Практика
функционального программирования

Выпуск 1
Июль 2009

ISSN 2075-8456

Последняя ревизия этого выпуска журнала, а также последующие выпуски могут быть загружены с сайта http://fprog.ru/.

Авторы журнала будут благодарны за любые замечания и предложения, присланные по адресам электронной почты, указанным в заголовках статей. Редакция журнала: editor@fprog.ru.

Журнал «Практика функционального программирования»

Авторы статей:	Дмитрий Астапов Роман Душкин Сергей Зефиров Евгений Кирпичёв Алексей Отт Дэн Пипони (в переводе Кирилла Заборского)
Редактор:	Лев Валкин
Корректор:	Ольга Боброва
Иллюстрации:	**Обложка** ©iStockPhoto/Matt Knannlein ©iStockPhoto/Branislav Tomic **Круги ада** ©iStockPhoto/Yuri Schipakin
Шрифты:	**Текст** Minion Pro © Adobe Systems Inc. **Обложка** Days © Александр Калачёв, Алексей Маслов Cuprum © Jovanny Lemonad
Ревизия:	495 (2009-09-13)

Сайт журнала: http://fprog.ru/

Оглавление

От редактора

Языки программирования, используемые в разработке промышленных систем,[1] не стоят на месте. Вводятся в оборот новые идиомы, да и сами языки расширяются с целью добавления возможности оперировать абстракциями более высокого порядка. Языки C# и Visual Basic обзавелись абстракцией LINQ, позволяющей декларативно производить запросы к структурированной информации, базам данных. В язык Java добавили обобщённые типы данных («generics»); велись жаркие дебаты по добавлению замыканий в стандарт Java 7 [2]. Замыкания и лямбда-выражения появляются в новой редакции стандарта C++. Да и предлагавшаяся[2] абстракция «концептов» в C++, структурирующая описание полиморфных интерфейсов, возникла из необходимости бороться с нарастающей сложностью создаваемых человеком информационных систем. Есть ли что-то общее между этими нововведениями, или разные языки развиваются независимо и разнонаправленно?

Как оказывается, общий вектор развития языков прослеживается без особого труда. С целью повышения уровня абстракции от аппаратуры и приближения к проблемным областям всё большее количество концепций, созданных в рамках парадигмы функционального программирования, находят своё место в императивных языках.

Функциональная парадигма программирования и функциональные языки уже долгое время являются источником «новых» идей в массовых языках программирования. Отказ от ручного управления памятью и использование сборщика мусора (Smalltalk, Java, C#) пришли из функциональных языков типа LISP, где сборка мусора появилась ещё в 1959 году. Генераторы списков[3] в языке Python и новом стандарте JavaScript пришли из функциональных языков Haskell и Miranda, впервые появившись в функциональном языке NPL. Некоторые абстракции в императивных языках разработаны почти независимо от соответствующих абстракций в функциональных языках, но развиваются десятилетиями позже. Так, например, проект «концептов» в

[1] Неформальный термин «промышленная система» или «промышленный язык» используется для контраста с «академическими», «учебными» или «экспериментальными» системами и языками.

[2] Пока верстался номер, комитет по стандартизации C++ принял решение воздержаться от включения «концептов» в новый стандарт языка [6], не сумев упростить предложенное расширение до состояния работоспособности.

[3] List comprehensions, нотация, позволяющая создавать и фильтровать списки элементов декларативным путём.

новом стандарте C++ соответствовал «классам типов» в языке Haskell, появившимся в нём в 1989 году [1].

Технология LINQ в .Net тоже является прямым результатом мышления в функциональном стиле. Не случайно, что автор LINQ, Эрик Мейер [3, 4], является одним из дизайнеров функционального языка Haskell. Вот что говорит ведущий архитектор языка C#, а также создатель Turbo Pascal и Delphi, Андерс Хейлсберг [5]:

> Функциональное программирование, по-моему, — чрезвычайно интересная парадигма. Если взглянуть на C# 3.0, то станет видно, что именно идеи из мира функционального программирования послужили источником вдохновения для LINQ и составляющих его элементов языка. Мне кажется, сейчас, наконец, наступил момент, когда функциональному программированию пора выйти в массы.

Как мы видим, элементы функционального программирования проникают в массы путём постепенного внедрения в обычные, императивные языки. Что необходимо программисту для эффективного использования этих элементов? Мы убеждены, что для наиболее эффективного усвоения того или иного метода или парадигмы программирования, их нужно изучать в максимально чистом виде, удалённом от «мусора» альтернативных подходов. Владение функциональной парадигмой в её «чистом» виде позволяет эффективно применять новые функциональные элементы современных языков для управления сложностью разрабатываемых систем, а также существенно обогащает набор инструментов, доступных для решения той или иной задачи.

Практика функционального программирования

Вашему вниманию представляется первый выпуск журнала, посвящённого практике функционального и декларативного программирования. Мы ставим своей задачей помочь вам сориентироваться в инструментарии функционального программирования, в используемых в функциональной парадигме подходах к декомпозиции задач, способах упрощения программирования и снижения количества дефектов в разрабатываемых системах.

Первый номер журнала посвящён погружению в предмет функционального программирования. Вводные статьи Сергея Зефирова «Лень бояться» и Романа Душкина «Функции и функциональный подход» затрагивают философию парадигм программирования. Более практически направленная часть журнала представлена статьёй Евгения Кирпичёва «Изменяемое состояние: опасности и борьба с ними», классифицирующей типы проблем, возникающих при небрежном использовании сущностей с изменяемым состоянием, и следующей за ней статьёй Дмитрия Астапова «Давно не брал я в руки шашек», на протяжении нескольких страниц раскрывающей подход проектирования «сверху вниз» на подробном примере написания игры в шашки на языке Haskell. Статья Дэна Пипони «Моноиды в Haskell и их использование» в переводе Кирилла Заборского простым языком обьясняет практическое применение моноидов

для создания элегантных полиморфных алгоритмов. Номер завершается внушительным «Обзором литературы о функциональном программировании» Алексея Отта, содержащим множество ссылок на русскоязычную и англоязычную литературу по разным языкам и аспектам декларативного программирования.

Авторский коллектив журнала состоит из профессионалов промышленного программирования, участников международных олимпиад, конкурсов и конференций по функциональному программированию и преподавателей вузов России, стран ближнего и дальнего зарубежья.

Приятного чтения!

Лев Валкин, vlm@fprog.ru

Литература

[1] A comparison of C++ concepts and Haskell type classes / J.-P. Bernardy, P. Jansson, M. Zalewski et al. // WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming. — New York, NY, USA: ACM, 2008. — Pp. 37–48.

[2] Closures for the Java Programming Language (v0.5). — Предложенное дополнение к языку Java: URL: http://www.javac.info/closures-v05.html (дата обращения: 20 июля 2009 г.).

[3] *Erik Meijer*. On LINQ. — Видеоинтервью: URL: http://www.infoq.com/interviews/erik-meijer-linq (дата обращения: 20 июля 2009 г.). — 2007. — Создатель LINQ, Эрик Мейер, рассказывает о дизайне и возможностях LINQ, о том, как его использовать, зачем его использовать, чем LINQ отличается от XQuery, как LINQ связывается с ORM и о многом другом.

[4] *Erik Meijer*. Functional Programming. — Канал 9 MSDN: URL: http://channel9.msdn.com/shows/Going+Deep/Erik-Meijer-Functional-Programming/ (дата обращения: 20 июля 2009 г.). — 2008. — Видео, в котором Эрик Мейер, архитектор Microsoft SQL Server, Visual Studio и .Net, рассказывает о функциональном программировании, академических и промышленных применениях функциональных языков.

[5] The A-Z of Programming Languages: C#. — Статья в Computerworld: URL: http://www.computerworld.com.au/article/261958/-z_programming_languages_c (дата обращения: 20 июля 2009 г.). — 2008. — Интервью с ведущим архитектором языка C#, Андерсом Хейлсбергом.

[6] The Removal of Concepts From C++0x. — Статья в InformIT: URL: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=441 (дата обращения: 20 июля 2009 г.). — 2009. — Рассматриваются причины отказа от включения «концептов» в стандарт языка C++.

Лень бояться

Неформальная статья о ленивых вычислениях

Сергей Зефиров
thesz@fprog.ru

Аннотация

Интересующиеся функциональным программированием часто встречают прилагательное «ленивый» в связи с такими существительными, как «язык» и «порядок вычислений». Встретив же, идут в интернет и находят что-то наподобие «ленивый порядок вычислений, иначе называемый вызов-по-необходимости, строго нормализующий порядок вычислений, удовлетворяющий условиям алмазной леммы Чёрча-Россера», после чего ближайшее знакомство с функциональными языками откладывается на потом.

Небольшой опус, введение к которому вы читаете, представляет собой попытку неформально объяснить, что же такое «ленивые вычисления». Они часто встречаются в повседневной жизни программиста и, на мой взгляд, по своей сути не сложнее новой платформы или компьютерной архитектуры.

1.1. Обязательное условие

Ни один язык программирования не обходится без условных конструкций. В большинстве языков они являются неотъемлемой частью синтаксиса и семантики (смысла конструкций языка) и поэтому воспринимаются как нечто, данное свыше. Соответственно, их редко рассматривают в подробностях, хотя они весьма интересны, ведь именно здесь программист впервые встречается с ленивыми вычислениями.

На Си и его спутниках самая распространенная условная конструкция — оператор ветвления **`if`**:

```
if (condition) {
    action1
} else {
    action2
}
```

При рассуждении о программе с условием мы делаем, фактически, подстановку действий: сначала из action1, а потом из action2. При её выполнении процессор и в самом деле выполняет подстановку — по крайней мере, результат выполнения программы с условными переходами на глаз неотличим от результата выполнения программы с условной подстановкой.

Умозрительная и физическая подстановки действий грубо соответствуют одному из порядков вычисления функциональных программ — порядку под названием call-by-name (передача-по-имени). Вкратце его можно описать так: ничего не вычисляй сразу, результаты вычислений могут не понадобиться, запоминай вычисления. В императивных языках считается, что вычисления внутри одной из ветвей понадобятся сразу, как только станет возможным вычислить условие.

Обычная реализация передачи-по-имени сводится к построению структур в памяти: вместо вычисления $x + y$ создаётся узел дерева $+(x, y)$, вместо тяжёлого вызова $f(x)$ создаётся узел $call(f, x)$. Когда же требуется значение вычисления, вызывается интерпретатор структур.

Более привычный порядок вычислений называется call-by-value (передача-по-значению). В нём всё честно: всё надо вычислять всегда. Увидев вычисление, надо немедленно его произвести, сохранить результат в переменную, подставить как аргумент или вернуть как результат выполнения функции. Большинство языков программирования так себя и ведут большую часть времени, пока не дойдут до условного оператора. Такая смесь двух способов вычислений с преобладанием call-by-value называется энергичным порядком вычислений [3].

Передача-по-значению приводит к необходимости защиты от излишних вычислений, как в примере из Таблицы 1.1. Сначала результат сложного и дорогого вычисления требовался в обеих ветвях условия. Потом у нас изменились требования, он стал требоваться только в одной ветви, и, по-хорошему, нам надо перенести вычисления внутрь условия.

Было	Отредактировали	Надо

`x = COMPUTATION;`
`if (condition) {`
`  a = x;`
`} else {`
`  b = −x;`
`}`
	`x = COMPUTATION;`
	`if (condition) {`
	`  a = x;`
	`} else {`
	`  b = 10;`
	`}`
		`if (condition) {`
		`  a = COMPUTATION;`
		`} else {`
		`  b = 10;`
		`}`

Таблица 1.1. Защита от ненужных вычислений

Если бы мы использовали порядок вычисления call-by-name, то можно было бы остановиться на втором варианте, сэкономив время на рефакторинг. При выполнении в x положат не результат, а запрос на вычисления, переменная a также будет работать с запросом, и он будет вычислен, только когда (и если!) понадобится.

Прямое использование call-by-name с её практически текстовыми подстановками само по себе не очень эффективно. Возьмем простую последовательность: $x = z + 1;\ y = x \times x$. В y попадёт запрос на вычисление $\times(+(z,1),\ +(z,1))$. Значение z будет вычислено два раза, два раза будет выполнено сложение. Пустая трата ресурсов.

Поэтому был изобретён ещё один способ вычислений — call-by-need. Это подстановка с запоминанием. Последовательность $x = z + 1;\ y = x \times x$ так и останется этой последовательностью. Если нам потребуется y, то мы вычислим левый аргумент выражения x, и далее по цепочке, и запомним, что x чему-то равен. Когда мы приступим к вычислению правого аргумента умножения, мы просто возьмём уже вычисленное значение. Вуаля! Вычисление z требуется всего один раз, и сложение срабатывает тоже один раз.

Метод call-by-need называется «ленивым порядком вычислений».

