти сущности нужно передавать в кортеже.

Тело функции, передаваемое `from`, должно быть определенного монадического типа, так что здесь довольно часто применяется блок `do`. Внутри этой монады для уточнения запроса может вызываться ряд специальных функций. Самой простой из них является функция `where_` (обратите внимание на завершающий символ подчеркивания, предотвращающий конфликт с ключевым словом `where`), включающая множество ограничений для полей сущностей внутри выражения:

- Доступ к полю сущности с помощью оператора `(^.)`.
- Постоянные значения, заключенные в оболочку выражения с помощью `val`, или их список, заключенный в оболочку с помощью `valList`.

- Сравнения с помощью операторов (==.), (!=.)... и всех остальных операторов, доступных в Persistent. При этом следует заметить, что оператор (<-.) превращается в in_, а (/<-.) — в notIn.
- Группировка из нескольких ограничений с оператором (&&.) (для «и») и (||.) (для «или»). Обратите внимание, что здесь есть отличия от библиотеки Persistent, поскольку в этом случае там не было объединительного оператора: набор фильтров в списке объединялся неявным образом.

Еще одной возможной функцией является orderBy, которая получает список выражений в форме asc (entity ^. field) или desc (entity ^. field), представляющих порядок следования полей по возрастанию и по убыванию значений. В качестве простого примера давайте просто вернем всех клиентов старше 25 лет, упорядоченных по фамилии и имени:

```
getPeopleOver25 =  -- возвращает [Entity Client]
  select $
  from $ \client -> do
  where_ (client ^. ClientAge >. just (val 25))
  orderBy [ asc (client ^. ClientLastName), asc (client ^. ClientFirstName) ]
  return client
```

Код явно показывает, что у поля ClientAge имеется значение Maybe. Постоянное значение, с которым сравнивается значение поля, должно быть заключено сначала в оболочку val, а затем в оболочку just.

Возвращаемое значение монады может не состоять из всех запрошенных сущностей: к следующему шагу может иметь отношение только подмножество информации, а все остальное должно быть отброшено. Давайте, к примеру, попробуем получить информацию обо всех клиентах из Испании или Германии старше 25 лет. В представленном далее коде для запроса выбраны как сущность Client, так и сущность Country, но при этом нужно убедиться, что обе сущности связаны правильно (то есть Country в Client совпадает с идентификатором Country). Кроме того, были наложены ограничения на название страны и возраст клиента. И наконец, мы возвращаем только информацию о клиентах, а не название страны:

```
getPeopleOver25FromSpainOrGermany =  -- возвращает [Entity Client]
  select $
  from $ \(client, country) -> do
  where_ (   client ^. ClientAge >. just (val 25)
         &&. country ^. CountryName `in_` valList [ "Spain", "Germany" ]
         &&. client ^. ClientCountry ==. country ^. CountryId )
  orderBy [ asc (client ^. ClientLastName), asc (client ^. ClientFirstName) ]
  return client
```

Здесь показан пример объединения, подразумеваемого в ссылке между клиентской страной и категорией, представляющей саму страну. Библиотека Esqueleto выражает связь более конкретно, используя для этого в качестве аргумента не просто кортеж, а конструктор InnerJoin. Затем с функцией on в монаде указываются условия объединения сущностей. Предыдущий пример может быть переделан следующим образом:

```
getPeopleOver25FromSpainOrGermanyJoin =  -- возвращает [Entity Client]
  select $
```

```
  from $ \(client `InnerJoin` country) -> do
  on (client ^. ClientCountry ==. country ^. CountryId)
  where_ (   client ^. ClientAge >. just (val 25)
         &&. country ^. CountryName `in_` valList [ "Spain", "Germany" ])
  orderBy [ asc (client ^. ClientLastName), asc (client ^. ClientFirstName) ]
  return client
```

Кроме внутренних объединений Esqueleto поддерживает внешние объединения, которые в ряде случаев могут пригодиться.

Чтобы завершить это краткое знакомство с Esqueleto, нужно узнать, как выражается *группировка* (grouping) в том виде, в каком это делается в SQL. Сама группировка выполняется с помощью функции `groupBy`, которой в качестве аргумента передается поле группировки. Затем в отношении других полей можно использовать операторы агрегирования, такие как `sum_`, `min_`, `max_`, `avg_` или `countRows`, чтобы выполнить эту операцию над всеми значениями каждой группы. Однако подробное объяснение порядка работы этих операторов займет много времени, к тому же эта тема больше подходит для книги по языку SQL, от которого библиотека Esqueleto взяла многие концепции.

В качестве примера группировки давайте попробуем вычислить сумму, потраченную каждым клиентом на все его покупки. Замысел решения этой задачи заключается в группировке информации обо всех элементах сущности `Purchase` по идентификатору `Client`, а затем в использовании оператора агрегирования `sum_`. Обратите внимание на то, что как для агрегирования, так и для тех клиентов, которые еще не приобретали товар, необходимо воспользоваться левым внешним объединением.

```
getMoneyByClient =  -- возвращает [(Entity Client, Value (Maybe Double))]
  select $
  from $ \(client `LeftOuterJoin` purchase) -> do
  on (client ^. ClientId ==. purchase ^. PurchaseClient)
  groupBy (client ^. ClientId)
  let s = sum_ (purchase ^. PurchaseAmount)
  return (client, s)
```

Безусловно, язык, поддерживаемый библиотекой Esqueleto, очень выразителен и позволяет выражать запросы, выходя далеко за пределы того, о чем рассказано в данной главе. Когда вам понадобится выполнять полноценные и эффективные запросы к реляционной базе данных, нужно внимательно прочитать документацию по этой библиотеке, чтобы полностью изучить все ее конструкции.

Вставка, обновление и удаление

В начале этой главы мы узнали о вставке в базу данных новых значений с помощью функции `insert`. Она возвращает идентификатор, присвоенный новому значению, сохраненному в базе данных. Этот идентификатор может пригодиться при ссылке на него из других значений базы данных. Однако в некоторых случаях вставка значения в базу данных может не произойти из-за конфликтов, вызванных ограничениями уникальности. В этом случае рекомендуется вместо функции `insert` воспользоваться функцией `insertUnique`, возвращающей значение `Maybe`, которое в случае невозможности вставки значения превратится в `Nothing`.

Как уже упоминалось, в Persistent используется парадигма чистоты функций, поэтому все изменения, вносимые в значение базы данных, должны быть явными. Проще всего изменить значение с помощью функции `replace`: этой функции передаются идентификатор и значение одной из сущностей, а она заменяет новым значением то значение, которое до этого хранилось в базе данных с упомянутым идентификатором. Например, очень медленным, но простым способом сделать так, чтобы все имена, имеющиеся в базе данных, начинались с прописной буквы, является использование следующего кода:

```
import Data.Char

capitalizeNamesSlow = do
  clients <- selectList [] []
  mapM_ (\(Entity ident client) ->
          let c:rest  = clientFirstName client
           in replace ident $ client { clientFirstName = (toUpper c):rest })
        clients
```

Это решение работает медленно, поскольку все данные из базы данных перемещаются в программу с последующей обратной передачей всей информации для каждого клиента. В простых случаях та же самая задача может быть решена прямо на стороне базы данных без передачи туда и обратно какой-либо информации. Библиотека Persistent предлагает для подобных действий механизм *обновления на месте* (in-place update). В этом случае операция либо является присвоением постоянного значения, либо касается только числовых данных, имеющихся в значении. Такую операцию выполняет функция `updateWhere`: в качестве аргумента ей передается список фильтров для указания тех элементов базы данных, которые будут затронуты обновлениями, а также список производимых на месте обновлений. Для каждого обновления указываются имя поля, один из операторов: `(=.)` — для присвоения, `(+=.)`, `(-=.)`, `(*=.)` или `(/=.)` — для числовых операций — и постоянное значение, являющееся аргументом обновления.

Давайте, к примеру, проведем 10-процентную скидку. Но делать такую большую скидку на все товары мы не собираемся, поскольку в случае очень дорогих товаров скидка получится слишком большой. Решение заключается в предоставлении только 3-процентной скидки на все товары с ценой, превышающей 10 000. В следующем коде значение 10 000 служит разделительной линией, показывающей, какая скидка должна применяться, 3 % или 10 %:

```
discount = do
  updateWhere [ ProductPrice <=. 10000 ] [ ProductPrice *=. 0.9 ]
  updateWhere [ ProductPrice >. 10000 ] [ ProductPrice *=. 0.97 ]
```

Как и в случае с запросами, библиотека Persistent недостаточно эффективна для представления ряда условий обновления в одном выражении. Поэтому приходится либо прибегать к нескольким запросам и обновлениям, либо использовать более эффективный язык, подобный языку библиотеки Esqueleto. Во втором случае нужно заменить вызовы `select` и `from` вызовом `update`. Напомним, что этой функции в качестве аргумента передается еще одна функция, чей тип проинформирует Esqueleto о том, какие сущности будут затронуты изменениями. Однако функция `update` не допускает использования кортежей и объединений, поскольку изменения

должны касаться только одной сущности. Для выполнения запросов внутри блока `update` можно воспользоваться функцией `sub_select`, которая работает примерно так же, как `select`, но позволяет вернуть только один результат.

Когда вместо `select` используется `update`, внутри монады вам разрешается задействовать еще одну дополнительную функцию, `set`. Эта функция описывает выполняемые в базе данных изменения. В качестве первого аргумента ей передается значение для изменения, а затем, как и в случае с Persistent, список обновлений на месте. Давайте, к примеру, предположим, что мы решили прибегнуть к более подходящей политике скидок: применять их только к тем товарам, у которых низкий уровень продаж. Этот низкий уровень продаж определяется через общее количество покупок, меньшее 10. Используя следующий код, можно одновременно и выполнить запрос, и внести изменения:

```
betterDiscount = update $ \product -> do
  let totalAmount = sub_select $
                from $ \purchase -> do
                where_ $ product ^. ProductId ==. purchase ^. PurchaseProduct
                groupBy (purchase ^. PurchaseProduct)
                return $ sum_ (purchase ^. PurchaseAmount)
  where_ $ isNothing totalAmount ||. totalAmount <. just (val 10)
  set product [ ProductPrice *=. val 0.9 ]
```

И тут мы подошли к самому узкому месту любой базы данных. Со временем может понадобиться удалить некоторую сохраненную информацию либо для освобождения места, либо по соображениям производительности. Сделать это весьма несложно, если известен способ уникальной идентификации удаляемого из базы данных значения по его идентификатору или по набору полей с ограничениями уникальности. В первом случае можно воспользоваться функцией `delete`, а для второго предоставляется функция `deleteBy`.

Однако бывают случаи, когда нужно удалить множество значений, удовлетворяющих определенным условиям. Использование упомянутых функций будет означать получение идентификаторов значений, возможно, через функцию `selectKeysList`, а затем проецирование функции `delete` на все эти идентификаторы. Однако с учетом всего предыдущего материала можно ожидать, что существует функция, позволяющая решить эту задачу без необходимости гонять туда-сюда данные между базой данных и приложением. Такая функция есть, и называется она `deleteWhere` (по аналогии с `updateWhere`).

Например, для очистки базы данных может потребоваться удаление из базы данных информации обо всех товарах, которых больше нет в наличии. В Persistent соответствующий вызов мог бы выглядеть так:

```
cleanProductStock = deleteWhere [ ProductInStock ==. 0 ]
```

Однако проанализировав ситуацию, можно понять, что этот код при всей своей правильности слишком опасен для базы данных. Если клиент приобрел один из товаров, которого больше нет в наличии, эта строка стала бы неправильно ссылаться на идентификатор товара, поскольку товара больше нет в базе данных. Возможным способом исправления ситуации является построение более приемлемого запроса, проверяющего, что сущность `Purchase` с этим товаром отсутствует. Но

тут опять возникает ситуация, когда сама по себе библиотека Persistent не позволяет выразить этот код и приходится обращаться к библиотеке Esqueleto и ее функции `delete`, которая работает как `select` и `update`, но при этом удаляет полученные в результате значения. Обратите внимание на то, как в следующем коде функция `notExists` выполняет проверку соответствующей покупки:

```
cleanProductStock' = delete $
  from $ \product -> do
    where_ $ product ^. ProductInStock ==. val 0
         &&. (notExists $ from $ \purchase ->
              where_ (purchase ^. PurchaseProduct ==. product ^. ProductId))
```

Как видите, обновление и удаление значений в базе данных при использовании обеих библиотек, Persistent и Esqueleto, особых трудностей не представляет. Главное в процессе обучения — понять, как создавать подходящие ограничения для значений, затрагиваемых этими операциями. Здесь проявляется одно из преимуществ таких высокоуровневых языков программирования, как Haskell: проблемно-ориентированные концепции можно выразить простым способом, сконцентрировав основное внимание на работе приложения.

Выводы

Из данной главы мы узнали, как Haskell обеспечивает взаимодействие с СУБД безопасным для типов образом:

- Мы познакомились с доступными в Haskell библиотеками доступа к базам данных, уделив особое внимание тем из них, которые предоставляют абстракции для работы с несколькими базами данных.
- В частности, мы рассмотрели библиотеки Persistent и Esqueleto, позволяющие получать доступ как к реляционным, так и к нереляционным базам данных, причем делать это безопасным для типов образом, затрудняющим написание неправильных запросов, что вполне возможно при использовании обычных инструкций баз данных.
- Мы научились подключаться к базам данных с использованием и без использования пула соединений.
- Мы поработали с языком задания схемы базы данных и выяснили, что использование этой информации в сочетании с механизмом Template Haskell позволяет создавать многочисленные типы данных и связующий код для взаимодействия с СУБД и переноса базы данных.
- Большое место в главе уделено запросам, выполняемым либо посредством идентификаторов, либо с помощью множества полей с ограничениями уникальности, либо в целом по любому полю. Для тех случаев, когда не хватало средств библиотеки Persistent (как в случаях с объединениями), мы задействовали средства библиотеки Esqueleto.
- И наконец, мы узнали, как с помощью этих библиотек осуществляются вставка, обновление и удаление данных.

Веб-приложения 12

Все, о чем мы узнали в предыдущих главах, было подготовкой к этому моменту: сейчас мы уже умеем сохранять информацию о клиентах и товарах, кроме того, нам известна пара алгоритмов анализа данных для интересующих нас образцов. Теперь настала пора приступить к созданию интерфейса для клиентов, позволяющего им совершать покупки: эти данные будут служить входной информацией для алгоритма K-средних и априорного алгоритма.

Наиболее подходящим для организации взаимодействия и самым востребованным способом создания интерфейса для такого приложения является написание его на базе веб-технологий. Клиентам останется только ввести веб-адрес в браузере, получить перечень товаров, поместить нужные товары в виртуальную корзину и в завершение оплатить покупку. Поэтому основное внимание в данной главе уделено созданию веб-приложения с использованием языка Haskell.

Разработка такого приложения разбивается на две части. В первой части мы займемся внутренним интерфейсом: сервером, который будет слушать HTTP-запросы и возвращать информацию о клиентах, товарах и покупках, а также обновлять эту информацию. Для этого нам снова понадобятся многие из навыков, которые были получены при изучении предыдущих глав, а также новые инструменты: Scotty, минималистская среда веб-разработки для Haskell, и `digestive-functors`, библиотека для работы с формами с применением соответствующего стиля.

Затем мы переключим свое внимание на внешний интерфейс, то есть на код, который в конечном счете будет выполняться в браузерах ваших клиентов, осуществляя корректные запросы к внутреннему интерфейсу. Обычно эта часть пишется на языке JavaScript, но как вы увидите далее, можно также использовать язык Haskell. Это приведет нас прямиком к вопросам взаимодействия с другими языками и управления графическим приложением с сохранением состояния.

Веб-экосистема Haskell

В наши дни веб-приложения становятся стандартным средством взаимодействия с пользователями. У такого подхода есть множество преимуществ: не нужно создавать разные исполняемые программы для каждой системы, с которой клиент может обратиться к вашему приложению (хотя вопросы функциональной совместимости между различными браузерами решать все-таки придется), не нужно заставлять пользователей загружать или выполнять какой-либо двоичный код, поскольку все прекрасно работает в браузере.

Если заглянуть в Hackage, можно увидеть постоянно увеличивающееся количество пакетов, предназначенных для работы с веб-приложениями. Диапазон таких пакетов распространяется от весьма традиционных решений с использованием образцов, подобных средам разработки в других языках, до экспериментальных механизмов компоновки приложений — в этой последней группе большую популярность набирают библиотеки функционального реактивного программирования (Functional Reactive Programming, FRP).

Учитывая столь широкие возможности, давайте познакомимся с наиболее важными библиотеками, которые можно использовать для разработки веб-приложений, как это было сделано при рассмотрении вопроса доступа к системам управления базами данных. Разумеется, в данной главе невозможно охватить все, что имеет отношение к веб-приложениям в Haskell. Поэтому основное внимание будет уделено лишь нескольким библиотекам, включая Scotty, `blaze-html`, Hamlet, `digestive-functors` и Fay. Вкратце мы упомянем также несколько других библиотек.

Среды веб-разработки

Среда веб-разработки (web framework) — это библиотека или набор библиотек, используемых при разработке веб-приложений и охватывающих все возможные технологические требования. К этим требованиям обычно относятся система маршрутизации (для связывания URL-адреса с конкретной частью кода, выполняемого при обслуживании этого URL-адреса), система шаблонизации (для генерирования окончательного HTML-вывода), несколько разновидностей кэша, доступ к базе данных и аутентификация через различные протоколы.

В других языках среда веб-разработки обычно представлена единственной библиотекой, объединяющей все инструменты в одном месте. Однако Haskell-разработчики обычно стремятся к большей модульности, поэтому наиболее распространенным образцом в Hackage является множество пакетов, разработанных вместе и имеющих общую часть имени (например, все пакеты, чьи имена начинаются с `yesod`), но не требующих обязательного совместного использования для реализации всей своей функциональности. Это означает, что вы можете создать свое веб-приложение, просто собирая различные части разных проектов. Хотя наличие единой среды разработки предполагает наличие наиболее удобных условий работы, применение лишь части пакетов не означает, что в результате может получиться какой-то совершенно запутанный код. И в самом деле, в магазине машин времени, представляемом в данной главе, мы будем использовать библиотеки маршрутизации, шаблонизации и обработки форм из разных проектов.

Одной из самых старых сред разработки является *Happstack*. Наиболее известной в Happstack является система управления контентом `clckwrks`. Основной пакет `happstack-server` предоставляет доступ практически ко всей информации о HTTP-запросах и HTTP-ответах, включая сжатие, cookie-файлы или аутентификацию. Для всего этого пользователь Happstack обычно объявляет в качестве зависимости модуль `happstack-foundation`, добавляющий множество пакетов, хорошо работающих вместе в среде Happstack. В частности, данные обрабатываются посредством пакета `acid-state`, для маршрутизации используется пакет `web-routes`, а обработка

форм осуществляется с помощью пакета `reform`. Какого-либо предпочтительного механизма шаблонизации не существует, но в документации перечисляются библиотеки HSP, Heist и Hamlet. Пакеты `web-routes` и `reform` построены на одинаковых принципах, которые мы рассмотрим применительно к среде Scotty. Идеи, заложенные в HSP (Haskell Server Pages), очень напоминают идеи других языков шаблонизации, например JSP (Java Server Pages): вы пишете HTML-код, который нужно сгенерировать, декорируя его небольшими фрагментами Haskell-кода.

Большинство других сред разработки не имеют сервера: они надстраиваются над обобщенным интерфейсом веб-приложений (Web Application Interface, WAI). В принципе, для запуска вашего кода может применяться любой сервер, распознающий данный интерфейс (точно так же, как для запуска Java-приложений можно использовать множество сервлетов). Однако на практике в производстве используется только Warp-сервер. Поэтому большинство других сред разработки предоставляют непосредственный доступ к запущенному веб-приложению с помощью Warp.

Snap — это имя одной из библиотек, построенных на WAI. В конструкции Snap присутствуют две основные особенности. Первая заключается в том, что предлагаемый этой библиотекой инструментарий нацеливает разработчика не на создание цельного приложения, а на написание модульных компонентов (так называемых «снеплетов»), которые впоследствии образуют некую композицию. Вторая особенность заключается в использовании в качестве библиотеки шаблонизации такого инструмента, как Heist, который, запрашивая информацию у других прикладных уровней, использует обычный язык HTML в сочетании со специальными тегами.

Третьим доступным вариантом является библиотека *Yesod*. Основной целью разработчиков Yesod было достижение максимально возможной безопасности в отношении типов: в качестве части этой среды разработки используется библиотека Persistent, с которой мы познакомились в предыдущей главе. Если помните, схемы в Persisent описываются на компактном языке, синтаксический разбор которого выполняется с использованием квазицитирования и механизма Template Haskell. В Yesod подобные идеи перенесены и на другие части веб-приложения: с помощью тех же блоков реализуются маршрутизация и шаблонизация. Для шаблонизации были разработаны различные подчиненные библиотеки, каждая из которых предназначается для обслуживания общепринятого языка вывода информации: Hamlet — для HTML-документов (которые будут рассматриваться в данной главе), Cassius и Lucius — для таблиц стилей, Julius — для JavaScript-кода.

Основное внимание в данной главе уделено минималистской среде веб-разработки *Scotty*. Ее главная особенность: простое описание маршрутов и чрезвычайно простой способ подключения библиотек шаблонов (далее мы узнаем, как использовать с этой средой библиотеки `blaze-html` и Hamlet). Благодаря минималистской природе эта среда не требует для работы с ней знания множества различных концепций: тем самым создается отличная стартовая позиция для того, чтобы приступить к изучению веб-библиотек для Haskell с последующим переходом к более сложным библиотекам, когда возникнет необходимость в дополнительной функциональности.

В частности, одной из сторон, где Scotty пасует, является обработка информации, передаваемой пользователями через формы. Закрыть эту брешь способна весьма популярная библиотека `digestive-functors` (в данном случае она даже предпочти-

тельнее Snap), с которой мы познакомимся в этой главе. Вспомните представленную в главе 10 библиотеку `aeson`, предназначенную для обработки формата JSON. Библиотека `digestive-functors` также опирается на аппликативный стиль при определении форм и проверке приемлемости данных.

Для упомянутых пакетов существует большой объем документации, причем даже в форме учебников и книг, при желании вы можете изучить эту документацию и составить собственное представление о том, чем эти пакеты отличаются друг от друга. Обновляемый перечень сред веб-разработки для Haskell (включая как упомянутые, так и многие другие), а также ссылки на соответствующие веб-сайты можно найти по адресу http://www.haskell.org/haskellwiki/Web/Frameworks.

Компиляция в JavaScript-код

Рассматриваемые до сих пор среды касались разработки внутреннего интерфейса приложения (известного также как «сторона сервера»). Они позволяют генерировать HTML-страницы, которые становятся частью того, что в конечном счете увидит пользователь. К сожалению, на языке HTML невозможно описать все варианты поведения, ожидаемого от динамической веб-страницы, поэтому разработчикам приходится прибегать к языку JavaScript. Каждый современный браузер включает в себя JavaScript-интерпретатор, который раскрывает перед пользователем широкий спектр функциональных возможностей и позволяет создавать очень сложные приложения (достаточно вспомнить разнообразные веб-страницы с интерактивными картами).

Проблема состоит в нежелании многих разработчиков полностью взять на себя программирование внешнего интерфейса (или стороны клиента) на языке JavaScript. Сам по себе язык JavaScript объединяет в себе императивный язык, язык, ориентированный на прототипы, и даже поддерживает некоторые особенности функциональной парадигмы, но его синтаксис и, что более важно, его концепции существенно отличаются от таковых для языка Haskell. Использование одного и того же языка как для внутреннего, так и для внешнего интерфейса упрощает разработку обоих интерфейсов и повышает возможности использования в них общего кода.

Однако возможность заставить браузеры поддерживать новый язык — не более чем иллюзия: все они ориентированы на JavaScript. Поэтому многие компиляторы кроме генерирования двоичного кода поддерживают также режим генерирования JavaScript-кода. Тогда предпочитаемый вами язык программирования можно использовать для написания динамических веб-страниц, сохранив при этом ту поддержку, которую JavaScript имеет в браузерах. К тому же в наши дни быстродействие и качество JavaScript-реализаций достигли весьма впечатляющего уровня.

Haskell-сообщество много сделало в данном направлении, и в Hackage появилось несколько пакетов, позволяющих компилировать Haskell-код в JavaScript-код. Для вывода этого альтернативного представления сам компилятор GHC был модифицирован с помощью пакетов *GHCJS* и *Haste*. Разница между ними в том, что первый задумывался как средство беспроблемного выполнения любого приложения, которое может быть скомпилировано компилятором GHC в среде веб-разработки, а второй был призван найти баланс между полной совместимостью с GHC и более качественным взаимодействием с другим JavaScript-кодом.

Третьим пакетом является Fay, который не поддерживает (пока) язык Haskell в целом (главной его особенностью является отсутствие классов типов), зато очень прост в применении, прекрасно взаимодействует с другим JavaScript-кодом и хорошо подходит для обмена данными с веб-серверами. Он станет нашим главным объектом изучения при описании дизайна внешнего интерфейса веб-приложения.

Как и в случае со средами веб-разработки, горизонты компиляции в JavaScript-код быстро расширяются, и поддержку этим методам предлагает множество других пакетов. Их актуальный перечень с указателями на документацию можно найти по адресу http://www.haskell.org/haskellwiki/The_JavaScript_Problem.

Структура RESTful

Перед тем как приступить к программированию, давайте познакомимся с несколькими общими образцами разработки приложений, основанными на принципах передачи репрезентативного состояния (Representational State Transfer, REST). Многие из этих концепций напрямую кодируются с помощью комбинаторов, предоставляемых библиотекой Scotty, и могут встречаться во многих других средах веб-разработки как внутри мира Haskell, так и за его пределами.

Основной замысел состоит в том, что веб-приложение предоставляет доступ к множеству *ресурсов* (resources), которые могут запрашиваться или обновляться. Примером ресурсов в магазине машин времени может служить информация о клиенте или товаре. Доступ к каждому из этих ресурсов осуществляется посредством уникального идентификатора, которым в веб-приложении является URL-адрес. Например, адрес /clients может идентифицировать список всех клиентов, а /product/3 — информацию о товаре, идентификатором которого в базе данных является число 3.

Учитывая, что URL-адреса играют основную роль в уведомлении о том, какие данные должны запрашиваться или обрабатываться по запросу, выбранная среда веб-разработки должна обеспечивать эффективную поддержку заданных образцов URL-адресов и правильно указывать на исполняемый код. Каждый такой образец называется *маршрутом* (route). Например, в Scotty можно дать описание маршрута для товаров в виде /product/:id. Это описание говорит о том, что если обнаружится любой URL-адрес, начинающийся с префикса /product/, то будет выполнен некий код с аргументом `id`, привязанным к остальной части URL-адреса.

Для любого ресурса у приложения могут быть несколько *методов запроса* (request methods), каждый из которых призван выполнять конкретное действие над ресурсом. Для веб-приложений, являющихся основной темой данной главы, наиболее востребованными методами запроса являются `GET`, `POST`, `PUT` и `DELETE`. Метод `GET` извлекает конкретный объект или списки элементов коллекции, метод `POST` служит для создания нового элемента внутри указанного ресурса, метод `PUT` заменяет информацию всего ресурса новой информацией, поступившей с запросом, метод `DELETE` удаляет ресурс из системы. Архитектурный стиль REST налагает ряд ограничений на то, каким образом будет вести себя каждый метод (например, `GET` вообще не должен изменять ресурс): об этом можно прочитать множество книг, поэтому мы не будем здесь вдаваться в подробности.

И наконец, для каждой поддерживаемой комбинации идентификатора ресурса и метода запроса приложение должно дать ответ с использованием стандартных форматов. Когда механизм REST работает через протокол HTTP, подразумевается использование HTTP-кодов состояния (например, возвращение ошибки 404, если какой-то ресурс не найден) и кодирование запрошенной информации в формате HTML, XML или JSON. Кроме того, потребитель может указать список поддерживаемых типов ответов, а сервер должен попытаться удовлетворить это требование.

Разработка внутреннего интерфейса в среде Scotty

В данном разделе нам предстоит разработать внутренний интерфейс для магазина машин времени. Чтобы уложиться в отведенный для книги объем, основное внимание будет уделено обслуживанию запросов и обновлению информации о товарах, а поддержка клиентов и покупок останется для упражнения, призванного собрать воедино большую часть полученных к настоящему моменту знаний.

Простой каркас приложения

Сначала нужно задать проект с выполняемой строфой, которая запустит сервер, и добавить модуль `scotty` в качестве зависимости. В первых четырех главах все это мы уже делали. Давайте теперь повторим пройденный материал при выполнении упражнения 12.1, чтобы заложить основу нашего магазина. Вспомним, что решения упражнений включены в загружаемый файл, находящийся на странице каталога на сайте Apress.com.

УПРАЖНЕНИЕ 12.1. СОЗДАНИЕ ПРОЕКТА

Создайте новый Cabal-пакет либо с использованием утилиты командной строки cabal, либо с помощью EclipseFP и назовите его chapter12. Этот пакет должен содержать строфу, генерирующую исполняемый файл time-machine-store. В качестве зависимостей исполняемого файла укажите модули scotty и text.

Теперь все готово для того, чтобы приступить к созданию вашего веб-приложения. В следующем фрагменте показано очень простое приложение, которое лишь отвечает на URL-адрес /about:

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Monoid (mconcat)
import Web.Scotty

main :: IO ()
main = scotty 3000 $
  get "/about" $
    html $ mconcat [ "<html><body>"
                   , "  <h1>Hello Beginning Haskell!</h1>"
                   , "</body></html>" ]
```

Точка входа в приложение, функция `main`, запустит наше Scotty-приложение через порт 3000. Кроме номера порта функции `scotty` требуется описание маршрутов, которые передаются в блоке `do`. В данный момент это только маршрут `/about`, который может быть доступен через `GET` запрос. Этот маршрут определяется с помощью имени метода, записанного строчными буквами (`get`), URL-адреса образца и, наконец, *обработчика* (handler), выполняемого по запросу кода, в который могут быть включены действия по вводу-выводу (`IO`).

ПРИМЕЧАНИЕ

Маршруты определяются посредством значения типа Text. В показанном коде для записи маршрутов в виде строковых литералов используется расширение OverloadedStrings.

Результат запроса в Scotty может быть определен просто как значение типа `Text`, которое представляет собой данные, отправляемые по линии связи. Однако если возвращаемые данные должны обрабатываться как HTML-код, принято определять значение заголовков ответов. Чтобы не делать это вручную, Scotty предоставляет вспомогательную функцию `html`, которая устанавливает для этих заголовков соответствующие значения, а затем возвращает контент.

Проверить работоспособность веб-приложения можно, создав пакет и вызвав `./dist/build/time-machine-store/time-machine-store`. В качестве альтернативы, если используется среда EclipseFP, исполняемый файл можно запустить непосредственно из IDE. В любом из двух случаев затем потребуется указать браузеру адрес http://localhost:3000/about, чтобы запустить соответствующий обработчик и вывести результаты на экран.

Однако если перейти на другой URL-адрес на том же самом сервере, будет получен прототип страницы ошибки. В качестве небольшого усовершенствования этого простого приложения давайте создадим для него нестандартную страницу «not found» («не найдена»), добавив к блоку `do` новый маршрут:

```
import Network.HTTP.Types

main = scotty 3000 $ do
  get "/about" $ ...

  notFound $ do
    status notFound404
    html "<h1>Not found :(</h1>"
```

Важно отметить, что в случае, когда URL-адрес не найден, состоянию ответа должно быть присвоено соответствующее число (в данном случае всем известное число 404). Это делается вызовом функции `status`, которой в качестве аргумента передается один из кодов, определяемых в модуле `Network.HTTP.Types` (этот модуль определен в пакете `http-types`, а не в пакете `scotty`, поэтому пакет `http-types` нужно дополнительно указать в качестве зависимости).

ВНИМАНИЕ

Scotty проверяет маршруты в том же порядке, в котором они появляются в коде. Маршрут notFound проверяется с каждым возможным URL-адресом: любой маршрут после него никогда не будет выполнен. Поэтому маршрут notFound в описании вашего приложения всегда нужно записывать последним.

Демонстрация товаров из базы данных

Перейдем к основной задаче этого раздела: демонстрации информации о товарах. Доступ к этим данным будет осуществляться посредством библиотеки Persistent (следовательно, нужно включить в качестве зависимостей пакеты `persistent`, `persistent-template` и `persistent-sqlite`) по той же схеме, что и в главе 11. В приводимых далее примерах подразумевается, что описание схемы базы данных находится в модуле `Chapter12.Database`.

Чтобы база данных была доступна обработчикам, в функцию `main` нужно внести два изменения. Прежде всего, нужно убедиться в том, что схема, описанная посредством библиотеки Persistent, совпадает с реальной схемой базы данных. Как вы уже знаете, это достигается путем переноса, который должен быть задан в `main` в первую очередь.

Каждый обработчик, которому необходим доступ к базе данных, может, в принципе, установить новое соединение с помощью функции `runSqlite` и выполнять любые действия по вводу-выводу, пользуясь этим соединением. Однако с точки зрения производительности рациональнее создать управляемый библиотекой Persistent пул соединений для совместного использования всеми возможными запросами. Для этого вызов `scotty` нужно заключить в оболочку `withSqlitePool`. Окончательная структура функции `main` теперь должна приобрести следующий вид:

```
import Data.Monoid (mconcat)
import qualified Database.Persist.Sqlite as Db
import Network.HTTP.Types
import Web.Scotty

import Chapter12.Database

main :: IO ()
main = do
  Db.runSqlite "example.db" $ Db.runMigration migrateAll
  Db.withSqlitePool "example.db" 10 $ \pool ->
    scotty 3000 $ do
      get "/about" $ ...
      notFound $ ...
```

Каждый товар будет доступен через URL-адрес вида /product/*n*, где *n* — уникальный идентификатор, назначенный этому товару системой управления базой данных. Маршрут, охватывающий подобный случай, является не константой, как было с адресом /about, а *параметром* с соответствующим именем. Для его кодирования в образце маршрута указывается двоеточие, за которым следует имя параметра. В данном случае параметр должен быть назван `productId`, что требует создания маршрута `/product/:productId`. Затем значение каждого параметра может быть получено с помощью функции `param`. Этой функции нужно сообщить, к параметру какого типа осуществляется доступ: в показанном далее коде это делается посредством локальной сигнатуры типа, следовательно, к исходному файлу нужно добавить расширение `ScopedTypeVariables`.

```
{-# LANGUAGE OverloadedStrings, ScopedTypeVariables, RecordWildCards #-}

import Control.Monad.IO.Class (liftIO)
import Data.Text.Lazy (pack)
```

```
main = do
  ...
  Db.withSqlitePool "example.db" 10 $ \pool ->  scotty 3000 $ do
      ...
      get "/product/:productId" $ do
        (productId :: Integer) <- param "productId"
        product <- liftIO $ flip Db.runSqlPersistMPool pool $
                  Db.get $ Db.Key (Db.PersistInt64 $ fromIntegral productId)
        case product of
          Just (Product { .. }) -> html $ mconcat [ "<html><body>"
                                                  , "  <h1>"
                                                  , pack productName
                                                  , "  </h1>"
                                                  , "  <p>"
                                                  , pack productDescription
                                                  , "  </p>"
                                                  , "</html></body>" ]
          Nothing               -> status notFound404
                                   html "<h1>Not found :(</h1>"
```

Весь остальной код реализует простое обращение к базе данных тем способом, который был описан в главе 11, для доступа к значению по его идентификатору и сборки возвращаемого HTML-значения. По поводу предыдущего блока кода можно отметить только четыре вещи:

- Самое важное здесь то, что вам необходимо добавить вызов функции `liftIO`, чтобы выполнить действие `IO` внутри монады `Scotty`. Роль у функции `liftIO` такая же, как и у функции `lift` в стеках преобразователей монад, но ограничена данными специфическими действиями. Из-за использования функции `liftIO` вам необходимо добавить пакет `transformers` в качестве зависимости.
- Чтобы упростить привязку переменных из записи `Product`, в коде используются символы подстановки.
- В `param` код указывает как имя получаемого параметра, так и ожидаемый тип результата. Интересно, что Scotty не рухнет, даже если значение не прошло санацию и не соответствует числу, выполнение кода просто продолжится на следующем правиле вашего веб-приложения.
- Если идентификатор товара не найден, для состояния кода ответа будет установлено значение 404, как и для маршрутов «not found» в предыдущем разделе.

Показанному коду присущи краткость и простота, исключая ту часть, в которой происходит сборка получаемой на выходе HTML-страницы, — сборка осуществляется простой конкатенацией строк. Теперь, когда мы уже знаем, как много разнообразных проверок компилятор Haskell может для нас провести, можно задать вполне резонный вопрос: а есть ли какое-либо средство написания HTML-кода, с помощью которого можно было бы убедиться в закрытии всех тегов, воспользоваться системой отступов для задания структуры документа и т. д.? Ответ положительный: далее мы познакомимся с двумя разными библиотеками, которые берут на себя решение разных частей этой задачи.

Первая возможность заключается в выборе пакета `blaze-html`. Лежащая в его основе идея состоит в том, что каждый HTML-элемент представляется функцией,

последним аргументом которой является монадическое значение, содержащее внутри все необходимые теги. Таким образом, в своем Haskell-коде для имитации структуры документа вы можете написать вложенные блоки `do`. Другим основным комбинатором является оператор `(!)`, позволяющий включать HTML-атрибуты для каждого элемента.

Поскольку существует несколько версий стандарта (X)HTML, библиотека `blaze-html` предлагает различные пакеты, определяя для каждого допустимые теги и атрибуты. В показанном далее коде вместо конкатенации строк, как было в предыдущем обработчике товаров, используются средства библиотеки `blaze-html`. Обратите внимание на то, что для преобразования строки в значение, распознаваемое библиотекой `blaze-html`, необходимо вызвать функцию `toHtml`, а для преобразования из значения, понятного библиотеке `blaze-html`, в значение типа `Text`, которое может быть возвращено в Scotty, — функцию `renderHtml`:

```
import qualified Text.Blaze.Html5 as H
import qualified Text.Blaze.Html5.Attributes as A
import Text.Blaze.Html.Renderer.Text (renderHtml)

main :: IO ()
... -- предыдущий код опущен
      get "/product/:productId" $ do
        ...
        case product of
          Just (Product { .. }) -> html $ renderHtml $
            H.html $ do
              H.head $
                H.title "Time Machine Store"
              H.body $ do
                H.h1 $ H.toHtml productName
                H.p  H.! A.id "descr" $ H.toHtml productDescription
          Nothing               -> status notFound404
```

Вторая возможность внедрения HTML-значений безопасным для типов способом заключается в использовании системы шаблонизации, которая называется Hamlet. Отличительной особенностью пакета `hamlet` является то, что для предоставления синтаксиса, близкого к HTML, в нем используются не Haskell-функции, а механизм Template Haskell. Давайте перепишем блок информации о товаре для этого случая:

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}

import Text.Hamlet

main :: IO ()
... -- предыдущий код опущен
      get "/product/:productId" $ do
        ...
        case product of
          Just p -> html $ renderHtml [shamlet|
            <html>
              <head>
                <title>Time Machine Store
              <body>
                <h1>#{productName p}
```

```
            <p id=descr>#{productDescription p}
        |]
    Nothing               -> status notFound404
```

В Hamlet-шаблонах открывающие теги используются точно так же, как в HTML, но вложенная структура указывается отступами, являющимися синтаксическим элементом, поэтому закрывающие теги, как видно из предыдущего примера, не нужны. В любой момент с помощью указателя блока `#{ }` можно «совершить побег» в мир Haskell: результат вычисления выражения внутри фигурных скобок будет безопасно выведен в этой точке.

В нашем простом веб-приложении функция `shamlet` служит для синтаксического разбора HTML-блока. Это простая функция, всего лишь возвращающая HTML-значение, которое может быть визуализировано с помощью функции `renderHtml` (точно так же, как это было показано для `blaze-html`). Однако Hamlet предлагает другие квазицитаты, обладающие намного большей эффективностью:

- Функция `hamlet` позволяет использовать *безопасные для типов URL-адреса* (type-safe URLs). Замысел заключается в определении всех возможных маршрутов в вашем веб-приложении с помощью алгебраического типа данных. Каждый из конструкторов будет описывать образец URL-адреса наряду с множеством параметров. Затем вместо построения HTML-ссылок вручную вы просто указываете один из этих конструкторов всякий раз, когда нужна ссылка. В пакете `yesod-routes` предоставляются функции из Template Haskell, которые генерируют весь необходимый для этого шаблонный код.
- Функция `ihamlet` добавляет ко всему этому поддержку *интернационализации* (internationalization): вы можете создать для своих шаблонов общую структуру документа, а затем включить разные сообщения для всех языков, поддерживаемых вашим приложением.

И наконец, для каждой из этих функций Hamlet предоставляет соответствующую функцию с окончанием `File` (`shamletFile`, `hamletFile`, `ihamletFile`), которая считывает шаблон для внешнего файла. Таким образом вы можете отделить представление ваших данных от программной логики сервера.