Таким образом, мы имеем дело с тремя методами вычислений:

call-by-value Известен как «энергичный порядок вычислений» или «строгий порядок вычислений». Встречается чаще всего. Вычислит всё, до чего дотянется, если не контролировать каждый шаг. Для полноценной работы[1] требует call-by-name в заметных дозах. Проще всего реализуется, поэтому столь распространён.

call-by-name Простая подстановка структуры вычислений. Вычислит только нужное, но может вычислять некоторые части нужного большее количество раз, чем требуется. В языках программирования встречается в малых объёмах из-за низкой эффективности реализации. Основной прием в рассуждениях о программах, как императивных, так и функциональных.

call-by-need Ленивый порядок вычислений. Со стороны выглядит как предыдущий, только работает быстрее. Считается, что встречается редко и только в специаль-

[1]Call-by-name и call-by-need Тьюринг-полны, в то время как основная для большинства языков семантика call-by-value не является Тьюринг-полной. Условные конструкции с семантикой call-by-name обеспечивают таким языкам Тьюринг-полноту.

но отведённых местах. Тем не менее, он часто встречается в обычных программах (вычисление по требованию с запоминанием) и используется при оптимизации программ для устранения повторных вычислений.

Только один известный язык программирования — Algol-60 — использовал семантику call-by-name в промышленных масштабах. Остальные либо подходят к делу не столь радикально и применяют её только по месту, либо делают гораздо более широкий шаг и сразу берутся за полностью ленивые вычисления.

Главное — то, что некоторая ленивость присутствует практически в любом языке программирования. Без неё никуда, всего же на свете не вычислишь.

1.2. Сравнение с лучшим образцом

Одним из немногих языков, использующих ленивый порядок вычислений, является Haskell. Вот пример небольшой функции на нём:

```
False && x = False
True  && x = x
```

Это определение оператора `&&` путём сравнения с образцом. Всё достаточно прозрачно: если **`False`** хотя бы слева, то что бы ни было справа, ответом будет **`False`**, поэтому переменная x игнорируется. Если же слева **`True`**, то результатом будет значение справа.

Результатом **`False && error`** `"unreachable!"` будет False — до **`error`** дело не дойдёт. А результатом `10/0 == 10 &&` **`True`** будет деление на ноль.

Для выбора одного из клозов `&&` нам надо вычислить левый аргумент. Если его значение равно **`False`**, то правый аргумент вообще не понадобится!

Внимательный читатель наверняка уже сопоставил эти правила с правилами вычисления логических связок в языке Си и его спутниках, где логические связки также вычисляются до понимания результата и не дальше. В условии **`if`** `(i≥0 && i<10 && arr[i] != EOF) ...` до проверки `arr[i]` дело может не дойти (как выше не запустился **`error`** `"unreachable!"`).

В стандартном Паскале, где нет закорачивания вычислений логических связок (short circuit evaluation [8]), все выражения в приведенном выше условии будут вычислены. В результате мы получим ошибку выхода за границы массива, несмотря на то, что проверили все условия.

В C++ нас ожидает другой сюрприз: если мы попытаемся переопределить оператор `&&` для нового типа данных, мы с удивлением обнаружим, что `&&` вычисляет оба аргумента.

Ниже я приведу ещё один пример на Хаскеле, на этот раз для логического типа со значением «Unknown»:

```
data BoolU = FALSE | TRUE | Unknown
FALSE  && x     = FALSE
```

```
TRUE    && x      = x
Unknown && FALSE = FALSE
Unknown && x      = Unknown
```

Только в случае, когда мы не уверены в левом аргументе, мы вынуждены вычислять правый. Поэтому ошибку обращения вне границ массива, как в примере тремя абзацами выше, мы получим, только если мы не сможем чётко понять, больше ли i нуля и меньше ли i десяти. Но в этом случае мы эту ошибку заслужили, по-моему.

При реализации на C++ типа `enum {False, True, Unknown}` оператор `&&` для него будет вести себя одинаково плохо вне зависимости от степени нашей уверенности в результатах сравнения — он всегда будет вычислять оба операнда. Нам придётся придумывать что-то ещё, например, применять указатели. Чего мы не заслужили — то, что может быть автоматизировано, должно быть автоматизировано.

Логические связки — это второе и последнее место в привычных языках программирования, где дела могут идти ленивым порядком. Остальное реализуется с помощью обычного программного кода, более или менее независимого от языка программирования, о чём будет следующая часть.

1.3. Ленивый код

Выборка данных по SQL запросу часто осуществляется чем-то вроде строящегося в процессе работы итератора. Мы создаём структуру данных с двумя функциями: получить голову и получить хвост. Голова будет очередной строкой выборки, а хвост будет вычислением, позволяющим получить последующие строки (если они есть), по одной строке с хвостом за раз. Это и есть итератор, только потенциально бесконечный.

И тут мы подходим к причине, по которой ленивые языки трудно использовать.

Мы создали запрос к БД и даже что-то из неё выбрали. В процессе мы решили, что в БД чего-то не хватает, и добавили туда информацию, которая может влиять на содержимое нашего SQL запроса. Совсем плохо, если это решили и добавили не мы.

Получается, что мы можем получить данные, в которых, допустим, нарушен порядок сортировки строк. Или можем не получить актуальные данные.

Иными словами, работая лениво, мы не можем вести простые рассуждения о последовательности действий. А раз не можем рассуждать о последовательности, то не можем просто предсказать время работы.

Это приводит к необходимости чётко разграничивать участки программы, где последовательность действий надо контролировать, и остальные, где такой контроль не нужен.[2]

В этом есть и плюсы, и минусы. С одной стороны, добавляется работы по разделению программы на разные участки — это минус. С другой стороны, как показывает

[2] В современных функциональных языках такое разграничение поддерживается системой типов.

практика, количество кода, где неважно, что за чем будет вычисляться, больше в разы, если не на порядки.

Получается, что мы экономим свой труд, поскольку в большем количестве мест дополнительная работа не требуется. Экономия состоит и в отсутствии необходимости рефакторинга, и в простоте самого рефакторинга: в коде, где нет зависимости от порядка вычислений, порядок определений не важен. Последовательности определений «$x = f(z); y = g(x);$» и «$y = g(x); x = f(z);$» имеют один и тот же смысл. Мы упрощаем для себя рассуждения о программе, поскольку можем проводить простую текстовую подстановку без оглядок на состояние мира или переменных. Мы, наконец, можем заглядывать глубоко в структуры данных и использовать их части напрямую, без боязни, что кто-то их поменяет.

1.4. Размышления

Уже достаточно долго меня мучает вопрос: почему ленивые вычисления так тяжело воспринимаются даже весьма умными людьми? Что их останавливает?

Более или менее приемлемый ответ пришёл ко мне после знакомства с различными параллельными архитектурами вычислительных систем. Если быть кратким, то обычный процессор умеет делать scatter (писать в любое место памяти) и gather (читать из любого места памяти). Трудности синхронизации двух параллельно выполняющихся процессов разбрасывания и собирания данных велики, поэтому их как-то ограничивают: отдавая предпочтение одному из них при реализации, разделяя их во времени[3] или действуя иначе. Например, процессоры GPU умеют только gather (причем, в ограниченном объёме), машины динамического потока данных — scatter (и тоже в ограниченном объёме). Труден и анализ произвольной рекурсии, поэтому от неё тоже избавляются (GPU).

Например, на GPU делать обработку изображений просто, а анализ — тяжело; с задачей «получить N координат максимальных[4] результатов обработки изображения» мы не смогли справиться за пару недель. На GPU нельзя сделать сортировку слиянием или быструю сортировку, надо делать bitonic sort, при этом управление сортировкой внешнее (рекурсии-то нет, как и локальных массивов); в цикле вычисляются параметры очередного прохода и запускается шаг параллельного алгоритма. Для программ типа компиляторов GPU вообще не применимо — в компиляторах используются практически неограниченные структуры, не ложащиеся на массивы.

Изучение программирования GPU имеет вполне определённую цель — ускорить выполнение программ. Но это подходит не для любых программ — компиляторы от использования GPU только проиграют, на данный момент. Поэтому создатели компиляторов на GPGPU[5] не смотрят.

[3] Что может сказаться на производительности.

[4] Максимальных по какому-то критерию, например, по величине результатов алгоритма определения подходящих для отслеживания точек — чем выше «яркость», тем лучше отслеживать.

[5] Техника использования графического процессора видеокарты для общих вычислений, см. [5].

Применение ленивых вычислений может сократить код логики программы, тех вычислений, что можно записать в нём с выгодой. Но они не помогут, когда большую часть времени занимает ввод-вывод или когда требуется удовлетворить неким временным условиям.

По-моему, в этом GPU и ленивые вычисления схожи: они подходят не для всех задач, надо с ними разбираться, чтобы понимать, могут ли они пригодиться в конкретных задачах. Многих это и останавливает, поскольку «известный путь самый короткий». Но наблюдения показывают, что области применения как GPU, так и ленивых вычислений расширяются: в GPU не так давно добавили практически произвольную рекурсию, а ленивые вычисления помогают лучше структурировать анализ при обработке в условиях временных ограничений.[6]

Уподобление ленивых вычислений компьютерной архитектуре позволяет провести ещё одну интересную параллель. Языки программирования с ленивым порядком вычислений можно представить как автокод немного непривычного компьютера или виртуальной машины. Ну, ещё одна архитектура, сколько их в карьере программиста. А автокоды бывают очень пристойными: «Советская Ада» Эль-76 Эльбруса, Java для JavaVM, C# для .Net. В свое время на автокоде МИР (Машины для Инженерных Расчётов) [10] решали весьма серьёзные задачи, к которым было трудно подступиться на машинах побольше.

1.5. Чистый параллелизм

Раз речь зашла о параллельных архитектурах, нельзя не упомянуть и о том, как распараллеливание связано с разбиением на чистый код (без побочных эффектов) и остальной.

Если в двух внешне независимых участках кода есть вызов функции с неизвестным побочным эффектом (допустим, отладочная печать), то выполнять эти участки кода параллельно нельзя, поскольку порядок вызова этих внешних функций может измениться.

Чистый же код можно переставлять, как угодно [1] — зависимости в нем выражены исключительно явным образом. Это не означает лёгкости анализа для автоматического распараллеливания, но упрощает ручное распараллеливание. Практика показывает, что это действительно так: изменения в двух строках одного многопроходного оптимизатора нашего проекта ускорили процесс оптимизации на 10% на двух ядрах по сравнению с одним. Код оптимизатора занимает порядка 800 строк, то есть, мы получили 10% ускорения за счёт 0.3% изменений кода. Получается, что ленивый порядок вычислений, заставивший нас трудиться над разбиением кода на отдельные части, подготовил код для параллельного выполнения.

[6] Приём очень простой: создаётся (практически бесконечный) генератор итераций анализа. Управляющая часть программы выбирает из потока итераций до тех пор, пока не кончится время на реакцию или не будет найдено удовлетворительное решение. Таким образом, анализ не знает о временных ограничениях, а управляющая часть — об алгоритме.

Более того, в чистом коде без побочных эффектов проще проверять всякого рода условия, например, что мы не зациклимся, или что не будет ошибок обращений к массивам [9].

1.6. Заключение

Приведу немного статистики. В gcc 4.2.2, большом проекте на C, есть файл bitmap.c. Он довольно средний по размерам: 1500 строк, 40 функций. Он содержит 132 оператора **`if`** и один тернарный оператор **`?:`**. Таким образом, на функцию приходится более трех операторов **`if`**, по одному условному оператору на каждые 12 строк. Получается, что мы, обычные программисты, используем грубый аналог call-by-name примерно 5—6 раз за рабочий день.[7]

Можно оценить и количество использования ленивых вычислений в gcc. Поиск «grep -E -i recog_memoiz *.c» дал 13 файлов из 299 (33 использования) всё того же gcc 4.2.2. Везде используется call-by-need — производится вычисление и запоминается промежуточный результат. Немного, но используется.

Если интересно, поищите у себя в коде что-либо наподобие установки в конструкторе некоего поля в NULL, и чтобы рядом где-нибудь был getter вида:

```
if (m_field == NULL)
    m_field = computation;
return m_field;
```

Наверняка найдётся нечто похожее, что означает наличие ленивых вычислений.

Не стоит бояться ленивых вычислений. Они встречаются даже чаще, чем я думал, и гораздо привычней, чем принято считать. Они позволяют структурировать программу так, что в ней открываются новые возможности для экономии времени программиста и увеличения скорости работы на современных системах. И уж тем более не стоит бояться ленивых языков программирования: они не страшней автокода многих существующих, или существовавших ранее, машин.

Литература

[1] Data dependencies. — Ссылка в Wikipedia, URL: http://en.wikipedia.org/wiki/Data_dependencies (дата обращения: 20 июля 2009 г.). — О зависимости по данным и их влиянии на параллелизм. В этой статье говорится только про параллелизм уровня инструкций, но если поставить части массивов или других структур данных на место переменных, то получим всё то же самое, только для параллелизма большего объёма.

[7] ≈ 17000 отлаженных строк кода в год для программистов вне США. 68 строк кода в день.

[2] Efficient gather and scatter operations on graphics processors / B. He, N. K. Govindaraju, Q. Luo, B. Smith. — 2008. — О реализации произвольного доступа к памяти на GPU.

[3] Evaluation strategy. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Evaluation_strategy (дата обращения: 20 июля 2009 г.). — Описываются все известные порядки вычислений.

[4] General-purpose computation on graphics processing units. — URL: http://gpgpu.org/ (дата обращения: 20 июля 2009 г.). — Об использовании GPU в гражданских целях.

[5] General-purpose computing on graphics processing units. — Ссылка в Wikipedia, URL: http://en.wikipedia.org/wiki/GPGPU (дата обращения: 20 июля 2009 г.). — О технике использования графического процессора видеокарты для общих вычислений.

[6] *Kiselyov O.* Incremental multi-level input processing with left-fold enumerator: predictable, high-performance, safe, and elegant. — О безопасном применении ленивых итераторов. Предназначено для сильных духом, поскольку содержит магию типов Haskell высшего уровня.

[7] *Mitchell N.* — Catch: Case Totality Checker for Haskell. — Проверка полноты разбора по образцу. По сравнению с тестами экономит время; отыскивает ошибки, которые тесты отловить не в состоянии.

[8] Short circuit evaluation. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Short_circuit_evaluation (дата обращения: 20 июля 2009 г.). — Логические операторы в разных языках программирования.

[9] Single Assignment C. — URL: http://sac-home.org/ (дата обращения: 20 июля 2009 г.). — Чего можно добиться, отказавшись от разрушающего присваивания.

[10] Машина для Инженерных Расчётов. — Ссылка в Wikipedia, URL: http://ru.wikipedia.org/wiki/МИР_(компьютер) (дата обращения: 20 июля 2009 г.). — Об одной из первых в мире персональной ЭВМ, созданной в 1965 году Институтом кибернетики Академии наук Украины. МИР выпускалась серийно и предназначалась для использования в учебных заведениях, инженерных бюро и научных организациях.

Функции и функциональный подход

Роман Душкин
darkus@fprog.ru

Аннотация

В статье в сжатой форме рассказывается про функциональный подход к описанию вычислительных процессов (и в общем к описанию произвольных процессов в реальном мире), а также про применение этого подхода в информатике в функциональной парадигме программирования. Примеры реализации функций даются на языке программирования Haskell.

Введение

Вряд ли можно подтвердить или даже доказать какую-либо закономерность, но можно предположить, что два способа вычисления — процедурный и функциональный — как-то связаны с особенностями человеческого разума, различающимися у разных людей. Такие особенности издревле приводили к попыткам классификации человеческих характеров по каким-либо дуальным шкалам. В качестве банальнейшего примера можно привести шкалу «интровертность — экстравертность», хотя причины, приведшие к появлению двух парадигм вычислений находятся в какой-то другой плоскости, нежели приведённый пример.

И процедурный, и функциональный стили вычислений были известны в далёком прошлом, и сейчас уже невозможно узнать, какой подход появился первым. Последовательности шагов вычислений — особенность процедурного стиля — можно рассматривать в качестве естественного способа выражения человеческой деятельности при её планировании. Это связано с тем, что человеку приходится жить в мире, где неумолимый бег времени и ограниченность ресурсов каждого отдельного индивидуума заставляют людей планировать по шагам свою дальнейшую жизнедеятельность.

Вместе с тем нельзя сказать, что функциональный стиль вычислений не был известен человеку до возникновения теории вычислений в том или ином виде. Такие методики, как декомпозиция задачи на подзадачи и выражение ещё нерешённых проблем через уже решённые, составляющие суть функционального подхода, также были известны с давних времён. Тут необходимо отметить, что эти методики вполне могут применяться и в рамках процедурного подхода как проявление в нём функционального стиля. Именно этот подход и является предметом рассмотрения настоящей статьи, а объясняться его положения будут при помощи функционального языка Haskell[1].

Итак, ниже читателю предлагается ознакомиться со способами определения функций, изучить дополнительные интересные методы в рамках функционального подхода, а также углубиться в теоретический экскурс для понимания основ функционального подхода. Автор надеется, что разработчик программного обеспечения с любым уровнем подготовки сможет найти для себя что-нибудь новое.

2.1. Простые примеры функций

В одной из компаний, где в своё время довелось работать автору, при отборе на вакантные должности инженеров-программистов кандидатам давалась задача: необходимо написать функцию, которая получает на вход некоторое целое число, а возвращает строку с представлением данного числа в шестнадцатеричном виде. Задача очень простая, но вместе с тем она позволяет легко выяснить, какими методами решения

[1]Описание языка можно найти на официальном сайте: на английском языке http://www.haskell.org/ или на русском языке http://www.haskell.ru/; также для изучения языка можно воспользоваться книгой [11].

задач руководствуются кандидаты, поэтому основной упор на собеседовании делался не на правильность написания кода, а на подход и канву рассуждений при написании этой функции. Более того, если кандидат затруднялся с алгоритмом, ему он полностью разъяснялся, поскольку интерес представляли именно ход рассуждений и окончательный способ реализации алгоритма. Для решения задачи разрешалось использовать любой язык программирования на выбор кандидата, включая псевдоязык описания алгоритмов, блок-схемы и прочие подобные вещи.

Сам алгоритм прост: необходимо делить заданное число на основание (в задаваемой задаче, стало быть, на 16), собирать остатки и продолжать этот процесс до тех пор, пока в результате деления не получится 0. Полученные остатки необходимо перевести в строковый вид посимвольно (учитывая шестнадцатеричные цифры), после чего конкатенировать все эти символы в результирующую строку в правильном направлении (первый остаток должен быть последним символом в результирующей строке, второй — предпоследним и т. д.).

Каковы были типовые рассуждения большинства приходящих на собеседование? «Получаем входное число — организуем цикл `while` до тех пор, пока параметр цикла не станет равен 0 — в цикле собираем остатки от деления параметра на основание, тут же переводим их в символы и конкатенируем с переменной, которая потом будет возвращена в качестве результата — перед возвращением переменную обращаем». Некоторые кандидаты оптимизировали эти рассуждения и уже в цикле конкатенировали символы в правильном порядке, так что впоследствии переменную обращать было не надо. Некоторые кандидаты пользовались циклом `for`, некоторые добавляли всякие «рюшечки». Но за всё время работы автора в этой компании ни один из кандидатов не предложил решения задачи в функциональном стиле.

Вот как выглядит типовая функция для описанной цели на языке C++:

```
std::string int2hex (int i) {
  std::string result = "";
  while (i) {
    result = hexDigit (i % 16) + result;
    i /= 16;
  }
  return result;
}
```

Здесь функция `hexDigit` возвращает символ, соответствующий шестнадцатеричной цифре.

Как же решить эту задачу при помощи функционального подхода? При размышлении становится ясно, что взяв первый остаток от деления на 16 и после этого целочисленно разделив само число на 16, задача сводится к той же самой. И такое сведение будет происходить до тех пор, пока число, которое необходимо делить, не станет равным 0. Налицо рекурсия, которая является одним из широко используемых методов функционального программирования. На языке Haskell эта задача может быть решена следующим образом:

```
int2hex :: Integer → String
int2hex 0 = ""
int2hex i = int2hex (i `div` 16) ++ hexDigit (i `mod` 16)
```

Здесь функции **div** и **mod**, записанные в инфиксном стиле, возвращают соответственно результат целочисленного деления и остаток от такого деления. Инфиксный стиль в языке Haskell позволяет записывать функции двух аргументов между ними при вызове — в данном случае имя функции необходимо заключать в обратные апострофы (`) (обычно инфиксный стиль используется для повышения степени удобочитаемости кода для функций с наименованиями вроде **isPrefixOf** и т. д.). Функция (++) конкатенирует две строки. Все эти функции определены в стандартном модуле **Prelude**. Первая строка определения функции, так называемая сигнатура, определяет тип функции. Для языка Haskell описание сигнатур не является необходимым, поскольку компилятор самостоятельно выводит типы всех объектов, но правилом хорошего тона при написании исходных кодов программ является простановка сигнатуры для каждой функции. Кроме того, сигнатура может являться ограничением на тип функции (в вышеприведённом примере автоматически выведенный тип функции `int2hex` будет более общим, чем записано в сигнатуре; более общий тип этой функции: **Integral** $\alpha \Rightarrow \alpha \rightarrow$ **String**, где **Integral** — это класс типов таких значений, над которыми можно производить целочисленные арифметические операции).

Вторая строка определяет результат функции `int2hex` в случае, когда значение её единственного входного параметра равно 0. Третья строка, соответственно, определяет результат функции в оставшихся случаях (когда значение входного параметра ненулевое). Здесь применён механизм сопоставления с образцами, когда для определения функции записывается несколько выражений,[2] каждое из которых определяет значение функции в определённых условиях. В других языках программирования для этих целей обычно используются **if-then-else** или **case**-конструкции. Вот как, к примеру, та же самая функция будет записана на языке C++:

```
std::string int2hex (int i) {
  if (i) {
    return int2hex(i / 16) + hexDigit (i % 16);
  } else {
    return "";
  }
}
```

Представленный пример уже достаточно показывает отличие двух подходов к представлению вычислений. Тем не менее, уже сейчас видно, что есть широкий простор для усовершенствования кода. В первую очередь это касается основания преобразования, ведь часто при программировании необходимы числа в двоичной и восьмеричной записи. Более того, почему бы не сделать универсальную функцию для пре-

[2] В литературе по функциональному программированию для обозначения одного такого выражения в определении функции иногда используется термин «клоз» (от англ. «clause»).

образования числа в произвольную систему счисления? Эта задача легко решается преобразованием уже написанной функции[3]:

```
convert :: Int → Int → String
convert _ 0 = ""
convert r i = convert r (i `div` r) ++ digit r (i `mod` r)
```

Здесь в сигнатуру внесены два изменения. Во-первых тип **Integer** изменён на тип **Int**, что связано с необходимостью ограничения (тип **Integer** представляет неограниченные целые числа, тип **Int** — ограниченные интервалом $[-2^{29}; 2^{29} - 1]$) для оптимизации вычислений. Во-вторых, теперь функция `convert` принимает два параметра. Первым параметром она принимает основание системы счисления, в которую необходимо преобразовать второй параметр. Как видно, определение функции стало не намного сложнее. Ну и в-третьих, в первом клозе определения на месте первого параметра стоит так называемая маска подстановки (_), которая обозначает, что данный параметр не используется в теле функции.

Соответственно, функция `digit`, возвращающая цифру в заданном основании, теперь тоже должна получать и само основание. Но её вид, в отличие от функции `hexDigit`, которая являлась простейшим отображением первых шестнадцати чисел на соответствующие символы шестнадцатеричной системы счисления, теперь должен стать совершенно иным. Например, вот таким:

```
digit r i | r < 37    = if (i < 10)
                          then show i
                          else [(toEnum (i + 55))::Char]
         | otherwise = "(" ++ (show i) ++ ")"
```

В определении функции `digit` используется несколько интересных особенностей языка Haskell. Во-первых, вместо механизма сопоставления с образцами в определении применен механизм охраны (охранных выражений), которые также позволяют сравнивать входные параметры с некоторыми значениями и осуществлять ветвление вычислений. Вторая особенность — использование выражения **if-then-else** для тех же самых целей в первом варианте. Особой разницы между этими подходами

[3]Для простоты изложения в статье приведены определения функций, работающих с положительными числами. Если передать им в качестве входного значения число 0, то в результате будет некорректное преобразование в пустую строку. Данная проблема решается несложно — например, функцию `int2hex` можно дополнить следующим образом:

```
int2hex :: Int → String
int2hex i = int2hex' i True
 where
  int2hex' 0 True  = "0"
  int2hex' 0 False = ""
  int2hex' i _     = int2hex' (i `div` 16) False ++ hexDigit (i `mod` 16)
```

В качестве упражнения читателю предлагается написать новое определение функции `convert` по аналогии с приведённым определением функции `int2hex`.

нет, вдумчивому читателю предлагается поэкспериментировать с охранными и условными выражениями (подробности синтаксиса — в специализированной литературе, рекомендуется использовать справочник [10]).

Функции **show** и **toEnum** опять же описаны в стандартном модуле **Prelude**, который подгружается всегда. Первая функция преобразует любое значение в строку (её тип — $\alpha \rightarrow$ **String**), вторая — преобразует целое число в заданный тип (её тип — **Int** $\rightarrow \alpha$, причём конкретно в данном случае она преобразует целое в код символа **Char**). Таким образом, алгоритм работы функции `digit` прост: если основание системы счисления не превышает 36 (это число — сумма количества десятеричных цифр и букв латинского алфавита, в исходном коде записывается как «меньше 37»), то результирующая строка собирается из символов цифр и латинских букв. Если же основание больше или равно 37, то каждая цифра в таких системах счисления записывается как соответствующее число в десятичной системе, взятое в круглые скобки. Для понимания способа работы функции `digit` можно запустить её с различными параметрами и посмотреть результат:

```
> digit 1 0
"0"

> digit 10 9
"9"

> digit 16 10
"A"

> digit 20 15
"F"

> digit 36 35
"Z"

> digit 100 50
"(50)"
```

Теперь можно легко определить несколько практичных дополнительных функций:

```
int2bin = convert 2
int2oct = convert 8
int2hex = convert 16
```

Такая запись может выглядеть необычно для тех, кто не знаком с функциональным программированием. Используемый здесь подход называется «частичным применением». В данных определениях производится частичный вызов уже определённой ранее функции `convert`, принимающей на вход два параметра. Здесь ей передаётся всего один параметр, в результате чего получаются новые функции, ожидающие на вход один параметр. Этот подход проще всего понять, представив, что первый па-

раметр функции `convert` просто подставлен во все места, где он встречается в теле функции. Так частичная подстановка `convert 2` превращает определение в:

```
convert :: Int → Int → String
convert 2 0 = ""
convert 2 i = convert 2 (i `div` 2) ++ digit 2 (i `mod` 2)
```

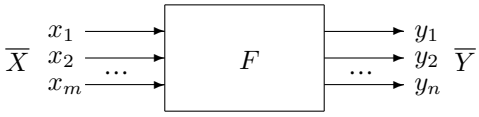
Поскольку данное определение можно легко преобразовать в функцию одного параметра (первый же теперь зафиксирован и является константой), современные трансляторы языка Haskell проводят именно такую оптимизацию, создавая дополнительное определение новой функции для частичных применений.