Hamlet включает не только теги и скобки для переменных, но и управляющие структуры для простых, часто встречающихся при создании HTML-страниц случаев. Например, в показанном далее коде для перебора всех товаров в базе данных и записи строки для каждого из них в таблицу используется структура `$forall`:

```
main :: IO ()
... -- предыдущий код опущен
      get "/products" $ do
        (products :: [Db.Entity Product]) <-
          liftIO $ flip Db.runSqlPersistMPool pool $ Db.selectList [] []
        html $ renderHtml [shamlet|
          <html>
            <body>
            <h1>Products
            <table>
              <tr>
                <th>Name
                <th>Description
              $forall Db.Entity _ p <- products
```

```
            <tr>
              <td>#{productName p}
              <td>#{productDescription p}
      |]
```

Полностью синтаксис Hamlet описывается в документации по этому пакету.

В качестве последнего штриха может понадобиться предоставить не только HTML-вывод, но и вывод в формате JSON для тех потребителей вашего веб-приложения, кто является не конечным пользователям, а другим разработчиком, создающим средства взаимодействия с вашим магазином. А теперь о приятном: если вы используете библиотеку `aeson`, Scotty предоставляет весьма полезную функцию `json`, которая берет на себя преобразование значения в соответствующий текстовый формат и устанавливает нужные обработчики. Например, добавление нового маршрута для получения в ответ информации о товаре в формате JSON не представляет никаких сложностей:

```
main :: IO ()
... -- предыдущий код опущен
      get "/json/product/:productId" $ do
        (productId :: Integer) <- param "productId"
        (product :: Maybe Product) <-
          liftIO $ flip Db.runSqlPersistMPool pool $
            Db.get $ Db.Key (Db.PersistInt64 $ fromIntegral productId)
        case product of
          Just p  -> json p
          Nothing -> status notFound404
```

В упражнении 12.2 от вас потребуется реализовать ту же функциональность магазина машин времени, что и в случае товаров, но на этот раз для клиентов.

УПРАЖНЕНИЕ 12.2. ЗАПРОСЫ О КЛИЕНТАХ

Включите новые маршруты /clients, /client/:clientId и /json/client/:clientId, возвращающие информацию обо всех клиентах и о конкретном клиенте в форматах HTML и JSON. Для этого с целью генерирования HTML-вывода воспользуйтесь любой из представленных ранее систем шаблонизации.

Вставка информации о новых товарах с использованием форм

Пока у нас есть только инструменты для написания маршрутов, использующие для ответа информацию из URL-адреса. Однако если предполагается, что пользователю нужно передать в интерактивном режиме некие данные, интерфейс должен предоставить форму, а веб-приложение должно получить введенную информацию из этой формы. То, как решить данную задачу наиболее элегантным и приемлемым образом, и является основной темой этого раздела.

Прежде чем продолжить, следует заметить, что ранее использованная нами для получения значений фрагментов URL-адреса функция `param` позволяет также получать значения полей формы. В этом случае вместо указания имени параметра в качестве аргумента `param` нужно задать идентификатор формы и идентификатор поля, отделив их друг от друга точкой. Проблема аргумента `param` состоит в том, что синтаксический разбор и визуализация форм постоянно оказываются «ябло-

ком раздора» для различных обработчиков: для ее решения нужно стремиться к централизованному описанию в веб-приложении каждой формы, которая может многократно использоваться разными обработчиками (например, форма для товаров может применяться как для создания новой товарной позиции, так и для обновления информации об уже существующей).

Пакет `digestive-functors`, который используется в нашем веб-приложении для обработки форм, разработал Джаспер Ван дер Югт (Jasper Van der Jeugt). У этой библиотеки есть интересная особенность: в ней применен аппликативный стиль, который придает формам тот же колорит, что и при обработке формата JSON с помощью уже рассмотренной в данной книге библиотеки `aeson`. Кроме того, пакет `digestive-functors` отделяет описание данных и верификацию при создании значения от способа визуализации, что обеспечивает сугубо модульный поход к обработке форм.

Каждая создаваемая с помощью этой библиотеки форма должна построить значение определенного типа, обычно относящегося к одному из типов, используемых в схеме базы данных. Кроме того, тип `Form` (описывающий каждую форму на странице), нуждается в переменной типа, задающей формат представления ошибок. Для каждого аргумента в конструкторе этого типа нужно указать, какое поле в форме будет обрабатывать его значение, один из основных видов полей, который может иметься в HTML-форме, исходное значение для пустой формы и любую дополнительную верификацию, если она требуется. К примеру, далее представлена форма для нашего типа `Country`, для которой требуется строковое и булево значения и которая показывает ошибки в виде строк. Обратите внимание на то, как с помощью функции `string` или `bool` указывается поле, а также на то, что исходное значение заключается в оболочку `Maybe`:

```
import Control.Applicative ((<$>), (<*>))
import Text.Digestive

countryForm :: Monad m => Form String m Country
countryForm = Country <$> "name" .: string Nothing
                      <*> "send" .: bool (Just True)
```

ВНИМАНИЕ

Как в aeson, так и в digestive-functors есть функция (.:). Обратите внимание, что если в одном и том же исходном файле используются оба этих модуля, хотя бы одну из функций необходимо уточнить.

Для последующей верификации нужно заключить одну из таких простых спецификаций в оболочку `validate` или `check`. Разница между ними состоит в том, что `validate` может также проводить синтаксический разбор значения в новое значение (возможно, даже другого типа), а `check` является всего лишь булевым предикатом, оставляющим значение в неизменном виде. Обе функции нуждаются в дополнительном вводе сообщения об ошибке, предоставляемом пользователю, когда значение не отвечает требованиям. Следующий код реализует форму для значений `Product`: в нем используется функция `validate` для преобразования из строки в числа и проверки того факта, что запасы по указанной товарной позиции превышают нуль или равны нулю:

```
import Text.Digestive.Util

productForm :: Monad m => Form String m Product
productForm = Product <$> "name"        .: string Nothing
                      <*> "description" .: string Nothing
                      <*> "price"       .: validate isANumber (string Nothing)
                      <*> "inStock"     .: check "Must be >= 0" (>= 0)
                                           (validate isANumber (string Nothing))

isANumber :: (Num a, Read a) => String -> Result String a
isANumber = maybe (Error "Not a number") Success . readMaybe
```

Закончив описание формы, нужно определить, как она будет *показана* пользователю. Создаваемый код должен получать *представление* (view) формы (то есть определение формы плюс значения для каждого из полей) и возвращать HTML-вывод. Доступные в Hackage пакеты для интеграции форм с практически любой системой шаблонизации уже были представлены в начале главы. В данном разделе мы для примера используем библиотеку `blaze-html`, поэтому в качестве зависимости нужно добавить пакет `digestive-functors-blaze`.

В модуль `Text.Digestive.Blaze.Html5` включены функции для всех типов ввода, поддерживаемых стандартом HTML. Каждая из функций в качестве аргументов получает как минимум имя поля, на которое она ссылается, и представление, которое она обрабатывает. Кроме элементов управления вводом этот пакет может также генерировать код для меток ввода и ошибок, касающихся конкретного поля. В качестве примера далее приводится возможный вариант демонстрации в форме информации по товару. Обратите внимание на включение функции `inputSubmit` для создания кнопки Submit, предназначенной для отправки данных формы.

```
import qualified Text.Blaze.Html5 as H  -- из предыдущих примеров
import Text.Digestive.Blaze.Html5

productView :: View H.Html -> H.Html
productView view = do
  form view "/new-product" $ do
    label     "name"    view "Name:"
    inputText "name"    view
    H.br
    inputTextArea Nothing Nothing "description" view
    H.br
    label     "price"   view "Price:"
    inputText "price"   view
    errorList "price"   view
    label     "inStock" view "# in Stock:"
    inputText "inStock" view
    errorList "inStock" view
    H.br
    inputSubmit "Submit"
```

Теперь давайте создадим форму, которая будет показана, когда пользователь веб-приложения захочет добавить новый товар. Поскольку о новом товаре никакой информации нет, нужно показать форму с исходными данными, определенными в `productForm`. Это делается путем создания пустого представления с помощью функ-

ции getForm. Форму с таким представлением можно интегрировать в более крупную веб-страницу, вызвав, как показано в следующем коде, функцию productView:

```
main :: IO ()
... -- предыдущий код опущен
      get "/new-product" $ do
        view <- getForm "product" productForm
        let view' = fmap H.toHtml view
        html $ renderHtml $
          H.html $ do
            H.head $ H.title "Time Machine Store"
            H.body $ productView view'
```

ПРИМЕЧАНИЕ

Функция getForm возвращает значение View String (поскольку определение productForm имеет тип Form String m Product), а функции productView требуется значение View H.Html. От первого к последнему можно перейти путем проецирования H.toHtml на каждое поле, как это сделано в предыдущем примере. Замысел, заложенный в это преобразование, состоит в том, что нам нужно спроецировать ошибки в форме из типа String на HTML-представление.

Когда данные формы отправляются браузером обратно веб-приложению, вместо метода GET запроса используется метод POST. Следовательно, в данном случае нам нужно прочитать значения, отправленные пользователем, с последующей записью информации о новом товаре в базу данных, если значения корректны, или возвращением той же самой формы с сообщениями об ошибках. Ключевым моментом является замена функции getForm функцией postForm, которая вместо исходных данных использует только словарь полей и значений из запроса. Кроме этого изменения, код данного запроса не отличается особой сложностью: если вводимые данные приемлемы (postForm возвращает значение Just), происходит контакт с базой данных и перенаправление на страницу товаров или же, в ином случае, возвращение той же самой формы (о выводе ошибок заботится функция productView).

```
main :: IO ()
... -- предыдущий код опущен
      post "/new-product" $ do
        params' <- params
        (view,product) <- postForm "product" productForm (paramsToEnv params')
        case product of
          Just p -> do
            Db.Key (Db.PersistInt64 newId) <- liftIO $ Db.runSqlPersistMPool (Db.
insert p) pool
            redirect $ mconcat ["/product/", pack $ show newId]
          Nothing -> do
            let view' = fmap H.toHtml view
            html $ renderHtml $
              H.html $ do
                H.head $ H.title "Time Machine Store"
                H.body $ productView view'
```

ПАРАМЕТРЫ СРЕДЫ

В предыдущем коде имеется вызов функции paramsToEnv. Эта функция берет на себя преобразование между кодировкой параметров запроса, принятой в Scotty, и таким представлением

этих параметров, которое способен распознать пакет digestive-functors. Ее определение имеет следующий вид:

```
import Data.Text.Lazy (toStrict)

paramsToEnv :: Monad m => [Param] -> Text.Digestive.Env m
paramsToEnv []           _ = fail "Parameter not found"
paramsToEnv ((k,v):rest) t = if toStrict k == fromPath t
                             then return [TextInput $ toStrict v]
                             else paramsToEnv rest t
```

Тип Text.Digestive.Env m является синонимом типа Path -> m [FormInput], который дает основание функции на получение двух параметров вместо одного. Способ работы функции очень прост: получая значение Path (идентификатор формы и поля), она в проекции params конструирует имя параметра, вызывая функцию fromPath. Затем происходит последовательный перебор элементов проекции, пока не будет найден ключ. Кроме того, код должен позаботиться о преобразовании между ленивыми и строгими значениями типа Text, потому что в этих двух библиотеках (Scotty и digestive-functors) используются их разные версии.

Этот раздел служит неплохим введением в тему создания в Haskell веб-приложений на базе технологии REST, в частности в тему использования среды Scotty. Чтобы закрепить новые понятия, попробуйте выполнить упражнение 12.3, завершив, как там предлагается, создание внутреннего интерфейса приложения. После этого, если вы интересуетесь разработкой веб-приложений в Haskell, можете переходить к углубленному изучению Scotty или любых других сред разработки, упомянутых ранее в этой главе.

УПРАЖНЕНИЕ 12.3. РЕГИСТРАЦИЯ КЛИЕНТОВ

Создайте новый обработчик для регистрации клиентов по маршруту /register. Как и в случае с товарами, нужно добавить обработчики как для методов GET запросов, так и для методов POST.

Подсказка: для создания на HTML-форме раскрывающегося списка с возможными названиями стран может понадобиться прочитать документацию по комбинатору choice из библиотеки digestive-functors.

Разработка внешнего интерфейса с помощью компилятора Fay

Этот раздел посвящен созданию для нашего веб-приложения полноценного внешнего интерфейса, реализующего магазин машин времени с использованием технологий компиляции из Haskell-кода в JavaScript-код. Как и в предыдущем разделе, из-за ограниченного объема книги в круг рассматриваемых вопросов входят только краткое знакомство с компилятором Fay и его использование для демонстрации перечня товаров.

Как уже отмечалось в начале этой главы, компилятор Fay отличается от компилятора GHC. Поэтому вам нужно установить его отдельно и компилировать свои Haskell-файлы, используя его вместо Cabal или самого компилятора GHC. Первый шаг выполняется очень просто. Поскольку компилятор Fay запакован в Hackage, для его установки нужно лишь ввести в командной строке следующую команду:

```
$ cabal install fay fay-base
```

Сам компилятор находится в пакете `fay`, а пакет `fay-base` содержит реализацию модуля Prelude и интерфейса внешних функций для языка. Чтобы работать с представленными в этой главе примерами, вам понадобятся оба пакета.

Давайте попробуем скомпилировать очень простое приложение, которое просто выводит что-нибудь на консоль браузера. Предполагается, что следующий код будет помещен на вашей системе в файл Simple.hs:

```
module Simple where

import Prelude

main :: Fay ()
main = print "Hello from Fay!"
```

Можно заметить, что модуль Prelude не импортируется в Fay-код по умолчанию, поэтому его нужно всегда добавлять явным образом. Кроме того, монадой, идентифицирующей действия для среды, является не `IO`, а `Fay`.

Чтобы скомпилировать код, откройте командную строку и выполните следующую команду:

```
$ fay Simple.hs
```

В результате создается файл Simple.js, содержащий трансляцию этого Haskell-кода в JavaScript-код. Поскольку функция `main` у нас задействована, можно напрямую включать JavaScript-файл в любую HTML-страницу, и его код будет выполнен. Создайте, к примеру, файл Simple.html со следующим контентом:

```
<html>
  <head>
    <title></title>
    <script src="Simple.js"></script>
  </head>
  <body>
  </body>
</html>
```

Если открыть этот HTML-файл в браузере и перейти на консоль JavaScript-кода, которая обычно доступна в меню Инструменты (Tools) браузера, то будет видна строка, содержащая массив символов `"Hello from Fay!"`.

Интерфейс внешних функций

Конечно, если вы используете только чистые функции, а также функции, доступные в Fay через модуль Prelude, очень сложно работать с веб-страницей, в которой выполняется ваш код. Но есть и хорошая новость: вызов JavaScript-кода из Fay осуществляется очень просто, что открывает двери к использованию любой JavaScript-библиотеки внутри Fay-кода. Такую возможность обеспечивает интерфейс внешних функций (Foreign Function Interface, FFI).

Чтобы получить возможность вызова JavaScript-функций, для них нужно создать *упаковщик* (wrapper). Он требуется компилятору для построения кода, который в конечном счете вызовет JavaScript-код. В Fay упаковщик состоит из двух частей:

- Из-за нетипизированной природы JavaScript для каждого FFI-упаковщика нужно предоставить сигнатуру типа. Можно воспользоваться обычными Haskell-типами, и Fay проведет их сериализацию в соответствующие JavaScript-объекты. Здесь нужно запомнить самое важное: если функция производит какой-нибудь побочный эффект, она должна вернуть свое значение внутри монады `Fay`, а если функция работает чисто, то может свободно возвращать чистый тип.
- Телом функции должен быть вызов функции `ffi` в модуле `FFI`, которой передается один строковый аргумент. Эта строка должна быть реальным JavaScript-кодом, который будет вызван там, где можно использовать идентификаторы формы `%n` для ссылки на *n*-й параметр в Haskell-сигнатуре.

Типичным примером FFI может служить вызов JavaScript-функции `alert`, показывающей всплывающее окно с произвольным сообщением, заданным строкой. Эта функция не возвращает никакого значения, поэтому возвращаемый ею тип нужно определить просто как элемент `()`, декорированный монадой `Fay`, поскольку будет иметь место побочный эффект. Собрав все вместе, можно создать новую версию `Simple` с использованием функции `alert` вместо `print`:

```
module Simple where

import Prelude
import FFI

main :: Fay ()
main = alert "Hello from Fay!"

alert :: String -> Fay ()
alert = ffi "alert(%1)"
```

Затем нужно выполнить этот код, как ранее объяснялось, после чего появится всплывающее окно с сообщением.

СОЗДАНИЕ ИНТЕРФЕЙСОВ ДЛЯ ДРУГИХ ЯЗЫКОВ

Fay предлагает для JavaScript очень простой интерфейс внешних функций. Однако создание из компилятора GHC интерфейсов для других языков программирования, особенно для C, ничуть не сложнее. Например, в Haskell 2010 Report представлено объявление, позволяющее вызвать функцию sin из стандартной C-библиотеки math.h:

```
import Foreign.C

foreign import ccall "math.h sin" sin :: CDouble -> CDouble
```

Как видите, сюда включены те же основные элементы, что и для JavaScript-упаковщиков: вызываемая функция (для C также имя заголовочного файла) и сигнатура, выраженная в понятиях языка Haskell. В этом коде CDouble является специальным типом, определенным в модуле Foreign.C для организации интерфейса с C-значениями двойной точности.

Планируя организовать интерфейс с «родным» кодом, лучше поискать в Hackage какой-нибудь пакет, который предоставляет необходимые мосты. Если ничего присмотреть не удастся, можно обратить внимание на программы hsc2hs и c2hs, облегчающие создание файлов интерфейса с C путем анализа заголовочных файлов.

Интерфейсы для C++, Java и .NET также определены в Haskell Report, хотя поддержка для этих сред еще далека от завершения.

Fay и jQuery

В качестве заключительного примера, иллюстрирующего эффективность Fay, в этом разделе показано, как выполнять обращения к серверу, разработанному в предыдущем разделе, используя в качестве вспомогательной библиотеку jQuery, являющуюся межбраузерной JavaScript-библиотекой, призванной упростить разработку функционально насыщенных веб-страниц путем предоставления высокоуровневых упаковщиков для таких функциональных возможностей браузера, как манипулирование элементами страницы, вызовы удаленных серверов, анимация и многое другое. То есть эта библиотека — очень хороший объект для организации с ее помощью интерфейса из Fay. Именно такой интерфейс и предлагает библиотека `fay-jquery`.

Чтобы воспользоваться библиотеками внутри Fay, сначала нужно установить их с помощью Cabal, как и любые другие Haskell-библиотеки. Командная строка, необходимая для установки библиотеки jQuery наряду с библиотекой `fay-text`, установка которой является необходимым условием работы библиотеки jQuery, имеет следующий вид:

```
$ cabal install fay-text fay-jquery
```

Теперь при создании кода с помощью ключа `--package` нужно сообщить компилятору Fay о необходимости использования этих новых пакетов. Предположим, что код, который мы разработаем в данном разделе, будет помещен в файл CommWithServer.hs. Тогда полностью командная строка должна выглядеть так:

```
$ fay --package fay-text,fay-jquery CommWithServer.hs
```

Разрабатываемая веб-страница будет очень простой: текстовое поле для ввода идентификатора товара и кнопка. При щелчке на кнопке страница должна вступить в контакт с веб-приложением, созданным в Scotty, доставить информацию о товаре с идентификатором, введенным пользователем, и показать ее пользователю. Этому описанию соответствует следующий HTML-код:

```
<html>
  <head>
    <title></title>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.
js"></script>
    <script src="CommWithServer.js"></script>
  </head>
  <body>
    <label for="productId">Product #:</label>
    <input type="text" id="productId" />
    <input type="submit" id="productButton" value="Get this product!" />
  </body>
</html>
```

Обратите внимание на строку, которая предназначена для загрузки основных библиотек для jQuery (тех, которые написаны на JavaScript) с публичного сервера Google, и на идентификаторы, присваиваемые текстовому полю и кнопке, — в последующем они позволят ссылаться на них в Fay-коде.

Сначала нужно добавить определение типа `Product`, которое будет использовано для десериализации информации, отправленной сервером. Haskell-сервер обеспечивает преобразование данных из Haskell-значений в формат JSON и обратно, а компилятор Fay получает возможность считывать эти JSON-значения, поскольку в его внутрен-

них структурах используются JavaScript-объекты. Нужно учесть, что `ToJson` является экземпляром, генерируемым для нас средой Persistent, и он содержит те же имена полей, что и имена полей сущности. Определение имеет следующий вид:

```
data Product = Product { name :: Text
                       , description :: Text
                       , price :: Double
                       , inStock :: Int
                       }
```

Далее нужно определить функцию, которая будет вызвана после щелчка на кнопке. В данном простом примере она посредством jQuery-функции `select` будет получать ссылку на текстовый ввод, а затем посредством функции `getVal` считывать текст, введенный пользователем. А теперь самое важное: с помощью функции `ajax` создается URL-адрес для запроса, и запрос отправляется на сервер. Эта функция получает два аргумента, указывающие на то, что делать при успешной обработке запроса и что делать в случае ошибки (в обоих случаях появляется всплывающее окно):

```
{-# LANGUAGE OverloadedStrings, RebindableSyntax #-}

import Fay.Text
import FFI
import JQuery
import Prelude hiding (concat)

onProductButtonClick :: Event -> Fay Bool
onProductButtonClick _ = do
  productField <- select "#productId"
  productId <- getVal productField
  let url = concat ["http://localhost:3000/json/product/", productId]
  ajax url
       (\p -> alert $ name p)
       (\_ _ _ -> alert "Error in AJAX request")
  return True

alert :: Text -> Fay ()
alert = ffi "alert(%1)"
```

ПРИМЕЧАНИЕ

В библиотеке jQuery для Fay используется пакет fay-text, предназначенный для естественного обмена данными посредством JavaScript-строк. Чтобы сократить объем показанного кода, в него включены расширения для перезагружаемых строк.

Теперь осталось только связать предыдущую функцию с событием щелчка на кнопке. Это делается посредством функции `onClick`. Но вы не можете включить этот код непосредственно в функцию `main`, поскольку нет никаких гарантий, что HTML-страница на момент выполнения кода уже загружена. Решение состоит в том, чтобы заключить функцию в оболочку `ready`, что заставит перед началом выполнения ожидать полной загрузки страницы:

```
main :: Fay ()
main = ready $ do
  productButton <- select "#productButton"
  onClick onProductButtonClick productButton
  return ()
```

Если теперь открыть HTML-страницу в браузере, ввести идентификатор существующего в базе данных товара и щелкнуть на кнопке, будет возвращена информация о наименовании товара. Великолепно!

ПОЛИТИКА ЕДИНОГО ПРОИСХОЖДЕНИЯ

Если код всех этих примеров выполнить в локальной среде, ваша веб-страница может вообще не заработать. Причина, скорее всего, кроется в политике единого происхождения (same origin policy). Эта политика, обычно реализуемая в каждом браузере, не позволяет странице из одного домена получать информацию из других доменов. В этом случае локальная страница не может передавать информацию в localhost:3000, где выполняется Scotty.

Простейшим решением будет добавление заголовка, разрешающего передачу данных из любого домена в Scotty-коде. Это делается путем вызова функции setHeader "Access-Control-Allow-Origin" "*" в обработчике JSON-маршрута. Так вы оповещаете браузер о том, что сервер допускает вызовы отовсюду.

После этих слов совершенно очевидно, что указанный заголовок не следует добавлять в каждое создаваемое вами веб-приложение, поскольку информация может быть доступна другим доменам. Более удачным решением является отделение от домена внутреннего и внешнего интерфейсов. В качестве альтернативы можно включить ограничительный список разрешенных клиентов в заголовок Access-Control-Allow-Origin.

На этом краткое знакомство с Fay завершается. Мы выяснили, как компилировать код в JavaScript-код и как взаимодействовать с библиотеками на языке JavaScript. В качестве примера использовалась библиотека jQuery, но сообщество Fay-разработчиков весьма активно, поэтому постоянно появляются пакеты для многих других библиотек.

Выводы

Эта глава послужила для нас введением в тему создания веб-приложений на языке Haskell.

- Мы узнали о многих библиотеках, предоставляемых Hackage для реализации каждой функциональной возможности, востребованной в веб-приложении: маршрутизации, шаблонизации, обработке форм и т. д.
- Основное внимание в этой главе было уделено разработке магазина машин времени на принципах архитектурного стиля *REST*.
- В качестве минималистской среды веб-разработки нам была представлена библиотека *Scotty*, и мы узнали, как создаются обработчики для различных маршрутов и методов запросов.
- Мы рассмотрели две *системы шаблонизации*: библиотека `blaze-html` предлагает множество комбинаторов для составления HTML-документов, а библиотека Hamlet служит для встраивания разметки непосредственно в Haskell-код с помощью механизма Template Haskell.
- Мы познакомились с библиотекой `digestive-functors`, обеспечивающей обработку пользовательского ввода.
- И наконец, в качестве средства программирования богатого внешнего интерфейса веб-страниц с использованием Haskell мы изучили компилятор Fay и познакомились с примером выполнения удаленных вызовов.

Часть IV

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ

Глава 13. Строгие типы для описания предложений

Глава 14. Интерпретация предложений с помощью атрибутов

Строгие типы для описания предложений

13

Мы уже решили две основные задачи, поставленные в книге: реализовали кое-какие алгоритмы анализа данных, позволяющие провести анализ клиентов и их покупок, и рассмотрели несколько способов сохранения информации о покупках в долговременном хранилище. Эта глава является стартовой позицией для решения последней задачи: создания *предметно-ориентированного языка* (Domain Specific Language, DSL), предназначенного для выражения предложений с целью их применения в магазине машин времени. Этот компактный язык предоставит множество комбинаторов для создания предложений, которые для начала будут очень простыми.

Не все предложения, выражаемые на этом нашем первом языке, будут корректными. Интересно, что система типов может помочь избежать многих некорректных предложений, и в этой главе мы рассмотрим необходимые для этого средства.

В первую очередь мы познакомимся с *обобщенными алгебраическими типами данных* (Generalized Algebraic Data Types, GADT), которые позволяют создавать более конкретные конструкторы для типов данных. Затем мы узнаем, как кодируются другие инварианты в условиях *зависимой типизации* (dependent typing), которая дает возможность использовать внутри типов обычные значения с применением языка Idris, напоминающего язык Haskell, но с более эффективной зависимой системой типов. Что касается Haskell, то в этом языке зависимые типы вообще отсутствуют. То есть нам придется кодировать инварианты с помощью других приемов. В частности, в этой главе рассматриваются *функциональные зависимости* (Functional Dependencies, FD) и *семейства типов* (Type Families, TF) наряду с библиотекой `singletons`.

После прочтения данной главы мы должны иметь полное представление обо всей системе Haskell-типов. Мы должны не только понимать код, в котором используются описываемые здесь нетривиальные возможности, но и применять их в своем коде.

Предметно-ориентированные языки

Понятие *предметно-ориентированного языка* относится, как правило, к языку, предназначенному не для выражения *какой-нибудь* компьютерной программы, а для выражения данных или вычислений, относящихся к пространству *конкретной проблемы*. Например, SQL может рассматриваться как DSL для выражения запро-

сов к базам данных. В то же время Haskell является не предметно-ориентированным языком, а языком общего назначения, поскольку он не привязан к какой-либо конкретной области применения.

Существование предметно-ориентированных языков не является в компьютерной науке чем-то новым. В ядрах многих применяемых нами программ используются один или несколько предметно-ориентированных языков. Например, CSS — это предметно-ориентированный язык, выражающий представления атрибутов для элементов разметки, регулярные выражения являются предметно-ориентированным языком для выражения строковых образцов, разработчики компьютерной аппаратуры задействуют специфические языки для проектирования электрических схем и т. д.

Еще раз отметьте, что предметно-ориентированный язык предназначен для выражения не просто какой-то из областей, а одной конкретной области. Это означает, что предметно-ориентированный язык должен быть очень тесно связан с предметной областью, характеристики которой вы намерены выразить, позволяя организовать взаимодействие и всецело используя абстракции, характерные для данной предметной области. Это может быть полезно в двух отношениях:

- Разработчикам больше не нужно записывать в программах бизнес-правила. Если предметно-ориентированный язык действительно выражает правила с помощью обычных абстракций предполагаемой предметной области, специалистами в этой конкретной области могут быть те же, кто пишет правила. Это снижает уровень недопонимания между специалистами данной предметной области и программистами.
- Нам достаточно реализовать ядро DSL, которое обычно весьма компактно, а затем использовать новые абстракции для построения остальной системы. Это упрощает написание программы, ее понимание и поддержку, поскольку абстракция определенной предметной области не распределяется по разным сменным частям программы.

Эти преимущества играют в процессе разработки программного обеспечения весьма важную роль. С учетом этого важность самой концепции предметно-ориентированного языка возросла, и теперь многие проблемы решаются с помощью специализированных языков для соответствующих предметных областей. На разработку предметно-ориентированных языков ориентированы многие инструментальные средства, и эта глава убедит вас в том, что Haskell в этом плане является одним из лучших.

Внедрение в Haskell своего языка

Любой разрабатываемый вами предметно-ориентированный язык в конечном итоге будет интегрирован в более крупное приложение, написанное на языке общего назначения, подобном языку Haskell. Следовательно, большую важность приобретает вопрос подключения DSL к этому другому языку, который обычно называют *хостом* (host).

Как вариант, можно сделать DSL полностью независимым от языка-хоста. В таком случае придется разработать для предметно-ориентированного языка полный набор

инструментов. В частности, для компилятора потребуется написать основные части, или, по крайней мере, парсер. В результате вы получите свободу в разработке, но от вас потребуется приложить максимум усилий для реализации языка. При таком способе интеграции, когда DSL действительно формирует ядро приложения, а инструментальные средства (отладчики, механизмы интеграции с IDE) вы предоставляете дополнительно, язык называется *внешним* (external), или *автономным* (standalone). В качестве подходящего примера такой разновидности DSL можно привести HTML.

В отличие от этого можно разработать *внедренный* (embedded), или *внутренний* (internal), предметно-ориентированный язык. В этом случае ваш язык выражается с помощью ряда конструкций языка-хоста. Основным преимуществом такого подхода является отсутствие необходимости писать многочисленные инструментальные средства, к тому же язык-хост может быть полезен в разработке DSL в том плане, что избавляет вас от дополнительных поисков недопустимых значений в вашем предметно-ориентированном языке. Больше того, вам совсем не обязательно использовать единственный предметно-ориентированный язык, в приложение может быть внедрено несколько таких языков. Основным недостатком является то, что вы ограничены языком-хостом, и в частности, ошибки могут сказываться на внутренних компонентах вашего предметно-ориентированного языка.

Ряд свойств языка Haskell делают его хорошим хостом для внедрения DSL:

- Его эффективная система типов позволяет выражать инварианты, которым должны удовлетворять корректные DSL-значения. Это означает, что безопасность, обеспечиваемая языком Haskell, отчасти распространяется и на ваш язык.
- Синтаксис языка Haskell весьма лаконичен, а значит, он не вводит в DSL множество странных символов. Используя операторы и группы, в вашем новом языке можно создавать значения, похожие на описания обычных языков.
- Классы типов, особенно аппликативы, монады и их do-нотация, предлагают удобную среду для выражения абстракций, являющихся частью DSL. Это означает, что Haskell-разработчики для многих концепций могут использовать хорошо им знакомую нотацию.

Многие библиотеки, о которых шла речь в данной книге, фактически являются внедренными предметно-ориентированными языками: комбинаторам парсеров, таким как `attoparsec`, которые предназначены для описания грамматики, нужны только специальные функции; язык запросов, такой как Esqueleto, применяет сочетание классов типов и нестандартных типов данных; библиотека `digestive-functors` выражает формы с помощью аппликативного стиля и множества вариантов трактовки значения формы. Как видите, упомянутые свойства языка Haskell используются каждым предметно-ориентированным языком DSL по-разному, в зависимости от того, насколько подходят они в каждом случае.

В том, как внедренные предметно-ориентированные языки проявляют себя в Haskell, есть две основные тенденции. При *поверхностном внедрении* (shallow embedding) значения кодируются в вашем языке непосредственно по их смысловому значению в Haskell (это смысловое значение называется их *интерпретацией*). Предположим, к примеру, что вы создаете DSL для стековых операций. Как показано в следующем коде, при поверхностном внедрении каждая DSL-операция просто представляется Haskell-функцией:

```
pop :: [a] -> [a]
pop (x:xs) = xs
pop _      = error "empty stack"

push :: a -> [a] -> [a]
push x xs = x:xs

-- Значение в нашем языке: эта интерпретация работает прямо в стеке
v :: [Int] -> [Int]
v = push 1 . pop . pop . push 2
```

В то же время при *глубоком внедрении* (deep embedding) использование языка происходит в два этапа. Сначала с помощью Haskell-типов в вашем языке создается представление значения в виде синтаксического дерева. Глубокое внедрение предыдущего примера реализовано в следующем коде:

```
data StackOps a = Pop (StackOps a) | Push a (StackOps a) | End
-- То же значение представлено в виде синтаксического дерева
v :: StackOps Int
v = Push 1 $ Pop $ Pop $ Push 2 $ End
```

Затем вы даете интерпретацию синтаксического дерева, в результате его значение преобразуется в Haskell-значение. Преимущество здесь в том, что теперь можно выполнять дополнительную оптимизацию, например `Pop` после `Push` аналогично отсутствию операции:

```
interpret :: StackOps a -> [a] -> [a]
interpret (Pop (Push _ ops)) stack = interpret ops stack
interpret (Pop ops)          stack = tail $ interpret ops stack
interpret (Push e ops)       stack = e : interpret ops stack
interpret End                stack = stack
```

Здесь упомянуты два преимущества глубокого внедрения по сравнению с поверхностным. Прежде всего, можно дать более одной интерпретации, то есть трактовать значения вашего внедренного предметно-ориентированного языка по-разному в зависимости от ситуации. Кроме того, перед созданием смыслового значения можно изучить синтаксическое дерево и реализовать для вашего значения некие оптимизационные или статистические действия.

В данной главе мы займемся глубоким внедрением языка предложений в Haskell, а в следующей главе — изучением UUAGC-компилятора, призванного помочь в выражении поведения одного из предложений применительно к корзине товаров.

Язык предложений

Давайте создадим предметно-ориентированный язык для выражения предложений в магазине машин времени. Помимо расширения прав и возможностей менеджеров в плане непосредственного кодирования предложений для нашего приложения (чтобы не приходилось всякий раз писать их вручную), реализация языка станет хорошей возможностью больше узнать о Haskell. Для достижения второй цели слишком объемный язык нам не нужен, поэтому ядро DSL будет весьма компактным. Вообще-то надо всегда стремиться к тому, чтобы предметно-ориентированный язык был как можно более компактным, поскольку с небольшим ядром проще работать.

А для решения сложных задач можно написать функции, генерирующие составные выражения, которые уже не будут относиться к создаваемому простому языку.

Первое, что должен делать язык предложений, — предоставлять основные предложения для использования в магазине. В данном случае пусть у нас будет три основных предложения: предоставление чего-нибудь в качестве подарка, снижение цены на определенный процент и снижение цены на некоторое абсолютное значение. Из этих трех основных предложений мы сможем генерировать DSL-значения, комбинируя предложения с дополнительными фрагментами данных. В нашем случае такие комбинации могут быть разбиты на три группы:

- Возможно ограничение действия предложения, чтобы оно было действительным только для некоего множества товаров.
- По умолчанию все предложения действительны на неопределенный период времени. Язык предоставит возможность ограничивать начальную и конечную даты действия предложения, а также распространять действие предложения на более длительные периоды времени.
- Возможно создание предложения, являющегося объединением двух предложений (например, предоставление бесплатного воздушного шарика и 10-процентной скидки), просто лучшего из двух предложений (либо предоставление 10-процентной скидки, либо скидки в 5 долларов) или условного предложения, зависящего от удовлетворения некоторых критериев покупки (вроде применения предложения только при приобретении товаров на сумму свыше 100 долларов).

Исходя из этого описания получить объявление типа `data` для синтаксического дерева довольно просто:

```
data Offer a = Present a | PercentDiscount Float | AbsoluteDiscount Float -- основа
  | Restrict [a] (Offer a)
  | From Integer (Offer a) | Until Integer (Offer a) | Extend Integer (Offer a)
  | Both (Offer a) (Offer a) | BetterOf (Offer a) (Offer a) | If (Expr a) (Offer a)
(Offer a)
  deriving Show
```

При использовании условных выражений может понадобиться выразить отсутствие предложений при неудовлетворении какого-нибудь условия. Для таких случаев подойдет значение «no offer» («предложение отсутствует»). Одним из подходов может быть включение в тип данных дополнительного конструктора. Однако если подумать, можно прийти к мысли, что эквивалентом отсутствия предложения может служить абсолютная скидка в 0 долларов. Таким образом, ядро можно оставить в прежнем виде и определить это предложение в других понятиях:

```
noOffer :: Offer a
noOffer = AbsoluteDiscount 0
```

В коде типа `Offer` не хватает объявления типа данных `Expr`, призванного кодировать те выражения, которые могут появляться в качестве условий покупки. В него должны быть включены количество и цены нескольких или всех товарных позиций в корзине покупок, сравнения между этими величинами и булевыми комбинаторами условий (и, или, не). Обратите внимание, как в следующем коде

с помощью конструкторов `IVal` и `FVal` в наш язык подтянуты базовые значения (целое и с плавающей точкой):

```
data Expr a = AmountOf a | PriceOf a              -- информация о корзине
  | TotalNumberProducts | TotalPrice
  | IVal Integer | FVal Float        -- подтягивание в DSL числовых значений
  | (Expr a) :+: (Expr a) | (Expr a) :*: (Expr a) -- арифметические операции
  | (Expr a) :<: (Expr a) | (Expr a) :<=: (Expr a) -- сравнения
  | (Expr a) :>: (Expr a) | (Expr a) :>=: (Expr a)
  | (Expr a) :&&: (Expr a) | (Expr a) :||: (Expr a) | Not (Expr a) -- булевы
операции
  deriving Show
```

Давайте, к примеру, выразим предложение «в течение 30 дней вы получите лучший из двух вариантов: либо скидку в 10 долларов на окончательный счет, либо шарик в качестве подарка, а если купите товаров более чем на 100 долларов, получите скидку в 5 %». Значение для такого предложения имеет следующий вид:

```
v :: Offer String
v = Until 30 $ BetterOf (AbsoluteDiscount 10.0)
                        (Both (Present "ballon")
                         (If (TotalPrice :>: IVal 100) (PercentDiscount 5.0)
                                                       noOffer))
```

Эти типы данных являются ядром нашего предметно-ориентированного языка. Однако как показывает упражнение 13.1, мы можем добавить ряд вспомогательных функций, чтобы упростить описания тех предложений, которые зачастую следуют одному и тому же образцу.

УПРАЖНЕНИЕ 13.1. ОБРАЗЦЫ ПРЕДЛОЖЕНИЙ

Мы выяснили, как определить значение noOffer в понятиях более общих конструкторов, сохраняя при этом простоту ядра языка. Следуя той же идее, напишите определения для следующих образцов предложений:

- значение `period f d o` будет ограничивать предложение «`o`» следующими днями «`d`» начиная с дня «f». Не забудьте, что в From и Until в качестве аргументов используются контрольные точки времени, а не интервалы времени.
- значение `allOf os` должно быть конъюнкцией всех предложений в списке «`os`».

Затем нужно выразить следующее предложение: «Начиная с третьего дня в течение пяти дней вы получите шарик, бесплатный шоколадный кекс и 10-процентную скидку в магазине машин времени». Проверьте, соответствует ли ваше выражение правильному предложению, выраженному через ядро языка DSL.

Кроме ограничения основных предложений есть еще два дополнительных требования, которым должны удовлетворять все значения в нашем предметно-ориентированном языке. Первое требование называется *правилом подарков* (presents rule): в определенные месяцы года количество подарков, бесплатно выдающихся вместе с покупками, ограничено. Таким образом, система должна проверять наличие ограничений. Второе требование называется *правилом времени действия* (duration rule): нельзя делать предложения, если из-за ограничения по времени в какой-либо отдельно взятый момент они становятся недействительными. Реализацией в языке этих двух правил мы займемся в следующих разделах.

Безопасность языка выражений

Как отмечено в начале этой главы, имеющаяся в Haskell строгая система типов помогает исключить из предметно-ориентированного языка некорректные значения. Однако в нашей первой реализации некорректные значения могли создаваться, не вызывая больших проблем. Посмотрите на следующий пример, в котором значение `Expr` создается путем дизъюнкции цены и некоторых других выражений. Однако сама по себе цена не является булевым значением, поэтому ей нельзя придать какой-либо смысл.

```
incorrectExpression :: Expr Char
incorrectExpression = TotalPrice :||: (TotalNumberProducts :<: PriceOf 'a')
```

Для исправления ситуации во всех примерах мы добавим некий тег к DSL-типам и ограничим доступные варианты комбинирования значений разных типов. В нашем случае:

- Количественные показатели, цены товарных позиций и постоянные значения (те, что создаются с помощью конструкторов `FVal` или `IVal`) должны иметь числовой тег.
- Аргументами сравнения должны быть значения с числовыми тегами, а результатом — значение с булевым тегом.
- Булевы операторы могут комбинировать только значения с булевыми тегами.
- Заключительное выражение внутри предложения должно иметь булев тег.

Лучше всего включить такой тег в наше выражение, добавив новый параметр типа к типу данных `Expr`. В таком случае выражение `Expr a t` будет относиться к товарам типа `a`, помеченным тегом типа `t`. Если бы теперь каждый из комбинаторов, составляющих выражение, был не конструктором, а обычной функцией, то такие ограничения можно было бы наложить, просто ограничивая сигнатуру функции. Например:

```
(:||:) :: Expr a Bool -> Expr a Bool -> Expr a Bool
```

И подобная функция должна быть у каждого нашего комбинатора.