Осталось упомянуть, что при частичном применении тип функции как бы сворачивается на столько параметров, сколько было частично применено. В рассматриваемом примере тип функций `int2bin`, `int2oct` и `int2hex` равен **`Int → String`**.

2.2. Теоретические основы функционального подхода

Несмотря на то, что фактически функциональный подход к вычислениям был известен с давних времён, его теоретические основы начали разрабатываться вместе с началом работ над вычислительными машинами — сначала механическими, а потом и электронными. С развитием традиционной логики и обобщением множества сходных идей под сводом кибернетики появилось понимание того, что функция является прекрасным математическим формализмом для описания реализуемых в физическом мире устройств [6]. Но не всякая функция, а только такая, которая: во-первых, не имеет побочных эффектов, и во-вторых, является детерминированной. Данные ограничения на реализуемость в реальности связаны с физическими законами сохранения, в первую очередь энергии. Именно такие чистые процессы рассматриваются в кибернетике при помощи методологии чёрного ящика — результат работы такого ящика зависит только от значений входных параметров.

Ну и классическая иллюстрация этой ситуации:



Таким образом, функциональное программирование предлагает практические методы реализации кибернетических идей. Сегодня такие методы всё больше распространяются в области промышленного создания информационных и автоматизированных систем, поскольку при проектировании этих систем применяются методы декомпозиции функциональности и связывания отдельных функций в цепочки исполнения вычислений. Так, к примеру, автоматизированные системы управления технологическими процессами (АСУ ТП) могут представляться в виде блоков обработки информации, соединённых друг с другом информационными потоками от датчиков

к пункту принятия решений и обратно к исполнительным устройствам. Каждый элемент на должном уровне абстракции представляет собой как раз такой чёрный ящик, представимый вычислимой детерминированной функцией.

Одним из ведущих ученых, заложивших формальные основы теории вычислений, был А. Чёрч, предложивший λ-исчисление в качестве формализма для представления вычислимых функций и процессов [4]. Данный формализм основан на систематическом подходе к построениюи исследованиям операторов, для которых другие операторы могут бытькак формальными аргументами, так и возвращаемым результатом вычислений. Это — проявление функций высших порядков, то есть таких функций, аргументами которых могут быть другие функции. Функциональные языки программирования основаны на λ-исчислении, поскольку функция является отображением λ-терма в конкретный синтаксис, включая функциональную абстракцию и применение (аппликацию).

Как формальная система λ-исчисление представляет собой достаточно сложную и содержательную теорию, которой посвящено множество книг (некоторые из них приведены в списке литературы [4, 11, 13]). Вместе с тем, λ-исчисление обладает свойством полноты по Тьюрингу, то есть теория предлагает нотацию для простейшего языка программирования. Более того, дополнения к теории, расширяющие её свойства, позволяют строить прикладные языки программирования на основе заданных денотационных семантик [8]. Так, к примеру, ядро языка программирования Haskell представляет собой типизированное λ-исчисление.

Также стоит упомянуть про комбинаторную логику [7], которая использует несколько иную нотацию для представления функций, а в качестве базового правила вывода в формальной системе использует только аппликацию (применение). В этой формальной системе отсутствует понятие связанной переменной, а объекты-функции (или «комбинаторы») просто прикладываются друг к другу. Базис системы состоит из одного комбинатора, то есть утверждается, что любая функция может быть выражена через этот единственный базисный комбинатор. Сама по себе комбинаторная логика изоморфна λ-исчислению, но обладает, по словам некоторых специалистов, большей выразительной силой. В дополнение можно отметить, что некоторые исследователи подходят к комбинаторной логике как к средству наименования λ-термов (например, $\lambda x.x \equiv \mathbf{I}$), что просто облегчает запись аппликативных выражений.

Необходимо отметить, что несмотря на глубокую теоретическую проработку вопросов теории вычислений и наличие прикладных инструментов в виде языков программирования, вопросы создания качественного инструментария непосредственно для процесса разработки для функциональной парадигмы рассматриваются мало. Так, к примеру, Ф. Уодлер отмечает [3], что отсутствие достаточного количества удобных и распространённых инструментальных средств оказывает негативное влияние на возможности использования функциональных языков программирования. Как следствие, функциональные языки программирования, многие из которых являются действительно универсальными и отличными средствами решения задач, до сих

пор не могут выйти из узких стен научных лабораторий и найти широкого пользователя в среде разработчиков программного обеспечения.

Вместе с тем уже сегодня имеются прекрасные методики функционального анализа и проектирования, применение которых на этапах подготовки требований и разработки проекта программного продукта позволит усовершенствовать процесс разработки и ввести в него элементы функционального программирования.

В первую очередь речь идёт о методологии структурного анализа и проектирования SADT [12]. Нотации DFD (англ. «Data Flow Diagrams» — диаграммы потоков данных) и в особенности IDEF0 (англ. «Integration Definition for Function Modeling» — интегрированная нотация для моделирования функций), предлагаемые в рамках этой методологии, отлично проецируются на методы и технологии функционального программирования. Так, например, в IDEF0 каждый блок представляет собой функцию, которая связана с другими функциями при помощи отношений декомпозиции и получения/передачи параметров. Диаграммы IDEF0 могут быть в автоматизированном режиме преобразованы в шаблоны модулей на каком-либо функциональном языке, а методика обратного проектирования позволит преобразовать модули на том же языке Haskell в диаграммы IDEF0. Тем самым можно построить инструментарий, в чём-то схожий с известными средствами для объектно-ориентированного программирования, на основе языка моделирования UML (англ. «Unified Modeling Language» — унифицированный язык моделирования).

К тому же и сам язык UML позволяет применять функциональный подход [5]. Диаграммы вариантов использования можно рассматривать как верхний уровень абстракции функциональности программных средств, выражаемой при помощи функций. В дальнейшем при декомпозиции каждого варианта использования при помощи диаграмм последовательностей или конечных автоматов можно также предусмотреть автоматизированный процесс кодогенерации.

Впрочем, эта тема ещё ждёт своего исследователя и реализатора.

2.3. Дополнительные примеры с отдельными элементами программирования

Для полноты изложения осталось привести несколько примеров функций, которые используют особые элементы функционального программирования, связанные с оптимизацией, улучшением внешнего вида исходного кода и т. д. Для демонстрации большинства таких элементов программирования приведем следующее преобразование уже рассмотренной функции `convert`:

```
convert' :: Int → Int → String
convert' r i = convert_a r i ""
  where
    convert_a _ 0 result = result
    convert_a r i result = convert_a r (i `div` r)
```

```
                                (digit r (i `mod` r)
                                  + result)
```

Данное определение необходимо разобрать подробно.

Функция `convert'` выполняет абсолютно то же вычисление, что и функция `convert`, однако оно основано на подходе, который называется «накапливающий параметр» (или «аккумулятор»). Дело в том, что в изначальном определении функции `convert` используется рекурсия, которая в некоторых случаях может приводить к неоптимальным вычислительным цепочкам. Для некоторых рекурсивных функций можно провести преобразование так, что они принимают вид хвостовой рекурсии, которая может выполняться в постоянном объёме памяти.

В функциональном программировании такое преобразование делают при помощи накапливающего параметра. Определение начальной функции заменяют на вызов новой функции с накапливающим параметром, а в данном вызове передают начальное значение этого параметра. Дополнительная же функция производит вычисления как раз в накапливающем параметре, делая рекурсивный вызов самой себя в конце всех вычислений (в этом и заключается смысл хвостовой рекурсии). Соответственно, здесь видно, что функция `convert_a` вызывает саму себя в самом конце вычислений, а приращение цифр в новой системе счисления производится в третьем параметре, который и является накапливающим.

Особо надо обратить внимание на вид функции `convert_a`. Её определение записано непосредственно в теле функции `convert'` после ключевого слова **`where`**. Это — ещё один из элементов программирования, который заключается в создании локальных определений функций или «замыканий». Замыкание находится в области имён основной функции, поэтому из его тела видны все параметры. Кроме того, замыкания могут использоваться для оптимизации вычислений — для некоторых функциональных языков программирования справедливо, что если в теле основной функции несколько раз вызвать локальную функцию с одним и тем же набором параметров, то результат будет вычислен один раз. Замыкания определяются в языке Haskell двумя способами: префиксно при помощи ключевого слова **`let`** и постфиксно при помощи рассмотренного ключевого слова **`where`** (у этих ключевых слов имеется семантическое различие, несущественное здесь).

Кроме того, представленный пример демонстрирует так называемый двумерный синтаксис, который применяется в языке Haskell для минимизации количества специальных разделительных символов. Два клоза определения локальной функции `convert_a` начинаются с одной и той же позиции символа, и это важно. Этому же принципу подчиняются все перечисления «операторов»: их можно записывать друг под другом в одном столбце, а можно отделять друг от друга точкой с запятой.

Дополнительные приёмы программирования, описание ключевых слов, а также описание метода преобразования функции к хвостовой рекурсии в деталях описаны в книге [10].

Здесь же осталось упомянуть то, что полученные функции `convert` и `convert'` можно использовать так, как любые иные: передавать в качестве аргументов, частично

применять и т. д. Например, для получения списка чисел в заданной системе счисления (в двоичной, скажем) можно воспользоваться таким вызовом:

```
map (convert 2) [1..100]
```

Данный вызов вернёт список двоичных представлений чисел от 1 до 100, поскольку стандартная функция **map** применяет заданную функцию к каждому элементу заданного списка и возвращает список результатов таких применений.

Для окончательного оформления исходного кода в исполняемом модуле необходимо разработать функцию `main`, которая будет использоваться как точка входа в откомпилированную программу. Пример такой функции ниже:

```
main :: IO ()
main = putStr $ convert' 2 14
```

Здесь стандартная функция **putStr** выводит на экран результат работы функции `convert'`. Оператор (`$`) позволяет записывать функции друг за другом без лишних скобок — это просто оператор применения функции с наинизшим приоритетом, используемый для облегчения исходного кода. Вместо такой записи можно было бы написать тождественную:

```
main = putStr (convert' 2 14)
```

Дело в том, что операция применения функции (аппликация) имеет в языке Haskell самый высокий приоритет исполнения, при этом она является левоассоциативной, то есть при записи **putStr** `convert' 2 14` транслятор языка выдал бы ошибку, поскольку к функции **putStr** производится попытка применения параметра `convert'`, который не проходит статической проверки типов.

2.4. Общие свойства функций в функциональных языках программирования

Осталось кратко суммировать всё вышеизложенное и изучить общие свойства функций, рассматриваемые в функциональном программировании. К таким свойствам наиболее часто относят чистоту (то есть отсутствие побочных эффектов и детерминированность), ленивость и возможность производить частичные вычисления.

Итак, как уже упоминалось, физически реализуемыми являются такие кибернетические машины, выход которых зависит только от значений входных параметров. Это положение относится и к таким кибернетическим машинам, которые имеют внутренний накопитель — память (например, автомат Мили); использование внутреннего состояния моделируется передачей его значения из вызова в вызов в последовательности функций так, что это внутреннее состояние может рассматриваться в качестве входного параметра. Данное положение нашло чёткое отражение в парадигме функционального программирования, поскольку в ней принято, что функции, являясь математическими абстракциями, должны обладать свойством чистоты. Это означает,

что функция может управлять только выделенной для неё памятью, не модифицируя память вне своей области. Любое изменение сторонней памяти называется побочным эффектом, а функциям в функциональных языках программирования обычно запрещено иметь побочные эффекты.

Так же и с детерминированностью. Детерминированной называется функция, выходное значение которой зависит только от значений входных параметров. Если при одинаковых значениях входных параметров в различных вызовах функция может возвращать различные значения, то говорят, что такая функция является недетерминированной. Соответственно, обычно функции в функциональной парадигме являются детерминированными.

Конечно, есть некоторые исключения, к примеру, системы ввода-вывода невозможно сделать без побочных эффектов и в условиях полной детерминированности. Также и генерация псевдослучайных чисел осуществляется недетерминированной функцией. Можно привести ещё несколько подобных примеров. Само собой разумеется, что универсальный язык программирования, каковым является язык Haskell, должен предоставлять средства для решения этих практических задач. В данном случае побочные эффекты и недетерминированность вынесены из ядра языка и обёрнуты в так называемую монаду, которая скрывает в себе все нефункциональные особенности (описание монад выходит за рамки настоящей статьи).

Очень интересным свойством функций является ленивость. Не все функциональные языки предоставляют разработчику возможность определять ленивые функции, но язык Haskell изначально является ленивым, и разработчику необходимо делать специальные пометки для функций, которые должны осуществлять энергичные вычисления. Ленивая стратегия вычислений заключается в том, что функция не производит вычислений до тех пор, пока их результат не будет необходим в работе программы. Так значения входных параметров никогда не вычисляются, если они не требуются в теле функции. Это позволяет, помимо прочего, создавать потенциально бесконечные структуры данных (списки, деревья и т. д.), которые ограничены только физическим размером компьютерной памяти. Такие бесконечные структуры вполне можно обрабатывать ленивым способом, поскольку вычисляются в них только те элементы, которые необходимы для работы. Передача на вход какой-либо функции бесконечного списка не влечёт зацикливания программы, поскольку она не вычисляет весь этот список целиком (что было бы невозможным).

В качестве примера, наглядно иллюстрирующего ленивую стратегию вычислений, можно привести определение следующей несколько бесполезной функции:

```
firstNumbers n = take n [1..]
```

Данная функция возвращает список из n первых натуральных чисел. Стандартная функция **`take`** возвращает n первых членов произвольного списка, а вторым аргументом ей на вход подаётся бесконечный список натуральных чисел, записываемый как `[1..]`. Соответственно, при вызове функции `firstNumbers` происходит вычисление только заданного количества целых чисел.

Ну и в качестве наиболее распространённого примера использования ленивых вычислений можно привести такой, который используется даже в императивных языках программирования. Операции булевской алгебры И и ИЛИ в реализации для языков программирования могут не вычислять второй аргумент, если значение первого равно **`False`** (в случае операции И) или **`True`** (в случае операции ИЛИ, соответственно).

Наконец, уже упоминалось, что у функций есть тип. В функциональных языках программирования принято, чтобы тип функций был каррированным, то есть имел такой вид:

$$A_1 \to (A_2 \to \ldots (A_n \to B) \ldots)$$

где A_1, A_2, $\ldots A_n$ — типы входных параметров, а B — тип результата.

Такой подход к определению типов функций был предложен М. Шейнфинкелем[4] как способ, позволяющий проводить частичные вычисления [2]. Метод был развит Х. Карри [11], в честь которого он, собственно, и назван.

Каррированность функций означает, что такие функции принимают входные параметры по одиночке, а в результате такого одиночного применения получается новая функция. Так, если в функцию указанного выше типа подать первый параметр типа A_1, то в итоге получится новая функция с типом:

$$A_2 \to (A_3 \to \ldots (A_n \to B) \ldots)$$

Когда на вход функции подаются все входные параметры, в результате получается значение типа B.

В свою очередь это означает не только возможность частичного применения, но и то, что функции сами по себе могут быть объектами вычислений, то есть передаваться в качестве параметров другим функциям и быть возвращаемыми в качестве результатов. Ведь никто не ограничивает указанные типы A_1, A_2, $\ldots A_n$ и B только атомарными типами, это могут быть также и функциональные типы.

Перечисленные свойства функций в функциональных языках программирования открывают дополнительные возможности по использованию функционального подхода, поэтому разработчикам программного обеспечения рекомендуется изучить этот вопрос более подробно.

Заключение

Оставим идеалистам споры о преимуществах и недостатках тех или иных подходов к программированию. Важно понимать, что знание обоих методов описания вычислительных процессов позволяет более полноценно взглянуть на проектирование

[4] Моисей Исаевич Шейнфинкель (в зарубежной литературе известен как Moses Schönfinkel [1]) — русский математик, обозначивший концепцию комбинаторной логики. *Прим. ред.*

и разработку программных средств. К сожалению, на уроках программирования (информатики) в средних учебных заведениях редко изучают оба подхода, в результате чего у начинающих специалистов и интересующихся имеется известный перекос в сторону процедурного стиля.

Владение функциональным стилем и его основными методиками (декомпозицией и выражением ещё нерешённых задач через уже решённые) позволяет более эффективно решать управленческие задачи, поскольку эти приёмы также повсеместно встречаются в области регулирования и управления. В виду вышеизложенного автор надеется, что распространение и популяризация парадигмы функционального программирования позволит не только взращивать более серьёзных и вдумчивых специалистов в области информационных и автоматизированных систем, но и решит некоторые проблемы подготовки управленческих кадров.

Литература

[1] Moses Schönfinkel. — Статья в Wikipedia: URL: `http://en.wikipedia.org/wiki/Moses_Schönfinkel` (дата обращения: 20 июля 2009 г.).

[2] *Schönfinkel M.* Über die baustein der mathematischen logik. // *Math. Ann.* — 1924. — Vol. 92. — Pp. 305–316.

[3] *Wadler P.* Why no one uses functional languages // *ACM SIGPLAN Not.* — 1998. — Vol. 33, no. 8. — Pp. 23–27.

[4] *Х. Барендрегт.* Ламбда-исчисление. Его синтаксис и семантика: Пер. с англ. — М.: Мир, 1985. — 606 с.

[5] *Буч Г., Рамбо Дж., Якобсон И.* Язык UML. Руководство пользователя. — М.: ДМК Пресс, 2007. — 496 с.

[6] *Винер Н.* Кибернетика, или Управление и связь в животном и машине: Пер. с англ. — М.: Советское радио, 1958. — 216 с.

[7] *Вольфенгаген В. Э.* Комбинаторная логика в программировании. Вычисления с объектами в примерах и задачах. — М.: МИФИ, 1994. — 204 с.

[8] *Вольфенгаген В. Э.* Конструкции языков программирования. Приёмы описания. — М.: АО «Центр ЮрИнфоР», 2001. — 276 с.

[9] *Душкин Р. В.* Функциональное программирование на языке Haskell. — М.: ДМК Пресс, 2007.

[10] *Душкин Р. В.* Справочник по языку Haskell. — М.: ДМК Пресс, 2008. — 544 с.

[11] *Карри Х. Б.* Основания математической логики. — М.: Мир, 1969. — 568 с.

[12] *Марка Д. А., Макгоуэн К.* Методология структурного анализа и проектирования SADT. — М.: Метатехнология, 1993.

[13] *Филд А., Харрисон П.* Функциональное программирование: Пер. с англ. — М.: Мир, 1993. — 637 с.

Изменяемое состояние: опасности и борьба с ними

Евгений Кирпичёв
jkff@fprog.ru

Аннотация

В этой статье рассматриваются опасности использования изменяемого состояния в программах, преимущества использования неизменяемых структур и способы минимизации нежелательных эффектов от изменяемого состояния в тех случаях, когда оно все-таки необходимо.

3.1. Введение

Одно из ключевых отличий многих функциональных языков от объектно-ориентированных и процедурных — в поощрении использования неизменяемых данных; некоторые языки, в частности Haskell, даже не содержат в синтаксисе оператора присваивания! Апологеты функционального программирования объясняют это решение, в частности, тем, что отказ от изменяемых данных резко повышает корректность программ и делает их значительно более удобными для анализа с помощью формальных методов. Это действительно так, и в данной статье мы в этом убедимся. Однако полный отказ от изменяемых данных зачастую не оправдан по следующим причинам:

1) Некоторые техники программирования, применяющиеся в функциональных языках без присваиваний (к примеру, ленивые вычисления), применимы в более традиционных языках, таких как Java или C++, лишь с огромным трудом.

2) Для некоторых алгоритмов и структур данных не известно или не существует столь же эффективных аналогов без использования присваиваний (к примеру, для хэш-таблиц и систем непересекающихся множеств).

3) Многие предметные области по своей сути содержат изменяемые объекты (например, банковские счета; элементы систем в задачах имитационного моделирования, и т. п.), и переформулировка задачи на язык неизменяемых объектов может «извратить» задачу.

В данной статье мы поговорим о том, как пользоваться изменяемыми данными, не жертвуя простотой и корректностью кода.

3.2. Опасности изменяемого состояния

Перед тем, как перейти к техникам нейтрализации опасностей изменяемых данных, перечислим сами эти опасности.

3.2.1. Неявные изменения

Необходимое условие корректности программы — целостность ее внутреннего состояния, выполнение некоторых инвариантов (к примеру, совпадение поля `size` у объекта типа «связный список» с реальным числом элементов в этом списке). Код пишется так, чтобы в моменты, когда состояние программы наблюдаемо, инварианты не нарушались: каждая отдельная процедура начинает работать в предположении, что все инварианты программы выполняются и гарантирует, что после ее завершения инварианты выполняются по-прежнему.

Инварианты могут охватывать сразу несколько объектов: к примеру, в задаче представления ненаправленных графов логично требовать инварианта «если узел `A` связан ребром с узлом `B`, то и узел `B` связан ребром с узлом `A`».

Сохранение такого инварианта представляет собой непростую задачу: всякая процедура, меняющая один из составляющих его объектов, обязана знать не только о существовании всех остальных составляющих этого инварианта, но и обо всех составляющих *всех* инвариантов, зависящих от этого объекта! В противном случае, процедура может, сама того не ведая, нарушить инвариант.

Добиться такого знания порой чрезвычайно сложно; еще сложнее сделать это без нарушения модульности. Поэтому программисты стремятся делать инварианты охватывающими как можно меньше объектов и зависящими от как можно меньшего числа их изменяемых свойств.

Рассмотрим классический пример, иллюстрирующий данную проблему.

Пример: Обходчик интернета. Предположим, что мы разрабатываем программу — обходчик интернета. Она ходит по графу некоторого подмножества интернета и собирает данные со встречаемых страничек. В графе интернета узлами являются страницы, ребрами — ссылки с одних страниц на другие. В результате работы программа записывает в базу ссылки на некоторые из найденных страничек вместе с определенной дополнительной информацией.

Структура классов выглядит примерно так:

```
public class Address {
    private String url;
    public String getUrl() {
        return url;
    }
    public void setUrl(String u) {
        this.url = u;
    }
    int hashCode() {
        return url.hashCode();
    }
    boolean equals(Address other) {
        return url.equals(other.url);
    }
}

public class Node {
    Address address;
    List<Node> inLinks, outLinks;
}

public class Graph {
    Map<Address,Node> addr2node = new HashMap<Address,Node>();
}
```

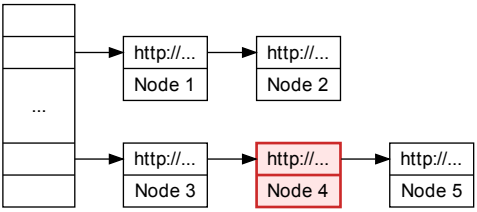


Рис. 3.1. Организация объекта класса HashMap

Во время разработки программы в один прекрасный момент выясняется, что доступ к некоторым страничкам приводит к перенаправлению (*redirect*) на другой адрес. Если в базе оказывается записан исходный адрес, то когда другая программа будет считывать адреса из базы и загружать соответствующие странички, она потратит лишнее время на перенаправление — поэтому лучше записать в базу новый адрес, полученный после перенаправления. В код добавляется следующее небольшое изменение:

```
Page download(Address address) {
    ...
    if(response.isRedirect()) {
        address.setUrl(response.getRedirectedUrl());
    }
    ...
}
```

И тут про некоторые адреса, определенно обязанные содержаться в графе, `addr2node.containsKey(address)` вдруг начинает отвечать **`False`**! Опытные читатели, скорее всего, заметят здесь проблему; однако, будучи встречена впервые, она может потребовать для решения пары часов отладки и сомнений в собственном душевном здоровье и качестве стандартной библиотеки. На деле проблема очень проста: метод `download` модифицировал объект `address`, но не учел, что его состояние является частью инварианта объекта `addr2node`.

Вспомним, как устроен класс `HashMap` в языке Java (рис. 3.1). Он реализует хэш-таблицу с закрытой адресацией: каждому хэш-коду (по модулю выделенной длины хэш-таблицы) соответствует «корзина» — список элементов, чей ключ обладает таким хэш-кодом.

Отмеченный на рисунке элемент соответствует адресу, измененному в методе `download`. В результате изменения поменялся и его хэш-код, однако элемент остался в корзине, соответствующей старому хэш-коду! В результате методом `download` оказывается нарушен инвариант класса `HashMap` — «Хэш-код всех ключей в одной корзине по модулю длины таблицы равен номеру этой корзины».

Теперь, к примеру, при попытке проверить наличие нового адреса в графе, поиск будет производиться в корзине, соответствующей хэш-коду нового адреса — конечно же, ключа там не окажется, т.к. он находится в другой корзине. При попытке прове-

рить наличие в графе старого адреса, поиск будет производиться в корзине, соответствующей старому адресу — однако самого адреса там также не окажется. Таким образом, после выполнения метода download в графе будут «отсутствовать» и старый, и новый адреса.

Как видно, объекты класса `Address` можно изменять только если известно, что они не содержатся ни в каком контейнере!