Проблема в том, что обычные объявления `data` не позволяют возвращать разные типы в зависимости от конструктора значения. Все конструкторы *единообразно* создают значения типа, указанного после ключевого слова `data`. Обойти это ограничение позволяет GHC-расширение, называемое *обобщенными алгебраическими типами данных* (GADT): каждый из конструкторов типа данных может возвращать для определяемого типа разное множество параметров типа (то есть в определении `Expr a t` можно возвращать `Expr a Bool`, `Expr Int Int` и т. д., но не `Offer Char`).

Синтаксис начинается с такого же ключевого слова `data`, за которым следуют имя и тип переменных определяемого типа. Но вместо знаков равенства нужно указать ключевое слово `where` и перечислить конструкторы посредством их сигнатур. Выражение GADT-типа выглядит следующим образом:

```
{-# LANGUAGE GADTs #-}

data Expr a r where
  AmountOf            :: a -> Expr a Integer
```

```
  PriceOf             :: a -> Expr a Float
  TotalNumberProducts :: Expr a Integer
  TotalPrice          :: Expr a Float
  IVal                :: Integer -> Expr a Integer
  FVal                :: Float -> Expr a Float
  (:+:)               :: Expr a Integer -> Expr a Integer -> Expr a Integer
  (:+.:)              :: Expr a Float -> Expr a Float -> Expr a Float
  (:*:)               :: Expr a Integer -> Expr a Integer -> Expr a Integer
  (:*.:)              :: Expr a Float -> Expr a Float -> Expr a Float
  (:<:)               :: Expr a Integer -> Expr a Integer -> Expr a Bool
  (:<.:)              :: Expr a Float -> Expr a Float -> Expr a Bool
  (:<=:)              :: Expr a Integer -> Expr a Integer -> Expr a Bool
  (:<=.:)             :: Expr a Float -> Expr a Float -> Expr a Bool
  (:>:)               :: Expr a Integer -> Expr a Integer -> Expr a Bool
  (:>.:)              :: Expr a Float -> Expr a Float -> Expr a Bool
  (:>=:)              :: Expr a Integer -> Expr a Integer -> Expr a Bool
  (:>=.:)             :: Expr a Float -> Expr a Float -> Expr a Bool
  (:&&:)              :: Expr a Bool -> Expr a Bool -> Expr a Bool
  (:||:)              :: Expr a Bool -> Expr a Bool -> Expr a Bool
  Not                 :: Expr a Bool -> Expr a Bool
```

Учтите, что для полной безопасности типов код нуждается в собственных версиях оператора сравнения для целого значения и для значений с плавающей точкой. Можно скомбинировать оба оператора, воспользовавшись контекстом класса, поскольку оба типа являются экземплярами класса типов `Num`. В таком случае значения и сравнения приобретут следующий вид:

```
data Expr a r where
  ...
  Val                 :: Num n => n -> Expr a n
  (:+:)               :: Num n => Expr a n -> Expr a n -> Expr a n
  (:*:)               :: Num n => Expr a n -> Expr a n -> Expr a n
  (:<:)               :: Num n => Expr a n -> Expr a n -> Expr a Bool
  (:<=:)              :: Num n => Expr a n -> Expr a n -> Expr a Bool
  (:>:)               :: Num n => Expr a n -> Expr a n -> Expr a Bool
  (:>=:)              :: Num n => Expr a n -> Expr a n -> Expr a Bool
```

В следующих примерах подразумевается, что конструкторы с контекстом `Num n` такие же, как в типе данных `Expr`.

Если теперь попробовать создать код с неверным выражением, с которого начинался этот раздел, произойдет ошибка компиляции, поскольку выражение не пройдет проверку типов. Это наш первый пример использования строгой системы типов для ограничения видов значений, которые могут быть выражены в DSL.

Кроме того, GADT-типы решают еще одну проблему, которая так или иначе будет возникать в дальнейшем. Предположим, нам необходимо интерпретировать одно выражение с помощью исходного типа данных `Expr`: эта интерпретация будет представлять собой функцию, получающую список товаров в виде пар (*наименование*, *цена*) и возвращающую результат применения к ним выражения. Поскольку результат выражения может иметь как типы `Integer`, `Float`, так и тип `Bool`, в качестве возвращаемого значения нужен тип суммы. Вот как выглядит небольшой фрагмент интерпретации для случая `(:||:)` (заметьте, что в коде нужно в явном виде позаботиться о несоответствиях типов):

```
data ExprR = EInt Integer | EFloat Float | EBool Bool

interpretExpr :: Expr a -> [(a,Float)] -> ExprR
interpretExpr (e1 :||: e2) list =
  case (interpretExpr e1 list, interpretExpr e2 list) of
    (EBool b1, EBool b2) -> EBool (b1 || b2)
    _                    -> error "type error"
interpretExpr ...               = ...
```

Однако если воспользоваться нашим GADT-типом, больше не придется создавать специальный тип данных для возвращения значения выражения, поскольку получаемое в результате значение может зависеть от тега в типе `Expr`. Это дает нам новый вариант полиморфизма: перед возвращением значения части входного типа этому типу не нужно быть единообразным для всех конструкторов, заставляя функцию возвращать значения из разных типов. Посмотрите, как это происходит в функции интерпретации для случаев `(:||:)` и `(:+:)`:

```
interpretExpr :: Eq a => Expr a t -> [(a,Float)] -> t
interpretExpr (e1 :+: e2)  list = interpretExpr e1 list + interpretExpr e2 list
interpretExpr (e1 :||: e2) list = interpretExpr e1 list || interpretExpr e2 list
interpretExpr ...               = ...
```

Этой функции интерпретации нужно уделить еще немного внимания. Завершить работу предлагается упражнением 13.2.

УПРАЖНЕНИЕ 13.2. ЗАВЕРШЕНИЕ ИНТЕРПРЕТАЦИИ ВЫРАЖЕНИЙ

Напишите недостающий код функции interpretExpr как для случая определения Expr с обычным ADT-типом, так и для случая определения с GADT-типом. В варианте с PriceOf нужно учесть, что некоторые товары могут появляться в перечне покупок более одного раза.

Хотя в данном случае создание тегов для типов осуществляется с использованием встроенных типов из пакета Prelude, ничто не мешает нам использовать и другие типы. Мы можем строить новые типы данных, чьим *единственным* предназначением будет создание тегов для других типов. Тогда в объявлении нам даже не понадобятся какие-либо конструкторы, поскольку они никогда не будут использоваться. Если включить расширение `EmptyDataDecls`, можно будет создавать пустые объявления `data`.

Давайте рассмотрим пример, показывающий, насколько это может быть полезным. В магазине машин времени может быть много пользователей. Но не все они будут играть в магазине одну и ту же роль: некоторые из них могут быть администраторами или управляющими (эти могут вносить любые изменения), другие — складскими работниками (этим разрешено лишь обновлять в базе данных сведения о товарах), кроме того, у вас будут и обычные пользователи (то есть покупатели). Применяя множество пустых типов данных, можно создать тег, указывающий на уровень доступа к магазину:

```
{-# LANGUAGE EmptyDataDecls, GADTs #-}

data AllowEverything
data AllowProducts
data AllowPurchases
```

```
data Person = Person :: { firstName :: String, lastName :: String }

data User r where
  Admin        :: Person -> User AllowEverything
  StoreManager :: Person -> User AllowEverything
  StorePerson  :: Person -> User AllowProducts
  Client       :: Person -> User AllowPurchases
```

Теперь для того, чтобы потребовать значение с тегом правильного типа, может быть определена функция, которую будут вызывать только те, кому доступно все:

```
changePurchaseFinalPrice :: User AllowEverything -> Purchase -> Float -> Purchase
changePurchaseFinalPrice = ...
```

Путешествие во времени — задача утомительная, поэтому пользователям наших машин времени периодически нужно подкрепляться. Однако на еду, которую можно брать с собой, накладываются определенные ограничения: она должна быть растительной (никакой свинины), слабо соленой и т. д. В упражнении 13.3 от вас требуется создать GADT-тип, представляющий такую еду и имеющий ограничительные теги.

УПРАЖНЕНИЕ 13.3. ЕДА ДЛЯ ПУТЕШЕСТВЕННИКОВ ВО ВРЕМЕНИ

Создайте GADT-тип, представляющий множество доступных блюд для магазина машин времени. У блюд должны быть теги с типом, разрешающим вегетарианскую еду и определяющим, содержится в еде свинина или нет. Используйте, как показано в этом разделе, пустые типы данных.

Зависимая типизация

Похоже, что идея снабжения типов тегами с дополнительной информацией оказалась весьма удачной, по крайней мере для выражений. Однако способ воплощения ее в жизнь крайне ограничен, поскольку мы можем лишь использовать другие типы, причем только в качестве констант или переменных, которые нами не проверяются.

Одним из способов получения более эффективной системы типов является разрешение участвовать в других типах не только типам, но и значениям. Например, было бы интересно снабдить списки тегов их длиной и выразить что-то подобное следующему: «Длина `append l1 l2` является суммой `l1` и `l2`». Как видите, в этом случае тег — это не другой тип, а число, то есть значение в мире Haskell-понятий.

Haskell-значениям разрешается зависеть от других значений (посредством обычных аргументов функций) или от типов (посредством параметрического или специального полиморфизма), а типам разрешено зависеть только от других типов (посредством переменных типа). *Зависимые системы типов* (dependent type systems) снимают последний барьер, устанавливаемый Haskell между значениями и типами, и позволяют создавать типы, зависящие от значений, как уже упомянутые списки, дополненные тегами со значениями их длины.

Зависимые типы являются характерным атрибутом нескольких языков, в числе которых Coq, Twelf, Agda и Idris. Agda и Idris наиболее похожи по концепции

и синтаксису на Haskell и оба языка помогут знакомству с этими эффективными системами типов. В частности, в данном разделе в качестве инструмента обучения мы используем язык Idris.

ВНИМАНИЕ

Функциональность зависимой типизации гораздо богаче, чем описывается в данном разделе. Основное внимание здесь уделяется той части функциональности, которая каким-либо образом может быть выражена средствами Haskell. Для более глубокого изучения этой темы можно порекомендовать книги Эдвина Брэди (Edwin Brady) «Programming in Idris: a tutorial» (http://www.idris-lang.org/documentation), Ульфа Норелла (Ulf Norell) и Джеймса Чапмэна (James Chapman) «Dependently Typed Programming in Agda» (http://wiki.portal.chalmers.se/agda), Адама Члипала (Adam Chlipala) «Certified Programming with Dependent Types» (http://adam.chlipala.net/cpdt) и Бенджамина С. Пирса (Benjamin C. Pierce) и др. «Software Foundations» (http://www.cis.upenn.edu/~bcpierce/sf). Последние две касаются языка Coq.

Знакомство с языком Idris

Как отмечено в начале этого раздела, Idris является функциональным языком, система типов которого включает зависимые типы. Первоначально он разрабатывался в Сент-Эндрюсском университете, но теперь у него целое сообщество сторонников. Из всех языков с зависимой типизацией он наиболее близок к Haskell: синтаксис почти такой же, специальный полиморфизм реализован классами типов, поддерживается do-нотация и т. д. Кроме зависимых типов к другим характерным особенностям языка Idris относятся предметно-ориентированные языки, реализуемые посредством перегрузки синтаксиса, чем, возможно, вам будет интересно заняться после прочтения данной главы. Основное отличие Haskell от Idris состоит в том, что в последнем вместо модели ленивых вычислений используется модель упреждающих вычислений, хотя лень доступна через аннотации.

ПРИМЕЧАНИЕ

Чтобы использовать в Haskell приемы программирования на уровне типов, полностью разбираться в Idris вам не нужно. Технические части следующих разделов являются необязательными. Сконцентрируйтесь на том, чтобы понять концепции, заложенные в примеры, и не переживайте, если сначала заработают не все примеры. Вы всегда сможете перечитать эту часть книги после того, как наберетесь опыта в функциональном программировании.

В Haskell имеется реализация языка Idris, то есть он доступен в Hackage в виде еще одного пакета. Проще всего получить его на вашей системе, введя в командной строке следующую команду:

```
$ cabal install idris
```

ВНИМАНИЕ

Изначально для компиляции Idris понадобится присутствие в вашей системе нескольких библиотек. Однако установка всех этих библиотек может оказаться в Windows и Mac OS X непростой задачей. Для последней системы самым простым решением является компиляция Idris без внутреннего интерфейса LLVM, что можно сделать, если вместо показанной команды выполнить команду cabal install -f -LLVM idris. Соответствующую пошаговую процедуру для Windows можно найти по адресу https://github.com/idris-lang/Idris-dev/wiki/Idris-on-Windows.

Между Haskell и Idris есть небольшая синтаксическая разница, о которой следует знать:

- Символы (`::`) и (`:`) имеют противоположные значения. В Idris, чтобы выразить фразу «`x` имеет тип `Int`», нужно написать `x : Int`, а для списка, состоящего из значения 3, к которому добавляется значение l, делается запись `3 :: 1`.
- Тип для списков типа `a` записывается как `List a`, а не как `[a]`.

Теперь посмотрим, как объявляются списки, имеющие теги с их длиной. Создадим новый тип для этой цели и назовем его `ListL`.

В первую очередь нужно иметь в виду, что в Idris необходимо явное объявление типа параметров `ListL`. В Haskell этого делать не нужно, потому что только типы могут появляться в качестве параметров других типов. В данном случае имеются два параметра: натуральное число, выражающее длину, и тип элементов, содержащихся внутри списка, а возвращаться будет новый тип. Для второй части нужно создать выражение «этот параметр является типом», для которого используется ключевое слово `Type`. Результатом объявления данных станет еще один тип, то есть то, что вы должны включить явным образом. Итак, сигнатура для `ListL` приобретает вид `Nat -> Type -> Type`. Затем, как показано в следующем коде, посредством конструкторов GADT-стиля осуществляется объявление конструкторов:

```
data ListL : Nat -> Type -> Type where
  EmptyL : ListL 0 a
  ConsL  : a -> ListL n a -> ListL (S n) a
```

Обратите внимание на то, как в коде определяется изменение тега при применении различных конструкторов. В данном случае пустой список должен получить тег 0, а добавление к списку одного элемента должно увеличить его тег на единицу (для этого используется функция следования `S` для натуральных чисел, что эквивалентно прибавлению единицы).

Можно скомпилировать и загрузить код в Idris-интерпретатор точно так же, как это делалось при использовании GHCi. Например:

```
$ idris ListL.idr
```

```
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9.9.3
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Type checking ./ListL.idr
```

Чтобы проверить, что наши списки записывают свою длину правильно, давайте запросим у интерпретатора тип списка из трех элементов:

```
*ListL> :t ConsL 'a' (ConsL 'b' (ConsL 'c' EmptyL))
ConsL 'a' (ConsL 'b' (ConsL 'c' (EmptyL))) : ListL 3 Char
```

Как видите, было возвращено значение 3, что согласуется с определением `ListL`.

Система типов, используемая в Idris, настолько строга, что выводом типа больше нельзя получить тип каждого выражения. Поэтому компилятору нужно предоставить сигнатуру всех функций верхнего уровня. Разумеется, эта сигнатура должна

быть совместима с определяемой функцией. Например, можно написать функцию, дублирующую каждый элемент в списке, но указать в сигнатуре неверный тип. В следующем примере, иллюстрирующем подобный случай, код сообщает, что число элементов будет равно удвоенной длине исходного списка плюс единица, но это неверно:

```
duplicateEachElem : ListL n a -> ListL (n * 2 + 1) a
duplicateEachElem EmptyL      = EmptyL
duplicateEachElem (ConsL x r) = ConsL x (ConsL x (duplicateEachElem r))
```

Компилятор возмутится, и описание ошибки будет выглядеть примерно так:

```
$ idris ListL.idr
Type checking ./ListL.idr
./ListL.idr:When elaborating right hand side of duplicateEachElem:
Can't unify
    ListL 0 a
with
    ListL 1 a

Specifically:
    Can't unify
        0
    with
        1
```

Решение сводится к написанию правильной сигнатуры:

```
duplicateEachElem : ListL n a -> ListL (n * 2) a
```

Добавление тегов с натуральными числами может пригодиться во многих других структурах данных. В упражнении 13.4 вам предлагается создать GADT-тип для двоичных деревьев с целью отслеживания высоты каждого дерева.

УПРАЖНЕНИЕ 13.4. ДВОИЧНЫЕ ДЕРЕВЬЯ С ТЕГАМИ, ПОКАЗЫВАЮЩИМИ ВЫСОТУ

Создайте тип данных BinaryTree с двумя типами конструкторов: Leaf содержит только один элемент; Node содержит элемент и два поддерева. Тип должен иметь тег с высотой дерева. Воспользуйтесь языком Idris и описанными ранее приемами.

Подсказка: функция maximum возвращает наибольшее из двух натуральных чисел.

Предупреждение: если в Idris-интерпретаторе вы попытаетесь получить тип значения BinaryTree, то с большой степенью вероятности увидите примерно следующее:

```
*BinaryTree> :t Node 'a' (Leaf 'b') (Node 'c' (Leaf 'd') (Leaf 'e'))
Node 'a' (Leaf 'b') (Node 'c' (Leaf 'd') (Leaf 'e'))
  : Tree (S (boolElim (lte 1 (S (boolElim (lte 1 1) 1 1)))
                      (S (boolElim (lte 1 1) 1 1)) 1)
         Char
```

Решение состоит в том, чтобы потребовать от Idris вычислить выражение, которое также вычислит значения внутри типа. То есть включать символы :t в командную строку не нужно:

```
*BinaryTree> Node 'a' (Leaf 'b') (Node 'c' (Leaf 'd') (Leaf 'e'))
Node 'a' (Leaf 'b') (Node 'c' (Leaf 'd') (Leaf 'e')) : Tree 3 Char
```

В зависимо-типизированных языках грань между типами и значениями размыта, поэтому иногда приходится вычислять выражение, чтобы получить урезанный тип этого выражения.

ВНИМАНИЕ

При программировании с использованием зависимых типов можно столкнуться с немалыми сложностями, особенно когда тегами служат числа. В некоторых случаях компилятор не в состоянии проверить, эквивалентно одно выражение другому или нет, и тогда ему нужно предоставить подсказку. Однако вопрос о том, как это делается, выходит за рамки темы данной главы.

Введение правила подарков в Idris

Введение правила подарков в Idris посредством системы типов является простым уточнением идеи отслеживания количества элементов в списке или высоты дерева. В этом разделе мы не покажем весь код сразу, а представим каждый фрагмент объявления нового типа данных, снабдив его комментариями.

Сначала при трансляции кода из Haskell нужно добавить сигнатуру типа данных. Как и раньше, у `Offer` должна быть переменная типа для тех товаров, к которым будут применяться предложения. Затем нужно добавить натуральное число, чтобы снабдить предложение тегом с максимальным количеством подарков:

```
data Offer : Type -> Nat -> Type where
```

Основной конструктор `Present` должен отражать тот факт, что с ним дается один подарок. В то же время ни процентная, ни абсолютная скидки не добавляют к предложению ни одного подарка, поэтому их теги должны быть равны нулю:

```
Present          : a -> Offer a 1
PercentDiscount  : Float -> Offer a 0
AbsoluteDiscount : Float -> Offer a 0
```

Ни один из связанных со временем комбинаторов для `Offer` не изменяет количество выдаваемых подарков:

```
From             : Nat -> Offer a p -> Offer a p
Until            : Nat -> Offer a p -> Offer a p
Extend           : Nat -> Offer a p -> Offer a p
```

Однако при ограничении предложения только некоторыми товарами подарки также подпадают под это ограничение (они могут выдаваться только для ограниченного круга товаров). Таким образом, нужно указать минимальное количество товаров, подпадающих под ограничения, и максимальное возможное количество подарков. В данный момент мы теряем возможность узнать точное количество подарков, а можем лишь получить оценку их максимального количества. Проверьте, как необработанный список товаров также был изменен в `Vect` (встроенный тип, соответствующий ранее определенному типу `ListL`), чтобы гарантировать, что даны только те списки, в которых есть как минимум один товар:

```
Restrict         : Vect (S n) a -> Offer a p -> Offer a (minimum (S n) p)
```

Комбинация нескольких предложений представлена довольно просто: когда два предложения применяются одновременно, мы получаем в подарок оба, таким образом, тег изменяется с добавлением. Однако конструктор `BetterOf` не настолько прост. Поскольку заранее неизвестно, какая из ветвей будет принята, подсчитать

точное количество подарков для этих других вариантов невозможно. Но можно все же получить некоторые границы: когда применяется лучшее из двух предложений, мы получаем самое большое количество подарков, которое можем получить исходя из любого из этих предварительных предложений (то есть выбирается максимум). Таким образом, фокус переводится с получения точного количества подарков на получение верхней границы их количества.

```
Both            : Offer a p -> Offer a q -> Offer a (p + q)
BetterOf        : Offer a p -> Offer a q -> Offer a (maximum p q)
```

В понятиях количества подарков условное предложение действует как дизъюнктивное, поскольку самое большее, что можно получить, — это максимальное количество подарков в любой из ветвей. Еще раз фокус смещается с наличия точного значения к получению максимальной верхней границы. Это происходит из-за неопределенности результата условного выражения. Таким образом, последний конструктор нашего типа данных получает следующее прочтение:

```
If          : Expr a Bool -> Offer a p -> Offer a q -> Offer a (maximum p q)
```

В данном случае тип `Expr`, использованный внутри условий, остается таким же, каким он был в Haskell (за исключением того, что для указания сигнатуры конструкторов вместо оператора `::` применяется оператор `:`), поскольку выражения не оказывают влияния на количество подарков.

Контролировать правильность работы этого типа данных можно, создав предложение и проверив, что максимальное количество подарков вычисляется системой типов правильно. Например:

```
*CheckPresents> let p = Present 'a' in Both p (BetterOf p (Both p p))
Both (Present 'a') (BetterOf (Present 'a') (Both (Present 'a') (Present 'a'))) :
Offer Char 3
```

Программирование на уровне типов в Haskell

Haskell *не относится* к зависимо-типизированным языкам. В нем имеется четкое разделение между миром понятий, или выражений, и миром типов. В частности, в нем запрещено использовать понятия в качестве параметров типов. Однако как было показано в предыдущем разделе, теги со значениями могут быть очень полезными.

В разделе, посвященном GADT-типам, в качестве тегов использовались пустые типы данных. Этот подход отчасти имитирует зависимую типизацию, но позволяет лишь иметь теги с постоянными значениями, не допуская каких-либо операций над типами. То есть мы можем выразить тот факт, что конструктор `Present` включает только один подарок (поскольку он представляет собой константу), но не может, как конструктор `BetterOf`, выбрать максимум из обоих тегов. Вся остальная часть главы посвящена демонстрации различных способов описания операций над типами. В совокупности все эти методы известны как приемы *программирования на уровне типов* (type-level programming).

ИСПОЛЬЗОВАНИЕ В КОДЕ ПРИЕМОВ ПРОГРАММИРОВАНИЯ НА УРОВНЕ ТИПОВ

Приемы программирования, представленные в остальной части главы, обычно относят к нетривиальным возможностям Haskell. Чтобы стать хорошим и опытным Haskell-программистом, разбираться в каждой тонкости функциональных зависимостей и семейств типов не нужно. Действительно, эти возможности стали доступны в Haskell совсем недавно, поэтому их использование пока еще не получило широкого распространения.

Тем не менее программирование на уровне типов становится все более востребованным. Самые новые библиотеки, такие как Persistent или Yesod, применяют его достаточно интенсивно. Хотя при первом прочтении часть данного материала можно пропустить, впоследствии к нему все же нужно вернуться: это поможет вам понимать многие сообщения об ошибках, разбираться в конструкторских решениях, заложенных в эти библиотеки, и разрабатывать более качественные приложения.

Два стиля программирования

Программирование на уровне типов в Haskell является источником множества идей, касающихся совершенствования специального полиморфизма. Существуют два разных способа частичной имитации зависимой типизации в Haskell:

- *Функциональные зависимости* (Functional Dependencies, FD) позволяют ограничивать параметры классов типов. При правильно наложенных ограничениях Haskell-компилятор может вывести уникальный тип, который можно рассматривать как результат выполнения функции на уровне типа.
- *Семейства типов* (Type Families, TF) позволяют создавать функции, присваивающие тип, заданный множеством других типов. Последние версии GHC включают две разновидности семейств типов: *закрытые семейства типов* (closed type families) ближе всего к интуитивному понятию функции типа. В то же время *открытые семейства типов* (open type families) похожи на классы типов в том смысле, что в любой момент к открытому семейству типов можно добавить новое правило, точно так же, как к классу типов можно добавить экземпляр.

Оба способа обладают сопоставимой эффективностью, поэтому в принципе все равно, какой из них вы выберете для кодирования своих инвариантов. В одних случаях проще использовать функциональные зависимости, в других — семейства типов, поскольку у них разные синтаксис и акцент на некоторые концепции. Как правило, начинают с семейств типов, поскольку раскрываемые ими концепции на уровне типов ближе к простому уровню Haskell, а функциональные зависимости используют для более выразительной реализации отношений между типами.

ВНИМАНИЕ

Хотя у этих способов примерно равная эффективность, смешивать функциональные зависимости и семейства типов в одном и том же коде не стоит, поскольку это может привести к серьезным осложнениям. То есть если применяемая вами библиотека экспонирует функциональные зависимости, то и вы должны во избежание дальнейших проблем использовать функциональные зависимости, а если она экспонирует семейства типов, то и вы должны задействовать тот же прием. Хотя в новых версиях компилятора GHC ситуация может улучшиться.

Представление натуральных чисел

Поскольку теги, применяемые для проверки правил подарков, являются натуральными числами, перед тем, как воспользоваться функциональными зависимостями или семействами типов, для их представления на уровне типов нам нужно узнать о том, как они кодируются в качестве значений. В данном разделе вам предлагается краткое введение в тему натуральных чисел: если вы уже знакомы со стандартным типом данных для представления натуральных чисел и порядком кодирования сложения, вывода максимума и минимума, можете этот раздел спокойно пропустить.

Наиболее распространенный способ представления натуральных чисел в качестве типа данных основывается на аксиомах, выведенных математиком XIX века Джузеппе Пеано. В частности, он вывел два правила для построения таких чисел:

- *Нуль* является натуральным числом.
- Для каждого натурального числа *n* существует *число*, *следующее за n*, которое также является натуральным числом.

Эти аксиомы можно закодировать в объявлении Haskell-данных:

```
data Number = Zero | Succ Number deriving Show
```

Например, число «один» следует за нулем; число «два» следует за числом «один», и т. д.

```
one :: Number
one = Succ Zero
two :: Number
two = Succ one  -- Succ (Succ Zero)
```

ПРИМЕЧАНИЕ

Такое кодирование натуральных чисел можно представить в виде списков, в которых не нужно заботиться об элементах. Тогда сложение будет означать конкатенацию списков, получение минимума — возвращение списка с наименьшим количеством элементов и т. д.

Перейдем к операциям. Сначала нам понадобится сложение. Как и для многих других Haskell-типов, наилучшим способом создания функции, работающей с данными типа `Number`, является применение каждого конструктора и рекурсия вложенных значений. В данном случае у нас есть два разных конструктора:

- Если происходит сложение нуля и любого натурального числа y, результатом будет y.
- Если происходит сложение числа, следующего за x, и числа y, это эквивалентно числу, следующему за результатом сложения чисел x и y. Поскольку следующее число эквивалентно прибавлению единицы, это можно рассматривать как алгебраический закон, который читается так:

$$(x + 1) + y = (x + y) + 1.$$

В Haskell-синтаксисе ветви функции `plus'` записываются следующим образом:

```
plus' :: Number -> Number -> Number
plus' Zero     y = y
plus' (Succ x) y = Succ (plus' x y)
```

Работу функции можно протестировать путем сложения 1 и 2, например:

```
*Chapter13.Numbers> plus' one two
Succ (Succ (Succ Zero))
```

Как видно в выводе, результатом будет число 3, представленное таким необычным способом.

Для вычисления максимума также есть два варианта. Прежде всего, любое из чисел может быть нулем, в таком случае мы точно знаем, что другое число больше или равно нулю. Другой вариант — когда оба числа являются следующими числами, например $x + 1$ и $y + 1$. В этом случае максимум может быть вычислен путем обращения функции к x и y с последующим добавлением единицы.

```
max' :: Number -> Number -> Number
max' Zero y = y
max' x Zero = x
max' (Succ x) (Succ y) = Succ (max' x y)
```

Каркас функции по вычислению минимума очень похож на предыдущий, а для написания всего ее кода придется выполнить упражнение 13.5.

УПРАЖНЕНИЕ 13.5. МИНИМУМ НАТУРАЛЬНЫХ ЧИСЕЛ

Напишите функцию min' типа Number -> Number -> Number, которая вычисляет минимальное значение двух натуральных чисел, заданных в качестве аргументов.

Функциональные зависимости

Как утверждалось в предыдущем разделе, *функциональные зависимости* в Haskell — это один из инструментов кодирования операций на уровне типов. Однако изначально они предназначались для усовершенствования механизма класса типов путем ограничения набора типов, которые могут быть связаны родством через класс типов с несколькими параметрами. В данном разделе мы обсудим их первоначальное назначение, а затем перейдем к программированию правил подарков посредством функциональных зависимостей.

Определение категорий товаров с помощью функциональных зависимостей

Давайте ненадолго отклонимся от темы языка предложений и сконцентрируемся на совершенно другой проблеме. До сих пор все товары в магазине машин времени были представлены с использованием одного и того же типа данных `Product`. Однако информация для описания машины времени не похожа на ту, которая нужна для описания книги или костюма. Поэтому нам может быть интересно создание не только класса типов `Product`, но и типов данных, представляющих различные категории товаров.

У разных товаров не только разные поля для описания, они используются поразному. Например, на машине времени мы путешествуем, а книгу читаем. Было бы интересно указать для каждой категории товаров, какие операции над ней можно

совершать. Затем можно было бы включить определенные инструкции о том, как выполнять каждую операцию, а также спецификацию того, какая операция должна использоваться для тестирования товара.

Из данных размышлений можно сделать вывод, что нужно создать класс типов `Product` с двумя параметрами: один будет определять категорию товаров, а другой — операции, поддерживаемые для этой категории. В следующий код в качестве операций включены цена товара и упомянутые операционные функции:

```
class Product p op where
  price :: p -> Float
  perform :: p -> op -> String
  testOperation :: p -> op
```

ПРИМЕЧАНИЕ

Для того чтобы этот код был принят GHC, не забудьте включить для него расширение MultiParamTypeClasses.

С учетом простого типа данных для представления машин времени и операций над ними написать объявление экземпляра не составит труда. В показанном здесь коде демонстрируется возможный вариант решения данной задачи:

```
data TimeMachine = TimeMachine { model :: String } deriving Show

data TimeMachineOps = Travel Integer | Park deriving Show

instance Product TimeMachine TimeMachineOps where
  price _ = 1000.0
  perform (TimeMachine m) (Travel y) = "Travelling to " ++ show y ++ " with " ++ m
  perform (TimeMachine m) Park       = "Parking time machine " ++ m
  testOperation _ = Travel 0
```

Разумеется, главная цель создания класса типов заключается в написании функции, способной работать с любым видом `Product`. Например, одна функция может получать суммарную цену товаров одной категории, а другая проводить тестовую операцию с конкретным товаром. Их определения имеют следующий вид:

```
totalAmount :: Product p op => [p] -> Float
totalAmount = foldr (+) 0.0 . map price

performTest :: Product p op => p -> String
performTest p = perform p $ testOperation p
```

Проблема в том, что этот код не компилируется. Вместо этого выводится несколько сообщений об ошибках, например таких:

```
src/Chapter13/CategoriesFnDeps.hs:
    Could not deduce (Product p op0) arising from a use of `price'
    from the context (Product p op)
      bound by the type signature for
                 totalAmount :: Product p op => [p] -> Float
      at src/Chapter13/CategoriesFnDeps.hs:
    The type variable `op0' is ambiguous
    Possible fix: add a type signature that fixes these type variable(s)
```

```
    Note: there is a potential instance available:
      instance Product TimeMachine TimeMachineOps
        -- Defined at src/Chapter13/CategoriesFnDeps.hs:18:10
```

В сообщениях об ошибках говорится о недостатке информации для вывода типа каждой операции, соответствующего категории каждого товара. Тем не менее суть понятна: для `TimeMachine` соответствующим типом операции является `TimeMachineOps`, и никакой другой. Компилятор должен каким-то образом прийти к выводу, что при использовании `TimeMachine` операции всегда будут принадлежать к типу `TimeMachineOps`.

Проблема в том, что объявление класса типов в своем нынешнем виде никоим образом не подразумевает этот вывод. Через `Product` могут быть связаны любые два типа `p` и `op`. Например, можно добавить экземпляр, использующий в качестве категории `TimeMachine`, со следующими операциями, проводимыми над книгами:

```
data Book = Book { title :: String, author :: String, rating :: Integer } deriving
Show

data BookOps = Read | Highlight | WriteCritique deriving Show

instance Product TimeMachine BookOps where
  price _ = 500.0
  perform _ _ = "What?!"
  testOperation _ = Read -- ??
```

Компилятор возражать не будет, поскольку к коду нет никаких претензий. Однако теперь можно понять, почему определения `totalAmount` и `testOperation` были неоднозначными: потенциально может применяться другой экземпляр, где объявление отличается от `Product TimeMachine TimeMachineOps`.

Решение состоит в добавлении к объявлению класса типов ограничения, которое четко выразит, что в данной конкретной категории товаров для множества операций доступна только одна возможность. Делается это через *функциональную зависимость*, — данное понятие взято из теории баз данных, в точности описывающей такие сценарии. Функциональные зависимости записываются в заголовке объявления `class` и отделяются от переменных имен и типов символом `|`, а между ними ставятся запятые. Каждая функциональная зависимость в свою очередь следует той же схеме: $x_1 \ldots x_n$ `->` $y_1 \ldots y_m$, выражая тот факт, что для каждого уникального замещения типов от x_1 до x_n имеется только одно возможное составное значение от y_1 до y_m. Следует заметить, что для использования данного синтаксиса в своем типовом классе нужно включить расширение `FunctionalDependencies`.

В данном случае категории накладывают ограничения на операции, поэтому добавляемой функциональной зависимостью будет `p -> op`. Уточненный заголовок определения класса типов должен выглядеть так:

```
class Product p op | p -> op where
```

После того как это будет сделано, компилятор возмутится из-за двух разных экземпляров `Product`, заданных для `TimeMachine`:

```
src/Chapter13/CategoriesFnDeps.hs:
  Functional dependencies conflict between instance declarations:
```

```
instance Product TimeMachine TimeMachineOps
  -- Defined at src/Chapter13/CategoriesFnDeps.hs
instance Product TimeMachine BookOps
  -- Defined at src/Chapter13/CategoriesFnDeps.hs
```

Теперь компилятор допускает определение `totalPrice` и `performTest`, поскольку он знает, что для заданной категории товаров будет доступно только одно множество операций, поэтому он сможет их выбрать.

Если разобраться, функциональные зависимости весьма полезны, когда в них возникает необходимость в объявлении класса типов. Их использование в классах монад описывается в следующей врезке. А следующее далее упражнение 13.6 призвано закрепить понимание данных идей.

ФУНКЦИОНАЛЬНЫЕ ЗАВИСИМОСТИ В КЛАССАХ МОНАД

Когда классы типов поддерживают подтягивание основных операций в каждом из преобразователей монад (MonadState, MonadReader, MonadWriter и т. д.), о чем рассказывалось в главе 7, в объявлениях класса появляются функциональные зависимости, но до сих пор мы не знали почему.

Суть такого класса типов, как MonadState, состоит в том, что в нем объявляется как монада, выполняющая операции, так и тип элементов, сохраненных в виде состояния, поскольку то и другое требует сигнатур некоторых операций. Функциональная зависимость устанавливает, что для заданной конкретной монады тип переменных состояния известен автоматически. Возьмем, к примеру, монаду State Int: из ее сигнатуры уже заранее известно, что типом элементов, сохраненных в виде состоянии, может быть только Int.

УПРАЖНЕНИЕ 13.6. ТОВАРЫ И ПАКЕТЫ

В магазине доступны две разновидности пакетов: большие и малые. Создайте для представления каждого вида пакетов два новых типа данных, BigBag и SmallBag. Затем добавьте новый параметр к типовому классу Product, чтобы установить, какой из пакетов должен использоваться для каждой категории товаров. В принципе, машины времени должны попадать в большие пакеты, а книги — в малые. Тщательно продумайте дополнительные функциональные зависимости.

Задание правила подарков с помощью функциональных зависимостей

На первый взгляд кажется, что в Haskell функциональные зависимости неприменимы к операциям на уровне типов. Однако если присмотреться, можно увидеть, как с их помощью можно кодировать функции на уровне типов.

Для начала давайте создадим на уровне типов представление натуральных чисел. И здесь опять лучше всего задействовать пустые типы данных. Для каждого конструктора в исходном объявлении нужен пустой тип данных. В случае с числами мы используем два из них: один для представления нулевого тега, второй для последующих чисел.

```
{-# LANGUAGE EmptyDataDecls #-}

data Zero
data Succ n
```

ВНИМАНИЕ

Следует учесть, что почти в каждом примере этого раздела будет встречаться все больше и больше GHC-расширений, которые задаются с помощью директивы транслятора LANGUAGE. Большинство расширений нужны лишь для включения конкретных синтаксических конструкций, а если вы забудете это сделать, компилятор напомнит вам об этом, поэтому особо волноваться незачем.

Располагая лишь теми типами данных, которые мы только что задали, можно представить списки, снабженные тегами, указывающими на их длину, как это было в Idris, за исключением того, что вместо встроенных чисел будут использоваться эти новые натуральные числа. Для проверки того факта, что любой вызов функции `head` производится со списком из, как минимум, одного элемента (что делает язык более безопасным), уже могут использоваться типы данных `Zero` и `Succ`. Следуя Idris-соглашениям, эти списки, снабженные тегами с числами, будут называться Vect-списками:

```
{-# LANGUAGE GADTs #-}

data Vect n a where
  VNil :: Vect Zero a
  VCons :: a -> Vect n a -> Vect (Succ n) a
```

Пришло время вывести на первый план систему классов типов. Каждая операция на уровне типов кодируется как класс типов, у которого в качестве входных аргументов операции на уровне типов будут переменные, а дополнительная переменная представит результат операции. Например, `class Plus x y z` представляет следующее: «результат сложения x и y равен z», или в других понятиях: $x + y = z$. Но чтобы получить функцию, нужно явно указать, что для любой пары значений x и y есть только один возможный результат z. Для такого указания лучше всего подойдет функциональная зависимость. Таким образом, все объявление класса типов, представляющее сложение на уровне типов, должно выглядеть так:

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}

class Plus x y z | x y -> z
```

ПРИМЕЧАНИЕ

Поскольку класс типов Plus используется только для своих результатов на уровне типов, включать в его тело какую-либо функцию нет необходимости. В данном случае Haskell позволяет опускать в объявлении ключевое слово where.

Объявление класса типов просто описывает количество аргументов функции на уровне типов. Чтобы выразить правила, составляющие операцию, нужно написать разные экземпляры. Обычно эти экземпляры соответствуют каждому из случаев в определении функции. Взгляните, как это выглядит для сложения:

```
{-# LANGUAGE FlexibleInstances, UndecidableInstances #-}

instance Plus Zero x x
instance Plus x y z => Plus (Succ x) y (Succ z)
```

В этом коде выражена та же логика, которая была предложена Джузеппе Пеано для сложения, но в обратном стиле рассуждения. В первом экземпляре закодировано

правило сложения нуля с числом, чей результат будет точно таким же, как второй аргумент. Второе правило немного сложнее: оно говорит, что если вы знаете, что $x + y = z$, то можете логически вывести сложение $(x + 1) + y$, результатом которого несомненно будет $z + 1$. В некотором смысле объявление реверсировало способ записи сопоставления с образцом в отношении аргументов и обработки рекурсии посредством обращения к меньшему экземпляру.

Следующим шагом станет использование этих новых типов данных и класса типов внутри объявления `Offer`. Основные комбинаторы для подарков и скидок просто переносятся из их Idris-объявлений в Haskell: единственное отличие состоит в том, что вместо обычных чисел нужно писать их Пеано-представления.