Единственное, по-видимому, решение данной проблемы — никогда не использовать изменяемые поля в качестве ключей контейнеров, в частности, в методах `equals`, `hashCode`, `compareTo`. Эта проблема настолько часта и опасна, что некоторые среды разработки генерируют предупреждение, если в одном из этих методов используется изменяемое поле.

В рассматриваемой задаче компромиссное решение таково: иметь в классе `Address` два поля: одно, неизменяемое, соответствует изначальному адресу странички, без учета перенаправлений, и именно им индексируются узлы в графе; второе, изменяемое, соответствует конечному адресу с учетом перенаправлений, и именно оно записывается в базу, но не используется в качестве ключа.

3.2.2. Кэширование и разделение

Следующая опасность изменяемых данных заключается в том , что их наличие существенно усложняет корректное кэширование. Рассмотрим один из классических примеров этой проблемы, широко известный в сообществе Java-программистов.

Пример: Геометрические классы GUI-библиотеки AWT. AWT содержит классы `Point`, `Dimension`, `Rectangle`, обозначающие соответственно: точку на плоскости, размеры прямоугольника и прямоугольник. Методы-аксессоры у классов окон возвращают объекты этих классов на запросы о положении и размере окна. Все три класса изменяемы:

```
class Point {
    int x, y;
}
class Dimension {
    int width, height;
}
class Rectangle {
    int x, y, width, height;
}
```

Как должен выглядеть метод получения размеров окна, возвращающий Dimension?

```
class Component {
    private Dimension size;
    ...
    Dimension getSize() {
```

```
            return size;
        }
        ...
    }
```

Этот код, конечно же, неверен! Клиент *может* изменить возвращенный объект, тем самым нарушив инвариант класса `Component` — «Всякое изменение размеров объекта типа `Component` оповещает всех клиентов, подписавшихся на это изменение (с помощью метода `addComponentListener`)». Заметим, что клиент может и не иметь никакого злого умысла при изменении такого возвращенного объекта — например, его может интересовать центр окна, который он станет вычислять таким образом:

```
Point center = w.getLocation();
center.x += w.getSize().width/2;
center.y += w.getSize().height/2;
```

Такая реализация `getSize` недопустима. Правильная реализация обязана возвращать объект, изменение которого не может повлиять на окно.

```
Dimension getSize() {
    return new Dimension(size.width, size.height);
}
```

Однако такая реализация обладает другим недостатком — низкой производительностью: всякий раз при вызове `getSize` создается новый объект. В ситуации, к примеру, вызова менеджера размещения окон для сложного интерфейса методы `getSize`, `getLocation`, `getBounds` могут вызываться десятки тысяч раз, и издержки на создание объекта становятся совсем не безобидными.

Точно такие же проблемы возникают при возвращении массивов методов.[1]

Рассмотрим еще один пример.

Пример: Корзина в интернет-магазине. В программе, реализующей интернет-магазин, есть класс «корзина». Общая стоимость продуктов зависит от стоимости каждого продукта и скидки, вычисляющейся по некоторым сложным правилам, зависящим от самих продуктов, от покупателя и т. п. Правила настолько сложные, что всякий раз вычислять стоимость корзины заново — неэффективно, поэтому она кэшируется и сбрасывается при изменении набора продуктов.

```
class Cart {
    private Customer customer;
    private List<Product> products;
    private int totalPrice = -1;
```

[1]При возвращении списков и других коллекций проблемы несколько меньше, поскольку они допускают *инкапсуляцию* изменений, давая возможность переопределить изменяющие методы (`add`, `set`, …) и, к примеру, запретить изменения.

```
    private int computeTotalPrice() {
        // Scary code here
    }

    public int getTotalPrice() {
        if(totalPrice == -1)
            totalPrice = computeTotalPrice();
        return totalPrice;
    }
    public void addProduct(Product p) {
        products.add(p);
        totalPrice = -1;
    }
    public void removeProduct(Product p) {
        products.remove(p);
        totalPrice = -1;
    }
}
```

Стоимость продуктов может изменяться во время работы магазина, поэтому в классе `Product` есть метод `setPrice`.

Близится праздник, и наш герой (назовем его Петром) подбирает подарки для своей семьи; это нелегкое дело отнимает у него 2 дня. С приближением праздника в интернет-магазине начинается распродажа, и некоторые товары дешевеют. К концу второго дня корзина Петра полна подарков, и он уже готов нажать «Checkout», но тут он замечает, что — о ужас! — указанная в корзине сумма заказа не соответствует суммарной стоимости товаров. Пётр негодует: закэшированное значение `totalPrice` не было обновлено при изменении цен продуктов — нарушился инвариант «`totalPrice` равно либо –1, либо истинной суммарной цене содержащихся в корзине продуктов», поскольку метод `setPrice` действовал лишь над объектом класса `Product`, ничего не зная об объекте `Cart`, в чьем инварианте этот `Product` присутствовал.

Для решения этой проблемы придется либо отказаться от кэширования цены вовсе, либо сделать так, чтобы класс `Product` позволял подписываться на изменения цены. Оба решения одинаково плохи: первое неэффективно, второе — сложно и подвержено ошибкам: класс `Product`, бывший обычной структурой данных, обрастает всевозможными оповещателями, а все его пользователи обязаны на эти оповещения подписываться. Легко представить, какая путаница будет в коде, учитывая, что бизнес-область содержит множество взаимосвязанных объектов с изменяемыми свойствами — гораздо больше, чем просто `Product` и `Cart`.

3.2.3. Многопоточность

Подавляющее большинство багов в многопоточных программах связано с изменяемыми данными, а именно — с тем, что две (или более) корректных последователь-

ности изменений, переплетаясь в условиях многопоточности, вместе образуют некорректную. Вот классический пример такой ошибки.

Пример: Банковские транзакции. Пусть есть класс `BankAccount`, поддерживающий операции `deposit` (положить деньги на счет) и `withdraw` (снять деньги со счета).

```
class BankAccount {
    void deposit(int amount) {
        setMoney(getMoney() + amount);
    }
    void withdraw(int amount) {
        if(amount > getMoney())
            throw new InsufficientMoneyException();
        setMoney(getMoney() − amount);
    }
}
```

Предположим, у супругов Ивана да Марьи есть общий семейный счет, на котором лежит 100 рублей. Иван решает положить на счет 50 рублей, а Марья в это же время решает положить на счет 25 рублей.

Действия Ивана	Действия Марьи	Деньги на счете
$deposit(50)$	$deposit(25)$	100
$getMoney() \to 100$		100
	$getMoney() \to 100$	100
$setMoney(100 + 50)$		150
	$setMoney(100 + 25)$	125
Итого		125

В результате деньги Ивана оказываются выброшенными на ветер.

Причина этого — переплетение трасс: каждая из операций по отдельности работает правильно, однако лишь в предположении, что состояние системы во время ее работы контролируется только ею; это предположение оказывается неверным. Такая проблема может возникнуть не только в условиях многопоточности, но в этих условиях она проявляется особенно часто и ярко. Возможные пути решения — использование обыкновенных примитивов синхронизации или специальных средств, таких как транзакции.

3.2.4. Сложный код

С введением изменяемых данных в коде появляется измерение времени как на высоком уровне (взаимодействия компонентов), так и на низком — уровне последовательности строк кода. Вслед за ним приходит дополнительная сложность: необходимо

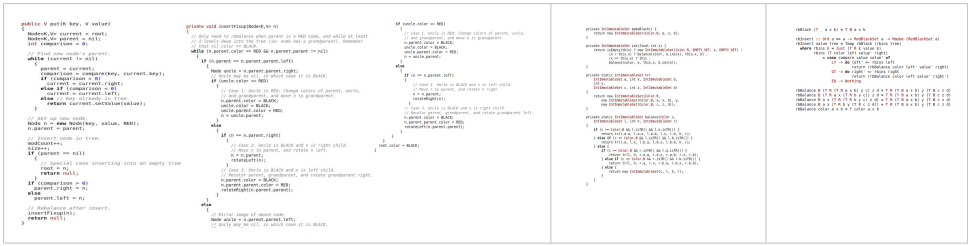
не только решить, *что* надо сделать, но и *в каком порядке*. Она проявляется, в основном, в реализациях сложных структур данных и алгоритмов.

Пример: Двусвязный список. Корректная реализация этой структуры данных — на удивление трудное дело: немногие могут реализовать двусвязный список правильно с первой попытки. Вот (слегка сокращенный) фрагмент кода, осуществляющего вставку в двусвязный список в GNU Classpath. Он выглядит невинно, но можете ли вы *в уме* убедиться в его корректности, не рисуя на бумаге диаграмм для трех возможных случаев и не отслеживая последовательно влияние каждой строки кода на диаграмму?

```
public void add(int index, Object o) {
    Entry e = new Entry(o);
    if (index < size) {
        Entry after = getEntry(index);
        e.next = after;
        e.previous = after.previous;
        if (after.previous == null)
            first = e;
        else
            after.previous.next = e;
        after.previous = e;
    } else if (size == 0) {
        first = last = e;
    } else {
        e.previous = last;
        last.next = e;
        last = e;
    }
    size++;
}
```

Пример: Красно-черные деревья. Более радикальный пример — реализация красно-черных деревьев: на рис. 3.2 представлен вид «с высоты птичьего полета» на процедуры вставки элемента в такую структуру данных: в изменяемое дерево на Java (из GNU Classpath), в неизменяемое на Java (из библиотеки functionaljava) и в неизменяемое на Haskell.

Даже использование диаграмм на бумаге не решает проблемы наличия времени: для отражения изменений во времени приходится в каждой строке кода либо перерисовывать диаграмму заново на чистом участке листа, либо зачеркивать ее части, увеличивая путаницу.



Java, mutable Java, immutable Haskell

Рис. 3.2. Вставка в красно-черное дерево

3.2.5. Наследование

Классы с изменяемым состоянием плохо поддаются наследованию. Класс-наследник, согласно принципу подстановки Лисков,[2] должен быть пригоден к использованию вместо базового класса в любом контексте — в частности, должен поддерживать все его операции и сохранять все его инварианты и спецификации операций. По отношению к операциям, способным изменять состояние объекта базового класса, это означает, что *класс-наследник не имеет права накладывать дополнительные ограничения на это изменяемое состояние*, т.к. тем самым он нарушит спецификацию и инварианты изменяющих методов. Рассмотрим классическую иллюстрацию этой проблемы.

Пример: Геометрические фигуры.

```
class Rectangle {
    private int width, height;
    public Rectangle(int w,h) {
        this.width = w;
        this.height = h;
    }
    int getWidth()              {...}
    int getHeight()             {...}
    void setWidth(int width)    {...}
    void setHeight(int height) {...}
}
class Square extends Rectangle {
    public Square(int side) {
        super(side,side);
    }
}
```

[2]Имеется в виду «принцип подстановки Барбары Лисков», также известный как LSP (Liskov Substitution Principle), гласящий «Если тип S унаследован от типа T, то должно быть возможным подставить объект типа S в любом месте программы, ожидающем тип T, без изменения каких-либо желаемых свойств программы — в т. ч. корректности» [3].

В классе `Rectangle` спецификация операций `setWidth` и `setHeight` такова:

- `r.getWidth() == w && r.getHeight() == h`
  ⇒ после `r.setWidth(w2)` верно
  `r.getWidth() == w2 && r.getHeight() == h`

- `r.getWidth() == w && r.getHeight() == h`
  ⇒ после `r.setHeight(h2)` верно
  `r.getWidth() == w && r.getHeight() == h2`

Вызов `setWidth` или `setHeight` на объекте класса `Square` обязан также удовлетворять этим спецификациям, однако при этом, очевидно, будет разрушен инвариант класса `Square` «`getWidth() == getHeight()`».

Правило стоит повторить еще раз: *Класс-наследник не имеет права накладывать дополнительные ограничения на изменяемое состояние базового класса.*

3.3. Круги ада

Ознакомившись с некоторыми недостатками изменяемого состояния, приступим к организации борьбы с ними. Первый этап борьбы — подробное изучение врага. Выполним классификацию вариантов изменяемого состояния по степени их «вредности».

Прекрасная классификация предложена Скоттом Джонсоном в [2]; приведем ее с небольшими изменениями и добавлениями. Чем больше номер круга ада, тем больше опасностей подстерегает нас. Избавление от опасностей будет зачастую заключаться в переходе с большего номера к меньшему.

1) **Отсутствие изменяемого состояния.** Этот «нулевой» круг ада абсолютно безопасен с точки зрения вышерассмотренных проблем, но достижим лишь в теории.

2) **Невидимое программисту изменяемое состояние** — код алгоритмов, не использующих изменяемое состояние, компилируется в машинный код, использующий изменяемые регистры, стек, память, что при правильной реализации компилятора заметить невозможно. Этот круг так же безопасен с практической точки зрения, как и предыдущий.

3) **Невидимое клиенту изменяемое состояние** — скажем, локальные переменные-счетчики внутри процедуры: изменение таких переменных ненаблюдаемо извне самой процедуры.[3] Этот круг безопасен с точки зрения клиента.

[3] В некоторых языках используются *системы эффектов* [1], позволяющие компилятору делать подобные суждения автоматически.