```
data Offer a p where
Present          :: a -> Offer a (Succ Zero)
PercentDiscount  :: Float -> Offer a Zero
AbsoluteDiscount :: Float -> Offer a Zero
```

Конструктором, использовавшим сложение в Idris, был `Both`. Соответствующий Haskell-код выглядит так:

```
Both :: Plus p q r => Offer a p -> Offer a q -> Offer a r
```

Как видите, код включает контекст с классом типов `Plus`. Тем самым выражается, что если $p + q = r$, то объединение предложения с p подарками и еще одного предложения с q подарками становится предложением с r подарками. Если в интерпретаторе проверить тип тестируемого подарка, компилятор для получения финального типа последует правилам для класса типов `Plus`. Вот как выглядит один из примеров вывода, соответствующий тому же предложению, которое было протестировано в Idris:

```
*Chapter13.CheckPresentsFnDeps> :t let p = Present 'a' in Both p (Both p (Both p p))
let p = Present 'a' in Both p (BetterOf p (Both p p))
  :: Offer Char (Succ (Succ (Succ (Succ Zero))))
```

Еще одним интересным конструктором является ограничение, которое должно выбирать минимум между количеством элементов в списке и количеством подарков. Давайте повторим те же шаги получения функциональных зависимостей для кодирования этой операции на уровне типов. Сначала объявим класс типов с дополнительным аргументом для результата и укажем, что последний аргумент функционально зависит от всех остальных:

```
class Min x y z | x y -> z
```

Для каждого правила в определении функции нужно включить объявление экземпляра. Не забудьте, что стиль программирования должен быть обратным — перед полным результатом нужно указать повторяющиеся условия:

```
instance Min Zero y Zero
instance Min (Succ x) Zero Zero
instance Min x y z => Min (Succ x) (Succ y) (Succ z)
```

Финальная часть использует в объявлении конструктора класс типов. Как это было в Idris, код запрашивает число с как минимум одним последующим числом, чтобы ограничиться списками с как минимум одним элементом.

```
{-# LANGUAGE FlexibleContexts #-}

Restrict :: Min (Succ n) p r => Vect (Succ n) a -> Offer a p -> Offer a r
```

Однако для реализации полноценного языка предложений нам не хватает еще нескольких конструкторов. В упражнении 13.7 предлагается написать недостающие конструкторы, кодируя на уровне типов функции, которые необходимы в качестве функциональных зависимостей. Предложения и списки не являются единственными типами данных, которые могут выиграть от их снабжения тегами с натуральными числами. Как уже объяснялось, в Idris деревья могут кодировать свою высоту. В упражнении 13.8 предлагается реализовать то же самое в Haskell.

УПРАЖНЕНИЕ 13.7. ПРЕДЛОЖЕНИЯ С ФУНКЦИОНАЛЬНЫМИ ЗАВИСИМОСТЯМИ

Включите конструкторы для остальной части языка предложений: From, Until, Extend, BetterOf и If. В последних двух вам понадобится класс типов, кодирующий максимум два натуральных числа: напишите его, следуя приему, представленному в данном разделе.

ЛОГИЧЕСКИЙ ЯЗЫК НА УРОВНЕ ТИПОВ

Если вам когда-либо приходилось использовать язык логического программирования, например Prolog или Datalog, то можно найти некоторое сходство между способом кодирования функций на уровне типов с объявлениями экземпляров и тем, как пишутся предикаты на этих языках. Такая связь действительно имеет место: программирование с функциональными зависимостями демонстрирует в системе типов Haskell стиль логического программирования.

Эта привнесенная в Haskell дополнительная парадигма является объектом основной критики функциональных зависимостей: программист не должен менять стиль программирования при переходе от кодирования на уровне понятий к кодированию на уровне типов. Далее вы увидите, что в объявлениях семейств типов присутствует более функциональный стиль.

УПРАЖНЕНИЕ 13.8. ДЕРЕВЬЯ С ФУНКЦИОНАЛЬНЫМИ ЗАВИСИМОСТЯМИ

Создайте тип BinaryTree с такими же конструкторами, что и в упражнении 13.4. В данном случае для снабжения деревьев тегами с их высотой задействуйте комбинации GADT-типов и операций на уровне типов с функциональными зависимостями. Подсказка: воспользуйтесь натуральными числами из этого раздела и функцией вычисления максимума из упражнения 13.7.

Семейства типов

Мы уже видели, что функциональные зависимости могут помочь в построении более строгой системы типов. В этом разделе показано, как выражаются те же разновидности инвариантов, но уже на языке *семейств типов*. Если не вдаваться в детали, то семейство типов является функцией уровня типов: она дает нам тип при задании в качестве параметров каких-либо других типов, но особым способом, а не так, как при параметрическом полиморфизме. Эта функция может располагаться на верхнем уровне вашего модуля или внутри класса типов: в этом разделе описано назначение каждого из вариантов.

Задание правил подарков с помощью семейств типов

Определение семейства типов обычно не сложнее соответствующего определения на уровне понятий. В примере, с которым мы работали, первой востребованной функцией уровня типов была функция сложения, поэтому продолжим работу именно с ней. С семействами типов будут использоваться те же типы данных, которые использовались для кодирования натуральных чисел в разделе, посвященном функциональным зависимостям:

```
{-# LANGUAGE EmptyDataDecls #-}

data Zero
data Succ n
```

Также мы снова задействуем тип данных `Vect`:

```
{-# LANGUAGE GADTs #-}

data Vect n a where
  VNil :: Vect Zero a
  VCons :: a -> Vect n a -> Vect (Succ n) a
```

Семейство типов начинается с ключевых слов `type family`, затем следует объявление имени семейства и аргументов, после которых указывается ключевое слово `where`. В этой первой строке определяется сигнатура функции уровня типов. После сигнатуры семейство типов определяется путем сопоставления с образцом в отношении его аргументов. Между семействами типов и обычными функциями есть два небольших синтаксических различия: в то время как функции уровня понятий должны начинаться с буквы в нижнем регистре, имена семейств типов должны начинаться с буквы в верхнем регистре. В данном случае код будет ссылаться на сложение уровня типов с именем `Plus`. Другое различие состоит в том, что все правила для семейства типов должны иметь отступы, этим они отличаются от обычных функций, где правила располагаются с теми же отступами, что и сигнатуры. Определение сложения с использованием семейств типов имеет следующий вид:

```
{-# LANGUAGE TypeFamilies #-}

type family Plus x y where
  Plus Zero x = x
  Plus (Succ x) y = Succ (Plus x y)
```

Как видите, экземпляры типа полностью имитируют определение функции `(+)` для обычных (на уровне понятий) натуральных чисел. Если используются семейства типов, эта схожесть упрощает перенос кода с уровня понятий на уровень типов.

ВНИМАНИЕ

В предыдущем коде используются закрытые семейства типов, а это расширение доступно в компиляторе GHC только начиная с версии 7.8.1. Если вы работаете с предыдущей версией, нужно задействовать открытые семейства типов, описываемые в следующем разделе.

Следующим шагом станет использование только что объявленного семейства типов внутри определения типа данных `Offer`. Как и в случае функциональных

зависимостей, единственным изменением в основных конструкторах из Idris-кода будет Пеано-представление нуля и чисел нашего типа данных:

```
data Offer a p where
  Present          :: a -> Offer a (Succ Zero)
  PercentDiscount  :: Float -> Offer a Zero
  AbsoluteDiscount :: Float -> Offer a Zero
```

Местом ожидаемого применения семейства типов `Plus` является конструктор `Both`. При использовании семейства типов та часть контекста типа, которая относится к сигнатуре, не нужна; просто там, где ожидается тип, можно указать семейство типов. Например, в данном конструкторе `Plus p q` оно заменяет тип:

```
Both :: Offer a p -> Offer a q -> Offer a (Plus p q)
```

С целью подкрепления шагов, которые необходимо предпринять для использования семейства типов при кодировании функций на уровне типов, давайте для краткости повторим весь процесс. Если вернуться к предыдущим разделам, можно увидеть, что в данном случае следует определить на уровне типов функцию минимума. Именно это и делает следующий код с помощью семейства типов `Min`:

```
type family Min x y where
  Min Zero y = Zero
  Min x Zero = Zero
  Min (Succ x) (Succ y) = Succ (Min x y)
```

Соответствующий конструктор `Restrict` с помощью семейств типов переделывается из Idris- в Haskell-код очень просто:

```
Restrict :: Vect (Succ n) a -> Offer a p -> Offer a (Min (Succ n) p)
```

Как и в предыдущем коде, перед вами в упражнении 13.9 ставится задача написать код для всех остальных типов данных.

УПРАЖНЕНИЕ 13.9. ПРЕДЛОЖЕНИЯ С СЕМЕЙСТВАМИ ТИПОВ

Для остальной части языка предложений включите конструкторы From, Until, Extend, BetterOf и If. На данный момент нам уже известно, что для двух последних случаев на уровне типов необходима функция получения максимума двух натуральных чисел: определите ее с использованием семейств типов.

Создание категорий товаров с помощью семейств типов

При знакомстве с программированием на уровне типов в Haskell утверждалось, что у функциональных зависимостей и семейств типов примерно одинаковая выразительность. Мы уже видели, как с помощью обеих этих технологий можно кодировать функции на уровне типов, но все же остается открытым вопрос о том, как использовать семейства типов для решения проблемы категорий товаров.

Одной из возможностей является определение семейства типов, назначающего тип операций каждому типу товаров. Если последовать инструкциям из предыдущего раздела, у такой функции, как `Operation`, будет следующий вид:

```
type family Operation x where
  Operation TimeMachine = TimeMachineOps
  Operation Book = BookOps
```

Но это решение нельзя признать целиком удовлетворительным. Когда наряду с функциональной зависимостью применялся класс типов, у нас была возможность в любой момент добавить новый товар с его множеством операций, воспользовавшись для этого новым объявлением `instance`. Однако если следовать показанному подходу с определением семейства типов, нам необходимо добавить не только экземпляр к типовому классу `Product`, но и новое уравнение к семейству типов `Operation`. Это приведет к тому, что класс типов и семейство типов окажутся тесно связанными, что затрудняет их поддержку.

Проблема в данном случае относится не к использованию семейств типов, а к тому факту, что семейство типов определено как *закрытое*. Начиная с версии 7.8 компилятором GHC поддерживаются две разновидности семейств типов: закрытые и открытые. Закрытое семейство типов, с которым мы только что познакомились, не может быть после определения расширено дополнительными правилами. Этим оно отличается от *открытого семейства типов*, определяющего частичную функцию, которая может быть уточнена или дополнена в других частях кода.

Чтобы определить семейство типов в качестве открытого, из его определения нужно убрать ключевое слово `where`:

```
type family Operation x
```

Когда к семейству типов потребуется добавить новое правило, нужно указать его после ключевых слов `type instance`. Например, предыдущие два отношения между товарами и операциями приобретут тогда следующий вид:

```
type instance Operation TimeMachine = TimeMachineOps
type instance Operation Book        = BookOps
```

Обратите внимание на то, что в данном случае объявления экземпляров располагаются на том же уровне, что и сигнатуры, поскольку они могут объявляться в совершенно разных модулях.

ЭКЗЕМПЛЯРЫ ТИПОВ НЕ ДОЛЖНЫ ПЕРЕКРЫВАТЬСЯ

Между закрытыми и открытыми семействами типов есть важное различие. Закрытые семейства типов следуют обычной модели вычислений: сначала предпринимается попытка применения первого правила, а если соответствие не обнаруживается, предпринимается попытка применения второго правила и так далее, до тех пор, пока соответствие в конечном итоге не будет найдено. Такая возможность предоставляется в силу того, что у правил из закрытого семейства типов имеется определенный порядок применения. Однако правила из открытого семейства типов поступают из разных не связанных друг с другом мест и не имеют определенного порядка применения. Если компилятор обнаружит удачные сопоставления сразу с двумя образцами, он не сможет понять, какое из правил выбрать. Таким образом, GHC требует, чтобы объявления type instance не перекрывались.

Возьмем в качестве примера показанное ранее определение семейства типов Min. Если компилятор должен вычислить результат Min Zero Zero, то соответствовать могли бы оба правила: и первое, и второе. Он знает, что должен воспользоваться первым, поскольку семейство типов является закрытым. Однако если бы вам пришлось определить Min как открытое семейство типов (например, по причине того, что ваша версия GHC ниже 7.8.1), вам потребовалось бы уточнить объявление, чтобы экземпляры, объявляемые с помощью ключевых слов type instance, не перекрывались. Этого можно добиться, вводя в действие второе правило, только если первый аргумент больше нуля:

```
type instance Min Zero y            = Zero
```

```
type instance Min (Succ x) Zero     = Zero
type instance Min (Succ x) (Succ y) = Succ (Min x y)
```

Теперь каждое применение открытого семейства типов Min подразумевает соблюдение одного, и только одного правила.

Во многих случаях при использовании функциональных зависимостей может потребоваться создание функции на уровне типов, но это также может стимулировать у каждого реализатора класса типов желание добавить к этой функции новое правило. Или (с другой точки зрения) добавить в класс типов функцию уровня типов. Когда включается расширение `TypeFamilies`, в объявления класса типов можно включать сигнатуры функций как на уровне понятий, так и на уровне типов, если того потребуют обстоятельства. Сигнатуры уровня типов внутри классов типов известны как *ассоциированные типы* (associated types).

ПРИМЕЧАНИЕ

Не забудьте, что у типов имеется очень простая система видов (единственными возможными видами являются вид * и виды, подобные функциям, такие как * -> *), которая проверяет правильность применения конструкторов типов. В том случае, если вы не помните всех подробностей, можете обратиться к первой части данной книги, в которой представлена система видов.

Давайте посмотрим, как переделать[1] класс типов `Product` для использования ассоциированных типов вместо функциональных зависимостей. Если вспомнить исходное определение, то переменная `p` представляла там категорию товаров, а переменная `op` — операции для этой категории. Последняя имела функциональную зависимость от первой, поэтому у нас есть неплохой кандидат на переделку в ассоциированный тип. Таким образом, в новом коде убирается параметр `op` и добавляется функция уровня типов под названием `Operation`:

```
class Product p where
  type Operation p :: *
  price :: p -> Float
  perform :: p -> Operation p -> String
  testOperation :: p -> Operation p
```

Как видите, все появления `op` в старом объявлении теперь заменены выражением `Operation p`, возвращающим тип, соответствующий операциям. Реализация `Operation` должна также появиться в каждой реализации экземпляра класса типов `Product`, как в следующей реализации экземпляра для `TimeMachine`:

```
instance Product TimeMachine where
  type Operation TimeMachine = TimeMachineOps
  price _ = 1000.0
  perform (TimeMachine m) (Travel y) = "Travelling to " ++ show y ++ " with " ++ m
  perform (TimeMachine m) Park = "Parking time machine " ++ m
  testOperation _ = Travel 0
```

[1] Показанный далее код не сможет функционировать в модуле введенного ранее семейства типов `Operation`. При попытке определения `Operation` внутри типового класса одновременно с семейством типов `Operation` произойдет конфликт имен, в результате компилятор не сможет продолжить свою работу.

Как и в предыдущем разделе, тип пакета, в котором нуждается каждая из категорий товаров, должен быть закодирован на уровне типов. Эту задачу, но теперь уже с использованием ассоциированных типов, вам предлагается решить при выполнении упражнения 13.10.

УПРАЖНЕНИЕ 13.10. И СНОВА ТОВАРЫ И ПАКЕТЫ

Используя ранее определенные типы данных BigBag и SmallBag, создайте представление вида пакетов, необходимых для каждой категории товаров. Добавьте новый ассоциированный тип к типовому классу Product. В заключение включите экземпляры, выражающие тот факт, что машины времени должны помещаться в большие пакеты, а книги — в малые.

В простом сценарии, в котором один из типов полностью зависит от всех остальных переменных в типовом классе (как операции и пакеты в данном разделе), ассоциированные типы обычно делают более явным тот факт, что в каждом экземпляре может быть выбрана только одна возможность. Однако функциональные зависимости тоже не нужно игнорировать, поскольку они позволяют выразить зависимость еще более ярко.

Семейства типов привносят в систему типов еще одно, последнее, дополнение. Рассмотрим следующую функцию `performTestFromOther`, которая выполняет тестовую операцию на совершенно другом товаре:

```
performTestFromOther p q = perform p $ testOperation q
```

Теперь задача заключается в предоставлении сигнатуры типа, имеющей как можно более общий характер, что позволяет применять функцию в максимально широком спектре ситуаций. Первым приближением станет требование о том, чтобы значения `p` и `q` были одного и того же типа, что может быть реализовано классом типов `Product`. Разумеется, если попытаться добавить к функции следующую сигнатуру типа, то компилятор примет это определение:

```
performTestFromOther :: Product p => p -> p -> String
```

Однако данная сигнатура накладывает чрезмерные ограничения: для получения точно такого же типа оба аргумента не нужны. Нужна лишь поддержка одних и тех же операций, то есть совпадение их ассоциированного типа `Operation`. Такой вид оснастки может быть выражен в Haskell с помощью *ограничения в виде равенства* (equality constraint) `x ~ y`, которое определяет, что `x` и `y` должны после разрешения всех семейств типов быть синтаксически равны. Тогда для данного случая наиболее общим ограничением будет следующее:

```
performTestFromOther :: (Product p, Product q, Operation p ~ Operation q) => p ->
q -> String
```

Вообще-то если попытаться скомпилировать код без предоставления явно выраженной сигнатуры, GHC предложит в качестве подсказки именно этот наиболее общий тип. Можете напрямую скопировать эту сигнатуру в код, убрав начальную часть `forall`, которая подразумевалась для каждой свободной переменной в сигнатуре. Например:

```
Warning: Top-level binding with no type signature:
  performTestFromOther :: forall p p1.
                          (Product p, Product p1, Operation p ~ Operation p1) =>
                          p -> p1 -> String
```

Ограничения в виде равенства встречаются в написанном вручную коде довольно редко, но они важны для понимания сообщений об ошибках, генерируемых GHC. В большинстве экземпляров, где семейства типов не совпадают, компилятор будет предупреждать вас о несоблюдении ограничений в виде равенства (содержащих символ ~).

СЕМЕЙСТВА ДАННЫХ

Если быть до конца точным, в предыдущем разделе были представлены не семейства типов, а их подкласс, называемый семействами синонимов типов (type synonym families). В случае применения этого подкласса его нужно использовать как в типах параметров, так и в типах возвращаемых выражений, которые определены где-то в другом месте.

Есть еще один вид семейств типов, который называется семейством данных (data family). В данном случае вместо использования другого типа непосредственно перечисляются конструкторы для каждого экземпляра семейства. Например, класс типов Product и его экземпляр TimeMachine могут быть определены в следующем виде:

```
class Product p where
  data Operation2 p
  price :: p -> Float
  perform :: p -> Operation2 p -> String
  testOperation :: p -> Operation2 p

instance Product TimeMachine where
  data Operation2 TimeMachine = Travel Integer | Park
  price _ = 1000.0
  perform (TimeMachine m) (Travel y) = "Travelling to " ++ show y ++ " with " ++ m
  perform (TimeMachine m) Park = "Parking time machine " ++ m
  testOperation _ = Travel 0
```

Однако семейства синонимов типов и семейства данных не являются взаимозаменяемыми, поскольку они обладают разными свойствами (например, семейства данных могут применяться частично, а семейства синонимов типов нет).

Многие технологии, рассмотренные в данной главе, используются в Hackage-библиотеках с целью придания более весомых гарантий работоспособности выполняемого кода. Мы уже задействовали ассоциированные типы, когда составляли описание схемы в библиотеке Persistent[1], но тогда мы об этом не знали. Еще одним интересным пакетом для программирования на уровне типов является пакет HList, позволяющий создавать списки элементов разных типов.

Продвижение типа и одноэлементные типы

Начиная с версии 7.4.1 GHC поддерживает новую концепцию, которая называется *продвижением типа данных* (data type promotion). При включении соответствующего расширения для каждого типа данных уровня понятий на уровне типов создается новое множество типов (то есть этот тип *продвигается*). Кроме того, с каждым из этих продвинутых типов данных ассоциируется их новый вид, что обеспечивает

[1] Создаваемые определения можно проверить, запустив GHC с ключом -ddump-splices.

безопасное программирование на уровне типов. В данном разделе мы узнаем, как использовать эту возможность и как помимо типов данных продвигать функции (с помощью пакета `singletons`).

Дальнейшее уточнение правила подарков

Давайте начнем реализацию правила подарков с самого начала (если вы пишете код по мере чтения книги, начните с нового пустого модуля). Но теперь вместо создания пустых типов данных для представления нуля и последующих чисел мы применим в коде обычное объявление данных, а в компиляторе включим расширение `DataKinds`. Код должен выглядеть так:

```
{-# LANGUAGE DataKinds #-}

data Nat = Zero | Succ Nat
```

Если в этот же файл включить объявление списков, снабженных тегами их длины (как в разделе, посвященном функциональным зависимостям), код успешно откомпилируется. В качестве напоминания приведем определение типа данных `Vect`:

```
{-# LANGUAGE GADTs #-}

data Vect n a where
  VNil :: Vect Zero a
  VCons :: a -> Vect n a -> Vect (Succ n) a
```

Похоже, что внутри типа был использован тип данных уровня понятий, как это было в языке с зависимой типизацией. Но тем самым игнорируется строгое разделение между типами и понятиями, характерное для языка Haskell. Истина в том, что когда включается расширение `DataKinds`, компилятор на уровне типов создает *копию* типа данных. Концептуально об исходном файле с расширением `DataKinds` и теперь можно думать, что он эквивалентен следующему коду:

```
data Nat = Zero | Succ Nat

data Zero
data Succ nat
```

Компилятор может различать конструкторы и имена типов, поскольку они существуют в разных мирах. В тех редких случаях, когда компилятор будет не в состоянии их различить, можно воспользоваться синтаксисом `'Идентификатор`, чтобы явно сослаться на `Идентификатор` уровня типов. Например, можно написать определение `Vect`, явно указав на то, что `Zero` и `Succ` являются продвинутыми типами:

```
data Vect n a where
  VNil :: Vect 'Zero a
  VCons :: a -> Vect n a -> Vect ('Succ n) a
```

Следующим шагом в направлении переноса кода в этот новый файл станет объявление семейства типов `Plus`, в котором будет закодировано сложение. Но если копировать код в том виде, в каком он был представлен в предыдущем разделе (включая директиву транслятору `TypeFamilies`), компилятор сообщит об ошибке:

```
src/Chapter13/CheckPresentsPromotion.hs::
  Kind mis-match
  The first argument of `Plus' should have kind `*',
  but `Zero' has kind `Nat'
  In the type `Zero'
  In the type instance declaration for `Plus'
```

Эта ошибка похожа на ошибку типа, но в сообщении об ошибке упоминается не тип, а вид (kind). Более того, имеется ссылка на вид `Nat`, который вообще не объявлялся в исходном файле. Виновно здесь опять расширение `DataKinds`, которое не только создает типы `Zero` и `Succ` для представления натуральных чисел в мире типов, но и продвигает тип `Nat` до статуса вида и подчиняет этому виду `Nat` члены `Zero` и `Succ` (вместо вида *, который предлагается по умолчанию).

ПРИМЕЧАНИЕ

Не забудьте, что виды служат для классификации типов так же, как типы служат для классификации значений. В Haskell без продвижения типов данных все применяемые типы данных относятся к виду *. Однако как показано в примере, указанное расширение открывает двери видам, определяемым пользователями.

По умолчанию в любом объявлении семейства типов исходным видом аргументов является *. Когда не используется никаких расширений на уровне типов, это существенно упрощает синтаксис, но в данном случае за нас делается неверный выбор. Объявление `Plus`, таким образом, эквивалентно следующему объявлению, где оператор `::` показывает вид переменной типа:

```
type family Plus (x :: *) (y :: *) :: * where
  Plus Zero x = x
  Plus (Succ x) y = Succ (Plus x y)
```

Теперь понятно, почему компилятор показал ошибку: `Zero` и `Succ x` относятся не к виду *, а к виду `Nat`. Можно успокоить компилятор, явно указав, что семейству типов требуются типы данного вида, просто изменив объявление:

```
type family Plus (x :: Nat) (y :: Nat) :: Nat where
```

Следующим шагом станет использование этого нового подхода. Например, в данный момент у нас нет никаких явных запретов на запись такого типа, как `Vect Int Char`. Но в этом нет никакого смысла: единственным возможным значением для первой переменной в `Vect` будет значение вида `Nat` (этот вид представляет числа). В объявлении типа это можно записать непосредственно:

```
data Vect (n :: Nat) a where
```

ПРИМЕЧАНИЕ

В данном случае никакого пояснения давать не нужно, поскольку компилятор способен логически вывести, что видом n должен быть Nat. Но все же в явном указании вида есть свой резон, хотя бы в целях документирования и гарантии того, что внесение изменений в код не заставит компилятор сделать логический вывод о принадлежности к другому типу.

Такой же подход можно применить к типу `Offer` — его параметр типа должен принимать значения только из натуральных чисел уровня типов. Соответствующее уточнение этого типа данных имеет следующее объявление:

```
data Offer a (p :: Nat) where
```

Как видите, расширение `DataKinds` привносит в программирование на уровне типов более высокий уровень безопасности, чем тот, который типы дают Haskell-программам на уровне понятий. Больше того, автоматическое продвижение упрощает объявление типов, которые будут использоваться при программировании на уровне типов. Тем не менее следует отметить, что система видов не настолько эффективна, как система типов: например, у нее нет ничего похожего на классы типов (что-то вроде *видовых классов*). Однако в основном для программирования на уровне типов этого расширения более чем достаточно.

Расширение `DataKinds` при всей своей реальной пользе не предлагает нам весь пакет функций. Типы данных могут продвигаться без каких-либо шероховатостей, но нам по-прежнему приходится определять свои функции на уровне типов с помощью либо функциональных зависимостей, либо семейств типов. Это означает, что для применения какой-либо функциональности, которая доступна на уровне понятий, необходимо дублировать код, причем делать это с использованием других стилей программирования в ущерб простоте чтения кода и удобству его сопровождения.

Практическое решение данной проблемы предоставляется переменными пакета `singletons`. Используя средства метапрограммирования, предлагаемые механизмом Template Haskell, библиотека этого пакета создает такие версии искомых функций уровня понятий, которые работают на уровне типов. Хотя эти два мира по-прежнему разделены, возникает иллюзия того, что на обоих уровнях без проблем используются одни и те же конструкторы и функции.

Чтобы начать работу с пакетом, его нужно добавить в проект в качестве зависимости и импортировать модуль `Data.Singletons.TH`[1] в исходный файл. Это расширение механизма Template Haskell блокирует пользовательские семейства типов и продвижение типов данных, поэтому кроме расширений, связанных с метапрограммированием, вам необходимо включить все нужные режимы в вашем исходном файле, в противном случае код не будет компилироваться.

Модуль предлагает массу функциональности, но наибольший интерес для нас представляет функция `promote`. С ее помощью можно создать версии уровня типов как для типов данных, так и для функций. Функции кодируются на уровне типов с помощью семейств типов, причем имя получается заменой первой буквы в имени функции со строчной на прописную. Например, в следующем примере показано, как продвинуть операции с натуральными числами, о которых шла речь в этой главе:

```
{-# LANGUAGE DataKinds, TypeFamilies, TemplateHaskell, QuasiQuotes #-}
import Data.Singletons.TH

$(promote [d|
  data Nat = Zero | Succ Nat
           deriving (Show, Eq)

  plus :: Nat -> Nat -> Nat
  plus Zero y = y
  plus (Succ x) y = Succ (plus x y)
```

[1] В версиях singletons, предшествующих версии 0.9, нужно вместо этого модуля импортировать модуль Data.Singletons.

```
  min :: Nat -> Nat -> Nat
  min Zero _ = Zero
  min _ Zero = Zero
  min (Succ x) (Succ y) = Succ (min x y)
  |])
```

ВНИМАНИЕ

Для правильного функционирования код внутри блока квазицитирования (который находится между символами [d| и |]) должен иметь отступ. При работе с пакетом singletons нужно внимательно следить за этим.

Задание правила времени действия

Поскольку об обычном Haskell-программировании на уровне типов, реализуемом посредством продвижения, мы теперь знаем достаточно, можно подумать о кодировании более сложных инвариантов, а именно о кодировании правила времени действия. Чтобы решить эту задачу, давайте создадим тип данных для интервалов времени. При этом нам нужно обрабатывать три случая: интервал, открытый только в конце, означает, что предложение применимо начиная с определенного момента времени (например, бесконечный интервал является интервалом, открытым с нуля); закрытый интервал — это интервал с начальной и конечной точками; пустой интервал относится к запрещенному предложению. В качестве дополнительного примера давайте путем продвижения определим на уровне типов бесконечный интервал (`Infinite`). В данном коде используется функция продвижения `Nat` из предыдущего раздела:

```
$(promote [d|
  data Range = Empty | Open Nat | Closed Nat Nat

  infinite :: Range
  infinite = Open Zero
  |])
```

Теперь давайте создадим семейство типов, которое в конечном итоге будет вызываться при применении конструкторов `From` и `Until` типа данных `Offer`. Для первого случая функция будет называться `restrictFrom` и получать в качестве аргументов интервал дней, в которые предложение доступно до вступления в силу ограничения, и новую отметку времени для начального дня предложения. Код получается достаточно объемным, но разобраться в нем несложно:

```
{-# LANGUAGE UndecidableInstances #-}

data Comparison = Less | Equal | Greater

$(promote [d|
  compare :: Nat -> Nat -> Comparison
  compare Zero Zero = Equal
  compare Zero (Succ _) = Less
  compare (Succ _) Zero = Greater
  compare (Succ x) (Succ y) = compare x y
```

```
restrictFrom :: Nat -> Range -> Range
restrictFrom _ Empty = Empty
restrictFrom n (Open f) = restrictFrom1 n f (compare n f)
restrictFrom n (Closed f t) = restrictFrom2 n f t (compare n f) (compare n t)

restrictFrom1 :: Nat -> Nat -> Comparison -> Range
restrictFrom1 n _ Greater = Open n
restrictFrom1 _ f Equal = Open f
restrictFrom1 _ f Less = Open f

restrictFrom2 :: Nat -> Nat -> Nat -> Comparison -> Comparison -> Range
restrictFrom2 _ _ _ Greater Greater = Empty
restrictFrom2 _ _ _ Greater Equal = Empty
restrictFrom2 n _ t Greater Less = Closed n t
restrictFrom2 _ f t Equal _ = Closed f t
restrictFrom2 _ f t Less _ = Closed f t
|])
```

Обычно при написании такого кода нужно заботиться о том, чтобы функции `restrictFrom1` и `restrictFrom2` появлялись в условиях `let` или `where`. Но пакет `singletons` пока не поддерживает эти конструкции продвижения, поэтому приходится создавать другие функции. Включение расширения `UndecidableInstances` связано с тем, что компилятор не может удостовериться, что создаваемые семейства типов будут всегда прекращать выполняться при наличии вложенных семейств типов. Такое вложение возникает в продвинутых семействах типов из-за вызова `compare` внутри `restrictFrom`. Однако если известно, что создаваемая вами функция прекратит свое выполнение, можно с уверенностью представить это ограничение и сообщить компилятору, что функцию можно принять, больше не проверяя ее на предмет прекращения выполнения.

Первым шагом в направлении наделения наших предложений тегами с информацией о времени действия является расширение исходного типа за счет новой переменной типа, продвинутой нами ранее до вида `Range`. Чтобы сократить объем примера, мы воспользуемся исходным типом данных `Offer` вместо типа данных, снабженного тегами с количеством подарков. Объявление GADT-типа имеет следующий вид:

```
data Offer a (r :: Range) where
```

Основные комбинаторы для предложений (подарков и скидок) по умолчанию имеют бесконечное время действия. Для их записи нужно воспользоваться семейством типов `Infinite`, созданном при предыдущем продвижении:

```
Present          :: a -> Offer a Infinite
PercentDiscount  :: Float -> Offer a Infinite
AbsoluteDiscount :: Float -> Offer a Infinite
```

В первом приближении можно было бы написать следующее ограничение по времени `From`:

```
From :: (n :: Nat) -> Offer a d -> Offer a (RestrictFrom n d)
```

Однако компилятор такому коду явно не обрадуется и сообщит об ошибке типа:

```
src/Chapter13/CheckDurationPromotion.hs:
  Kind mis-match
```

```
  Expected kind `OpenKind', but `n' has kind `Nat'
  In the type `(n :: Nat)'
  In the definition of data constructor `From'
  In the data declaration for `Offer'
```

Проблема в том, что в конструкторе нельзя использовать продвинутые виды, разрешены только типы вида *. В частности, запрещено применение вида `Nat`.

К счастью, существует конструкция, позволяющая преодолеть эту сложность. Идея заключается в создании обычного типа данных, который перенесет в теге нужное нам число уровня типов. Таким образом, у нас появится значение, которое можно использовать в ходе выполнения программы, и тег, передающий информацию уровня типов. У этого типа данных будет только одно возможное значение для каждого доступного тега, по этой причине такие типы называются *одноэлементными типами данных* (singleton data types). Например, одноэлементный тип `SNat`, соответствующий продвинутому виду `Nat`, выглядит так:

```
data SNat (n :: Nat) where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
```

Благодаря способу конструирования заданный тип вида `Nat` может иметь только одно значение `SNat`. Например, единственным обитателем `SNat (Succ (Succ Zero))` является `SSucc (SSucc SZero)`. Следовательно, как нам и требовалось, в форме значения времени выполнения мы отразили тип, соответствующий двойке. Если посмотреть на вывод GHCi, можно заметить явную ссылку на продвинутые конструкторы, которые помечены спереди знаком одинарной кавычки (`'`):

```
*Chapter13.CheckDurationPromotion> :t SSucc (SSucc SZero)
SSucc (SSucc SZero)
  :: SNat ('Succ ('Succ 'Zero))
```

Следующим шагом станет использование этого одноэлементного типа в конструкторе `From`, который в качестве аргументов времени выполнения применяет только значения вида *. Обратите внимание на то, как реализуется доступ к числу `n` уровня типов (из аргумента в значение `SNat`):

```
From :: SNat n -> Offer a d -> Offer a (RestrictFrom n d)
```

Еще один простой конструктор подразумевает перемещение из одноэлементного типа данных `SNat` обратно в исходный тип данных `Nat` до его продвижения. Когда это преобразование произойдет, у вас будет значение времени выполнения, которое можно использовать в обычных функциях. Следующий фрагмент кода определяет это преобразование и задействует его с целью вывода на экран ограничения для предложения:

```
toNat :: SNat n -> Nat
toNat SZero = Zero
toNat (SSucc n) = Succ (toNat n)

printDateRestriction :: Offer a r -> String
printDateRestriction (From n _)  = "From " ++ show (toNat n)
printDateRestriction (Until n _) = "Until" ++ show (toNat n)
printDateRestriction _           = "No date restriction"
```

Повторим: создание одноэлементного типа для заданного продвинутого типа данных является просто шаблонным кодом. В библиотеку `singletons` включена еще одна функция `singletons`, которая заменяет `promote` и генерирует одноэлементные типы наряду с продвижением типов данных и функций. Например, если нужно сгенерировать тип `SNat` в автоматическом режиме, в исходный файл можно включить следующий код:

```
$(singletons [d|
  data Nat = Zero | Succ Nat
         deriving (Show, Eq)
  |])
```

Кроме того, в библиотеку `singletons` входит класс типов `SingI` с единственной функцией `sing`. Эта функция предназначена для создания уникального значения одноэлементного типа для соответствующего ему продвинутого типа данных. Таким образом, нам не нужно создавать конструкторы, как это делалось ранее при работе с типом `SNat`. Например, обитатели, соответствующие первым четырем натуральным числам, на уровне типов могут быть записаны следующим образом:

```
zero :: SNat Zero
zero = sing  -- результаты в SZero
one :: SNat (Succ Zero)
one = sing   -- результаты в SSucc SZero
two :: SNat (Succ (Succ Zero))
two = sing   -- результаты в SSucc (SSucc SZero)
three :: SNat (Succ (Succ (Succ Zero)))
three = sing -- результаты в SSucc (SSucc (SSucc SZero))
```

Пакет `singletons` предоставляет также класс типов для преобразования из одноэлементного типа обратно в обычный тип данных. Он называется `SingE` и содержит единственную функцию `fromSing`. Этой функцией можно воспользоваться вместо самодельной функции `toNat`, чтобы преобразовать одноэлементный тип `three` в `Nat` уровня понятий:

```
*Chapter13.CheckDurationPromotion> fromSing three
Succ (Succ (Succ Zero))
```

Для создания полноценного типа данных `Offer` не хватает еще нескольких конструкторов. Завершить работу предлагается упражнением 13.11.

УПРАЖНЕНИЕ 13.11. ПРЕДЛОЖЕНИЯ С ОДНОЭЛЕМЕНТНЫМИ ТИПАМИ ДАННЫХ

Добавьте к GADT-типу Offer все остальные конструкторы: Restrict должен сохранять время действия, каким оно было; Until должен изменять время действия способом, похожим на использовавшийся в From; Both, BetterOf и If должны вычислять наименьший интервал времени действия, который включает интервалы из обоих аргументов (то есть их пересечение). На этой стадии вы можете использовать тип Offer для построения полноценного предложения и вычисления его интервала:

```
*> let one = SSucc SZero -- создание одноэлементного типа 1
*> let three = SSucc (SSucc (SSucc SZero)) -- создание одноэлементного типа 3
*> :t let p = Present 'a' in Both (From one p) (BetterOf p (Until three p))
let p = Present 'a' in Both (From one p) (BetterOf p (Until three p))
  :: Offer Char ('Closed ('Succ 'Zero) ('Succ ('Succ ('Succ 'Zero))))
```

ЛИТЕРАЛЫ УРОВНЯ ТИПОВ

GHC предлагает дополнительные возможности для типов, снабженных тегами с натуральными числами или строками. Если включить расширение TypeOperators и импортировать модуль GHC.TypeLits, можно воспользоваться числовыми и строковыми литералами в тех местах, где вам нужен тип вида Nat (для натуральных чисел) или вида Symbol (для строк).

Используя эти литералы, тип данных Vect можно объявить следующим образом:

```
data Vect n a where
  VNil :: Vect 0 a
  VCons :: a -> Vect n a -> Vect (n + 1) a
```

Следует заметить, что этот модуль предоставляет только небольшое множество операций с натуральными числами, а именно сложение, умножение и возведение в степень, а также класс типов, который кодирует определение того, не связаны ли два числа уровня типов соотношением «меньше или равно».

Одноэлементные типы оказались в самом конце обзора Haskell-программирования на уровне типов. Отношения между типами данных и функциями уровня понятий и соответствующими им продвинутыми типами и видами, а также одноэлементными типами являются достаточно размытыми, тем не менее каждый из них служит вполне конкретной цели:

- Объявления данных и функций показывают, как создавать значения, которые Haskell может использовать для вычислений во время выполнения.
- Продвинутые типы данных и виды, а также функции уровня типов, выраженные либо как функциональные зависимости, либо как семейства типов, вычисляются во время компиляции и позволяют использовать с более строгими типами теги со значениями, которые вводят дополнительные инварианты.
- Одноэлементные типы являются мостом между двумя мирами: когда вам требуется значение уровня типов, которое должно быть также отражено во время выполнения, нужно использовать именно эти типы.

Чтобы помочь вам глубже усвоить эти отношения, в упражнении 13.12 предлагается выполнить задание из совершенно другой предметной области.

УПРАЖНЕНИЕ 13.12. ПРЯМОУГОЛЬНИКИ И ОБРАМЛЕНИЯ

Для выполнения данного упражнения мы воспользуемся следующим типом данных, который представляет изображения, построенные из прямоугольников. Конструктор Rect представляет отдельный прямоугольник, а затем изображения можно комбинировать (с конструктором Union), брать только общие части (с конструктором Intersection) или помещать рядом несколько копий в строке (с конструктором Replicate):

```
data Rectangle = R { topLeft :: (Nat, Nat), bottomRight :: (Nat, Nat) }

data Image = Rect Rectangle
           | Union Image Image
           | Intersection Image Image
           | Replicate Nat Image
```

Изображения нужно снабжать тегами с их обрамлениями, то есть с минимальным прямоугольником, содержащим все изображение. Например, если имеется объединение прямоугольников от (1,0) до (5,4) и от (0,1) до (3, 2), то обрамление будет простираться от (0,0) до (5,4): чтобы убедиться в этом, можете нарисовать прямоугольники на бумаге.

Вычисления уровня типов должны разрабатываться с использованием приемов, предлагаемых пакетом singletons. Вопрос о том, где именно требуются одноэлементные типы, нужно тщательно продумать.

Выводы

В данной главе в процессе разработки предметно-ориентированного языка, призванного обеспечить выражение предложений для магазина машин времени, мы исследовали многие нетривиальные возможности системы Haskell-типов:

- Нам была представлена концепция внешнего, или автономного, и внутреннего, или внедренного, *предметно-ориентированного языка*, а также показана разница между глубоким и поверхностным внедрением.
- *Обобщенные алгебраические типы данных* (GADT) позволяют конструкторам типов данных создавать значения различных типов, и эта дополнительная функциональность используется для создания безопасных в отношении типов выражений.
- Мы познакомились с идеей наделения типа *тегами* с дополнительной информацией, позволяющей проверять более строгие инварианты во время компиляции.
- Первые методы создания этих тегов, которые мы изучили, имели отношение к *зависимым типам*, в частности к языку программирования Idris. Главной особенностью зависимой типизации является возможность использования значений в качестве параметров типов.
- Мы изучили несколько механизмов введения в Haskell *программирования на уровне типов*, включая *функциональные зависимости*, *семейства типов* и *продвижение типов*. В данном аспекте основной характеристикой Haskell является раздел между мирами понятий и типов.
- Мы проанализировали функциональные зависимости и *ассоциированные типы*, уточняющие механизм классов типов в Haskell.
- *Продвижение типов данных* и библиотека `singletons` позволили нам переместить объявления с уровня понятий на уровень типов. Больше того, они расширили *систему видов*, что позволило реализовать в Haskell безопасное программирование на уровне типов.

Интерпретация предложений с помощью атрибутов

14

В этой главе мы продолжим начатую в предыдущей главе работу по созданию предметно-ориентированного языка, призванного обеспечить выражение предложений для магазина машин времени. Вы, наверное, помните, что в качестве механизма моделирования языка предложений в Haskell было выбрано глубокое внедрение. Язык глубокого внедрения разбивается на *синтаксис*, выраженный в виде множества Haskell-типов, и на *интерпретацию*, отвечающую за придание смысла каждому значению в языке. В данном примере интерпретацию предложения можно было бы реализовать как функцию, которая, получив список товаров и цен на них, применяет предложения и скидки, после чего возвращает новую цену.

То есть, в принципе, варианты интерпретации можно было бы определять как обычные Haskell-функции, работающие с DSL-значениями. Однако почти каждый вариант интерпретации, который мы можем получить, следует множеству образцов. Поэтому интересно будет рассмотреть такие образцы и выстроить концептуальную модель, которая более точно определяет варианты интерпретации: именно это делают за нас *атрибутивные грамматики* (attribute grammars). Затем для генерирования Haskell-кода может быть использован соответствующий конкретной атрибутивной грамматике инструмент, который избавит нас от написания большого объема шаблонного кода. А если конкретно, то в этой главе основное внимание уделено теме программирования с помощью компилятора UUAGC (Utrecht University Attribute Grammar Compiler).

Помимо простого программирования в данной главе рассматриваются отношения между атрибутивными грамматиками и другими Haskell-концепциями, такими как монады. В конце главы мы вернемся к идеям *оригами-программирования* (origami programming) из главы 3 и применим их к основным типам данных (а не только к спискам).

Варианты интерпретации и атрибутивные грамматики

В данном разделе исследуется основная идея атрибутивной грамматики. Для начала мы будем интерпретировать в Haskell очень простой язык. Затем мы переработаем этот язык с использованием атрибутов.

Простая интерпретация

Давайте рассмотрим очень простой язык, содержащий только три комбинатора для построения выражений. У вас имеется основной комбинатор `AmountOf` с символом в качестве параметра, а также комбинаторы сложения и получения произведения двух выражений. Давайте выразим наш язык в простом Haskell-типе:

```
data Expr = Plus  Expr Expr
          | Times Expr Expr
          | AmountOf Char
```

Желаемая интерпретация этого типа данных получает в качестве параметра строку, а возвращаемым результатом является число. В случае комбинатора `AmountOf` подсчитывается количество символов заданного типа. Остальные два комбинатора просто выполняют сложение или умножение чисел в своих подвыражениях. Соответствующий Haskell-код для этой интерпретации написать несложно:

```
meaning :: Expr -> [Char] -> Int
meaning (Plus  l r)  p = meaning l p + meaning r p
meaning (Times l r)  p = meaning l p * meaning r p
meaning (AmountOf c) p = length $ filter (== c) p
```

Введение в атрибутивные грамматики

Интерпретация, представленная в предыдущем разделе, весьма проста. И все же она содержит немалый объем шаблонного кода: путем вызова `meaning` для подвыражений должно быть явно задано рекурсивное определение, а список `p` должен явно проходить через всю функцию. *Атрибутивные грамматики* предоставляют высокоуровневую модель написания вариантов интерпретации (и как показано далее, множества других видов функций), фокусируясь на используемых и производимых данных, что избавляет нас от обдумывания всех деталей, связанных с информационным потоком. В результате код становится намного более удобным для сопровождения.

Кроме того, атрибутивные грамматики вносят свой вклад в модульность кода. Интерпретацию можно разделить на несколько частей (и даже на несколько файлов), а затем потребовать от системы атрибутивной грамматики объединения всех этих частей с целью вычисления всей информации, которую нужно получить из типа данных при минимальном количестве проходов.

Основным недостатком атрибутивных грамматик является то, что у них иная вычислительная модель, чем у обычного Haskell-кода, хотя обе модели достаточно близки. Как мы увидим в этой главе, атрибутивные грамматики совсем не сложно выразить в понятиях Haskell, поэтому данный недостаток не должен стать препятствием для их использования.

Давайте вернемся к примеру с выражениями. Если вместо обычного Haskell-кода использовать атрибутивные грамматики, то об этой интерпретации можно думать в совершенно других понятиях. С точки зрения атрибутивной грамматики тип данных является *деревом*, узлы которого содержат в себе некие *атрибуты*. В данном примере атрибутами являются исследуемые строки и результат интерпретации.

При внимательном рассмотрении можно заметить, что эти части информации сильно отличаются друг от друга. С одной стороны, исследуемая строка передается из каждого выражения в его подвыражения в качестве параметра. Мы видим, что этот атрибут в информационном потоке направляется из выражения *сверху вниз*. С другой стороны, числовые результаты в каждой точке выражения создаются с учетом результатов его подвыражений и путем их объединения тем или иным способом. Таким образом, в данном случае информационный поток направляется *снизу вверх*. Эту ситуацию иллюстрирует рис. 14.1.

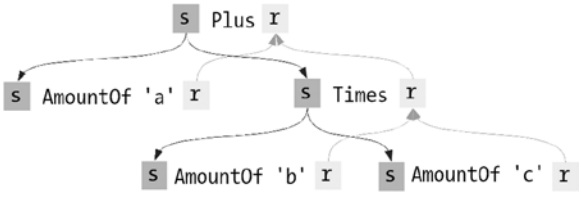


Рис. 14.1. Дерево и атрибуты для `AmountOf 'a' `Plus` (AmountOf 'b' `Times` AmountOf 'c')`

На жаргоне атрибутивной грамматики числовой результат является *синтезированным атрибутом* (synthesized attribute), а исследуемая строка — *унаследованным атрибутом* (inherited attribute). Данные названия приблизительно отражают разные направления информационных потоков, что является важным отличительным признаком этих атрибутов. Чтобы определить этот вариант интерпретации как атрибутивную грамматику, требуются, как показано в примере, три информационные составляющие:

- Из каких возможных узлов состоит дерево? Эта информация берется из определения множества типов данных, в котором каждый конструктор отражает вид узла.
- Являются вычисляемые атрибуты, а также признак унаследованными или синтезированными?
- Как передать информацию от родительских узлов дочерним узлам в случае унаследованных атрибутов и как вычислить значение синтезированных атрибутов по их дочерним узлам?

Назначением *системы атрибутивных грамматик* (attribute grammar system) является получение всей этой информации и генерирование необходимого кода для вычисления финального значения атрибутов в каждом узле. Здесь можно увидеть основное преимущество от использования атрибутивной грамматики в сравнении с написанием интерпретатора непосредственно на языке Haskell. Система атрибутивных грамматик в автоматическом режиме заботится о следующем:

- Когда приходится транспортировать атрибут по более сложному маршруту, чем просто сверху вниз или снизу вверх (например, между узлами одного уровня), Haskell-код может значительно усложниться. Но благодаря атрибутивной грамматике это поведение выражается декларативно, и система берет отслеживание на себя.
- Иногда значение атрибута зависит от другого атрибута. Это означает, что для вычислений атрибуты должны каким-то образом упорядочиваться, что может

потребовать более одного обхода того же дерева. Система атрибутивных грамматик выполняет это упорядочивание автоматически (и во многих случаях делает это оптимальным образом), поэтому думать об этом нам уже не нужно.

Как и для многих других идей, в хранилище пакетов Hackage имеется несколько реализаций атрибутивных грамматик. Одним из вариантов может стать использование генератора парсеров *Happy*, включенного в Haskell Platform и поддерживающего атрибуты в своих грамматиках. Happy является внешней утилитой, выполняющей предварительную обработку грамматического описания и генерирующей Haskell-код, необходимый для его синтаксического разбора.

Имеется также библиотека *AspectAG*, обеспечивающая внедрение атрибутивных грамматик непосредственно в Haskell. При ее использовании перед компиляцией не нужно предпринимать никаких дополнительных созидательных шагов. Она представляет интерес в том плане, что позволяет работать с атрибутивными грамматиками непосредственно в Haskell-коде. Однако у нее имеется недостаток, связанный с большим объемом программирования на уровне типов.

Средством, которое рассматривается в данной главе, является компилятор *UUAGC* (Utrecht University Attribute Grammar Compiler). Как и Happy, он является препроцессором, генерирующим Haskell-код на основе описания грамматики. Однако в отличие от Happy компилятор UUAGC предназначен исключительно для определения атрибутивных грамматик. Таким образом, в данном случае связь между формальной стороной атрибутивных грамматик и UUAGC-кодом выражена более явно, что делает этот инструмент наиболее подходящим для изучения атрибутивных грамматик.

Наша первая атрибутивная грамматика

Давайте создадим атрибутивную грамматику, соответствующую простым числовым выражениям, представленным в предыдущем разделе. Что касается исходного файла, нам нужно последовать двум соглашениям: во-первых, имена исходных UUAGC-файлов должны заканчиваться расширением `.ag`; во-вторых, файлы должны находиться внутри структуры папок, названной по имени модуля. Это аналогично соглашениям, которым следуют исходные Haskell-файлы. В данном случае модуль будет называться `Chapter14.Simple`, следовательно, файл должен находиться в папке src/Chapter14/Simple.ag.

Синтез результата

В предыдущем разделе нам были представлены три информационные составляющие, необходимые для определения атрибутивной грамматики. В UUAGC-программировании каждая из этих составляющих объявляется отдельно. Первыми объявляются те типы данных, на которые будут ссылаться атрибуты. Синтаксис очень похож на Haskell-синтаксис `data`, но есть два отличия:

- После имени типа данных никаких знаков равенства ставить не нужно. Вместо них каждое объявление конструктора должно начинаться с символа вертикальной черты (`|`).

- Все аргументы конструктора должны быть названы по именам, подобно тому, как делалось с записями в Haskell. Однако никаких скобок вокруг полей ставить не нужно.

Кроме того, при использовании переменных типов или составных типов их нужно помещать внутрь фигурных скобок. Это заставит UUAGC выполнить синтаксический разбор контента, указанного внутри скобок, в качестве обычного Haskell-кода, а не в качестве атрибутивной грамматики. С учетом всех этих различий тип данных для выражений должен иметь следующий вид:

```
data Expr
  | Plus  left :: Expr right :: Expr
  | Times left :: Expr right :: Expr
  | AmountOf c :: Char
```

ПРИМЕЧАНИЕ

UUAGC поддерживает два вида синтаксиса. Тот, который используется в данной главе, называется Haskell-синтаксисом, поскольку похож на синтаксис языка Haskell.

Следующим шагом является объявление атрибутов, которые будут содержаться в каждом узле. В UUAGC все узлы, представляющие конструкторы одного и того же типа данных, содержат одинаковые атрибуты. Синтаксис их определения состоит из ключевого слова `attr` и следующих за ним имен типов данных, к которым нам нужно добавить атрибуты. Затем нужно включить строку для каждого атрибута, которая начинается с префикса `inh` или `syn` в зависимости от того, какой атрибут нужно создать, унаследованный или синтезируемый соответственно, после чего следуют имя атрибута и его тип. Атрибуты в нашем примере относятся к типу данных `String`, следуют сверху вниз и имеют синтезируемый числовой результат:

```
attr Expr
  inh string :: String
  syn result :: Int
```

И наконец, нужно задать способ вычисления атрибутов, основанный на других атрибутах и параметрах конструкторов. Это делается в блоке `sem` исходного UUAGC-файла. Этот блок начинается с имени типа данных, затем следует блок для каждого описываемого конструктора. В каждом из этих блоков содержится код, связанный с определением каждого атрибута данного узла. Правая часть каждого конструктора следует обычному Haskell-синтаксису с тремя синтаксическими модификациями:

- Значение атрибута в дочернем узле доступно через синтаксис *узел*`.`*атрибут*. Специальное имя `lhs` зарезервировано для ссылки на *родителя* текущего узла. Идея родителя в Haskell-коде не поддерживается, поэтому давайте поясним ее на примере. Предположим, у нас имеется узел `Plus (AmountOf 'a') (AmountOf 'b')`. В каждом узле (в узле `Plus` и в двух узлах `AmountOf`) будут обрабатываться как атрибуты `string`, так и атрибуты `result`. С точки зрения `AmountOf 'a'` синтаксис `lhs.string` ссылается на атрибут в родительском узле, то есть на значение атрибута `string` в узле `Plus`.

- В правой части объявления каждого атрибута перед ссылками на другие атрибуты или параметры конструктора должен ставиться знак `@`.
- Для разграничения блоков в UUAGC-коде используются отступы. Однако вы можете потребовать от препроцессора сохранить фрагмент кода точно таким же, каким он был в исходном файле, заключив его в фигурные скобки.

В частности, это соглашение предполагает, что поскольку мы определяем значение такого атрибута в родительском узле, синтаксис объявления синтезируемого атрибута должен быть примерно таким:

```
lhs.атрибут = ...
```

В данном случае функция `meaning` может быть преобразована в следующий блок `sem`:

```
sem Expr
  | Plus     lhs.result = @left.result + @right.result
  | Times    lhs.result = @left.result * @right.result
  | AmountOf lhs.result = { length $ filter (== @c) @lhs.string }
```

Выполнение атрибутивной грамматики

Разумеется, написание и компиляция атрибутивной грамматики окажутся совершенно бесполезным занятием, если ее нельзя будет вызывать в нашем Haskell-коде. В зависимости от параметров, задаваемых при запуске UUAGC[1], препроцессор генерирует несколько функций и типов данных, позволяющих прочувствовать преимущества атрибутивной грамматики:

- Объявление `data` в грамматике преобразуется в объявление `data` в Haskell-коде, но с конструкторами, имена которых имеют префикс из имени типа данных, в данном случае — `Expr`. Если нужно, чтобы конструкторы появлялись такими, как они были записаны в грамматике, не используйте в UUAGC параметр `rename`. Однако в этом случае нам придется самостоятельно позаботиться о том, чтобы имена конструкторов не конфликтовали друг с другом.
- Типы данных `Inh_Expr` и `Syn_Expr` представляют унаследованные и синтезированные атрибуты в каждом узле типа данных `Expr`. Оба типа объявлены в виде записей; имена полей включают также имена атрибутов и тип записи, в которой они находятся. В данном примере именами являются `string_Inh_Expr` и `result_Syn_Expr`.
- Функция `sem_Expr` преобразует значение типа данных в функцию, которая получает в качестве аргументов исходные унаследованные атрибуты и возвращает кортеж синтезированных атрибутов. Однако атрибуты получаемой в результате функции могут располагаться в любом порядке, поэтому принято использовать функцию `wrap_Expr`, которая получает в качестве аргументов записи атрибутов.

Например, функция, способная выполнять действия по вычислению атрибутивной грамматики с целью получения в корне дерева синтезированных значений для типа данных `Expr`, могла бы выглядеть так, как показано в следующем фрагменте.

[1] Как указывать эти параметры, рассказывается в следующем разделе.

Обратите внимание на то, что эта функция должна находиться в *другом файле*, а не в том, где находится само определение атрибутивной грамматики, поэтому модуль, сгенерированный компилятором UUAGC, нужно импортировать.

```
import Chapter14.First

executeExpr :: Expr -> String -> Int
executeExpr e s =
  let syn = wrap_Expr (sem_Expr e) (Inh_Expr s)  -- возврат записи Syn_Expr
   in result_Syn_Expr syn
```

А теперь эту функцию можно использовать для тестирования нашей атрибутивной грамматики:

```
*Chapter14.Simple> :{
*Chapter14.Simple| let e = Expr_Times (Expr_AmountOf 'e')
*Chapter14.Simple|       (Expr_Plus (Expr_AmountOf 'a') (Expr_AmountOf 'o'))
*Chapter14.Simple| in executeExpr e "hello"
*Chapter14.Simple| :}
1
```

В следующем разделе мы узнаем, как собрать весь этот код в единое целое в нашем Cabal-пакете.

Интеграция UUAGC-кода в пакет

Чтобы использовать компилятор UUAGC, его сначала нужно установить в системе. Поскольку это препроцессор, включения его в пакет в качестве зависимости будет недостаточно. Вместо этого нужно включить его непосредственно в Cabal-файл Setup.hs. Как это делается, рассказывается в данном разделе.

Установка UUAGC

Компилятор UUAGC имеется в хранилище Hackage, поэтому для появления его в нашей системе достаточно выполнить обычную команду установки:

```
$ cabal install uuagc
```

Эта команда установит как сам компилятор, так и механизм его интеграции с Cabal, который находится в пакете `uuagc-cabal`. Данный пакет нам потребуется для создания атрибутивных грамматик в следующих разделах.

UUAGC и Cabal

Как утверждалось в начале этой главы, UUAGC для Haskell-кода является не библиотекой, а препроцессором. Это означает, что записываемые нами атрибутивные грамматики не могут напрямую компилироваться компилятором GHC или каким-нибудь другим Haskell-компилятором, сначала препроцессор UUAGC должен преобразовать их в исходный Haskell-код. При каждом изменении атрибутивной грамматики преобразование требуется выполнять заново. Делать это можно само-

стоятельно, но лучше сообщить Cabal об использовании UUAGC и позволить этому компилятору позаботиться о предварительной обработке изменившихся файлов так же, как он заботится о файлах, требующих перекомпиляции.

Для начала создайте новый пакет с исполняемой строфой либо используя команду `cabal` в командной строке, либо в Eclipse. Следуя соглашениям предыдущих глав, назовем этот пакет `chapter14`, а исполняемый блок должен называться `uuagc-examples`. Cabal-строфа, соответствующая этому исполняемому блоку, должна иметь следующий вид:

```
executable uuagc-examples
  hs-source-dirs:  src
  build-depends:   base >= 4
  ghc-options:     -Wall
  other-modules:   Chapter14.Simple
  main-is:         Main.hs
```

Следующим шагом нужно сообщить Cabal о необходимости вызова препроцессора перед компиляцией. Это делается путем подготовки сценария сборки, которому Cabal следует для получения откомпилированных файлов. Сценарий, который находится в файле Setup.hs, должен быть изменен следующим образом:

```
import Distribution.Simple
import Distribution.Simple.UUAGC (uuagcLibUserHook)
import UU.UUAGC (uuagc)

main = defaultMainWithHooks (uuagcLibUserHook uuagc)
```

Однако этого еще недостаточно: в Cabal-файле нужно также изменить поле `build-type`, чтобы сообщить процессу сборки об использовании нового файла Setup.hs вместо обычного файла. Откройте файл chapter14.cabal и измените соответствующую строку:

```
build-type: Custom
```

Завершающий шаг состоит в том, чтобы сообщить препроцессору UUAGC о файле, который должен быть им обработан, и о применяемых параметрах компиляции. Для этого создайте в корневой папке проекта файл uuagc_options (следом за файлами .cabal и Setup.hs) со следующим содержимым:

```
file : "src/Chapter14/Simple.ag"
options : data, semfuns, catas, pretty, wrappers, rename,
          module "Chapter14.Simple", haskellsyntax, signatures
```

Учтите, что нам нужно указать не только путь к атрибутивной грамматике (в поле `file`), но и имя создаваемого модуля (в данном случае `Chapter14.Simple`). Кроме того, если вы собираетесь задействовать Haskell-синтаксис, как в примерах данной главы, не забывайте всегда включать параметр `haskellsyntax`.

После всех этих приготовлений можно запустить Cabal для сборки нашего проекта. Выводимая информация будет отличаться от обычной, поскольку используется нестандартный сценарий установки и препроцессором UUAGC осуществляется предварительная обработка файлов. Далее показан пример вывода, но в вашей системе картина может быть несколько иной, что зависит от установленных версий:

```
$ cabal configure
Resolving dependencies...
[1 of 1] Compiling Main             ( Setup.hs, dist/setup/Main.o )
Linking ./dist/setup/setup ...
Configuring chapter14-0.1...
$ cabal build
Building chapter14-0.1...
Preprocessing executable 'uuagc-examples' for chapter14-0.1...
 [UUAGC] processing: src/Chapter14/Simple.ag generating:
dist/build/uuagc-examples/uuagc-examples-tmp/Chapter14/Simple.hs
[1 of 2] Compiling Chapter14.Simple ( dist/build/uuagc-examples/uuagc-examples-
tmp/Chapter14/Simple.hs, dist/build/uuagc-examples/uuagc-examples-tmp/Chapter14/
Simple.o )
[2 of 2] Compiling Main ( src/Main.hs, dist/build/uuagc-examples/uuagc-examples-
tmp/Main.o )
Linking dist/build/uuagc-examples/uuagc-examples ...
```

Для каждой новой атрибутивной грамматики, требующей компиляции, нужно включать новую пару строк, определив их в файле uuagc_options. Преимущество такой методологии состоит в том, что Cabal заботится о повторной сборке только тех частей, которые претерпели изменения, не вызывая препроцессор без необходимости.

EclipseFP сам позаботится о сборке атрибутивных грамматик, и вам вообще не придется модифицировать Cabal-проект. Не забудьте только создать исходный файл, выбрав в мастере создания файла вариант UU Attribute Grammar. Среда разработки задействует даже параметры из соответствующего файла uuagc_options, а в случае его отсутствия создает этот файл. Кроме того, как показано на рис. 14.2, для этого файла имеется даже собственный редактор.

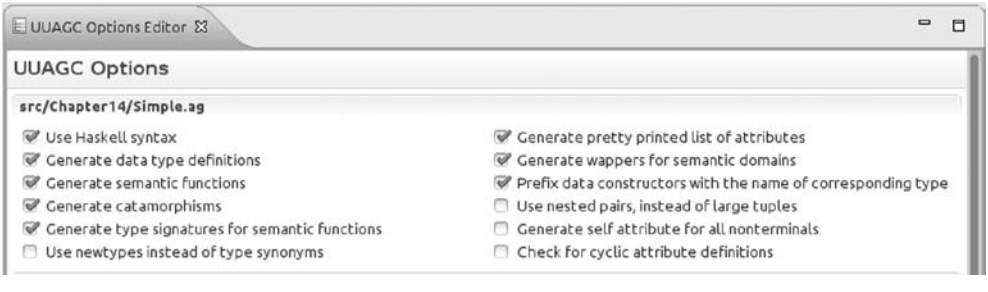


Рис. 14.2. Редактор файла uuagc_options

ГЕНЕРАЦИЯ КОДА ПРЕПРОЦЕССОРОМ UUAGC

При выполнении атрибутивной грамматики UUAGC по умолчанию прибегает к ленивым вычислениям. Однако препроцессор можно заставить оптимизировать код. В этом случае задается определенная последовательность обхода дерева, что требует создания более строгого кода.

Основное внимание в данной главе уделяется применению UUAGC в Haskell. Однако препроцессор может генерировать код и для других функциональных языков, включая OCaml и Clean. В таких случаях правые части инструкций должны писаться не на языке Haskell, а на соответствующем языке внутреннего интерфейса.

Интерпретация выражений

Теперь давайте перейдем к более объемной атрибутивной грамматике. В данном случае основное внимание уделено грамматике, соответствующей условным выражениям внутри предложения. Целью ставится получение функции, которая, получив выражение и список товаров, возвращает результат применения выражения к этому списку.

Использование атрибутивной грамматики

В примере, в котором использовались простые числовые выражения, мы уже видели, как UUAGC разделяет три информационные составляющие, необходимые для определения полной атрибутивной грамматики. В первом блоке определяется базовый тип данных, в котором будут вычисляться атрибуты. Следующий код является простым преобразованием типа данных `Expr`, которое можно найти в предыдущей главе. Имейте в виду, что с целью сокращения объема кода в него включен только один вид булева сравнения чисел.

```
data Expr a
  | AmountOf product :: {a}
  | PriceOf  product :: {a}
  | TotalNumberOfProducts
  | TotalPrice
  | IVal val :: Int
  | FVal val :: Float
  | Plus  right :: (Expr {a})  left :: (Expr {a})
  | Times right :: (Expr {a})  left :: (Expr {a})
  | LessThan right :: (Expr {a})  left :: (Expr {a})
  | Or  right :: (Expr {a})  left :: (Expr {a})
  | And right :: (Expr {a})  left :: (Expr {a})
  | Not inner :: (Expr {a})
```

ПРИМЕЧАНИЕ

Чтобы скомпилировать эту атрибутивную грамматику, нужно включить для GHC расширение ScopedTypeVariables. Проще всего это сделать, добавив в Cabal строфу вида extensions: ScopedTypeVariables.

В данном случае вернемся к простому моделированию результатов выражений: они будут содержаться в трех разных атрибутах в виде целочисленного значения, значения с плавающей точкой или булева значения. В любой отдельно взятый момент времени значение `Just` будет содержаться только в одном из компонентов, а другие получат значение `Nothing`. Как известно из предыдущей главы, чтобы не приходилось создавать кортежи, определение `Expr` может быть уточнено с помощью GADT-типов. Однако поскольку темой данной главы являются атрибутивные грамматики, давайте попробуем сохранять типы максимально простыми. Результат выражения зависит от подвыражений, поэтому данный атрибут должен быть синтезированным.

Независимо от результата нужно пройти список товаров в составе подвыражений, поскольку имеющаяся в нем информация нужна конструкторам `AmountOf`, `PriceOf`,

TotalAmountOfProduct и TotalPrice. В параметрах атрибутивной грамматики эта информация будет представлена с помощью унаследованных атрибутов. Объявление набора атрибутов для Expr имеет следующий вид:

```
attr Expr
  inh products  :: {[(a, Float)]}
  syn intValue  :: {Maybe Int}
  syn fltValue  :: {Maybe Float}
  syn boolValue :: {Maybe Bool}
```

ПРИМЕЧАНИЕ

Не забудьте, что составные типы, такие как [(a, Float)] или Maybe Bool, должны помещаться в скобки, чтобы не дать UUAGC провести их синтаксический разбор в качестве специального синтаксиса атрибутивной грамматики.

Следуя примеру из этого раздела, давайте напишем для Expr блок sem. Исходное объявление в данном случае будет несколько сложнее, поскольку для определения операций нам нужно объявить, что элементы в выражении являются экземплярами класса типов Eq. Получается следующее объявление:

```
sem Eq {a} => Expr
```

Конструктор TotalNumberOfProducts сложностью не отличается. В данном случае единственным атрибутом с конкретным значением должна быть целочисленная единица, остальные должны иметь значение Nothing. Поскольку узлы этого вида не имеют дочерних узлов, никакого кода для наследования атрибута products включать не нужно. В этом случае весь код сводится к следующему:

```
  | TotalNumberOfProducts lhs.intValue  = Just $ length @lhs.products
                          lhs.fltValue  = Nothing
                          lhs.boolValue = Nothing
```

Более сложным примером представляется Plus. В этом случае должны быть обновлены целочисленное значение и значение с плавающей точкой, но каждое из них должно получать только значение Just, если оба подвыражения также получают это значение. Добиться этого при минимальном объеме кода можно с помощью экземпляра Applicative типа Maybe. Однако для размещения этого экземпляра в области видимости необходимо импортировать модуль Control.Applicative. Не забудьте, что любой код, написанный внутри скобок, копируется в неизменном виде в сгенерированный Haskell-код: именно так и должен обрабатываться импорт. Случаи импорта трактуются компилятором UUAGC особым образом, поэтому чтобы обеспечить трактовку импорта в соответствии с правилами языка Haskell, перед блоком нужно поставить ключевое слово imports. Например, чтобы ввести модуль в область видимости, в начало файла нужно включить следующий код:

```
imports
{
import Control.Applicative
}
```

Теперь в коде нашей атрибутивной грамматики будут доступны операторы <$> и <*>, используемые в следующем коде для определения синтезированных атри-

бутов конструктора `Plus`. Код для унаследованных атрибутов также не отличается особой сложностью, поскольку он всего лишь копирует значение товаров из узла в свои дочерние узлы.

```
| Plus  lhs.intValue   = {(+) <$> @right.intValue <*> @left.intValue }
        lhs.fltValue   = {(+) <$> @right.fltValue <*> @left.fltValue }
        lhs.boolValue  = Nothing
        right.products = @lhs.products
        left.products  = @lhs.products
```

С учетом этих примеров будет несложно написать и все остальное. Именно это вам и нужно сделать при выполнении упражнения 14.1.

УПРАЖНЕНИЕ 14.1. ПОЛНОСТЬЮ ВЫРАЖЕННАЯ АТРИБУТИВНАЯ ГРАММАТИКА

Завершите создание блоков sem Expr. Подсказка: в качестве шаблона возьмите предыдущие примеры, которые либо производят вычисления для списка products, либо объединяют значения подвыражений с помощью синтаксиса Applicative.

Например, функция, которая должна выполнять вычисления атрибутивной грамматики для получения синтезированных значений в корне дерева для типа данных Expr, будет иметь следующий вид:

```
executeExpr :: Ord a => Expr a -> [(a,Float)] -> (Maybe Int, Maybe Float, Maybe
Bool)
executeExpr e products =
  let syn = wrap_Expr (sem_Expr e) (Inh_Expr products)  -- возврат записи Syn_Expr
   in (intValue_Syn_Expr syn, fltValue_Syn_Expr syn, boolValue_Syn_Expr syn)
```

И эту функцию можно использовать для тестирования вашей атрибутивной грамматики:

```
*Chapter14.Expr> :{
*Chapter14.Expr| let e = Expr_And (Expr_AmountOf 'a' `Expr_LessThan` Expr_IVal 2)
*Chapter14.Expr|                  (Expr_FVal 300.0 `Expr_LessThan` Expr_TotalPrice)
*Chapter14.Expr|     p = [('a',15.0), ('b',400.0)]
*Chapter14.Expr| in executeExpr e p
*Chapter14.Expr| :}
(Nothing,Nothing,Just True)
```

Предварительное вычисление некоторых значений

Атрибутивная грамматика для `Expr` работает исправно, но может быть усовершенствована с точки зрения производительности. В частности, при каждом обращении к конструктору `TotalNumberOfProducts` или `TotalPrice` должен осуществляться перебор всего списка товаров. Поскольку такие основные комбинаторы встречаются в выражениях нередко, было бы полезно кэшировать результаты для их повторного использования в случае необходимости. Давайте посмотрим, как такое кэширование реализуется в атрибутивной грамматике.

Наиболее распространенным способом, позволяющим справиться с этой ситуацией, является добавление нового типа данных с одним конструктором, который становится оболочкой для всего дерева выражений. Этот тип данных обычно называют

Root. Далее представлено его определение плюс указание на вывод экземпляра класса типов Show, поскольку такие деревья было бы интересно изучить:

```
data Root a
  | Root expr :: (Expr {a})

deriving Root Expr : Show
```

ПРИМЕЧАНИЕ

Добавление дополнительного типа данных Root является простой условностью, используемой для инициализации унаследованных атрибутов. Во многих случаях потребности в этом не возникает, и конечно же, именем не обязательно должно быть Root.

Предварительно вычисленные значения будут сохраняться в атрибутах numberOfProducts и totalPrice. Эти атрибуты должны быть переданы сверху вниз от корневого узла дерева. Следовательно, они должны быть наследуемыми атрибутами типа данных Expr. Кроме того, если нужно вернуть результаты вычислений из Root, придется включать такой же синтезируемый атрибут, который имеется у Expr. Часть грамматики attr должна быть приведена к следующему виду:

```
attr Root Expr
  inh products  :: {[(a, Float)]}
  syn intValue  :: {Maybe Int}
  syn fltValue  :: {Maybe Float}
  syn boolValue :: {Maybe Bool}

attr Expr
  inh numberOfProducts :: Int
  inh totalPrice       :: Float
```

Обратите внимание на то, что UUAGC допускает включение в одно и то же объявление нескольких типов данных, совместно использующих атрибуты. Чтобы применить эту функциональность, нужно просто указать все типы данных после ключевого слова attr. В данном примере атрибуты products, intValue, fltValue и boolValue объявляются как для типа данных Root, так и для типа данных Expr.

И наконец, следует добавить правила вычислений. Правила вычисления новых атрибутов, наследуемых Expr от Root при условии использования грамматики из предыдущего раздела, довольно просты:

```
sem Eq {a} => Root
  | Root expr.numberOfProducts = length @lhs.products
         expr.totalPrice       = foldr (\(_,p) x -> p + x) 0.0 @lhs.products
```

Теперь можно ожидать дополнительных объявлений для прохождения новых унаследованных атрибутов внутри Expr и получения синтезированных атрибутов из Expr для сохранения их в Root. Приятной новостью может стать то, что они вообще не нужны. В UUAGC имеется ряд встроенных правил на случай отсутствия указаний относительно того, как вычислять некоторые атрибуты. Для наследуемых атрибутов вычисление проходит путем копирования значения из родительского узла в дочерний. Для синтезируемых атрибутов результат берется из дочернего узла (если нужно объединить информацию из нескольких дочерних узлов, существуют

другие виды правил: с ними вы познакомитесь в следующем разделе). Обобщенно эти правила называются *правилами копирования* (copy rules), потому что в простых случаях атрибуты просто копируются. Фактически в предыдущей грамматике информационный поток наследуемых атрибутов `products` указываться не должен: об этом позаботится данное правило.

Изменить нужно вычисление синтезированных атрибутов в тех конструкторах, которые желательно усовершенствовать. Вместо того чтобы каждый раз вычислять результат, они должны использовать предварительно вычисленные значения из `Root`. Например, нужно изменить конструктор `TotalNumberOfProducts`:

```
| TotalNumberOfProducts lhs.intValue  = Just @lhs.numberOfProducts
                        lhs.fltValue  = Nothing
                        lhs.boolValue = Nothing
```

Суть идеи состоит в том, что каждый раз при необходимости инициализировать значения для вычисления остальных атрибутов полезно заключить все дерево в оболочку типа данных всего с одним конструктором, который включает эту инициализацию в блок `sem`.

Очевидно, теперь атрибутивная грамматика должна выполняться начиная со значения `HtmlRoot`. Соответствующая функция `describeOffer` имеет следующий вид:

```
describeOffer :: (Eq a, Show a) => Offer a -> H.Html
describeOffer o = html_Syn_HtmlRoot $ wrap_HtmlRoot (sem_HtmlRoot $ HtmlRoot_
HtmlRoot o)
                                                    Inh_HtmlRoot
```

Другой (монадический) взгляд

Изучая новую концепцию, весьма полезно сравнить ее с уже знакомой концепцией. Мы уже видели, как при попытке разложения информационного потока на части возникают атрибутивные грамматики. В первой простой интерпретации аргументы функций превратились в унаследованные атрибуты, а кортеж результатов стал соответствовать синтезированным атрибутам. Для прохождения информации таким способом в главах 6 и 7 использовалась другая концепция: монады `Reader` и `Writer`. И действительно, атрибутивную грамматику, разработанную в данном разделе, можно выразить через монадические параметры. Давайте посмотрим, как это сделать. Обратите внимание, что код из этого раздела не имеет никакого отношения к атрибутивной грамматике и записывается в другом файле, который должен содержать определение типа данных `Expr` из предыдущей главы.

Прежде всего, нужно соблюсти небольшую формальность: тип, используемый в монаде `Writer`, должен быть моноидом. Однако самое интересное здесь заключается только в последнем значении, выводимом функцией, поэтому можно «подделать» такое поведение, предоставив экземпляр `Monoid`, который именно это и делает:

```
import Data.Monoid

newtype Result = Result (Maybe Int, Maybe Float, Maybe Bool) deriving Show
instance Monoid Result where
  _ `mappend` r2 = r2
  mempty         = Result (Nothing, Nothing, Nothing)
```

ВНИМАНИЕ

Хотя тип вполне корректен, этот экземпляр не является выражением настоящей монады. Проблема в том, что x `mappend` mempty = mempty — это не то, что необходимо получить. Правильным решением данной задачи было бы сохранение списка выходных результатов и получение последнего списка в конце вычисления атрибутов.

Как уже упоминалось, поведение унаследованных атрибутов может считаться заключенным в оболочку монады `Reader`, а синтезированных — в оболочку монады `Writer`. Таким образом, сигнатура создаваемой функции должна иметь следующий вид:

```
sem :: Eq a => Expr a -> ReaderT [(a,Float)] (Writer Result) ()
```

А соответствующий код, вызывающий функцию, передающую исходные значения, должен быть таким:

```
import Control.Monad.Reader
import Control.Monad.Writer

executeExpr :: Eq a => Expr a -> [(a, Float)] -> Result
executeExpr e p = execWriter (runReaderT (sem e) p)
```

Давайте посмотрим, как в этих параметрах выражено вычисление двух конструкторов. Сначала конструктор `TotalNumberProducts` берет наследуемый атрибут (который теперь является информацией в монаде `Reader`) и производит новый синтезированный атрибут с длиной списка:

```
sem TotalNumberProducts = do products <- ask
                             tell $ Result (Just (length products), Nothing, Nothing)
```

Следующий рассматриваемый комбинатор занимается сложением двух чисел. В данном случае нужно вычислить синтезируемые атрибуты локально и получить эти результаты, чтобы создать новые синтезированные атрибуты. Первая часть может быть выполнена с помощью `listen`, аналогом локальности монады `Reader` для монады `Writer`:

```
sem (e1 :+: e2) = do (_, Result (i1, f1, _)) <- listen (sem e1)
                     (_, Result (i2, f2, _)) <- listen (sem e2)
                  tell $ Result ((+) <$> i1 <*> i2, (+) <$> f1 <*> f2, Nothing)
```

Как видите, атрибутивные грамматики выражают концепцию, заложенную в монады `Reader` и `Writer` в новом параметре. Основное преимущество заключается в возможности ссылаться на атрибуты по именам вместо создания пакета из нескольких преобразователей монад и обращения к каждому из них путем использования разного количества функций `lift`. Следующее более глубокое погружение в атрибутивные грамматики дает нам интуитивное понимание того, что монады `Reader` и `Writer` могут легко и в любом порядке объединяться, чтобы в результате была получена новая монада, и этот очевидный вывод делается благодаря возможности объединения атрибутивных грамматик просто путем слияния их атрибутов.

Варианты интерпретации предложений

Пора уже двигаться вперед от условных выражений к полноценным значениям типа `Offer`, представляющим возможные скидки в магазине машин времени.

Как и в предыдущем разделе, с целью сохранения чистоты и краткости кода примеры будут связаны с исходным определением этого типа данных, а не с обеспечением дополнительной безопасности типов.

Проверка правила подарков

Мы уже рассмотрели несколько возможных вариантов проверки правила подарков, относящегося к предложениям (напомним, количество бесплатных подарков в предложении может ограничиваться в зависимости от сезона). Используя строгую типизацию, мы изучали способы подсчета максимального количества подарков, но реализация не давала никаких зацепок относительно того, о каких именно подарках идет речь. Целью данного раздела является создание компактной атрибутивной грамматики, вычисляющей максимально допустимое множество подарков, которое может быть бесплатно получено отдельным покупателем.

Как и в предыдущем разделе, основной тип данных является прямым преобразованием Haskell-синтаксиса в UUAGC-синтаксис. Используемое объявление `data`, которое содержит имена, присвоенные полям каждого конструктора, выглядит следующим образом:

```
data Offer a
  | Present  present :: {a}
  | PercentDiscount  discount :: Float
  | AbsoluteDiscount discount :: Float
  | Restrict products :: {[a]} inner :: (Offer {a})
  | From   from  :: Int inner :: (Offer {a})
  | Until  until :: Int inner :: (Offer {a})
  | Extend times :: Int inner :: (Offer {a})
  | Both     left :: (Offer {a}) right :: (Offer {a})
  | BetterOf left :: (Offer {a}) right :: (Offer {a})
  | If cond :: (Root {a}) then :: (Offer {a}) else :: (Offer {a})

deriving Offer : Show
```

Список подарков относится к той категории, которая вычисляется снизу вверх, поэтому он должен быть синтезированным атрибутом. В примере кода атрибут называется `presents`. Часть грамматики `attr` выглядит так:

```
attr Offer
  syn presents :: {[a]}
```

Следующий шаг заключается в написании блока `sem`, сообщающего препроцессору о том, как вычислять значение этого атрибута. В большинстве случаев, касающихся типов данных, подарки просто аккумулируются из всех подчиненных предложений. Но есть также два особых случая, требующих отдельного рассмотрения:

- Как известно, при наличии конструктора `Present` подарки, возвращенные предложением, являются одноэлементным списком, содержащим значение с параметром.
- При наличии конструктора `Restrict` нужно удалить из списка подарков все те товары, которых нет в списке ограничений.

В коде это соответствует следующему блоку `sem`:

```
imports
{
import Data.List
}

sem Eq {a} => Offer
  | Present  lhs.presents = [@present]
  | Restrict lhs.presents = { @products `intersect` @inner.presents }
```

Настала пора написать код вычислений для всех остальных случаев, к каковым относятся конкатенация списков для таких конструкторов с подчиненными предложениями и пустого списка для других основных комбинаторов, таких как скидки. К счастью, компилятору UUAGC этот распространенный образец известен, и он предоставляет для него специальный синтаксис, расширяющий для синтезированных атрибутов ранее упомянутый встроенный механизм правила копирования. Следуя данному синтаксису, нам нужно предоставить некую функцию, выполняющую слияние атрибута с дочернего узла и основного варианта для тех узлов, у которых вообще нет дочерних узлов. В данном случае часть грамматики `attr` должна быть дополнена оператором `(++)` в качестве объединительной функции и оператором `[]` в качестве исходного значения:

```
attr Offer
  syn presents use {++} {[]} :: {[a]}
```

Если не указывать, как вычислять значение атрибута `presents`, правило, которое его просто копирует, больше применяться не будет. Вместо этого препроцессор задействует операции в объявлении `use`.

Если присмотреться к указанным операциям, можно заметить, что они действительно являются экземпляром `Monoid` для списков. Это возвращает нас к предыдущей дискуссии, посвященной взаимоотношениям между атрибутивными грамматиками и монадой. При использовании синтезированных атрибутов с `use` можно представить их в качестве монады `Writer`, где экземпляр `Monoid` типа атрибута явным образом определен операциями в `use` (вместо операции `mappend`, которая просто забывает о первом аргументе).

С помощью атрибутивной грамматики может быть также проверено правило времени действия. Именно эту проверку требуется реализовать в упражнении 14.2.

УПРАЖНЕНИЕ 14.2. ПРАВИЛО ВРЕМЕНИ ДЕЙСТВИЯ

Вычислите максимальный интервал времени действия предложения, используя атрибутивную грамматику. Правила должны быть такими же, как в главе 13. Чтобы указать на общее поведение объединяемых предложений, добавьте, как показано в данном разделе, в ваши атрибуты объявление use.

Показ HTML-описания

В последнем примере атрибутивных грамматик мы рассмотрим описание предложения в HTML-коде. Это может быть интересно в плане того, что грамматика

может использоваться не только для потребления информации, но и для ее показа новым способом. Кроме того, наша грамматика продемонстрирует ряд тонкостей, связанных с обработкой атрибутов.

Генерируемый HTML-код составляют три разные части:

- Список всех подарков, которые могут быть получены в рамках предложения. Он создается путем повторного использования атрибута `presents` из предыдущего раздела.
- Небольшая таблица контента, в которой нужно показать краткое описание первого и второго уровней дерева предложений. Элементом таблицы является ссылка на приводимое ниже полное описание.
- И наконец, полное описание предложения в виде серии вложенных списков.

Разметка для полного описания будет храниться в атрибуте `html` и кодироваться с помощью библиотеки `blaze-html`, представленной в главе 12. Таким образом, чтобы продолжить решение задач данного раздела, нужно добавить библиотеку `blaze-html` в качестве зависимости. Для получения окончательной разметки, состоящей из трех частей, в код включен дополнительный корневой тип данных с именем `HtmlRoot`. В следующем блоке кода собран импортируемый код, относящийся к `blaze-html`:

```
imports
{
import Data.List
import Data.String
import qualified Text.Blaze.Html5 as H
import qualified Text.Blaze.Html5.Attributes as A
}

data HtmlRoot a
  | HtmlRoot root :: (Offer {a})
```

Кроме того, в каждом узле будет храниться его заголовок, а сразу под ним — список заголовков дочерних узлов. Таким образом, самый верхний узел будет иметь информацию, необходимую для таблицы контента. Основной задачей при создании таблицы является генерирование ссылок на полное описание. Для этого нужно каждому узлу присвоить уникальный идентификатор. Однако этот идентификатор ведет себя и как унаследованный, и как синтезированный атрибут: независимо от движения вверх и вниз он обладает эффектом *прохождения* атрибутов *по узлам одного уровня*. На рис. 14.3 показан информационный поток такого идентификатора через основное предложение.

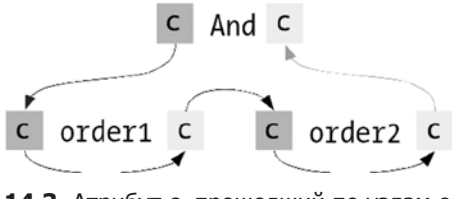


Рис. 14.3. Атрибут c, прошедший по узлам одного с узлом And уровня

Эти виды атрибутов, являющиеся как унаследованными, так и синтезированными, известны в области атрибутивной грамматики как *сцепленные* (chained). Если соотносить поведение сцепленных атрибутов с монадой, то в данном случае это могла бы быть монада `State`. Атрибут можно рассматривать как значение, которое меняется по мере исследования дерева в целом.

Объявление такого атрибута внутри исходного UUAGC-файла может быть сделано путем написания такого же атрибута, названного как `inh` и `syn`. Но чтобы подчеркнуть, что атрибут является сцепленным, в качестве идентификатора в UUAGC используется ключевое слово `chn`. Следуя общему образцу, используемому в компьютерной науке для названия источника уникальных чисел, этому атрибуту присвоено имя `counter` (счетчик). Определение всех применяемых атрибутов имеет следующий вид:

```
attr Offer
  chn counter   :: Int
  syn title     :: String
  syn subtitles :: {[(String,Int)]}
  syn presents use {++} {[]} :: {[a]}  -- из предыдущего раздела

attr HtmlRoot Offer
  syn html        :: {H.Html}
```

Давайте сначала посмотрим, как вычисляется атрибут `counter`. В каждом узле окончательное значение должно отличаться от значений всех его дочерних узлов и от значения его родительского узла. Этого можно добиться путем добавления единицы к значению последнего дочернего узла. Например, значение для основных комбинаторов, являющееся значением из родительского узла плюс 1, выражается следующим образом:

```
sem Eq {a}, Show {a} => Offer
  | Present PercentDiscount AbsoluteDiscount  -- применяется ко всем конструкторам
      lhs.counter = @lhs.counter + 1
```

Поскольку `counter` является как унаследованным, так и синтезированным атрибутом, `lhs.counter` ссылается в каждой стороне на разные атрибуты. В той стороне, что справа, `@lhs.counter` является значением, исходящим из родительского узла. А в той стороне, что слева, `lhs.counter` является значением, которое будет возвращено как идентификатор для данного узла.

Но этот код нельзя признать полностью правильным. Если внутри другого атрибута (например, при создании HTML-разметки для данного узла) ссылаться на `lhs.counter`, то получится ссылка на значение из родительского узла. Однако это не соответствует требуемому результату: два дочерних узла одного и того же родительского узла получат одно и то же значение `counter`, поскольку у них общий родительский узел. Решение заключается в сохранении нового значения в *локальном* атрибуте, который работает внутри узла как локальная переменная. Таким образом, каждый узел будет сохранять свое значение `ident`, несмотря на любые изменения в своем родительском узле. Объявлять локальные атрибуты не нужно, а в узле их можно использовать, только задав префикс из ключевого слова `loc`. Чтобы сделать значение счетчика доступным для изменения другим атрибутам, в примере применяется локальный атрибут `ident`:

```
| Present PercentDiscount AbsoluteDiscount
    loc.ident   = @lhs.counter + 1
    lhs.counter = @loc.ident
```

Для конструктора с более чем одним дочерним узлом нужно обеспечить поочередный проход атрибута от одного узла к другому. В данном примере конструкторы `Both` и `BetterOf` именно так и делают для дочерних узлов `left` и `right`:

```
| Both BetterOf
    left.counter  = @lhs.counter
    right.counter = @left.counter
    loc.ident     = @right.counter + 1
    lhs.counter   = @loc.ident
```

А теперь в упражнении 14.3 продумайте весь оставшийся путь.

УПРАЖНЕНИЕ 14.3. ПРОХОЖДЕНИЕ АТРИБУТА COUNTER

Завершите вычисление атрибутов counter и ident для конструкторов Restrict, From, Until, Extend и If.

Идея использования локальных атрибутов поможет написать более компактный код для вычисления HTML-разметки. Например, разметку каждого узла можно поделить на две части: HTML-код, соответствующий описанию самого конструктора (`htmlText`), и HTML-код, соответствующий его дочерним узлам (`htmlChild`). Затем, как показано здесь, оба атрибута объединяются в одно значение `Html`:

```
| Present PercentDiscount AbsoluteDiscount Restrict From Until Extend Both BetterOf If
  lhs.html    = { do H.a H.! A.name (fromString ("elt" ++ show @loc.ident))
                     $ H.toHtml @loc.htmlText
                @loc.htmlChild }
```

ПРИМЕЧАНИЕ

Фактически все правила можно написать в том виде, в котором они показаны, одно под другим. UUAGC позаботится о слиянии вычислений для всех атрибутов каждого конструктора.

Теперь каждый из конструкторов должен предоставить значение этих локальных атрибутов для создания их HTML-разметки. Это несложно: в качестве примера приведен код, который делает все это для конструкторов `PercentDiscount` и `BetterOf`:

```
| PercentDiscount
    loc.htmlText  = show @discount ++ "% discount"
    loc.htmlChild = H.toHtml ""
| BetterOf
    loc.htmlText  = "Better of"
    loc.htmlChild = { H.ul $ do H.li @left.html
                                H.li @right.html }
```

Независимо от разметки каждый из узлов, как уже упоминалось, должен сохранить для окончательной таблицы контента атрибуты `title` и `subtitle`. Первый атрибут будет простым значением типа `String`, а второй должен собрать заголовки

и идентификаторы каждого из ближайших дочерних узлов. Для используемых конструкторов этот код выглядит так:

```
| PercentDiscount
    lhs.title     = { "DISCOUNT: " ++ show @discount ++ "%" }
    lhs.subtitles = []
| BetterOf
    lhs.title     = { "BETTER OF" }
    lhs.subtitles = [(@left.title, @left.counter), (@right.title, @right.counter)]
```

Код для остальных конструкторов предлагается создать при выполнении упражнения 14.4.

УПРАЖНЕНИЕ 14.4. СОЗДАНИЕ ДЕСКРИПТОРОВ

Завершите вычисление атрибутов htmlText, htmlChild, title и subtitle для конструкторов Present, AbsoluteDiscount, Restrict, From, Until, Extend, Both и If.

Последний штрих заключается в том, чтобы взять всю эту информацию и создать все значение `Html`, представляющее описание. В показанном далее коде монада `blaze-html` с помощью `mapM_` генерирует список элементов. Что же касается всего остального, код просто генерирует разметку из внутренних элементов:

```
sem Eq {a}, Show {a} => HtmlRoot
  | HtmlRoot root.counter = 1
             lhs.html = {
               do H.h1 $ H.toHtml "Description of an offer"
                  H.h2 $ H.toHtml "Possible presents:"
                  H.ul $ mapM_ (\e -> H.li $ H.toHtml (show e)) @root.presents
                  H.h2 $ H.toHtml "Main offer"
                  H.a H.! A.href (fromString ("#elt" ++ show @root.counter))
                        $ H.toHtml @root.title
                  H.ul $ mapM_ (\(s, e) -> H.li $ H.a H.! A.href (fromString
("#elt" ++ show e))
                                                          $ H.toHtml st)
                               @root.subtitles
                  H.h2 $ H.toHtml "Complete offer"
                  @root.html }
```

Одно из важных преимуществ описания механизма генерирования разметки в качестве атрибутивной грамматики состоит в том, что вам не нужно заботиться об *упорядочении* вычислений. Компилятор UUAGC сам способен решить, как сортировать атрибуты, учитывая, что каждый из них для вычисления нуждается лишь в предыдущих атрибутах. Если компилятор не сможет с этим справиться, он выводом предупреждения сообщит, что атрибутивная грамматика *зациклена* (circular).

Оригами-программирование для любого типа данных

В главе 3 было показано, насколько эффективным средством могут быть свертки для понимания кода списков. Основная идея свертки заключается в замене кон-

структора (:) некоей функцией f и конструктора [] неким исходным значением i, как показано на рис. 14.4.

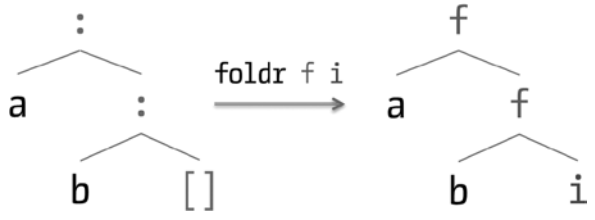


Рис. 14.4. Напоминание о поведении свертки

Интересно, что нотация свертки может быть определена для каждого Haskell-типа: каждому конструктору предоставляется функция, которая будет «заменять» его, а затем вычислять выдаваемое в качестве результата дерево. Кроме того, свертка должна вызывать саму себя рекурсивно в тех местах, где определение типа данных также носит рекурсивный характер. Эти обобщенные свертки называют также *катаморфизмами* (catamorphisms).

Рассмотрим, к примеру, простой язык из первого раздела данной главы, где жестко закодированный символьный аргумент AmountOf был обобщен до переменной типа в типе Expr:

```
data Expr a = Plus  (Expr a) (Expr a)
            | Times (Expr a) (Expr a)
            | AmountOf a
```

В данном случае у свертки должны быть в качестве аргументов функция для Plus, еще одна функция — для Times и еще одна, последняя, функция — для AmountOf. Как утверждалось ранее, реализация должна вызывать каждую из этих функций в зависимости от конструктора значения и иметь рекурсивные вызовы в тех местах, где в определении Expr в качестве аргумента тоже используется Expr. Со всеми этими идеями в целом получается следующий код:

```
foldExpr plusFn timesFn amountFn e =
  let f = foldExpr plusFn timesFn amountFn
  in case e of
       Plus  e1 e2 -> plusFn  (f e1) (f e2)
       Times e1 e2 -> timesFn (f e1) (f e2)
       AmountOf x  -> amountFn x
```

Интерпретация, показанная в первом разделе, теперь должна быть переработана в свертку, как показано в коде:

```
meaning :: Eq a => [a] -> Expr a -> Int
meaning s = foldExpr (+) (*) (\x -> length $ filter (==x) s)
```

Типы аргументов должны извлекаться из определения типа данных. Конкретная свертка будет производить элемент некоего типа b для каждого значения. Таким образом, для каждого места, где может быть получено значение из рекурсии, в его типе должен появляться тип b. У нас есть также дополнительные аргументы для

информации, закодированной в каждом конструкторе. С учетом этих инструкций типы приобретают следующий вид:

```
plusFn   :: b -> b -> b
timesFn  :: b -> b -> b
amountFn :: a -> b
```

В литературе о свертках, или катаморфизмах, иногда встречается термин «алгебра». *D-алгебра* для типа данных `D` — это всего лишь кортеж, содержащий все функции, необходимые для свертки этой структуры данных. Вы можете создать новый тип `newtype` для алгебры `Expr`:

```
newtype ExprAlgebra a b = ExprAlgebra (b -> b -> b, b -> b -> b, a -> b)
```

Таким образом, свертка типа данных `D` может рассматриваться как функция, получающая D-алгебру и значение `D` и возвращающая соответствующее значение. Давайте выразим свертку `Expr`, используя понятия алгебры:

```
foldExpr' :: ExprAlgebra a b -> Expr a -> b
foldExpr' a@(ExprAlgebra (plusFn,timesFn,amountFn)) e =
  case e of
    Plus  e1 e2 -> plusFn  (foldExpr' a e1) (foldExpr' a e2)
    Times e1 e2 -> timesFn (foldExpr' a e1) (foldExpr' a e2)
    AmountOf x  -> amountFn x
```

Практику в реализации этих идей можно получить, добавляя новые конструкторы при выполнении упражнения 14.5.

УПРАЖНЕНИЕ 14.5. БОЛЕЕ СЛОЖНЫЕ ВЫРАЖЕНИЯ

Расширьте тип данных Expr за счет конструкторов Minus и DividedBy и завершите разработку соответствующих функций свертки для этих новых конструкторов.

Свертки тесно связаны с атрибутивными грамматиками. При работе с атрибутивной грамматикой мы определяем конкретную алгебру своего типа данных, то есть алгебру, вычисляющую значение атрибутов. И действительно, если посмотреть на Haskell-код, сгенерированный компилятором UUAGC, можно заметить, что в нем есть ряд частей с метками `cata`: это уже знакомые нам функции, чьи имена начинаются с префикса `sem`. Эти функции занимаются сверткой структуры, ожидая исходные унаследованные значения, задаваемые с помощью функции, чье имя начинается с префикса `wrap_`.

Если вы усвоили эти идеи, можете познакомиться с другими способами, с помощью которых типы данных могут конструироваться и применяться в общей форме. Например, может быть также обобщено понятие развертки (называемой еще *анаморфизмом*), а из комбинаций сверток и разверток возникают другие интересные функции. Тот, кого эта тема интересует, может изучить библиотеку Эдварда А. Кметта (Edward A. Kmett) `recursion-schemes`. Изучение предлагаемых в ней образцов может помочь в плане многократного использования больших объемов готового кода и следования тем же структурам в отношении разных типов данных.

Кроме библиотек существует множество источников информации о том, как использовать концепцию свертки любого типа данных с целью создания более

элегантного кода. В главе 3 уже упоминалась книга Джереми Гиббонса (Jeremy Gibbons) «Origami programming», кроме нее есть еще книга Эрика Мейера (Erik Meijer), Маартена Фоккинга (Maarten Fokkinga) и Росса Патерсона (Ross Paterson) «Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire», в которой рассматриваются все эти обобщения сверток и разверток, а также показано, как применять эквациональные рассуждения не только к спискам (о чем рассказано в главе 3), но и к любым другим типам данных.

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ ТИПОВ ДАННЫХ

Мы уже знаем о двух видах полиморфизма в Haskell. В параметрическом полиморфизме значения рассматриваются как черные ящики, никакого исследования содержимого нет. В то же время специализированный полиморфизм позволяет трактовать каждый случай в отдельности.

Но есть еще и третья разновидность полиморфизма, доступная в GHC и относящаяся к обобщенному программированию типов данных (data type generic programming), или просто к обобщениям (generics). В этом случае функции должны работать с любым типом данных, но при этом разрешается зависеть от структуры объявления data, что может быть реализовано лишь отчасти, поскольку обычные Haskell-типы определяются через небольшие последовательности комбинаторов: выбор конструкторов, задание кортежей параметров и т. д.

Чтобы сделать доступным в Haskell этот вид программирования, существует ряд библиотек, хотя и в самом компиляторе GHC реализована такая возможность. Если есть интерес в получении дополнительных сведений по этой проблеме, хорошей стартовой позицией может стать модуль GHC.Generics.

Выводы

В данной главе мы познакомились с атрибутивными грамматиками в качестве средства предоставления на предметно-ориентированном языке вариантов интерпретации, закодированных как Haskell-тип:

- Мы усвоили основную идею, касающуюся наделения *атрибутами* дерева, узлы которого являются конструкторами конкретного Haskell-типа.
- Мы выделили три вида атрибутов: *синтезированные* (снизу вверх), *унаследованные* (сверху вниз) и *сцепленные* (проход одноуровневых узлов).
- С атрибутивными грамматиками связан класс типов `Monad`. В частности, монада `Reader` соответствует унаследованным атрибутам, монада `Writer` — синтезированным, монада `State` — сцепленным.
- Мы научились конфигурировать и использовать в Haskell-коде препроцессор *UUAGC*, предназначенный для предварительной обработки атрибутивных грамматик.
- UUAGC предлагает множество средств написания компактных атрибутивных грамматик: в данной главе мы познакомились с *объявлениями* `use` и с *локальными атрибутами*.
- И наконец, мы обобщили идею *свертки* (или *катаморфизма*) списка до любого доступного Haskell-типа.

Часть V
ИНЖИНИРИНГ МАГАЗИНА

Глава 15. Документирование, тестирование и проверка

Глава 16. Создание архитектуры приложения

Документирование, тестирование и проверка

15

На данный момент нам уже известны многие возможности и особенности языка Haskell и его библиотек. В этой главе мы не будем изучать какие-то его дополнительные механизмы, а уделим основное внимание тем утилитам, которые призваны помогать разработчику при программировании библиотек и приложений. Эти утилиты привносят в Haskell хорошие инжиниринговые методы работы.

Важным требованием при написании программы, о котором часто забывают, является создание *качественной документации* для исходного кода. Даже при том, что Haskell является весьма высокоуровневым языком, в нем всегда найдется место для документации, описывающей назначение каждого типа данных и каждой функции в модулях. В комплект Haskell Platform входит утилита *Haddock*, позволяющая из специальных комментариев в коде создавать очень симпатичную документацию.

Большая часть предлагаемых утилит касается *тестирования* (testing). Автоматическое тестирование кода гарантирует, что любые вносимые в него изменения не повлияют на его целевое поведение. В наши дни программисты считают *блочное тестирование* (unit testing) весьма важным видом деятельности, и вы увидите, как с помощью утилиты *HUnit* можно создавать блочные тесты в Haskell.

Haskell был первым языком, в котором с помощью библиотеки *QuickCheck* помимо блочного проводилось еще и *рандомизированное тестирование* (randomized testing). При этой форме тестирования генерируются случайные вызовы функций, что позволяет удостовериться в приемлемости определенных свойств выходных данных и избавляет тестера от написания сотен вариантов тестов. На этой идее построена еще одна утилита под названием *SmallCheck*, но она выполняет *исчерпывающее тестирование* (exhaustive testing). Это означает, что для определенной части вызываемого кода используются не случайные, а все возможные входные данные вплоть до предельных величин.

В последнем разделе мы вернемся к языку Idris. Мы уже знаем, как дополнительная информация или инварианты могут выражаться посредством зависимых типов. Следующим шагом в использовании этой дополнительной функциональности станет *формальная проверка* (formally verify) того, что та или иная реализация соответствует заявленным свойствам. Мы получим представление о том, как проводится формальная проверка.

Документирование двоичных деревьев с помощью утилиты Haddock

В примерах данной главы мы будем работать с двоичными деревьями, подобными тем, что были представлены в главе 4. Чтобы освежить в памяти соответствующий материал, давайте рассмотрим определение типа данных `BinaryTree`: конструкторы ссылаются либо на лист, либо на узел, который обрабатывает ту или иную внутреннюю информацию. Кроме того, в определении имеются две функции для работы с деревом:

```
data BinaryTree a = Node a (BinaryTree a) (BinaryTree a)
                  | Leaf
                  deriving Show

treeInsert :: Ord a => a -> BinaryTree a -> BinaryTree a
treeInsert x Leaf = Node x Leaf Leaf
treeInsert x (Node y l r) | x <= y    = Node y (treeInsert x l) r
                          | otherwise = Node y l (treeInsert x r)

treeMerge :: Ord a => BinaryTree a -> BinaryTree a -> BinaryTree a
treeMerge t Leaf         = t
treeMerge t (Node x l r) = treeInsert x $ treeMerge (treeMerge t l) r
```

Даже без дополнительных комментариев понятно, что мы можем сгенерировать сводную информацию по модулям, находящимся в пакете, запустив в корневой папке проекта команду `cabal haddock` или перейдя к пункту Export (Экспорт), а затем к пункту Cabal documentation (Cabal-документация) в EclipseFP. В нашем проекте внутри папки dist/doc/html будет создано множество HTML-страниц. При посещении этих страниц мы увидим документацию в том же формате, в котором она представлена в Hackage. Однако предлагаемая информация носит весьма поверхностный характер: это просто список модулей, и среди сведений о каждом модуле имеется список всех определений. Было бы неплохо включить комментарии и примеры в каждую функцию, каждый тип данных и каждый класс типов.

Передать дополнительную информацию для включения в такие HTML-страницы можно через *комментарии для документирования* (documentation comments), которые записываются в том же файле, что и код, — это обеспечивает синхронизацию кода и документации. Эти комментарии следуют тому же синтаксису, что и обычные комментарии, но начинаются со специального маркера. Точно такая же идея реализована и в других системах документирования, например в Javadoc и Doxygen.

Маркер в комментариях меняется в зависимости от того, в какое место нужно включить описание элемента, до самого объявления или после. Чтобы описание располагалось перед элементом, комментарий нужно пометить символом `|`, после — символом `^`. Комментарии, начинающиеся символом `|`, обычно служат для документирования типов данных, функций и классов типов, а комментарии, начинающиеся символом `^`, чаще всего используют при описании конструкторов и аргументов функций. В качестве примера можно создать документацию для типа `BinaryTree`:

```
-- | A typical binary tree
data BinaryTree a = Node a (BinaryTree a) (BinaryTree a) -- ^Inner nodes
                  | Leaf                                 -- ^Liaves
                  deriving (Eq, Show)
```

На рис. 15.1 показано, как представленные в этом фрагменте комментарии выглядят на HTML-странице, созданной с помощью утилиты Haddock.

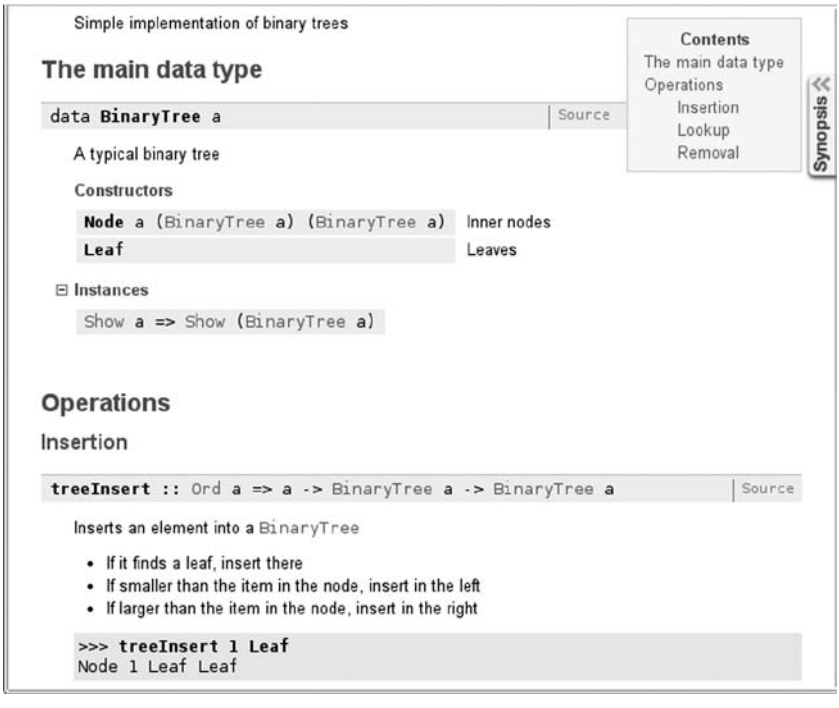


Рис. 15.1. Документация для бинарного дерева, сгенерированная утилитой Haddock

ВНИМАНИЕ

Однострочные комментарии Haskell начинаются с группы символов --, за которой следует пробел. Этого синтаксиса нужно придерживаться и в комментариях для документирования. Пробел также нужно ставить между группой символов -- и символом | или ^.

Для написания документации Haddock предлагает довольно развитый синтаксис:

- Если вы ссылаетесь на другие объявления, нужно процитировать их в одинарных кавычках. Тогда Haddock автоматически сгенерирует нужную ссылку. Эта функциональность доступна также для создания ссылок на модули: в этом случае нужно писать их имена в двойных кавычках.
- Ненумерованные списки объявляются путем установки перед каждым из пунктов списка символа *. При создании нумерованных списков для каждого n-го пункта нужно указывать либо символы $n.$, либо (n). Имейте в виду, что каждый пункт должен быть окружен пустыми строками.

- Фрагменты кода можно вставлять путем установки в начале каждой строки префикса >. Дополнительно Haddock поддерживает объявление свойств, перед которыми ставится префикс prop>, и демонстрацию примеров интерпретатора — в них входные данные записываются после префикса >>>, а выходные данные — сразу же под входными.

Далее показан пример документации для функции `treeInsert`. Обратите внимание на то, что комментарии являются многострочными. Что из них получилось, можно увидеть на уже упомянутом рис. 15.1.

```
{-|
Inserts an element into a 'BinaryTree'

 * If it finds a leaf, insert there

 * If smaller than the item in the node, insert in the left

 * If larger than the item in the node, insert in the right

>>> treeInsert 1 Leaf
Node 1 Leaf Leaf
-}
treeInsert :: Ord a => a -> BinaryTree a -> BinaryTree a
treeInsert x Leaf = Node x Leaf Leaf
treeInsert x (Node y l r) | x <= y    = Node y (treeInsert x l) r
                          | otherwise = Node y l (treeInsert x r)
```

Еще одной интересной особенностью Haddock является возможность систематизации объявлений в коде и деления документации на разделы. Единственное требование заключается в предоставлении явного списка экспорта: этот прием программирования используется повсеместно, поэтому он не должен стать для вас неожиданным. В список экспорта включаются комментарии для документирования, начинающиеся символом * (новый раздел) или символами ** (новый подраздел). Вот как выглядит возможный перечень объявлений для реализации полномасштабных двоичных деревьев:

```
-- | Simple implementation of binary trees
module Chapter15.BinaryTree (
  -- * The main data type
  BinaryTree(..),
  -- * Operations
  -- ** Insertion
  treeInsert, treeMerge,
  -- ** Lookup
  treeFind, treeFindMin,
  -- ** Removal
  treeDelete
) where
```

Запуск Haddock для данного модуля приведет к созданию документации, подобной той, что показана на рис. 15.1.

Если присмотреться к рис. 15.1, можно заметить несколько гиперссылок с пометкой Source (Источник). Целью каждой из этих ссылок является определение этого элемента в версии исходного файла, снабженной комментариями для документи-

рования. В своей документации эти описания можно получить, выполнив вместо команды `cabal haddock` команду `cabal haddock --hyperlink-source`. Следует заметить, что для работы этого механизма в системе должен быть установлен пакет `hscolour`.

HLINT

Haskell-компилятор проверяет, соблюдаются ли в компилируемом исходном коде синтаксис и семантика языка. Однако бывает так, что код правильный, но читается с трудом или содержит фрагменты, которые трудно понять, что может в будущем привести к ошибкам. Сделать исходный Haskell-код понятнее позволяет утилита HLint.

Заполучить HLint в свою систему можно с помощью обычной процедуры установки, то есть выполнения из командной строки команды cabal install hlint. Затем можно выполнить команду hlint <исходный файл> и получить перечень рекомендаций. Предположим, к примеру, что мы внесли изменения в третью строку treeInsert, придав ей следующий вид:

```
treeInsert x (Node y l r) | (x <= y) = Node y (treeInsert x l) r
```

Если запустить HLint, утилита предложит внести следующее изменение:

```
src/Chapter15/BinaryTree.hs:39:29: Warning: Redundant bracket
Found:
  (x <= y)
Why not:
  x <= y
```

Если в качестве среды разработки используется EclipseFP, то такие HLint-предложения автоматически появляются в окне редактора (рис. 15.2). Более того, вы можете сразу же реализовать эти предложения в своем коде.

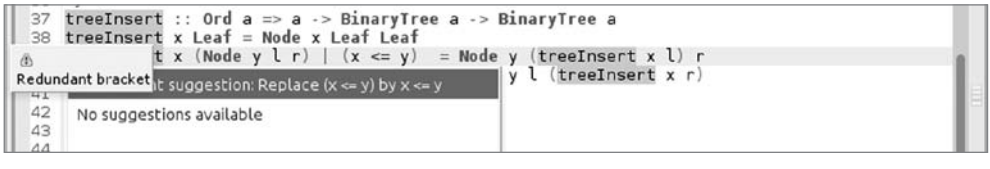


Рис. 15.2. Редактор EclipseFP показывает HLint-предложение

Блочное тестирование с помощью утилиты HUnit

Блочное тестирование представляет собой методологию проверки того, что ваши функции должным образом делают свою работу. Следуя этим методам, программист пишет множество вариантов тестов: каждый из них выполняет часть кода и описывает ожидаемый результат. Эти тесты могут запускаться по усмотрению разработчика или автоматически с целью проверки того, что после внесения в код изменений он продолжает нормально выполняться, — это называется *регрессивным тестированием* (regression testing).

В целях предоставления различным доступным методам тестирования единого интерфейса Haskell-сообщество создало несколько *сред тестирования* (test frameworks). В данном разделе мы будем говорить о библиотеке Tasty, но доступны и многие другие интересные средства, среди которых HTF, `test-framework` и Hspec. Хорошая новость состоит в том, что любая из этих сред тестирования может интегрироваться с вашим Cabal-проектом и запускаться с таким же ключом командной строки.

Объявление тестов в Cabal

В принципе, можно создать приложение, запускающее тесты как исполняемые файлы исключительно в вашем пакете. Однако Cabal воплощает идею тестов в своем ядре и имеет для них специальную поддержку. В данном разделе рассматривается способ добавления в Cabal-проект тестового исполняемого файла.

Внутри Cabal-файла тест определяется в собственной строфе как библиотека и исполняемые части. Поэтому объявление теста в файле проекта имеет заголовок `test-suite`, за которым следует имя набора тестов. Ключевым полем, включаемым в эту строфу, является `type`: Cabal поддерживает разные виды «поставщиков тестов», которые могут запускать тесты в проекте различными способами.

Основное внимание в данном разделе уделяется полю `type` с объявлением `exitcode-stdio-1.0`. На момент написания книги так объявлялся вариант поставщика, который поддерживался лучше других. Указывая его, мы сообщаем Cabal, что наши тесты должны быть откомпилированы в качестве исполняемых файлов, которые будут запускаться для выполнения тестов. Затем Cabal через выходной код сообщит, были эти тесты пройдены успешно или нет. Поскольку такой способ запуска известен всем доступным в Hackage средам тестирования, существенно облегчается их интеграция в Cabal-тест.

Остальное содержимое строфы `test-suite` при указании в качестве типа `exitcode-stdio-1.0` ничем не отличается от определения обычного исполняемого файла. Для примера рассмотрим строфу, соответствующую тестированию кода, рассматриваемого в данной главе. В нем в качестве зависимости указана среда Tasty наряду с пакетами, необходимыми для запуска в Tasty тестов из HUnit, QuickCheck и SmallCheck:

```
test-suite Tasty
  type:           exitcode-stdio-1.0
  build-depends:  base >= 4, tasty, tasty-hunit, tasty-quickcheck, tasty-smallcheck
  hs-source-dirs: test
  main-is:        Tasty.hs
```

Обратите внимание на то, что согласно объявлению исходные файлы для тестов находятся в папке test. Хранить исходный код и тесты в разных папках стало уже практикой — так проще находить места, куда нужно включать новую функциональность или новые проверки поведения.

Как обычно, вы можете добавить набор тестов в графической среде, используя EclipseFP. Для этого нужно в редакторе Cabal перейти на вкладку Test Suites (Наборы тестов) и щелкнуть на стрелке, следующей за знаком «плюс». Появится перечень шаблонов для всех наборов тестов, доступных в EclipseFP, что позволяет легко выбрать нужный набор. На рис. 15.3 показан вид редактора при добавлении нового набора тестов `exitcode-stdio`. Многим популярным средам тестирования EclipseFP позволяет запускать тесты непосредственно из пользовательского интерфейса и получать результаты в специальной области просмотра Test Results (Результаты теста).

И последний штрих. Часто требуется получить доступ к функциям, объявленным в строфе `library` внутри строфы `test-suite`. Это можно сделать, сославшись в той же строфе `test-suite` на папку источника. Однако это приведет к двойной компи-

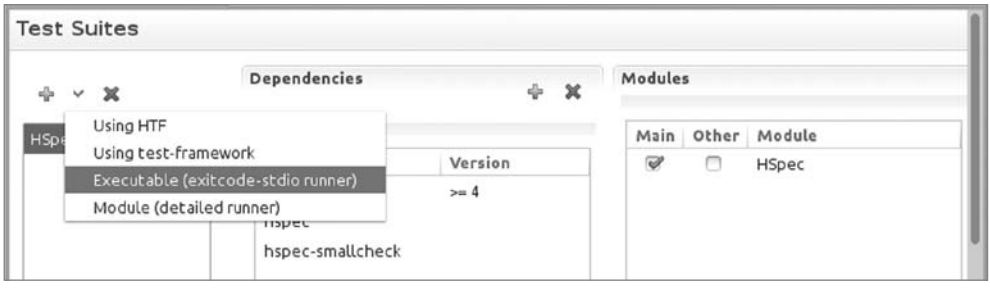


Рис. 15.3. Добавление нового набора тестов в EclipseFP

ляции (один и тот же файл компилируется, поскольку указан как в строфе `library`, так и в строфе `test-suite`) и не обозначит границы ответственности этой строфы. Вместо этого Cabal поддерживает зависимость набора тестов или исполняемого файла от той части Cabal-проекта, которая относится к библиотекам. Чтобы объявить эту зависимость, нужно просто добавить имя пакета в соответствующий список `build-depends`. Например, если взять предыдущий случай, получится следующий код:

```
build-depends:   base >= 4, tasty, tasty-hunit, tasty-quickcheck, tasty-smallcheck,
chapter15
```

Однако этого еще недостаточно для Cabal-файла, созданного либо с помощью команды `cabal init`, либо в мастере EclipseFP. Эта разновидность зависимости доступна только в самых последних версиях Cabal, поэтому исполняемые файлы для реализации этой функциональности требуют, чтобы о нужной версии было явно объявлено. Поэтому поле `cabal-version` в самом начале Cabal-файла должно выглядеть следующим образом:

```
cabal-version:  >=1.8
```

Теперь все готово к проведению тестов, которые будут объявлены в следующем разделе с помощью утилиты `cabal`. Эта функциональность весьма полезна, поскольку вся проектная информация концентрируется в одном месте.

Написание блочных тестов

Сначала нужно создать файл test/Tasty.hs, который в строфе указывается как основной. Этот файл должен содержать модуль `Main`. Основной исполняемый файл строфы является единственным исключением из правил именования файлов и модулей одинаковым идентификатором, что упрощает создание проектов с более чем одним исполняемым файлом или тестом. В примерах будут использоваться среда тестирования Tasty и утилита HUnit, поэтому нужно импортировать соответствующие модули. Основой для остальной части раздела служит следующий каркас:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit as HU
```

Набор тестов в Tasty состоит из базовых *вариантов тестов* (test cases), которые позже можно будет сгруппировать с целью получения иерархической структуры. В зависимости от выбранной утилиты (HUnit, QuickCheck или SmallCheck) применяются разные варианты тестов. В случае использования утилиты HUnit вариант теста определяется функцией `testCase`, которой в качестве аргументов передаются имя и *утверждение* (assertion). В утверждении кодируется конкретная проверяемая функциональность. Например, в следующем варианте теста при использовании HUnit-комбинатора `assertEqual` ожидается равенство единице результата вставки единицы в пустое дерево.

```
import Chapter15.BinaryTree

hunitTestInsertOnLeaf :: TestTree
hunitTestInsertOnLeaf = HU.testCase "Insert 'a' on empty tree" $
  assertEqual "Insertion is wrong" (treeInsert 'a' Leaf) (Node 'a' Leaf Leaf)
```

ПРИМЕЧАНИЕ

TestTree — это один из Tasty-типов. К двоичным деревьям, используемым в качестве примеров в данной главе, он никакого отношения не имеет.

Ранее уже отмечалось, что Tasty позволяет создать иерархическую структуру вариантов тестов за счет их размещения в группах тестов. Группа тестов определяется посредством функции `testGroup`, которой нужно передать имя группы и список элементов. Эти элементы должны быть вложенными группами или базовыми вариантами тестов. Пример организации тестов:

```
allTests :: TestTree
allTests = testGroup "Tasty Tests" [
    testGroup "HUnit Tests" [ hunitTestInsertOnLeaf ]
  ]
```

В данном случае имеется группа высшего уровня «Tasty Tests», внутри которой есть еще одна группа «HUnit Tests». В эту вложенную группу и помещен ранее определенный вариант теста.

Чтобы иметь возможность запускать тесты, нужно сделать еще один последний шаг. Поскольку, как уже отмечалось, за их автоматическое выполнение отвечает исполняемый файл, нам нужно предоставить точку входа `main`. Для решения этой задачи в Tasty имеется очень простая функция, которую можно включать во все свои тесты. Функции `defaultMain` нужно указать вариант теста или группу тестов:

```
main :: IO ()
main = defaultMain allTests
```

Пришла пора выполнить этот первый тест. Прежде всего, нужно сообщить Cabal, что в проекте требуется создать тесты, библиотеку и исполняемый файл. Для утилиты командной строки это делается путем повторного конфигурирования с дополнительным аргументом (при использовании среды EclipseFP этот шаг не нужен, поскольку там тесты включаются автоматически):

```
$ cabal configure --enable-tests
```

Затем для компиляции только что определенного набора тестов нужно выполнить команду `cabal build`:

```
$ cabal build
Preprocessing test suite 'Tasty' for chapter15-0.1...
[1 of 1] Compiling Main             ( test/Tasty.hs, dist/build/Tasty/Tasty-tmp/
Main.o )
...
Linking dist/build/Tasty/Tasty ...
```

В заключение нужно просто запустить команду `cabal test`. Будет выполнен набор тестов, и основные результаты появятся на экране:

```
$ cabal test
Running 1 test suites...
Test suite Tasty: RUNNING...
Test suite Tasty: PASS
Test suite logged to: dist/test/chapter15-0.1-Tasty.log
1 of 1 test suites (1 of 1 test cases) passed.
```

Если теперь открыть файл журнала, упомянутый в сообщении, можно будет увидеть более подробное описание результатов выполнения набора тестов:

```
Tasty Tests
  HUnit Tests
    Insert 1 on empty tree: OK
```

Как видите, в файле показана иерархическая структура, объявленная в исходном файле.