Рис. 3.3. Круги ада

4) **Монотонное изменяемое состояние** — переменные, присваивание которых происходит не более 1 раза: переменная вначале не определена, а затем определена. Это довольно безобидный тип изменяемого состояния, поскольку у переменной всего 2 состояния, лишь одно из которых не целостно (к тому же, *перевести* переменную в неопределенное состояние невозможно!), и обычно легко обеспечить, чтобы в нужный момент такая переменная оказалась определена. Монотонное изменяемое состояние часто используется для реализации ленивых вычислений.

5) **Двухфазный цикл жизни.** Это разновидность п. 4, при которой состояний у объекта более двух, при этом жизнь объекта поделена на две фазы: инициализация («наполнение»), при которой к нему происходит доступ только на запись, и мирная жизнь, при которой доступ производится только на чтение. Например, система сначала собирает статистику, а затем всячески анализирует ее. Необходимо гарантировать, что во время фазы чтения не будет производиться запись, и наоборот. Позднее будет рассмотрен прием («заморозка»), позволяющий давать такую гарантию.

6) **Управляемое изменяемое состояние** — такое, как внешняя СУБД: в системе присутствуют специальные средства для координации изменений и ограничения их опасностей, например, транзакции.

7) **Инкапсулированное изменяемое состояние** — переменные, доступ к которым производится только из одного места (скажем, private поля объектов). Контроль за целостностью состояния лежит целиком на реализации объекта, и если реализация правильная, то не существует способа привести объект в не-целостное состояние (инварианты самого объекта не нарушаются). Достаточно правильно реализовать сам объект. Тем не менее, для контроля инвариантов, охватывающих несколько объектов, по-прежнему необходимы специальные средства; либо же необходимо инкапсулировать весь контроль за состоянием этих несколь-

ких объектов в другом объекте.

8) **Неинкапсулированное изменяемое состояние** — глобальные переменные. Всем известно, что это — страшное зло. В этом случае нельзя сказать *ничего* о том, кто и когда изменяет глобальную переменную, не изменяет ли ее кто-нибудь прямо сейчас, между *вот этими* двумя строками кода, и т. п.

9) **Разделяемое между несколькими процессами изменяемое состояние.** В условиях многопоточности управление изменяемым состоянием превращается в ад во всех случаях, кроме тривиальных. Огромное количество исследований посвящено разработке методик, позволяющих хоть как-то контролировать корректность многопоточных программ, но пока что главный вывод таков: хотите избежать проблем с многопоточностью — минимизируйте изменяемое состояние. Основная причина трудностей заключается в том, что если имеется N потоков, каждый из которых проходит K состояний, то количество возможных последовательностей событий при одновременном выполнении этих потоков имеет порядок K^N. Техники, позволяющие минимизировать подобные эффекты, рассмотрены ниже.

3.4. Борьба

Прежде чем перейти к обсуждению техник обезвреживания изменяемого состояния, обсудим следующие глобальные идеи:

- **Минимизация общего числа состояний:** Чем меньше у системы состояний, тем меньшее количество случаев надо учитывать при взаимодействии с ней. Следует помнить и о том, что чем больше состояний, тем больше и *последовательностей* состояний, а именно непредусмотренные *последовательности* состояний зачастую являются причинами багов.

- **Локализация изменений:** Чем более локальные (обладающие меньшей областью видимости) объекты затрагивает изменение, тем из меньшего числа мест в коде изменение может быть замечено. Чем менее изменения размазаны между несколькими объектами во времени, тем легче логически сгруппировать их в несколько крупных и относительно независимых изменений объектов. К примеру, при сложном изменении узла структуры данных лучше сначала вычислить все новые характеристики этого узла, а затем произвести изменения, нежели производить изменения сразу в самом узле.

- **Разграничение права на наблюдение и изменение состояния (инкапсуляция):** Чем меньше клиентов могут изменить состояние, тем меньше действия этих клиентов нужно координировать. Чем меньше клиентов могут прочитать состояние, тем легче его защищать от изменений во время чтения.

- **Исключение наблюдаемости промежуточных не-целостных состояний:** Если систему невозможно застать в таком состоянии, то снаружи она всегда выглядит целостной и корректно работающей.
- **Навязывание эквивалентности состояний и их последовательностей:** Если некоторые состояния или их последовательности в каком-то смысле эквивалентны, то клиент избавлен от необходимости учитывать их частные случаи.

И, наконец, сами техники.

3.4.1. Осознание или отказ

Самый первый и самый важный шаг, который следует предпринять, разрабатывая систему, использующую изменяемое состояние — понять, чем в действительности обусловлено его наличие. Действительно ли в предметной области есть понятие объектов, изменяющихся во времени? Действительно ли в предметной области меняются *именно те* объекты, которые вы собираетесь сделать изменяемыми?

К примеру, неудачное архитектурное решение об изменяемости геометрических классов `java.awt` было бы отброшено уже на этом этапе: в геометрии не бывает изменяемых точек и прямоугольников! Авторы библиотеки неверно определили изменяющиеся сущности: в действительности меняются не координаты точки — левого верхнего угла окна, а меняется то, *какая именно* точка является левым верхним углом окна.

Аналогично этому примеру, множества и словари, в их математическом понимании, также не являются сами по себе изменяемыми объектами — во многих задачах оправдано использование *чисто функциональных структур данных* (структур данных, не использующих присваивания). К примеру, в интерфейс чисто функционального множества вместо операции «добавить элемент» входит операция «получить множество, отличающееся от данного наличием указанного элемента». Такие структуры допускают реализацию, совершенно не уступающую по эффективности изменяемым структурам, а в некоторых задачах и существенно превосходящую их (как ни странно, по потреблению памяти) — см. [5].

Если предметная область диктует наличие изменяющихся во времени объектов, то необходимо признать этот факт и помнить о нем постоянно. Особенно важно это в условиях многопоточности.

Необходимо осознать, что код — это больше не спецификация работы алгоритма, а разворачивающаяся во времени последовательность событий с ненулевой длительностью: ни один вызов метода, ни одно присваивание не проходит мгновенно, и между любыми двумя строками кода есть временной «зазор».

Это кажется тривиальным, но если постоянно помнить об этих правилах, то многие глупые (и не только) ошибки многопоточного программирования или неявного взаимодействия становятся отчетливо видны на этапе написания, а не на этапе отладки продакшн-системы.

3.4.2. Инкапсуляция

Большинство описанных ранее проблем с изменяемым состоянием возникает из-за того, что слишком многие клиенты (функции, классы, модули) изменяют состояние, и слишком многие его читают. При этом скоординировать чтение и изменение корректным образом не удается без нарушения модульности и сильного усложнения кода. Минимизировать проблемы такого рода можно, разграничив доступ к состоянию — предоставив клиентам как можно меньше способов прочтения и изменения состояния. Вместе с этим крайне важно, чтобы доступ был согласован с разбиением системы на компоненты. Так компонент, читающий состояние в процессе своей работы, должен либо быть тесно связан с компонентами, могущими его записывать, либо получать доступ к состоянию только в моменты, когда оно не может изменяться, либо проектироваться с учетом того, что состояние может измениться в любой момент.

Вот несколько советов, связанных с инкапсуляцией:

Не возвращайте изменяемые объекты из «читающих» методов Если метод, по своей сути, предназначен для *выяснения* какого-то свойства объекта (скажем, даты создания), а не для *получения доступа* к этому свойству, то он должен возвращать неизменяемый объект. В крайнем случае, он должен возвращать «свежий» объект (как сделано в `java.awt`), но через возвращенный объект должно быть невозможно повлиять на исходный объект.

- Возвращать дату создания в виде объекта класса `Calendar`, возвращая `private`-поля, недопустимо, т.к. объекты класса `Calendar` изменяемы. Гораздо лучше возвращать дату создания в виде `long` — например, в виде числа миллисекунд с 1 января 1970 года (число, возвращаемое функцией **`System.`**`currentTimeMillis()` в Java).
- Возвращая коллекцию, возвращайте ее неизменяемую обертку.
- Возвращая коллекцию, позаботьтесь о том, чтобы составляющие ее объекты были неизменяемы.

Не возвращайте массивы — возвращайте коллекции Если необходимо *получить доступ* к свойству типа «набор объектов», то возвращайте его в виде коллекции, а не в виде массива. Доступ к массивам не инкапсулирован: имея в своем распоряжении массив, всякий клиент может прочитать или перезаписать какие-то из его элементов; при этом невозможно гарантировать атомарность доступа в условиях многопоточности, тогда как коллекции позволяют переопределять операторы чтения и записи и делать их атомарными (`synchronized`).

Предоставляйте интерфейс изменения, соответствующий предметной области Скажем, класс `BankAccount` должен предоставлять не метод `getMoney`/`setMoney`, а методы `deposit` и `withdraw`. Благодаря этому реализация атомарности и контроль

за инвариантами банка (скажем, ненулевое количество денег на счете и сохранение общего числа денег в банке) ляжет на реализацию `BankAccount`, а не на клиентов, вызывающих `getMoney`/`setMoney`. Сюда же можно отнести атомарные операции: `AtomicInteger.addAndGet`, `ConcurrentMap.putIfAbsent`, и т. п. Об этой технике речь пойдет ниже, в разделе «Многопоточные техники».

3.4.3. Двухфазный цикл жизни и заморозка

Довольно часто изменяемые данные требуются для того, чтобы накопить информацию и проинициализировать некоторый объект, который затем будет использоваться уже без изменений. Таким образом, цикл жизни объекта делится на две фазы: фаза записи (накопления) и фаза чтения; причем во время фазы накопления не производится доступ на чтение *извне*, а во время фазы чтения не производится запись. В результате все, кто читают объект, видят его неизменным. Для корректной работы системы достаточно обеспечить, чтобы чтение и запись не пересекались во времени.

В точке перехода от фазы накопления к фазе чтения можно подготовить информацию (закэшировать какие-то значения, построить индексы), что сделает чтение более эффективным.

Существует два подхода к организации двухфазного цикла жизни: статический и динамический.

Статический подход предполагает, что интерфейсы объекта в фазе чтения и фазе записи отличаются.

```
public interface ReadFacet {
    Foo getFoo(int fooId);
    Bar getBar();
}
public interface WriteFacet {
    void addQux(Qux qux);
    void setBaz(Baz baz);

    ReadFacet freeze();
}

class Item implements WriteFacet {
    ...
    ReadFacet freeze() {
        return new FrozenItem(myData);
    }
}
class FrozenItem implements ReadFacet {
    FrozenItem(ItemData data) {
        this.data = data.clone();
        prepareCachesAndBuildIndexes();
```

```
    }
    ...
}
```

Клиент получает объект типа `WriteFacet`, производит в него запись, а затем, когда запись окончена, замораживает объект и в дальнейшем пользуется для чтения полученным `ReadFacet`. Конечно, необходимо, чтобы созданный объект `ReadFacet` был полностью независим от замораживаемого, в противном случае дальнейший вызов записывающих методов испортит его.

Зачастую организовывать два лишних интерфейса неудобно, или же такой возможности может просто не оказаться: скажем, клиент ожидает библиотечный интерфейс, содержащий и методы чтения, и методы записи. При этом все же желательно иметь возможность гарантировать отсутствие записей после начала чтения, и возможность построить кэши после фазы чтения с гарантией их будущей целостности. Тут пригодится второй подход к заморозке.

Динамический подход предполагает, что формально интерфейсы объекта в фазе чтения и записи одинаковы, однако фактически поддерживаемые операции отличаются, как и в случае статического подхода.

```
class Item implements QuxAddableAndFooGettable {
    private boolean isFrozen = false;
    ...
    void addQux(Qux qux) {
        if(isFrozen) throw new IllegalStateException();
        ...
    }

    Foo getFoo(int fooId) {
        if(!isFrozen) throw new IllegalStateException();
        // Efficiently get foo using caches/indexes
    }

    void freeze() {
        prepareCachesAndBuildIndexes();
        isFrozen = true;
    }
}
```

Здесь статическая проверка заменяется на динамическую, однако общая идея остается той же: на фазе записи объект предоставляет только интерфейс записи, на фазе чтения — только интерфейс чтения.

В качестве иллюстрации такого подхода рассмотрим пример из практики автора.

Пример: Кластеризация документов. Каждый документ в программе описывается длинным разреженным битовым вектором свойств. В начале программа анализи-

рует документы, составляя их векторы свойств и пользуясь изменяющим интерфейсом битового вектора (установить/сбросить бит). Затем, при вычислении матрицы попарных расстояний между документами выполняется лишь две операции: вычисление мощности пересечения и мощности объединения двух векторов. Эти операции можно реализовать очень эффективно, если «подготовить» векторы, построив на них «пирамиду» (не будем углубляться в подробности), однако обновлять пирамиду при вставках в битовый вектор слишком дорого. Поэтому пирамида строится для каждого вектора сразу после вычисления векторов и перед вычислением матрицы попарных похожестей, и матрица затем строится очень быстро (ускорение в рассматриваемой задаче достигло двух порядков).

Поскольку объекты с двухфазным циклом жизни легко использовать корректно, то стоит попытаться разглядеть их в своей задаче или свести ее к ним; вовсе не обязательно при этом даже использовать заморозку — само наличие двухфазности вносит в программу стройную структуру.