Кроме «длинных» комбинаторов, таких как `assertEqual`, в HUnit включены инфиксные комбинаторы `(@?=)` и `(@=?)`, позволяющие создавать более короткие выражения для ожидаемых равенств. Оба оператора задают проверку, являются ли элементы по обе стороны от них равными. Разница лишь в месте записи проверяемого и правильного значений. Правило заключается в том, что первое значение (проверяемое) должно записываться за знаком `?`, а второе (ожидаемое значение) — за знаком `=`. Таким образом, предыдущий пример может быть также записан в следующих двух формах (не забудьте, что выражение с `treeInsert` является проверяемым, поэтому оно должно быть возле знака `?`):

```
hunitTestInsertOnLeaf' = HU.testCase "Insert 'a' on empty tree" $
  treeInsert 'a' Leaf HU.@?= Node 'a' Leaf Leaf
hunitTestInsertOnLeaf'' = HU.testCase "Insert 'a' on empty tree" $
  Node 'a' Leaf Leaf HU.@=? treeInsert 'a' Leaf
```

Равенство является наиболее распространенной проверкой в блочных тестах, но HUnit позволяет также проверять булево свойство результата. Например, можно проверить, что после вставки элемента в дерево этот элемент может быть найден. В данном случае давайте создадим шаблон для нескольких блочных тестов, написав функцию, возвращающую вариант теста на основе исходного дерева и вставляемого элемента:

```
import Data.Maybe

hunitTestInsertFind :: Ord a => a -> BinaryTree a -> TestTree
```

```
hunitTestInsertFind e t = HU.testCase "Insert can be found" $
  assertBool "Cannot find element" (isJust $ treeFind e $ treeInsert e t)
```

Как и в предыдущем случае, у HUnit имеется краткая версия комбинатора `assertBool`, а именно — `(@?)`:

```
hunitTestInsertFind' e t = HU.testCase "Insert can be found" $
  (isJust $ treeFind e $ treeInsert e t) HU.@? "Cannot find element"
```

К списку можно добавить несколько блочных тестов, предоставляя шаблону различные параметры:

```
allTests = testGroup "Tasty Tests" [
    testGroup "HUnit Tests" [
      hunitTestInsertOnLeaf
    , hunitTestInsertFind 'b' Leaf
    , hunitTestInsertFind 'c' (Node 'd' Leaf Leaf)
    ]
  ]
```

Как видите, хотя библиотека HUnit весьма компактна, она поддерживает наиболее распространенные варианты блочных тестов. В упражнении 15.1 вам предлагается написать ряд дополнительных тестов для проверки реализации двоичных деревьев. Выполняя упражнение, можно составить начальное представление о среде HSpec.

УПРАЖНЕНИЕ 15.1. БЛОЧНОЕ ТЕСТИРОВАНИЕ ДВОИЧНЫХ ДЕРЕВЬЕВ

Напишите варианты тестов для проверки того, что при добавлении элемента к двоичному дереву размер (количество внутренних узлов) дерева увеличивается на единицу. Дополнительно напишите варианты тестов для проверки того, что удаление элемента фактически приводит к его исключению из двоичного дерева.

HSPEC

В данной главе средой тестирования, используемой для интеграции всех тестов, является Tasty. Однако как уже отмечалось в начале этой главы, в Hackage есть ряд других доступных сред тестирования. Одной из них является среда Hspec, специально предназначенная для тех групп разработчиков, которые используют методологию разработки через реализацию поведения (Behavior-Driven Development, BDD). Пример применения среды Hspec выглядит следующим образом:

```
import Data.Maybe
import Test.Hspec
import Test.HUnit

main = hspec $ do
  describe "Insertion in binary tree" $ do
    it "Inserts correctly 1 in empty tree" $
      treeInsert 1 Leaf @?= Node 1 Leaf Leaf
    it "Finds 1 after inserting it on a tree" $
      isJust $ treeFind 1 $ treeInsert 1 (Node 2 Leaf Leaf)
    it "Gets the minimum correctly" $
      pendingWith "Needs to be implemented"
```

Как видите, стиль описания тестов в Hspec напоминает обычный текст. Цель заключается в том, чтобы обеспечить максимальную близость с языком, используемым на этапе спецификации. Тем самым облегчается написание тестов, проверяющих правильность последующей реализации.

Рандомизированное тестирование с помощью библиотеки QuickCheck

Блочное тестирование является наиболее распространенным подходом к проверке того, что реализация отвечает предъявляемым к ней требованиям. Но это тестирование требует слишком большого объема работы: для каждого требования нужно создать несколько вариантов тестов, с помощью которых попытаться охватить достаточно большое количество возможных сценариев. Обычно приходится создавать варианты тестов для экстремальных значений, пустых коллекций, неверных данных и т. д.

Чтобы не продумывать все эти детали самостоятельно, было бы неплохо иметь возможность выразить свои спецификации на более высоком уровне, а затем проверить, поддерживаются ли нужные свойства вашей программой. Характерное для языка Haskell использование функций высшего порядка облегчает выражение таких свойств. Например, свойство «двойное реверсирование списка аналогично оставлению его в неизменном виде» может быть записано как `reverse . reverse = id`. Плохо только то, что решить задачу путем автоматической проверки каждого отдельного свойства в реальных условиях невозможно (хотя в следующем разделе мы увидим, что когда полное доказательство корректности кода получить невозможно, может пригодиться формальная верификация).

Объединить эти два мира пытается библиотека *QuickCheck*. При ее использовании мы с помощью высокоуровневых свойств показываем, каким должно быть поведение программы. Затем библиотека создает множество случайных значений, с применением которых программа тестируется. Если тестов достаточно много (сотни или тысячи), повышается уверенность в том, что данное свойство поддерживается. Кроме того, QuickCheck включает компонент *сужения* (shrinking), который используется, когда находится значение, не удовлетворяющее заданной спецификации. Задача этого компонента заключается в том, чтобы попытаться максимально уменьшить (сузить) это значение, чтобы можно было воспроизвести ошибку и отследить ее источник.

Тестирование свойств списков

Давайте начнем с тестирования простых свойств списочных функций. В частности, давайте обратим внимание на функцию `reverse`, которая выстраивает список в обратном порядке. Исходная реализация будет содержать небольшую ошибку, позволяющую увидеть, как Tasty выявляет проблемы, связанные с неудачным прохождением QuickCheck-теста.

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' [x]    = [x, x]
reverse' (x:xs) = reverse' xs ++ [x]
```

Первым проверяемым свойством станет то, что после реверсирования списка его длина остается ожидаемой (конечно же, в данном примере функция тест не

пройдет). Соответствующее определение для этого QuickCheck-свойства имеет следующий вид:

```
{-# LANGUAGE ScopedTypeVariables #-}

import Test.Tasty.QuickCheck as QC

reverseTests :: TestTree
reverseTests = testGroup "Tests over reverse"
  [ QC.testProperty "reverse respects length" $
      \(lst :: [Integer]) -> length (reverse' lst) == length lst ]
```

Как видите, определение начинается точно так же, как и HUnit-тест: мы вызываем функцию `testProperty` и даем свойству имя. Затем мы определяем тело свойства как функцию, которая должна выполняться для каждого возможного значения своих аргументов.

ПРИМЕЧАНИЕ

В некоторых случаях вам понадобится ограничить типы аргументов для QuickCheck-свойств. Причина в том, что в Haskell невозможно создать случайные значения для каждого возможного типа. Например, в случае со списками невозможно просто сгенерировать список функций. Таким образом, в примере код работает только для списков целых чисел даже при том, что функция reverse' применима к спискам любого вида.

Если к перечню Tasty-тестов, который был назван `allTests`, добавить `reverseTest`, провести компоновку и запустить новый набор тестов, будет получено следующее сообщение об ошибке:

```
Test suite Tasty: RUNNING...
Tasty Tests
  Tests over reverse
    reverse respects length: FAIL
      *** Failed! Falsifiable (after 3 tests and 3 shrinks):
      [0]
```

В сообщении говорится, что после трех попыток[1] встретился пример, в котором свойство не поддерживается. Затем список сужается до тех пор, пока пример не становится совсем небольшим. Разумеется, наименьшим примером, на котором `reverse'` спотыкается, является одноэлементный список.

Проверьте, как вы усвоили принцип работы библиотеки QuickCheck, выполнив упражнение 15.2.

УПРАЖНЕНИЕ 15.2. БЫСТРАЯ ПРОВЕРКА РЕВЕРСИИ С ПОМОЩЬЮ QUICKCHECK

Добавьте дополнительные QuickCheck-свойства к функции reverse', определенной в начале данного раздела. Например, примените ее дважды, чтобы получить исходный результат (reverse' . reverse' == id), или проверьте, что голова списка является последним элементом реверсированного списка.

После проверки того, что свойства для исходной реализации не поддерживаются, внесите в код исправления и посмотрите, пройдет ли новая версия все эти тесты.

[1] Поскольку в QuickCheck тесты проводятся в случайном порядке, в вашем случае количество попыток может оказаться другим.

Тестирование свойств двоичного дерева

Хотя тестирование списков и представляет интерес, в большинстве случаев придется создавать тесты для собственных типов данных. Для этого библиотеке QuickCheck нужно предоставить *генератор* случайных значений требуемого типа. Это делается путем создания экземпляра класса типов `Gen`, в частности путем реализации функции `arbitrary`.

Для генерирования случайных значений двоичных деревьев можно использовать несколько инструментов:

- Значения других типов данных можно генерировать, вызывая соответствующие экземпляры класса `Gen`. Относительно двоичных деревьев это должно быть сделано для элементов в узлах.
- Случайный выбор из нескольких генераторов можно делать с помощью функции `oneof`, которая получает их список, и в ходе выполнения программы возвращает один из указанных генераторов. Более совершенным вариантом `oneof` является функция `frequency`. Эта функция кроме возможных результатов включает относительную частоту, с которой должен появляться каждый генератор. Кроме того, вы узнаете, как использовать эту вторую функцию, чтобы в каждый отдельно взятый момент времени решить, что именно нужно генерировать, узел или лист дерева.
- Чтобы ваши значения удовлетворяли конкретному условию, можно добавить вызов функции `suchThat`. Эта функция обеспечит в отношении случайных значений сохранение заданного предиката.

Одним из самых важных свойств генератора случайных значений является то, что он должен *остановиться* и в определенный момент выдать значение. QuickCheck уточняет этот замысел и требует от генераторов возвращать значения определенного *максимального размера*. Под размером нужно понимать некоторое внутреннее измерение того, «насколько большим» может быть значение. Для части примеров могут использоваться длина списка, количество листьев на дереве и абсолютное значение целого числа. Если наш тип данных допускает реализацию подобного замысла, мы можем получить информацию относительно требуемого размера через QuickCheck-функцию `sized`.

Для нашего примера идея размера трактуется как максимальное количество листьев, которое может иметься у дерева. Стратегия генерирования случайных деревьев, удовлетворяющих этому свойству, заключается в выборе между созданием листа или узла с уменьшающейся вероятностью второго из вариантов. Если в процессе генерирования будет достигнута точка, в которой размер дерева должен быть равен 0, возвратится просто лист. Преобразование этой идеи в код выглядит следующим образом:

```
instance Arbitrary a => Arbitrary (BinaryTree a) where
  arbitrary = sized $ \n ->
    if (n == 0)
       then return Leaf
       else frequency [(1, return Leaf),
                       (n, resize (n-1) arbitrary)]
```

Если посмотреть на класс типов `Gen`, можно заметить еще одну функцию `shrink`, которая отвечает за сжатие значения в мелкие части. Для заданного значения функция `shrink` должна вернуть список этих мелких фрагментов, который должен быть проверен. Как показано в следующем фрагменте кода, в случае с двоичными деревьями вполне естественными вариантами выбора будут поддеревья заданного дерева:

```
shrink Leaf         = []
shrink (Node _ l r) = [l, r]
```

Имея в своем распоряжении этот экземпляр, можно протестировать свойства двоичных деревьев. После вставки элемента давайте проверим факт его наличия в дереве. Объявление теста имеет следующий вид:

```
qcTestInsert :: TestTree
qcTestInsert = QC.testProperty "insert => you will find it" $
  \(n :: Int) t -> treeFind n (treeInsert n t) == Just n
```

Еще одним возможным тестом является проверка того, что при вставке элемента в дерево с его последующим удалением найти его в дереве больше не удастся. В понятиях QuickCheck это выглядит так:

```
qcTestDelete = QC.testProperty "delete => not find it" $
  \(n :: Int) t -> (treeFind n $ treeDelete n $ treeInsert n t) == Nothing
```

Однако если хорошенько задуматься, это свойство для проверки не годится. Допускаются случаи, в которых у дерева `t` имеется копия элемента `n`, тогда после вставки и удаления элемента исходная копия останется. Поэтому результат `treeFind` будет равен не `Nothing`, а `Just n`.

Решить проблему можно, сообщив QuickCheck, что случайный элемент для тестирования должен отвечать некоторым предварительным условиям (в данном случае он не должен изначально содержать число `n`). Это делается с помощью комбинатора `(==>)`. В качестве аргумента ему передается булево условие, которому должен удовлетворять элемент, и тело текущего свойства. Например:

```
qcTestDelete = QC.testProperty "delete => not find it" $
  \(n :: Int) t -> (treeFind n t == Nothing) QC.==>
                   (treeFind n $ treeDelete n $ treeInsert n t) == Nothing
```

Как показано в этих примерах, QuickCheck допускает получение более высокоуровневого представления о тестах: вместо концентрации на отдельных вариантах тестов речь идет о свойствах, которые гораздо теснее связаны с исходной спецификацией.

SMALLCHECK

QuickCheck использует для тестирования свойств кода случайные числа. Утилита SmallCheck действует в том же духе, но вместо создания случайных значений она тестирует свойства с каждым возможным значением из некого множества. Например, свойство reverse . reverse == id можно проверить на каждом списке вплоть до определенной длины.

Открытый интерфейс SmallCheck очень похож на интерфейс QuickCheck. Он также интегрирован почти во все среды тестирования для Haskell, включая Tasty и Hspec.

Формальная верификация с использованием языка Idris

В главе 13 мы узнали, как кодировать интересную информацию о данных в системе типов. Основное внимание уделялось тому, чтобы на соответствующем предметно-ориентированном языке могли записываться только корректные предложения. Однако возможности систем со строгой типизацией этим не ограничиваются: строгую типизацию можно также использовать для проверки свойств кода.

ПРИМЕЧАНИЕ

Этот раздел — нечто вроде «пряника», которым вы награждаетесь за то, что зашли так далеко в чтении данной книги. Цель не в том, чтобы целиком и полностью раскрыть проблему формальной верификации в языках с зависимой типизацией, а в том, чтобы продемонстрировать возможности строгой системы типов при разработке программного обеспечения. Если вы заинтересовались данным вопросом и хотите поглубже погрузиться в тему, обратитесь к ссылкам на дополнительные источники, которые были даны в главе 13.

Основной причиной использования зависимых типов для проверки свойства является кодирование проверяемого свойства непосредственно в системе типов. То есть если компилятор допускает определение соответствующего типа, это означает, что свойство соответствует предъявляемым к нему требованиям. Большим преимуществом данного подхода является то, что формальная верификация кода гарантирует, что он работает так, как нужно для каждого возможного варианта ввода. В отличие от этого при тестировании проверяется только конечное количество значений, и отсутствие дефектов подтверждается лишь для этого множества входных данных. Основным недостатком технологии формальной верификации является более высокая трудоемкость полной верификации фрагмента кода по сравнению с созданием для него хорошего варианта теста.

Следуя все тому же рабочему примеру, давайте объявим в Idris тип данных двоичного дерева, снабженного тегами с количеством имеющихся в нем внутренних узлов. Вспомните, что в Idris объявление `data` всегда следует GADT-синтаксису, поэтому мы можем свободно задействовать понятия в качестве параметров типа. В данном случае в коде используется натуральное число типа `Nat`:

```
data BinaryTree : Type -> Nat -> Type where
  Leaf : BinaryTree a 0
  Node : a -> BinaryTree a l -> BinaryTree a r -> BinaryTree a (S (l + r))
```

Цель состоит в том, чтобы удостовериться в поддержании следующего свойства вставки в деревья: «заданное дерево с n внутренними узлами после вставки в него элемента имеет n+1 внутренних узлов». Поскольку количество внутренних узлов уже закодировано в типе `BinaryTree`, нужно создать функцию вставки со следующей сигнатурой:

```
treeInsert : Ord a => a -> BinaryTree a n -> BinaryTree a (S n)
```

Если удастся написать такую функцию, искомое свойство будет верифицировано. Давайте начнем с написания кода вставки в двоичное дерево, не заботясь о тегах:

```
treeInsert x Leaf = Node x Leaf Leaf
treeInsert x (Node y l r) with (x <= y)
  | True  = let l' = treeInsert x l in Node y l' r
  | False = let r' = treeInsert x r in Node y l r'
```

Если загрузить этот код в Idris, у компилятора появятся возражения:

```
$ idris src/Chapter15/BinaryTreeFV.idr
Type checking ./src/Chapter15/BinaryTreeFV.idr
./src/Chapter15/BinaryTreeFV.idr:11:When elaborating right hand side of Chapter15.
BinaryTreeFV.treeInsert:
Can't unify
    BinaryTree a (S (plus l (S r)))
with
    BinaryTree a (S (S (plus l r)))

Specifically:
    Can't unify
        plus l (S r)
    with
        S (plus l r)
```

Проблема в выделенном фрагменте: компилятор не может понять, что `l + S r` равно `S (l + r)`. Разумеется, из курса алгебры мы знаем, что оба выражения равны друг другу, но Idris-компилятор изначально такими знаниями не обладает. Стало быть, ему нужно подсказать, как сравниваются такие элементы.

Для подсказки нужно ввести определение с помощью группы символов `?=`. Тем самым компилятору будет сообщено, что мы предоставляем отдельный фрагмент кода, предписывая способ доказательства нужного Idris факта. В данном случае ошибки содержатся в последней строке, поэтому в нее нужно внести следующие изменения:

```
| False ?= let r' = treeInsert x r in Node y l r'
```

Если этот файл загрузить в Idris еще раз, сообщений об ошибках не будет. Компилятор запоминает, что у нас есть для него доказательство равенства `l + S r = S (l + r)`. Все, что нужно передать Idris, можно показать, выполнив в интерпретаторе команду `:m`:

```
*src/Chapter15/BinaryTreeFV> :m
Global metavariables:
    [Chapter15.BinaryTreeFV.treeInsert_lemma_1]
```

Здесь показано, что остальному доказательству нужно дать конкретное имя `Chapter15.BinaryTreeFV.treeInsert_lemma_1`. В данном случае мы перейдем непосредственно к выводу и представим код исходного файла, устраивающий Idris. Обратите внимание на то, что имя доказательства в точности совпадает с именем, которое Idris предлагает нам после выполнения команды `:m`:

```
Chapter15.BinaryTreeFV.treeInsert_lemma_1 = proof
  intros
  rewrite sym (plusSuccRightSucc l r)
  trivial
```

По сути, код сообщает компилятору последовательность шагов по перезаписи, которые нужно совершить, чтобы левая сторона превратилась в правую. В частности, в нем используется функция `plusSuccRightSucc`, определенная в Idris Prelude:

```
plusSuccRightSucc : (left : Nat) -> (right : Nat) -> S (left + right) = left + (S right)
```

Как видите, это почти нужное нам равенство. Единственное отличие состоит в том, что по сравнению с тем, что нам нужно, вывод получился зеркальным. Для таких случаев нужно использовать свойство симметрии равенства (то есть `a = b` является эквивалентом `b = a`), применив комбинатор `sym`.

В данный момент может показаться, что нужно быть суперпрограммистом, чтобы добраться до фрагмента кода, который убедит Idris в конкретном факте. Однако это совсем не так: Idris-интерпретатор поддерживает интерактивный режим, помогающий получать нужные доказательства.

В качестве второго примера давайте попробуем проверить тот факт, что слияние двух деревьев приводит к созданию нового дерева с количеством внутренних узлов, равным их сумме в двух исходных деревьях. Как и в предыдущем примере, начнем с описания типа, который нужен для проверки исходного фрагмента кода:

```
treeMerge : Ord a => BinaryTree a n -> BinaryTree a m -> BinaryTree a (n + m)
treeMerge t1 Leaf = t1
treeMerge t1 (Node x2 l2 r2) = treeInsert x2 (treeMerge (treeMerge t1 l2) r2)
```

Если загрузить файл в Idris, интерпретатор сообщит, что не может объединить `n` с `plus n 0`, указав на первое предложение. Поэтому давайте заменим `=` на `?=` и загрузим файл. Появится другая ошибка, теперь уже во втором предложении. Если и в нем заменить `=` на `?=`, Idris загрузит файл. Но вы должны предоставить интерпретатору два доказательства, которые можно увидеть, запустив в приглашении интерпретатора команду `:m`.

```
*src/Chapter15/BinaryTreeFV> :m
Global metavariables:
    [Chapter15.BinaryTreeFV.treeMerge_lemma_2,Chapter15.BinaryTreeFV.treeMerge_
lemma_1]
```

Чтобы приступить к доказательству, введите команду `:p`, за которой укажите имя доказываемой леммы. Начнем с леммы 1:

```
*src/Chapter15/BinaryTreeFV> :p Chapter15.BinaryTreeFV.treeMerge_lemma_1
----------                  Goal:                  ----------
{ hole 0 }:
 (a : Type) ->
 (n : Nat) ->
 (Ord a) ->
 (BinaryTree a n) -> (BinaryTree a n) -> BinaryTree a (plus n Z)
```

Здесь говорится, что нам нужно предоставить значение данного длинного типа. Но это значение является функцией, следовательно, это эквивалентно предоставлению значения типа `BinaryTree a (plus n Z)`, что обусловлено множеством значений типов, следующих перед ним. С помощью команды `intros` можно сообщить Idris о необходимости сделать данный шаг и предположить, что у нас есть все эти значения:

```
-Chapter15.BinaryTreeFV.treeMerge_lemma_1> intros
----------              Other goals:              ----------
{ .. опущено .. }
----------              Assumptions:              ----------
 a : Type
 n : Nat
 c : Ord a
 t1 : BinaryTree a n
 value : BinaryTree a n
```

```
----------                  Goal:                  ----------
{ hole 5 }:
 BinaryTree a (plus n Z)
```

Теперь нам осталось доказать, что если взять в расчет все эти предположения, целевое значение может быть получено. Интуитивно можно понять, что в складывающейся ситуации Idris не знает, что `n = n + 0`. Если компилятор смог бы это доказать, он смог бы изменить целевое значение на `BinaryTree a n`, и тогда значение `value` было бы в точности той же формы, что и целевое значение.

Огромным хранилищем многих равенств, которым удовлетворяют натуральные числа, является модуль `Prelude.Nat`. В этом модуле можно найти переписанное правило, подобное тому, что нам нужно, а именно `plusZeroRightNeutral`, тип которого имеет вид `(left : Nat) -> left + 0 = left`. Однако нам нужно симметричное равенство, то есть `left = left + 0`. Мы можем сообщить Idris о необходимости применения правила посредством команды `rewrite` или команды `rewrite sym`, чтобы воспользоваться симметричной версией. Давайте задействуем последний вариант:

```
-Chapter15.BinaryTreeFV.treeMerge_lemma_1> rewrite sym (plusZeroRightNeutral n)
----------              Other goals:              ----------
{ .. опущено .. }
----------              Assumptions:              ----------
 a : Type
 n : Nat
 c : Ord a
 t1 : BinaryTree a n
 value : BinaryTree a n
----------                  Goal:                  ----------
{ hole 6 }:
 BinaryTree a n
```

Это применение правила показывает, что нам также нужно предоставить параметры, к которым это правило должно применяться. В данном случае таким параметром будет просто переменная `n`, находящаяся в целевом значении, которое нам нужно доказать. Во многих других случаях этим параметром будет более длинное выражение, например `plus n m`.

Мы уже почти достигли желаемого результата. Осталось только сообщить Idris, что искомое значение `value` эквивалентно целевому значению, поэтому остальная часть доказательства весьма тривиальна: мы просто смотрим на подчиненные понятия и проверяем, что они те же самые.

```
-Chapter15.BinaryTreeFV.treeMerge_lemma_1> trivial
treeMerge_lemma_1: No more goals.
```

В этом последнем сообщении говорится, что доказательство завершено. Если ввести команду `qed` (на латыни это означает «что и требовалось доказать»), Idris вернет нам код, необходимый для вставки в исходный файл:

```
-Chapter15.BinaryTreeFV.treeMerge_lemma_1> qed
Proof completed!
Chapter15.BinaryTreeFV.treeMerge_lemma_1 = proof
  intros
  rewrite sym (plusZeroRightNeutral n)
  trivial
```

Есть и еще одна ветвь, за которую можно взяться. Нашим путеводителем по решению соответствующей задачи станет упражнение 15.3.

УПРАЖНЕНИЕ 15.3. ДОКАЗАТЕЛЬСТВО ДЛЯ СЛИЯНИЯ ДЕРЕВЬЕВ

Завершите код для treeMerge. Воспользуйтесь интерактивным режимом Idris, чтобы найти доказательство, необходимое для второго предложения. Соответствующие правила, которые можно применить, показаны ниже. Возможно существование более одного доказательства, поэтому нам может понадобиться применение простого поднабора правил:

- plusSuccRightSucc : (left : Nat) -> (right : Nat) -> S (left + right) = left + (S right)
- plusAssociative : (left : Nat) -> (centre : Nat) -> (right : Nat) -> left + (centre + right) = (left + centre) + right
- plusCommutative : (left : Nat) -> (right : Nat) -> left + right = right + left

Не забудьте, что можно использовать правила с выражениями, состоящими не только из простых переменных. Например, plusCommutative (l + r) n представляет собой доказательство того, что (l + r) + n = n + (l + r).

После завершения доказательств получите код обратно посредством команды qed и включите его в исходный файл. Idris сможет загрузить весь файл без выдачи предупреждений, а выполнение команды :m должно сообщить вам о том, что метапеременных не осталось.

Как видите, интерпретатор для языков с зависимой типизацией является также *помощником сбора доказательств* (proof assistant). Такой вид помощника оказывает помощь в поиске всего множества правил, которое должно быть применено для проверки свойства. Помощники сбора доказательств могут стать еще эффективнее: в данном случае мы выяснили, как все вводится вручную, но у помощников также имеется механизм *тактик* (tactics), который позволяет доказывать многие свойства автоматически. В этой сфере ведется интенсивная работа, облегчающая формальную верификацию больших фрагментов кода. Строгие системы типов, включая системы с зависимой типизацией, помогают убедиться, что код поддерживает заявленные свойства, и тем самым делают программное обеспечение более безопасным и удобным в поддержке.

Выводы

Основное внимание в данной главе было уделено инструментальным средствам, помогающим в тестировании и сопровождении Haskell-кода:

- Утилита *Haddock* — наиболее предпочтительный инструмент для связывания фрагментов документации с Haskell-объявлениями, к тому же он обеспечивает красивую визуализацию документации.
- Система Cabal поддерживает включение тестов в пакеты: мы узнали, как объявляется *строфа* `test-suite`, и познакомились с несколькими *средами тестирования* в Haskell, включая Tasty и Hspec.
- Мы познакомились с библиотекой блочного тестирования *HUnit*.
- Мы прочитали о преимуществах *рандомизированного тестирования* и узнали, как с помощью библиотеки *QuickCheck* создавать такие виды тестов.
- В качестве последнего урока мы изучили примеры того, как зависимые типы могут помочь при *формальной верификации* исходного кода.

Создание архитектуры приложения

16

В книгах по программированию новый язык обычно изучается на небольших примерах. Однако реальные приложения, как правило, намного больше по объему, чем 10–12 строк, и требуют дальнейшей работы по проектированию, обслуживанию и оптимизации кода. В этой главе с помощью ряда рекомендаций предпринята попытка перекинуть мост между этими двумя мирами. Разумеется, предлагаемые рекомендации не являются истиной в последней инстанции, но могут послужить хорошей подсказкой по функциональному программированию.

Сначала мы сравним применение образцов проектирования в объектно-ориентированном и в функциональном программировании. Мы увидим, что расхождение не слишком большое, и многие из них преобразуются в свои функциональные эквиваленты, но с другой реализацией.

В данной книге представлено множество различных средств Haskell-программирования, включая среды тестирования, инструменты профилирования и т. д. В этой главе мы рассмотрим сводную информацию обо всех этих средствах, а также познакомимся с несколькими предложениями по использованию других полезных программ, рассмотреть которые более глубоко из-за ограниченного объема книги не удалось.

И наконец, мы узнаем о ряде программных проектов, позволяющих отточить навыки программировании на языке Haskell, причем в каждом проекте будет задействован конкретный паттерн проектирования. В будущем они смогут стать основой ваших Haskell-приложений. Задумка состоит в том, чтобы укрепить навыки применения известных вам приемов и познакомить вас с несколькими новыми приемами.

Паттерны проектирования и функциональное программирование

Возможно, вы слышали, что для функционального программирования паттерны проектирования совершенно не подходят. Однако это слишком сильное заявление. Разумеется, многие паттерны проектирования объектно-ориентированного программирования невозможно напрямую применить к совершенно другой парадигме программирования. Но это не означает, что программный проект, разрабатываемый с использованием языка Haskell, не нуждается в тщательном анализе и проек-

тировании, которые предваряют создание программного кода. Более того, часто встречающиеся и многократно используемые пути решения проблем (называемые «образцами») можно встретить и в функциональном коде.

Во многих случаях утверждения, касающиеся сути образцов проектирования, необоснованно распространяются на те из них, которые иногда называют «шаблонами кода». Посмотрите на паттерн проектирования Singleton (Одиночка), предназначенный для хранения в памяти одного экземпляра конкретного класса. Когда возникает потребность в его применении, мы точно знаем, что именно нужно написать, и это написание всегда выглядит одинаково. Это просто шаблонный код, и именно поэтому в таких языках, как Scala, для подобных случаев предлагается специальный синтаксис. В аналогичных сценариях решения на языке Haskell, как правило, абстрагируют образец на более высоком уровне, обычно в типовом классе или в функции высшего порядка.

В ряде других случаев особенности языка позволяют беспрепятственно интегрировать конкретный образец в сам язык. Одним из примеров является паттерн проектирования Strategy (Стратегия): он применяется для определения вычислений с рядом перемещаемых частей, зависящих от последующего рассмотрения: например, изменения в коде, показывающем общую сумму, зависят от заданной валюты. В объектно-ориентированном окружении мы бы определили абстрактный класс или интерфейс, а производные классы содержали бы код для каждой валюты. В Haskell вместо этого мы задействуем функцию высшего порядка, которая получает в качестве аргументов все эти изменяемые части.

С концептуальной точки зрения некоторые образцы остаются работоспособными. В любой программной системе, будь она разработана в процедурном, объектно-ориентированном или функциональном стиле, существует проблема несовместимости интерфейсов между компонентами. В мире объектно-ориентированного программирования был бы определен общий интерфейс и созданы классы-оболочки для обращения к функциональности каждого компонента, то есть был бы реализован известный образец Adapter (Адаптер). В Haskell мы бы вместо этого воспользовались классами типов и сделали каждый из компонентов экземпляром того или иного класса. Таким образом, у нас был бы общий метод для обмена данными между всеми компонентами. В данном случае проблема (обмен данными между несовместимыми системами) и замысел решения (определение общего протокола и создание адаптированных к нему систем) остаются одинаковыми, изменения касаются лишь реализации.

Внутри базовых библиотек можно найти функторы, свертываемые классы, аппликативы, монады и многие другие классы типов, находящиеся на верхнем уровне абстракции. В подобных случаях применение образца проектирования эквивалентно созданию экземпляра класса типов. Некоторые из классов типов придают образцу дополнительную обобщенность: образец Iterator (Итератор) в объектно-ориентированных языках позволяет вначале работать только со стартовыми значениями коллекции. А в Haskell мы располагаем всем диапазоном операций с коллекциями: класс `Functor` обеспечивает применение функции внутри контейнера, класс `Foldable` — многократное использование функции для получения конечного результата, класс `Traversable` — перебор элементов с сохранением структуры.

Как мы уже видели, некоторые образцы относятся к особенностям языка или к высокоуровневой абстракции. Разумеется, в воплощении множества других образцов исходная идея сохранена. Одним из примеров может послужить паттерн проектирования Pool (Пул), представляющий собой механизм эффективного управления множеством ресурсов. Мы встречаем этот образец во многих пакетах: например, именно пул используется на уровне доступа к базе данных Persistent, чтобы управлять соединениями с ней.

Конечно же, некоторые особенности Haskell, например неизменность значений, функции высшего порядка или ленивые вычисления, оказывают влияние на способ проектирования приложений. Давайте подумаем о параллелизме: в условиях императивности для управления доступом при совместном использовании ресурсов приходится задействовать блокировки, семафоры или механизмы синхронизации. Во многих случаях причина применения этих низкоуровневых операций заключается в возможности возникновения побочных эффектов в любой части кода. В то же время Haskell обеспечивает четкое разделение между чистым кодом и кодом с побочными эффектами. В результате возможны более элегантные решения этой проблемы: при работе в параллельном режиме в Haskell используется программная транзакционная память, в которой воплощена концепция транзакций.

И наконец, можно отметить, что философия и особенности Haskell очень хорошо согласуются с методологиями итеративной и гибкой разработки. Возможность тестирования функций непосредственно в интерпретаторе помогает проверять работоспособность кода по мере его написания. А с помощью библиотеки QuickCheck можно генерировать намного больше тестов, чем при написании их вручную. Кроме того, в Haskell очень легко оптимизировать код: функции высшего порядка позволяют получить каркас алгоритма, а затем и доступные варианты путем применения функций, а строгие типы дают гарантию того, что код не изменится каким-либо неожиданным образом.

У языка, подобного Haskell, есть много других достоинств, это касается, например, строгой типизации или четкого разделения между чистым кодом и кодом с побочными эффектами, о чем рассказывается в главе 1. В общем, Haskell предлагает нам иные перспективы в проектировании программного обеспечения, полезность которых программист ощущает ежедневно.

Рекомендации среднего порядка

В этом разделе мы не будем говорить о мелких вариантах оптимизации или шаблонах Haskell-кода, а познакомимся с рядом общих рекомендаций, облегчающих чтение кода и позволяющих сделать код более понятным и удобным в сопровождении.

Высокоуровневые комбинаторы

Использование таких функций, как `map`, `filter` или `fold`, сделает назначение вашего кода более понятным для будущего читателя. В главах 3 и 14 уже было показано, как рекурсия применительно к типу данных зачастую может быть преобразована в серию вызовов этих функций, поэтому нужно стремиться свести ее к минимуму.

Переработка кода

Как показано в предыдущем разделе, Haskell очень хорошо уживается с методологиями гибкого программирования. Возможность передачи функций в качестве аргументов может способствовать созданию функций, инкапсулирующих общие части разнообразных алгоритмов. Из многих классов задач можно выделить общую схему, а затем создавать экземпляры для каждого случая путем введения небольших недостающих частей.

Переработка кода на более высоком уровне имеет двойное преимущество. Прежде всего, сокращается объем кода, который придется поддерживать, и соответственно снижается вероятность внесения новых дефектов, поскольку код можно будет лучше протестировать. Больше того, при абстрагировании в коде общего образца лучше видны правильные решения и узкие места.

Более разумный подход к использованию классов типов

Классы типов являются весьма эффективным средством абстрагирования общих идиом нескольких типов данных. В Haskell-библиотеках содержится множество классов типов, и почти все важные концепции программирования реализуются как классы типов. Попробуйте последовать примеру этих библиотек и реализуйте свои концепции в виде классов типов.

Здесь необходимо сделать одно небольшое предостережение. При чрезмерном увлечении классами типов код сложно читать. В компиляторе применяется механизм разрешения экземпляров классов типов, поэтому конкретный выбранный экземпляр может напрямую не появляться. В этих случаях, возможно, лучше отдать предпочтение более прямому подходу, например воспользоваться абстрактными типами данных или функциями высшего порядка. В частности, нужно иметь в виду следующие два сценария:

- Если все классы типов имеют только один экземпляр, возможно, это тот самый случай, когда абстрагировать именно эти концепции действительно не следует.
- Не следует непосредственно проецировать на классы типов объектно-ориентированные классы или интерфейсы.

Применение инвариантов посредством системы типов

В главе 13 рассмотрено множество способов применения инвариантов в ваших значениях посредством строгих типов. Использование в коде этих приемов полезно как в краткосрочной перспективе, поскольку компилятор выявит больше ошибок, так и в долгосрочной, потому что разрушительные для модификации инварианты будут отвергаться.

Система типов может посодействовать в выявлении множества ошибок, даже если не применять все эти приемы. Хорошим примером может служить тип `newtype`: вы можете отделить друг от друга разные понятия (например, деньги, идентификаторы записи или расстояния), даже если они имеют одинаковое компьютерное представление (которое в данном случае является целым числом). В библиотеке

Persistent можно найти варианты использования типа `newtype` для разделения типов идентификаторов каждой сущности.

Сохранение чистоты и полиморфизма (насколько это возможно)

Если при написании функций стремиться отделять вычисления с побочными эффектами от всех остальных вычислений, тестирование кода пройдет намного легче. Одним из полезных образцов предусматривается создание ядра чистых функций для основных типов данных, применение множества QuickCheck-тестов, а также формальная верификация и построение всего остального вокруг этого ядра.

Когда же нужно работать с монадическими контекстами, попробуйте задействовать классы монад (например, `MonadState`, `MonadError` и т. д.), чтобы четко указать, какую функциональность вам требуется получить от стека монад. Решение с самого начала указывать весь стек монад ухудшает возможность поддержки кода, потому что зачастую возникает потребность в добавлении новой функциональности. Больше того, сохранение кода полиморфным позволит вам делать это по-разному: например, можно тестировать Persistent-код по отношению к списку, а не к базе данных; это возможно, если вместо текущего преобразователя `SqlPersistT` указать в своей сигнатуре класс монад `PersistQuery`.

Суть этой рекомендации состоит в том, что полиморфизм открывает двери для многократного использования кода. Если вместо функции, работающей со списком целочисленных значений, написать функцию, работающую с любыми значениями `Traversable`, чьи элементы имеют отношение к `Num`, можно будет изменять как представление контейнера (дерево вместо списка), так и тип чисел (`Integer` вместо `Int`), не внося при этом никаких изменений в код.

СОВЕТ

Множество дополнительных рекомендаций можно найти в статье Нейла Митчела (Neil Mitchell) «Hoogle Overview» в номере 12 журнала «The Monad.Reader» (этот журнал, посвященный Haskell, издается сообществом) и в ряде обсуждений и ответов на вопросы в разделе StackOverflow Дона Стюарта (Don Stewart).

Утилиты

В книге было представлено множество утилит. В данном разделе вкратце упомянуты многие из них, а также представлены некоторые другие. Охватить каждую не позволяет дефицит места, но у всех этих утилит есть неплохая подробная документация в Интернете.

Управление проектом и зависимостями

Для управления проектами в Haskell достойна выбора система *Cabal*. Читая эту книгу, вы могли убедиться, что она предлагает декларативный способ задания типа

создаваемого программного продукта, необходимых зависимостей и ключей компиляции. Каждый создаваемый продукт определяется в так называемой *строфе*.

Основные свои возможности Cabal черпает из огромного хранилища библиотек под названием *Hackage*. Это хранилище управляется Haskell-сообществом, члены которого выкладывают свои самые последние разработки, открывая к ним доступ всему миру. Используя команду `cabal install`, вы можете автоматически получить копию пакета и откомпилировать его.

ПРИМЕЧАНИЕ

При использовании кода из хранилищ, подобных Hackage, обращайте внимание на лицензии. Эти лицензии должны перечисляться в каждом Cabal-файле, чтобы проще было проверить, что лицензия конкретного пакета подходит именно для планируемого вами варианта применения.

Стиль кода

Нужно писать код, удовлетворяющий ограничениям компилятора на создание библиотеки или исполняемого файла. Однако даже код, не вызывающий возражений у компилятора, может быть трудным в поддержке или соответствовать такому образцу, который способен запутать будущих читателей этого кода.

Частью совета, которому рекомендуется строго следовать, является постоянное использование флага `-Wall` компилятора. Этот флаг заставляет компилятор не только выискивать ошибки, но и выдавать *предупреждения*. Для генерирования предупреждений, способствующих обнаружению неудачных фрагментов кода, которые впоследствии могут причинить неприятности, может быть также полезна утилита *HLint*. В EclipseFP тоже есть дополнительный модуль для Eclipse, который может на лету выдавать подсказки и вносить исправления, помогая совершенствовать свои навыки программирования.

Зачастую требуется, чтобы код соответствовал ряду стилевых установок, то есть набору рекомендаций, касающихся использованию отступов, переходов на новую строку, выбору мест вызова безымянных функций, определения функций в блоках `let` и т. д. Направлять ваши усилия в этой сфере поможет утилита *Stylish Haskell*: она читает код и выдает новую версию, соответствующую ряду конфигурационных параметров. Кроме того, ее можно интегрировать в среды EclipseFP, Emacs и vi, сделав частью повседневно используемой среды разработки.

Документация

В предыдущей главе подчеркивалась важность хорошо продуманной и актуальной документации. Как мы уже знаем, рекомендуемой для этого случая утилитой является *Haddock*. Основное преимущество Haddock — это размещение информации об элементе в непосредственной близости от самого элемента. Кроме того, Haddock используется для создания большого объема справочной документации на веб-сайте Hackage. И наконец, выводимая Haddock документация позволяет сразу же увидеть, не забыли ли вы задокументировать какую-нибудь функцию в своем коде.

Поиск функций и типов данных в большом количестве пакетов может занять довольно много времени. Ранее уже упоминалась утилита *Hoogle*, позволяющая вести поиск объявлений в Haskell-коде не только по имени или описанию, но и по типам, используемым в сигнатурах функций.

Тестирование и верификация

В главе 15 довольно много места было уделено тестированию с помощью утилит *HUnit*, *QuickCheck* и *Smallcheck*. Важность тестирования для успешной разработки проекта трудно переоценить. Большим преимуществом ориентированных на свойства утилит тестирования, к числу которых относятся QuickCheck и SmallCheck, является то, что требуемое поведение программы задается на высшем уровне. Затем генерируются небольшие блочные тесты для различных сценариев. Таким образом удается охватить варианты поведения на более высоком уровне, чем тот, который предлагается традиционными инструментальными средствами.

Программирование на уровне типов и *зависимые типы* открывают двери к формальной верификации ряда свойств ваших типов данных и алгоритмов непосредственно внутри языка Haskell. Хотя на это тратится больше времени, чем при простом тестировании, это единственная технология, способная гарантировать полное отсутствие дефектов.

Контрольные испытания

Функциональная корректность, несомненно, является для фрагмента кода весьма важным оценочным показателем. Однако во многих случаях приложение должно выполняться с заданным уровнем производительности. Определять варианты тестов и получать статистические данные относительно времени выполнения поможет утилита *Criterion*. Оно запускает тесты на время, вполне достаточное для того, чтобы сделать продолжительность выполнения статистически значимой, а также вычисляет результат в целом (в некоторых случаях для определенных вычислений провести измерения невозможно из-за их ленивой природы).

Профилирование

В главе 15 был представлен *GHC-профайлер*, позволяющий собирать информацию о потреблении приложениями времени и памяти. По причине характерной для Haskell ленивой природы вычислений профилирование памяти играет более важную роль, чем в других языках. Разумное использование профайлера может сократить время расследования причин возникновения проблем производительности и указать на те места, которые требуют более тщательной оптимизации.

Основным недостатком профайлера является то, что он не рассчитан на многопоточные приложения, например такие, которые могут быть созданы с применением библиотек, представленных в главе 8. Для таких случаев будет интересно рассмотреть возможность применения графической утилиты *ThreadScope*, позволяющей читать журналы GHC-событий. Эти журналы включают информацию о том, когда

создаются и прекращают свою работу различные программные потоки, а также об активности каждого из них.

Охват

При проектировании тестов важно обеспечить охват каждого возможного пути выполнения кода. То есть нужно гарантировать, что охвачены все возможные ветви условий, все возможные образцы типов данных и сопоставлений и т. д. Включенная в GHC утилита `hpc` собирает важную статистическую информацию о конкретном выполняемом коде. Этой информацией можно воспользоваться для создания отчета с различными вариантами охвата и достигнутого процентного показателя.

Удаленный мониторинг

Зачастую приложения в силу своей природы работают не в ограниченном жизненном пространстве, а в качестве сервера. Одним из примеров такого рода являются веб-приложения. Пакет `ekg` позволяет получать статистику производительности и поведения приложения в процессе его выполнения. Кроме того, он делает это посредством веб-интерфейса, поэтому управлять им довольно просто. При использовании `ekg` вы не ограничены информацией, предоставляемой по умолчанию, а можете включать собственные счетчики. Например, для веб-приложения могут представлять интерес сведения о количестве страниц, обслуживаемых за минуту, или о количестве сохраняемых за определенный период подключений к базе данных.

Проекты

Полное раскрытие вопросов проектирования приложений и библиотек в Haskell, как и выявление в функциональном коде повторяющихся образцов, вполне определенно может составить тему отдельной книги. Вместо этого в данном разделе мы рассмотрим систему создания программного обеспечения и вопросы проектирования, которые ставятся во главу угла, если такая система разрабатывается на языке Haskell. Исходя из этих соображений можно будет предложить образцы для их применения в своих проектах.

Кроме того, эти проекты могут рассматриваться в качестве дополнительных более сложных упражнений для наработки навыков программирования на языке Haskell. В каждом из них решение должно быть в большей степени ориентировано на расширение по сравнению с заданиями из остальной части книги. А один совет подойдет для всех случаев жизни: для решения текущих проблем ищите готовую функциональность в библиотеках наподобие тех, что находятся в хранилище Hackage.

Библиотека для анализа данных

Вторая часть этой книги посвящена разработке двух алгоритмов анализа данных: алгоритму K-средних для кластеризации и априорному алгоритму для исследования ассоциативных правил. Этот наш первый проект подразумевает превращение

уже имеющегося кода в полноценную библиотеку для анализа данных с поддержкой различных алгоритмов для решения каждой задачи.

Даже если рассматривается всего одна задача, например кластеризация, интересно взглянуть на *общие черты* каждого алгоритма ее решения. Например, данные могут поступать из разных источников (списка, канала `Source`, запроса к базе данных), но мы должны предоставить для всех этих источников общий интерфейс, чтобы алгоритмы не зависели от выбора источника. В Haskell этот уровень абстракции достигается за счет *классов типов*. В данном случае может быть создан класс типов `ClusteringDataSource`.

Хорошая библиотека включает в себя не только типы данных, функции и классы типов, но и разъяснения правил и свойств, которые должны учитываться пользователем. Для типов данных очень полезно иметь в виду, является ли заданный тип экземпляром одной из общих абстракций, доступных в Platform. Например:

- Многие типы данных составлены так, чтобы можно было генерировать новое значение из двух других значений. С этим случаем хорошо справляется класс типов `Monoid`. Превращение типа в моноид может навести на мысль, что в вашем случае мог бы быть полезен нейтральный элемент.
- Другие типы больше вписываются в интуитивные понятия о контейнерах. В таких случаях постарайтесь создать экземпляры классов типов `Functor`, `Foldable` и `Traversable`.
- Те типы данных, которые подразумевают какой-нибудь вычислительный контекст или каким-то образом описывают множество предпринимаемых действий, хорошо вписываются в абстракции `Applicative` и `Monad`. Кроме того, если ваш тип допускает выбор, помочь могут такие абстракции, как `Alternative` и `MonadPlus`.

На рис. 16.1 показано большинство из этих классов типов, а также их родственные связи. Следует учесть, что номинально `Applicative` не является суперклассом `Monad`, равно как и `Alternative` не является суперклассом `MonadPlus`.

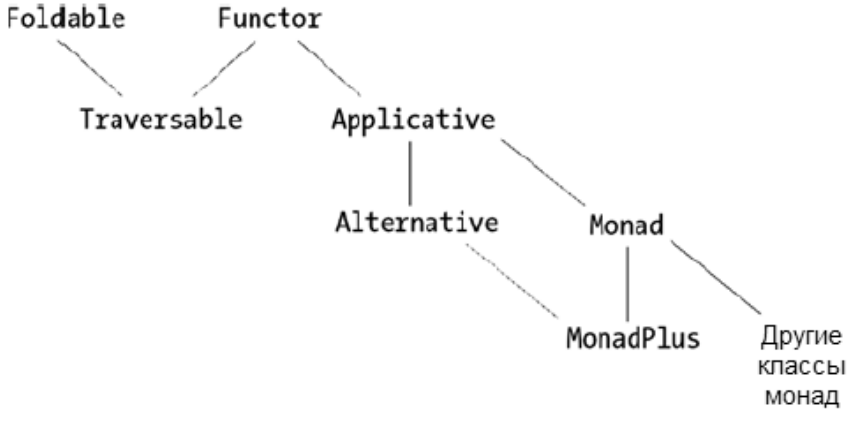


Рис. 16.1. Важные классы типов, доступные в Haskell Platform

При написании экземпляров этих широко известных абстракций следует соблюдать правила, с которыми они поставляются, даже если их соблюдение не проверяется компилятором. Пользователи вашей библиотеки будут очень удивлены, столкнувшись с экземпляром класса типов `Functor`, который, к примеру, не соблюдает правило `fmap id = id`. Общим является *правило «никаких сюрпризов»* (no-surprises rule), которое иногда называют *принципом наименьшего удивления* (principle of least astonishment): заставьте библиотеку делать только то, что от нее ожидают пользователи, либо предоставьте исчерпывающее описание ее поведения. Убедиться в неизменном соблюдении правил поможет формальная верификация или утилита QuickCheck.

Дополнительный анализ, который необходимо провести, касается производительности. Нужно понимать, что ваши прежние идеи насчет оптимизации Haskell-кода могут оказаться бесполезными там, где в игру вступают ленивые вычисления. Мы уже упоминали такие весьма ценные средства выявления дефицита производительности и его причин, как GHC-профайлер и пакет Criterion. Если вас интересует комплексный анализ Haskell-кода, вам непременно нужно прочитать книгу Криса Окасаки (Chris Okasaki) «Purely Functional Data Structures», выпущенную издательством Cambridge University Press в июне 1999 года.

Идеальным примером использования общих абстракций в конкретной обстановке является библиотека HLearn. Соответствующий пакет ориентирован на машинное обучение и предлагает множество алгоритмов для нескольких заданий и распределений. Например, для представления множеств данных используется класс типов `Monoid`.

НАПИСАНИЕ ПРИЛОЖЕНИЙ В ВИДЕ БИБЛИОТЕК

Разработка программного обеспечения может существенно выиграть, если предложить приложение в виде базовой библиотеки, которая впоследствии будет использоваться приложением внешнего интерфейса. Cabal воплощает эту стратегию, допуская как наличие строфы для библиотечной части, так и возможность ссылки на нее в исполняемой части.

Поводом для такого совета стала следующая мысль: зная, что другой разработчик может воспользоваться вашим кодом, вы, как правило, станете более тщательно проводить оптимизацию экспонируемых абстракций. Кроме того, следование данному совету упростит оптимизацию и многократное использование базовых составляющих вашего проекта, к тому же вам не придется возиться с кодом, призванным обеспечить взаимодействие с пользователем.

Сетевой клиент магазина

Во внутренней сети магазина необходимо создать сервер и ряд клиентов, которые могли бы использоваться работниками. В такого рода приложениях требуется самая разная функциональность: сохранение внутреннего согласованного состояния, обмен данными по сети, ведение журнала проблемных сценариев и т. д. Исходя из всего, о чем рассказано в главе 7, подобная функциональность идеально вписывается в стек монад, собирающий воедино механизмы из разных частей. В данном случае возможный выбор преобразователей монад может быть следующим:

- Для управления сетевыми подключениями подойдут монада `IO` и преобразователь `ConduitM`, если же вы отдаете предпочтение модели акторов, выбор следует сделать в пользу монады `Process`.
- Для обращения через сеть к базе данных Persistent следует выбрать преобразователь монад `SqlPersistT`.
- Монада `STM` может использоваться для сохранения согласованного внутреннего состояния в условиях одновременно проводимых модификаций. Если же вы являетесь приверженцем модели акторов, состояние может сохраняться более простым преобразователем `StateT`.
- Различные параметры конфигурации, например порт, используемый для обмена данными, или параметр подключения к базе данных, могут отслеживаться с помощью преобразователя монад `ReaderT`.
- Функциональность ведения журнала поможет реализовать преобразователь монад `WriterT`.

Однако вы не сможете просто создать большой стек монад и передавать его. Как уже упоминалось, код необходимо делать максимально полиморфным. Если вместо `ReaderT r (SqlPersistT IO)` использовать `MonadReader`, можно будет по мере разработки приложения увеличивать или уменьшать стек монад. В частности, не таким уж необычным станет решение о добавлении нового уровня. Функция, осуществляющая запрос к базе данных и регистрирующая некоторую информацию, должна иметь следующую сигнатуру:

```
getClientInformation :: (PersistQuery m, MonadWriter String m) => ClientId -> m
[Client]
```

Как видите, она отличается от более конкретной структуры:

```
getClientInformation :: ClientId -> WriterT String (SqlPersistM [Client])
```

Кроме того, разработка с опорой на классы монад помогает разделить задачи, которые призвано решать ваше приложение. В этом смысле применение классов монад напоминает *аспектно-ориентированное программирование* (aspect oriented programming), когда вы определяете каждую часть приложения, самостоятельно реализующую какой-то механизм, а затем собираете эти части воедино. Сигнатура, в которой используется список классов монад, необходимых для реализации той или иной функции, в точности определяет, какая функциональность потребуется, исходя из данного контекста. Задание конкретного воплощения посредством стека монад эквивалентно выбору реализации.

В табл. 16.1 представлены наиболее важные классы монад, а также монады (выделены курсивом) и преобразователи монад (обычный шрифт), которые являются экземплярами каждого класса наряду с указанием наиболее важных операций каждого класса. В этой таблице вы найдете больше монад, чем во всей книге. В частности, в ней можно найти две разновидности дополнений:

- В некоторых случаях представлена конкретная монада, а не класс монад, который абстрагирует образец. Это относится и к монаде `Par`, реализующей классы типов `ParFuture` и `ParIVar`, а также класс `MonadRandom`.

- Другие монады являются совершенно новыми, но они заслуживают упоминания. К этому разряду относится монада `MonadSupply`, используемая для приобретения источника новых значений (которые, к примеру, можно использовать в качестве уникальных идентификаторов).

Таблица 16.1. Обычные и полезные классы монад

Класс	Доступ	Пакет	Описание	Операции
	`IdentityT`	`mtl`	Применение функции: без дополнительных эффектов	Нет
`MonadPlus`	`MaybeT` `ListT`	`mtl` `mtl`	Выбор и неудачи	`mzero`: представление сбоя `mplus`: выбор между двумя результатами `guard`: проверка булева условия
`MonadLogic`	`LogicT`	`logict`	Возврат к предыдущему состоянию и честное чередование	`interleave`: честная дизъюнкция `(>>-)`: честная конъюнкция `ifte`: условная проверка с отсечением `once`: фиксация первого ответа и отсечение
`MonadZip`	`ListT`	`mtl`	Параллелизм	`mzip`: преобразование двух списков в список пар `munzip`: преобразование списка пар в два списка
`MonadReader`	`ReaderT r` `Monoid w =>` `RWST r w s`	`mtl` `mtl`	Добавление контекста только для чтения	`ask`: получение значения из контекста `local`: выполнение вычислений с новым контекстом
`MonadWriter`	`Monoid w =>` `WriterT w` `Monoid w =>` `RWST r w s`	`mtl` `mtl`	Вывод данных только для записи путем добавления нескольких значений	`tell`: добавление нового значения `listen`: получение выводимых данных подвычислений
`MonadState`	`StateT s` `Monoid w =>` `RWST r w s`	`mtl` `mtl`	Сохранение внутреннего состояния, которое можно читать и модифицировать	`get`: получение текущего значения состояния `put`: задание нового значения состояния `modify`: применение функции к состоянию
	`ST`	`base`	Только изменяемые переменные	Создание и модификация значений IORef

Класс	Доступ	Пакет	Описание	Операции
`MonadSupply`	`Monoid s =>` `SupplyT s`	`monad-supply`	Потребление значений из источника	`supply`: получение следующего значения
`MonadError`	*`IO`* `MaybeT` `EitherT e` `Error e =>` `ErrorT e`	`base` `mtl` `either` `mtl`	Неудача с дополнительной информацией: в зависимости от монады неудача представляется как чистые ошибки или как расширяемые исключения	`throwError`: сигнал о неудаче `catchError`: восстановление после ошибки
`MonadCatch`	*`IO`* `CatchT ==` `EitherT` `SomeException`	`base` `exceptions`	Генерация и перехват расширяемых исключений	`throwM`: генерация исключения `catch`, `handle`: восстановление для одного из типов исключений `catches`, `handles`: восстановление для нескольких типов исключений `bracket`: приобретение и удаление ресурсов
`MonadIO`	*`IO`* *`ParIO`* *`Process`*	`base` `monad-par` `distr-process`[1]	Выполнение неограниченных побочных эффектов, таких как чтение и запись файлов или обмен данными по сети	`liftIO`: подтягивание вычисления в монаду ввода-вывода внутри текущего стека монад
`MonadRandom`	*`IO`* `RandT`	`base` `MonadRandom`	Генерирование случайных значений	`getRandom`, `randomIO`: получение неограниченного случайного значения `getRandomR`, `randomRIO`: случайное значение с границами
`ParFuture`	*`Par`* *`ParIO`*	`monad-par`	Параллелизм на основе обещаний	`spawn`, `spawnP`: асинхронное выполнение функции и возвращение переменной `IVar`, дающей ее результат `get`: получение результата внутри `IVar` с блокировкой по необходимости
`ParIVar`	*`Par`* *`ParIO`*	`monad-par`	Параллелизм потока данных, где зависимости задаются через переменные `IVar`	`fork`: запуск параллельных вычислений `new`: создание новой переменной `IVar` для хранения значения `put`: запись значения внутри `IVar`

[1] distr-process ссылается на набор пакетов distributed-process.

Таблица 16.1 (продолжение)

Класс	Доступ	Пакет	Описание	Операции
	Eval	`parallel`	Детерминированный параллелизм на основе стратегий для вычисления ленивых типов	`rseq`, `rdeepseq`: последовательное вычисление аргументов `rpar`: параллельное вычисление аргументов
	STM	`stm`	Атомарные транзакции	`atomically`: выполнение транзакции атомарным образом `retry`: откат текущей транзакции и попытка ее повторного выполнения при изменении обстоятельств `orElse`: выполнение транзакции, если какая-нибудь другая транзакция оказалась неудачной Создание и модификация переменных `TVar`, `TQueue` и др.
	Process	`distr-process`	Актор одновременных вычислений, способный выполняться распределенно	`spawn`: создание нового актора в определенном узле `send`: отправка актору сообщения `expect`, `receiveWait`: получение сообщения определенного типа
`MonadResource`	`ResourceT`	`resourcet`	Безопасное выделение и освобождение ресурсов	`allocate`: получение ресурса и регистрация действия, необходимого для его финального освобождения `release`: досрочное освобождение ресурса
	`ConduitM i o`	`conduit`	Потоковая передача данных	`await`: получение следующего элемента из входного потока `leftover`: возвращение элемента во входной поток `yield`: генерирование элемента в выходной поток
	Parser	`attoparsec`	Соответствие списку символов предопределенному образцу	`Parser` чаще всего используется через интерфейс `Applicative`
`PersistStore`	`SqlPersistT`	`persistent`	Получение записей и управление ими в базе данных с использованием их ключей	get: получение записи по заданному ключу `insert`: создание новой записи в базе данных `repsert`: замена содержимого записи новой информацией или создание новой записи, если заданный ключ не существует `delete`: удаление записи по заданному ключу

Класс	Доступ	Пакет	Описание	Операции
`PersistUnique`			Получение записей и управление ими в базе данных с использованием уникальных ограничений	`getBy`: получение записи с заданным уникальным ограничением `insertUnique`: вставка проверяемых уникальных ограничений `deleteBy`: удаление записи с заданным ограничением
`PersistQuery`			Получение записей и управление ими в базе данных посредством запросов	`selectSource`, `selectList`: получение записи, удовлетворяющей заданному набору условий `update`: изменение информации заданной записи `updateWhere`: изменение всех записей из запроса `deleteWhere`: удаление всех записей из запроса
`MonadCont`	`ContT`	`mtl`	Вычисления, которые могут быть прерваны и возобновлены	`callCC`: вызовы функции с текущим продолжением ее работы
`MonadFree`	`FreeT f`	`free`	Свободная монада над функтором	
`MonadTrans`	Все преобразователи монад	`mtl`	Класс типов, экземплярами которого являются все преобразователи монад	`lift`: подтягивание вычислений в стеке на один уровень выше

В Haskell при разработке большинства веб-сред используется подход с опорой на классы монад, что является стимулом для создания собственного приложения путем наслаивания монад друг на друга. Одним из примеров может послужить библиотека Persistent. Применяемый там основной тип `SqlPersistM`, предназначенный для обращения к базе данных, определен в виде стека монад:

```
type SqlPersistM = SqlPersistT (NoLoggingT (ResourceT IO))
```

И в Yesod, и в Snap, и в Scotty определяются собственные преобразователи монад, которые можно объединять с любым необходимым вам уровнем базы данных, ведения журнала или конфигурирования.

Интерфейс администрирования и тетрис

Интерфейс администрирования магазина является средством управления товарами и запасами в системе, а также изменения покупок и отказа от них клиентов. Что касается тетриса, то это отличная игра для развлечения путешественников, пока машины времени творят свою магию. Общим у этих двух приложений является ограниченный набор совершаемых действий. В интерфейсе администрирования в зависимости от применяемого уровня безопасности можно получить доступ только к определенной функциональности, в игре «тетрис» можно только вращать текущую фигуру или перемещать ее по трем направлениям.

В главе 14 показано, как предметно-ориентированные языки помогают накладывать ограничения на значение, которое может быть представлено типом данных. В этом случае цель заключается в ограничении данных, которые могут быть выражены и обработаны приложением. Но действуя в том же направлении, можно создать *предметно-ориентированный язык* для *действий*, доступных вашему приложению. Существует несколько примеров применения этого образца: Esqueleto воплощает в себе SQL-подобный язык для выражения запросов к базе данных, `attoparsec` содержит язык, основанный на интерфейсе Applicative для описания парсеров, а Scotty использует собственный предметно-ориентированный язык для задания маршрутов в веб-приложении.

У подхода, предполагающего использование предметно-ориентированного языка, есть два основных преимущества:

- Прежде всего, он ограничивает действия, которые могут выполняться в определенном контексте. Это не является аналогом наличия сигнатуры при использовании типа `IO`, где допустимы неограниченные побочные эффекты, поскольку здесь для вашего приложения имеется специальная монада, допускающая подключение к сети и к базе данных. Система типов может предложить намного больше инвариантов.
- Если имеет место глубокое внедрение действий, то можно предоставить несколько интерпретаций одного и того же предметно-ориентированного языка. Особенно это может пригодиться при тестировании: например, если ваш предметно-ориентированный язык используется для обмена данными по сети, можно предоставить интерпретацию, которая имитирует сетевое соединение между двумя одноранговыми узлами.

Среди Haskell-библиотек и Haskell-приложений существует два разных способа реализации предметно-ориентированных языков для действий. Примеры обоих способов уже встречались в этой книге:

- Одна из возможностей заключается в разработке *библиотеки комбинаторов*. В этом случае определяются набор основных конструкций и серия функций, объединяющих эти базовые блоки. Подобный подход принят в библиотеке `attoparsec` для построения парсеров. Одним из его преимуществ является то, что вы можете исследовать структуру вычислений перед их фактическим выполнением, добавляя новые функциональные возможности или проводя оптимизацию.
- Другим популярным вариантом является *создание собственной монады*, то есть предоставление монадического интерфейса множеству поддерживаемых операций. С этой точки зрения могут рассматриваться многие монады. К примеру, монада `State` является ограниченным языком для выражения значений, которые могут быть считаны из контекста и обновлены. Весьма успешным примером построения приложения вокруг монады может служить диспетчер окон XMonad. Хотя он написан на языке Haskell, этот же образец можно также найти и в среде Scotty, где определяется преобразователь монад `ScottyT`, и в языке Esqueleto, где запросы выражаются в виде значений монады `SqlQuery`.

Создание собственной монады

Монады в Haskell-коде очень популярны. Специальная do-нотация предоставляет удобный синтаксис для выстраивания последовательностей и составления действий, к тому же существует множество библиотек и функций, работающих в монадическом коде. Поэтому написание монад представляется вполне естественным вариантом разработки предметно-ориентированного языка.

В Hackage имеется несколько пакетов, упрощающих создание таких монад. В качестве примера можно привести пакет `operational` или `free`. Основное внимание в данном разделе уделено последнему из них. В любом случае все эти пакеты включают абстракции, являющиеся «последовательностью действий»: каждое монадическое значение является списком элементарных операций. Для создания полноценной монады нужно предоставить два набора данных:

- Элементарные операции, которыми можно воспользоваться, и строительные блоки вашей монады.
- Сведения о том, как каждая операция влияет на связанный с ней контекст и какое значение предоставляется следующему вычислению в списке.

В качестве рабочего примера давайте рассмотрим компактный предметно-ориентированный язык, позволяющий управлять базой данных клиентов. Основными операциями будут добавление нового клиента, запрос клиента по его идентификатору и замена информации о клиенте. То есть в некотором смысле у нас получится ограниченная версия базы данных Persistent. Для определения клиентов требуются следующие типы данных:

```
newtype ClientId = ClientId Integer           deriving Show
data Client = Client { clientName :: String } deriving Show
```

При использовании пакета `free` каждая операция кодируется в виде конструктора в общем типе данных. Для каждой операции сначала нужно написать аргументы разрабатываемой операции. Например, функция записи в базу данных информации о новом клиенте получает в качестве аргумента клиента. Затем нужно будет определить, как результат операции проходит к следующему этапу вычислений. Это делается путем запроса функции из типа результата к еще не известному типу данных, который будет ссылаться на список вычислений. В примере новых клиентов результатом является `ClientId`, и этот результат должен пройти к следующему вычислению `r`. Код для всех операций имеет следующий вид:

```
data AdminOp r = GetClient  ClientId        (Client   -> r)
               | SaveClient ClientId Client r
               | NewClient  Client          (ClientId -> r)
```

Обратите внимание на то, что операция `SaveClient` вообще не возвращает значения, поэтому следующая операция не получает никакого аргумента. Таким образом, вам остается только указать `r` в конструкторе данных.

Пакет `free` требует, чтобы каждый тип данных операции следовал этой схеме, а также чтобы тип был экземпляром класса типов `Functor`. Позитивно то, что вы опять можете использовать функциональность `deriving` в GHC и генерировать такой экземпляр автоматически. Тогда код может приобрести следующий вид:

```
{-# LANGUAGE DeriveFunctor #-}

data AdminOp r = ...
               deriving Functor
```

Теперь ваша монада по отношению к рассматриваемому типовому классу `Functor` является *свободной монадой* (free monad). Сослаться на тип `Free`, параметризированный вашими операциями, можно в любой момент, но обычно, как в следующем примере, определяется синоним типа:

```
import Control.Monad.Free

type Admin = Free AdminOp
```

А вот использовать тип данных непосредственно в do-блоке по-прежнему нельзя. Сначала нужно подтянуть свои операции в свободную монаду. Поэтому нужно задействовать функцию `liftF` из `Control.Monad.Free`. Итак, остался последний штрих — разобраться, что предоставляется каждой операции в качестве последнего аргумента. Как правило, если в качестве аргумента должен быть передан следующий список операций, нужно предоставить `id`, иначе — `()`. В данном случае это означает следующее:

```
getClient :: ClientId -> Admin Client
getClient i = liftF $ GetClient i id

saveClient :: ClientId -> Client -> Admin ()
saveClient i c = liftF $ SaveClient i c ()

newClient :: Client -> Admin ClientId
newClient c = liftF $ NewClient c id
```

Итак, у нас, наконец, есть монада! Ее даже можно использовать внутри do-нотации:

```
import Data.Char

exampleAdmin :: String -> Admin String
exampleAdmin s = do i <- newClient $ Client s
                    n <- fmap clientName $ getClient i
                    return $ map toUpper n
```

Однако сама по себе функция `exampleAdmin` ничего не сделает. На данный момент это лишь описание вычислений, которые должны выполняться. Пакет `free` представляет список операций с использованием двух конструкторов: конструктор `Free` применяется, когда за одной операцией следует список других операций, а конструктор `Pure` служит меткой окончания вычислений. Это означает, что предыдущий пример эквивалентен следующему коду:

```
Free (NewClient (Client s) (\i ->
  Free (fmap clientName (GetClient i (\n ->
    Pure $ map toUpper n)))))
```

Финальным шагом является предоставление интерпретации каждой операции. Показанный далее пример интерпретирует операции в качестве выполняемых для ассоциативного списка идентификаторов и клиентов. Нетрудно заметить, что интерфейс очень похож на функции `run` других монад.

```
import Data.List

runAdmin :: Admin a -> ([(Integer,Client)],a)
runAdmin m = runAdmin' m []
  where runAdmin' (Free (GetClient  (ClientId i) n))   l =
          let Just c = lookup i l in runAdmin' (n c) l
        runAdmin' (Free (SaveClient (ClientId i) c n)) l =
          let l' = deleteBy (\(j,_) (k,_) -> j == k) (i, c) l
           in runAdmin' n $ (i,c):l'
        runAdmin' (Free (NewClient c n)) [] =
          runAdmin' (n $ ClientId 1) [(1,c)]
        runAdmin' (Free (NewClient c n)) l =
          let (i',_) = maximumBy (\(j,_) (k,_) -> compare j k) l
           in runAdmin' (n $ ClientId (i' + 1)) $ (i' + 1,c):l
        runAdmin' (Pure c)                   l = (l, c)
```

В данном случае эта интерпретация может использоваться в целях тестирования, а другая интерпретация обеспечит реальный доступ к базе данных. Проверить работоспособность нашей монады можно в интерпретаторе:

```
*Chapter16.FreeMonads> runAdmin $ exampleAdmin "Alejandro"
([(1,Client {clientName = "Alejandro"})], "ALEJANDRO")
```

Пакет `free` предлагает массу других функциональных возможностей для создания своих монад. Например, вместо простой монады можно решиться на создание преобразователя монад, просто используя `FreeT` вместо `Free`. Если вы собираетесь строить приложение вокруг собственной нестандартной монады, что в конечном итоге может обеспечить повышение производительности, полезно будет прочитать документацию по модулю `Control.Monad.Free.Church`. И наконец, пакет `free` предо-

ставляет не только свободные монады, но и свободные классы типов `Applicative`, `Alternative` и `MonadPlus`.

Выводы

Данная глава представляет собой взгляд с «высоты птичьего полета» на проектирование с использованием языка Haskell и функциональных образцов:

- Мы обсудили отношения между функциональным проектированием и традиционными образцами объектно-ориентированного проектирования: одни образцы остаются неизменными, в других изменяется способ их реализации, а надобность в третьих вообще отпадает.
- Мы совершили короткую прогулку в мир многочисленных инструментов, предлагаемых Haskell Platform и Hackage для документирования, тестирования, профилирования и управления проектом.
- Мы вспомнили о наиболее важных классах типов и связях между ними.
- Мы обсудили проектирование на основе преобразователей монад, а в конце главы познакомились с кратким обзором полезных классов монад.
- И наконец, мы узнали, как использовать предметно-ориентированный язык для программирования действий в приложении. Этот язык может быть реализован в виде библиотеки комбинаторов или путем создания собственной монады (например, с типом `Free`).

Приложение А. Дальнейшие перспективы

Поздравляем вас с тем, что вы добрались до этого места. Теперь, когда вы изучили многие функциональные возможности как самого языка Haskell, так и его библиотек, такие понятия, как функтор, монада или GADT, уже не должны вас пугать. Кроме того, вы разобрались с вопросами проектирования больших приложений в Haskell, а также с вопросами управления базами данных и веб-приложениями. И самое главное — вы поняли, что строгая система типов способствует более методичному написанию программ и помогает сократить время программирования и сопровождения.

Как и в случае любого другого языка, в книгу невозможно поместить каждый отдельно взятый вариант применения языка Haskell. Отправной точкой для исследования многочисленных Haskell-библиотек должно стать хранилище Hackage, где вся эта невероятная масса функциональности собрана в пакетах. Книга лишь приоткрыла завесу над библиотеками и средами веб-разработки кода для параллельных и одновременных вычислений, показав два места, где можно углубить свои знания. Продолжает появляться множество новых абстракций: весьма интересный подход к созданию изменяющихся во времени приложений, таких как пользовательские интерфейсы, в наши дни предлагает функциональное реактивное программирование.

Язык Haskell обладает множеством сюрпризов, встроенных в его компилятор. В главе 14 было упомянуто обобщенное программирование типов данных, которое наряду с программированием на уровне типов может еще больше сократить объем верифицируемого шаблонного кода при повышении уровня его безопасности. В этой книге довольно интенсивно использовался механизм метапрограммирования Template Haskell, хотя описание способов создания собственных квазицитат в книгу не вошло. Еще одной областью, где может ожидаться появление новых приемов написания программ, является программирование на уровне видов (kind-level programming).

Остается надеяться, что эта книга не станет конечным пунктом вашего путешествия в Haskell и в функциональное программирование, а подтолкнет вас к использованию этого языка в повседневной практике или в проектах, создаваемых ради интереса, а также станет поводом познакомиться с множеством информационных ресурсов, доступных в Интернете. Спасибо за то, что вы прочитали эту книгу!

Haskell-ресурсы

В Интернете имеется очень активное Haskell-сообщество, есть также огромная база готовых пакетов, кипучая деятельность прослеживается на форумах и в подписках. Среди многочисленных ресурсов, посещаемых с целью получения дополнительной информации, можно отметить следующие:

- На главной странице Haskell по адресу http://www.haskell.org есть указатели на многие руководства и ресурсы. Она построена по принципу вики, ее пользователи постоянно добавляют новую информацию, которая может заинтересовать Haskell-разработчиков.
- Периодическое издание, посвященное Haskell, журнал «Monad Reader», специализируется на статьях практического и просветительского направления, предназначенных для разработчиков. Все выходящие публикации можно найти по адресу http://themonadreader.wordpress.com.
- Если нужно посмотреть на элегантный и поучительный функциональный код, можно заглянуть в раздел «Functional Pearls» в Haskell-вики по адресу http://www.haskell.org/haskellwiki/Research_papers/Functional_pearls.
- Чтобы получать подпитку свежими новостями о Haskell и Hackage, можно подписаться на подборку новостей «Planet Haskell» по адресу http://planet.haskell.org. Вы убедитесь в наличии достаточного числа блогеров, старающихся поднимать и поддерживать тему Haskell.
- Еще одним неплохим способом быть на гребне волны развития Haskell является участие в подразделе новостного сайта по адресу http://www.reddit.com/r/haskell/. Поскольку здесь можно комментировать статьи, на сайте разворачивается немало весьма интересных дискуссий.
- Haskell-сообщество поддерживает подписку на рассылку под названием Caf , находящуюся по адресу http://www.haskell.org/mailman/listinfo/haskell-cafe. На ее просторах ведется множество дискуссий относительно языка и библиотек. Если просмотреть архивы, можно заметить, что подписка приветствует активность как новичков, так и опытных разработчиков. Если у вас есть вопросы, просто задавайте их, и вы получите весьма вежливые ответы.
- Если предпочтение отдается более прямым контактам, можно воспользоваться IRC. Канал #haskell в irc.freenode.net обычно полон людей, обсуждающих язык.
- Можно также обратиться к Haskell-сообществу на Google+ и поделиться своими проблемами и опытом.

Другие функциональные языки

Большинство концепций и идей, изложенных в данной книге, применимы ко многим другим языкам программирования. Одним из ближайших множеств языков являются языки семейства ML, включая OCaml и F# (последний интегрирован в Microsoft .NET Framework и позволяет легко взаимодействовать с программами, написанными на C#). Основные отличия от Haskell заключаются в степени использования строгих и ленивых вычислений. Языки из семейства Lisp, такие как Racket

или Clojure, также включают в себя функциональные концепции. В некоторых языках функциональные концепции смешиваются с другими парадигмами: одним из интересных примеров такого рода является язык Scala, где внутри Java-платформы благополучно сосуществуют функциональное и объектно-ориентированное программирование.

В этой книге упоминался ряд мест, где Haskell-абстракции напрямую применяются в других языках. Концепция монад положена в основу библиотек LINQ в C#, выражений `for` в Scala и вычислительных выражений в F#. Во многие другие языки внедрены библиотеки программной транзакционной памяти, а для трактовки текстовых данных все чаще используются комбинаторы парсеров. Таким образом, интуитивные представления, наработанные на языке Haskell, могут быть перенесены на многие другие сценарии.

HASKELL И SCALA

Навыки работы с языком Scala могут быть существенно обогащены благодаря Haskell. Scala-сообщество разделяет многие абстракции, представленные в данной книге. Вот небольшой перечень Scala-библиотек, при изучении которых может пригодиться знание Haskell:

- Scala-проект typelevel.scala предлагает множество механизмов уровня типов. В частности, библиотека scalaz включает в мир Scala моноиды, функторы, монады и многие другие классы типов, а механизмы shapeless — обобщенное программирование типов данных.
- ScalaCheck придает Scala эффективность рандомизированного тестирования. Такие инструментальные средства, как ScalaTest и specs2, предоставляют общий интерфейс многим видам тестирования, точно так же как Tasty и HSpec в Haskell.
- Система объектно-реляционного проецирования Squeryl очень похожа по замыслу на Persistent и Esqueleto. Здесь также можно определять собственные схемы и пользоваться предметно-ориентированным языком для написания запросов.
- Библиотека Scala предоставляет возможность использования комбинаторов парсеров подобно тому, как это делает библиотека attoparsec. Если вас интересует атрибутивная грамматика, эту функциональность предоставляет библиотека Kiama.

Этот неполный список показывает, что функциональные концепции могут быть полезны не только в Haskell, но и в других языках и средах проектирования.

Приложение Б. Путешествие во времени с Haskell

Раз уж вы добрались до этого места, значит, имеете полное право узнать один секрет, известный только приверженцам языка Haskell: Haskell-код может путешествовать во времени! Для этого нужно любезно попросить Доктора[1], чтобы он дал нам свою машину времени TARDIS. Хорошо, что машина имеет форму Haskell: она может быть преобразована в монаду.

В Haskell монаду `Tardis` предоставляет пакет `tardis`. Интерфейс похож на монаду `State`, но вы можете работать с состоянием, которое меняется обычным образом при перемещении во времени вперед, начиная от текущего этапа выполнения, а также с состоянием, способным обновляться при перемещении во времени назад. Для каждого из этих состояний в монаде имеется пара функций:

- При перемещении вперед во времени состояние обновляется с помощью функции `sendFuture`, а получить его можно в другой точке выполнения посредством функции `getPast`.
- При перемещении назад во времени состояние обновляется с помощью функции `sendPast`, а получается через функцию `getFuture`.

В обоих случаях важно понимать, что можно получить только последнюю версию состояния. Например, если вызвать `sendFuture` дважды, любой вызов `getPast` извлечет вторую версию `sendFuture`.

А теперь настало время проделать сам трюк. Давайте запрограммируем функцию, которая, получая список чисел, создает новый список кортежей, где каждый кортеж показывает текущую сумму вплоть до данного момента с начала списка и то же самое, но с конца списка. Результат можно увидеть как два состояния, перемещаемые во времени в противоположных направлениях: сумма, накапливающаяся с начала, движется вперед (прямое состояние), а сумма, накапливающаяся с конца, путешествует назад (обратное состояние). Графически это показано на рис. Б.1.

---

[1] Доктор — это имя персонажа сериала «Доктор Кто», путешествующего во времени и пространстве на своей машине временных и относительных размерностей (Time And Relative Dimension In Space, TARDIS).

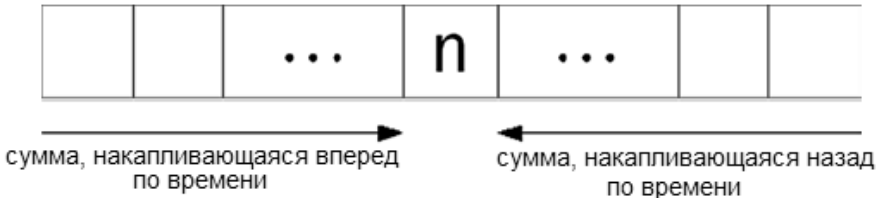


Рис. Б.1. Прямое и обратное состояния в Tardis

Соответствующий Haskell-код выглядит следующим образом:

```
{-# LANGUAGE RecursiveDo #-}

import Control.Monad.Tardis

sumListTardis :: [Int] -> [(Int,Int)]
sumListTardis lst = evalTardis (sumListTardis' lst) (0, 0)

sumListTardis' :: [Int] -> Tardis Int Int [(Int,Int)]
sumListTardis' (x:xs) = do
  sumFw <- getPast
  let newFw = sumFw + x
  sendFuture $ newFw
  rec let newBw = sumBw + x
      sendPast $ newBw
      sumBw <- getFuture
  rest <- sumListTardis' xs
  return $ (newFw, newBw):rest
sumListTardis' [] = return []
```

Обработка прямого состояния достаточно проста: на каждом шаге мы просто получаем предыдущее состояние из прошлого и отправляем в будущее новое значение, в котором для вычисления прямого состояния с предыдущим значением суммируется текущее число.

Обратное состояние получается более таинственным образом: как видите, информация отправляется в прошлое, а затем приносится из будущего. Такой способ работы необходим по причине того, что если сначала отправить состояние в прошлое, а затем попытаться принести его из будущего, то это будет означать, что вы пытаетесь получить значение последнего вызова функции `getPast`, происходящего в данный момент. А это невозможно, поскольку состояние будет зависеть от самого себя в прямом направлении. Вместо этого код требует создания *рекурсивного* do-блока, который начинается с ключевого слова `rec` и позволяет ссылаться на последнее значение.

ПРИМЕЧАНИЕ

Ключевое слово rec поддерживается не всеми монадами. Чтобы воспользоваться этим словом, монада, с которой вы работаете, должна быть экземпляром класса типов MonadFix.

Можно запустить Tardis и проверить, что все работает так, как ожидалось:

```
*> sumListTardis [1,2,3,4]
[(1,10),(3,9),(6,7),(10,4)]
```

Хотя Tardis выглядит несколько необычно, в реальной жизни тоже бывают случаи, когда вычисления получают обратную связь с самими собой. Основным примером могут послужить цепи, в которых выходной кабель подключается к входному порту. Как и в предыдущем случае, в отношении такой цепи нужно проявлять осторожность и не допускать зависимости значения от самого себя: обычно добавляется некоторая задержка, чтобы вход зависел от выхода в каком-то предыдущем моменте. Другим примером может послужить боулинг, где очки, набранные после удара или дополнительного удара, зависят от счета в последующих раундах[1].

Если бы мы только знали, что можем побывать в будущем! Вместо пошагового движения через примеры и упражнения можно было бы просто переместиться в будущее и попросить у самого себя в будущем научить себя всему, что он знает о Haskell. Но не все так просто, такова уж мистика ленивых вычислений, классов типов и функций высшего порядка. Так что теперь самое время стать фанатом Haskell и приступить к исследованию оставшейся части Haskell-вселенной!

[1] Более подробно о боулинге и Tardis можно прочитать в статье Дана Бартона (Dan Burton), http://unknownparallel.wordpress.com/2012/11/05/bowling-on-a-tardis