3.4.4. Превращение изменяемой настройки в аргумент

Довольно часто бывает так, что изменяемые данные используются для «настройки» объекта: объект настраивается с помощью нескольких сеттеров, а затем выполняет работу. Это похоже на двухфазный цикл жизни, однако процесс настройки сконцентрирован в одном месте кода и в одном коротком промежутке времени, известное количество настроек подаются одна за другой.

В некоторых случаях это не является проблемой, однако представим себе такой класс:

Пример: Подсоединение к базе данных.

```
class Database {
    void setLogin(String login);
    void setPassword(String password);
    Connection connect() throws InvalidCredentialsException;
}
```

Наличие такого класса может поначалу выглядеть оправданным, если он используется, скажем, из графического интерфейса, который сначала запрашивает у пользователя логин (и, запросив, вызывает `setLogin`), затем запрашивает пароль (и вызывает `setPassword`), а затем по нажатию кнопки «Connect» вызывает `connect`.

Однако если объект `Database` окажется разделяемым между несколькими пользователями, то вызовы `setLogin`, `setPassword`, `connect` начнут путаться между собой, пользователи будут получать чужие соединения, и т. п. — такая ситуация совершенно недопустима.

Гораздо лучше избавиться от изменяемости в интерфейсе `Database`:

```
class Database {
    Connection connect(String login,String password)
```

```
        throws InvalidCredentialsException;
}
```

При этом, конечно, клиенту придется управлять хранением значений `login` и `password` самостоятельно, однако путаница будет устранена.

Для ситуаций, когда хранение данных нежелательно (например, долгое хранение пароля в памяти может быть нежелательно по соображениям безопасности), пригодится, в частности, паттерн «Curried Object» («Каррированный объект»), о котором речь пойдет позже.

3.4.5. Концентрация изменений во времени

Родственный предыдущему метод — концентрирование изменений во времени. Он заключается в том, чтобы вместо последовательности мелких изменений выполнять одно большое, тем самым убивая сразу двух зайцев:

1) Избавление от промежуточных состояний между мелкими изменениями.

2) Избавление от необходимости устанавливать протокол последовательности внесения изменений.

П.1 позволяет решить проблемы, сходные с описанными в предыдущем разделе (путаницу между клиентами).

П.2 полезен в случаях, когда мелкие изменения в определенном смысле взаимозависимы, и не всякая последовательность мелких изменений является корректной, поэтому приходится навязывать клиенту протокол, предписывающий, в каком порядке нужно вносить изменения. Это может быть чрезвычайно неудобно.

Пример: Библиотека бизнес-правил. Рассмотрим библиотеку, позволяющую пользователю задать ряд бизнес-правил для вычисления некоторых величин. Правила (формулы) могут быть взаимозависимы.

Интерфейс библиотеки выглядит так:

```
interface RuleEngine {
    void addRule(String varName, String formula)
        throws ParseException, UndefinedVariableException;

    double computeValue(String varName);
}
```

В каждый момент правила должны быть целостными, поэтому `addRule` бросает `UndefinedVariableException` в случае, когда `formula` ссылается на переменную, для которой еще нет правила.

Большой минус такой библиотеки — в том, что если клиенту необходимо задать сразу несколько взаимозависимых правил (скажем, считать их из файла или электронной таблицы), то клиент должен *сам* заботиться о том, чтобы подавать правила в порядке топологической сортировки, т.е. добавлять зависимое правило только после добавления всех, от которых оно зависит!

Лучше перепроектировать интерфейс так:

```
interface RuleParser {
    RuleSet parseRules(Map<String,String> var2formula)
        throws ParseException, CircularDependencyException;
}
interface RuleSet {
    double computeValue(String varName);
}
```

Теперь конструирование набора взаимозависимых правил инкапсулировано в методе `parseRules`. Он сам выполняет топологическую сортировку передаваемых ему пар «переменная/формула» и детектирует циклические зависимости.

3.4.6. Концентрация изменений в пространстве объектов

Эта методика призвана устранить необходимость в поддержании инвариантов, охватывающих сразу несколько объектов: чем меньше объектов охватывает инвариант, тем проще его сохранять, и тем меньше шансов, что кто-то разрушит инвариант, изменив один из составляющих его объектов, но не изменив и ничего не зная о другом.

К примеру, сложность реализации двусвязных списков в значительной степени проистекает из того, что каждая операция, затрагивающая определенный узел, должна позаботиться о двух его соседних узлах и о самом списке (ссылках на первый/последний узел).

Методика состоит в том, чтобы минимизировать количество объектов, охватываемых инвариантами. Зачастую она сводится к разрыву зависимостей между объектами и, особенно, разрыву циклических зависимостей. К сожалению, с двусвязными списками, по-видимому, ничего не поделать; мы рассмотрим другой пример.

Пример: Многопользовательская ролевая игра. В рассматриваемой игре есть понятие «артефакта», и характеристики артефакта могут зависеть от характеристик игрока, носящего его. Поэтому в артефакт включается информация о его владельце:

```
class Artifact {
    Player getOwner();
    void setOwner(Player player);

    Picture getPicture();

    int getStrengthBoost();
    int getHealthBoost();
    int getDexterityBoost();
    int getManaBoost();
}
class Player {
    List<Artifact> getArtifacts();
    void addArtifact(Artifact a);
```

```
    void dropArtifact(Artifact a);
    void passArtifact(Artifact a, Player toWhom);
}
```

Должен выполняться следующий инвариант: «если `a.getOwner()==p`, то `p.getArtifacts().contains(a)`, и наоборот».

Все 4 метода `get*Boost()` учитывают расу игрока (`getOwner().getRace()`): у великанов умножается на 2 значение `getStrengthBoost` любого артефакта, у гномов — `getHealthBoost`, у эльфов — `getDexterityBoost`, у друидов — `getManaBoost`.

Пока артефакт лежит на земле, его `getOwner()` равен **`null`**. Когда игрок подбирает, роняет или передает артефакт, вызывается `setPlayer`.

Для каждого артефакта, присутствующего на карте мира, нужен отдельный экземпляр класса `Artifact`, хранящий в себе `Picture`, `strengthBoost`, `healthBoost`, `dexterityBoost` и `manaBoost` — использовать один и тот же экземпляр даже для совершенно одинаковых артефактов нельзя, т.к. экземплярами могут владеть разные игроки. Учитывая, что артефактов на карте может быть *очень* много, а характеристик у них может быть гораздо больше, чем указано здесь — имеет место лишний расход памяти.

Ситуацию можно улучшить, если сделать класс `Artifact` неизменяемым и разорвать зависимость от игрока (убрать методы `getPlayer`/`setPlayer`), тем самым избавившись полностью от необходимости поддерживать указанный инвариант, позволив использовать один и тот же экземпляр класса `Artifact` в рюкзаках разных игроков и существенно сократив потребление памяти.

Встает, конечно, вопрос — как же теперь быть с методами `get*Boost()`? Ведь метод без аргументов у артефакта больше не может давать правильный ответ.

Ответов несколько, и все они очень просты:

- Добавить аргумент типа `Player` к методам `get*Boost()`.
- Перенести методы в класс `Player`: `Player.getStrengthBoost(Artifact a)`.
- Переименовать методы в `getBaseStrenghBoost` и т. п., и вынести логику определения действия артефакта в отдельный класс `ArtifactRules`, в методы `getStrengthBoost(Artifact a, Player p)`. К таким методам можно будет легко добавить обработку и других условий: погоды, наличия вокруг других игроков и т. п.

Этот прием — избавление от изменяемых данных благодаря разрыву зависимостей — используется в чисто функциональных структурах данных и позволяет разделять значительную часть информации между двумя мало отличающимися структурами. Благодаря этому бывает возможна огромная экономия памяти (в задаче из практики автора, где требовалось хранить большое количество не очень сильно различающихся целочисленных множеств, переход от изменяемого красно-черного дерева к неизменяемому позволил сократить потребляемую память примерно на два порядка).

3.4.7. Каррирование

Название этой методики связано с аналогичным понятием из функционального программирования: при каррировании функции с несколькими аргументами некоторые из этих аргументов фиксируются, и получается функция от оставшихся.

Методика описана в статье [4] и предназначена для упрощения сложных протоколов, где каждый вызов требует большого количества аргументов, некоторые из которых меняются редко или не меняются вовсе.

Эта методика прекрасно подходит для разрыва зависимости по состоянию между несколькими клиентами, использующими объект, и попутно дает еще несколько преимуществ.

Можно сказать, что паттерн *Curried Object* — это частный случай инкапсуляции состояния; более точно, он предписывает выделить часть объекта, позволяющую хорошую инкапсуляцию состояния, в самостоятельный объект.

Пример: Улучшение безопасности подсоединения к БД. Рассмотрим упомянутый выше пример с подсоединением к базе данных. Проблема изначальной версии заключалась в том, что вызовы разных клиентов `setLogin`, `setPassword`, `connect` переплетались, а исправленной — в том, что программа могла долго хранить в памяти пароль. Фиксированным (хотя и неявным) аргументом в данном случае является то, какой клиент производит вызовы. Применим паттерн *Curried Object* и выделим каждому клиенту его личный объект для обработки протокола соединения.

```
class Database {
    Connector makeConnector();
}
class Connector {
    void sendLogin(String login);
    void sendPassword(String password);
    Connection connect()
        throws InvalidCredentialsException;
}
```

В такой реализации `makeConnector` будет создавать новый независимый объект, производящий сетевое соединение с базой данных и в методах `sendLogin`, `sendPassword` сразу отсылающий логин и пароль по сети. У каждого клиента объект `Connector` будет свой, поэтому клиенты не будут мешать друг другу.

Рассмотрим еще одну иллюстрацию паттерна «Curried Object».

Пример: Загрузчик данных. Программа предназначена для загрузки в базу большого количества данных от разных клиентов: клиент подключается к программе и загружает большое количество данных, затем отключается. Программа обязана обеспечить транзакционность загрузки данных от каждого клиента. Изначально API программы проектируется так:

```
class DataLoader {
    ClientToken beginLoad(Client client);
```

```
    void writeData(ClientToken token, Data data);
    void commit(ClientToken token);
    void rollback(ClientToken token);
}
```

Клиент вызывает `beginLoad` и с помощью полученного `ClientToken` многократно вызывает `writeData`, затем вызывает `commit` или `rollback`. Эта программа избавлена от проблем переплетения запросов между клиентами, однако код `DataLoader` довольно сложен: он хранит таблицу соответствия клиентов и транзакций и соединений БД, и в каждом из методов пользуется соответствующими элементами таблицы.

В некоторых задачах клиенты сами по себе могут быть многопоточными: скажем, клиент может в несколько потоков вычислять данные и выполнять их запись. Если метод `writeData` не обладает потокобезопасностью при фиксированном `token`, то код еще усложнится: добавится таблица соответствия «клиент / объект синхронизации», и метод `writeData` будет синхронизироваться по соответствующему ее элементу.

Нельзя забывать и о том, что доступ к таблицам также должен быть синхронизирован.

Существенно упростить код можно, если перепроектировать `DataLoader`, применив *Curried Object*:

```
class DataLoader {
    PerClientLoader beginLoad(Client client);
}
class PerClientLoader {
    void writeData(Data data);
    void commit();
    void rollback();
}
```

Теперь класс `DataLoader` полностью избавлен от изменяемого состояния! В классе `PerClientLoader` нет никаких таблиц, и синхронизация выполняется тривиально — достаточно объявить все три метода как `synchronized`. Клиенты также никак не могут помешать друг другу. Код получился простым и безопасным.

3.4.8. Многопоточные техники

Как уже было сказано, трудности с написанием корректных многопоточных программ проистекают от большого количества возможных совместных трасс выполнения нескольких процессов. Корректность необходимо гарантировать на всех трассах, в связи с чем приходится рассматривать большое количество возможных переплетений трасс и проверять на корректность каждое из них. Напомним, что количество трасс при N потоках, у каждого из которых K состояний, можно оценить как K^N.

Возможные пути уменьшения числа трасс, которые необходимо учитывать, таковы:

Использование критических секций для синхронизации. Охватывание критической секцией блока кода, содержащего M состояний, уменьшает число состояний на $M - 1$, превращая весь блок в один переход между двумя состояниями. Благодаря этому соответственно уменьшается количество совместных трасс.

Использование атомарных операций. Это двоякий совет. С одной стороны, речь идет о том, чтобы пользоваться эффективными аппаратно реализованными операциями, такими как «прочитать-и-увеличить» (get-and-add), «сравнить-и-поменять» (compare-and-exchange) и т. п. С другой стороны, что более важно, речь идет о том, чтобы предоставлять API в терминах атомарных, соответствующих предметной области, операций:

- «Перевести деньги с одного счета на другой» (помимо «снять деньги, положить деньги»).
- «Добавить элемент, если он еще отсутствует» (помимо «добавить элемент, проверить наличие») — кстати, сюда же относятся SQL-команды MERGE и INSERT IGNORE.
- «Выполнить соединение с данным логином и паролем» (помимо «установить логин, установить пароль, выполнить соединение»).
- и т. п.

Локализация изменяемого состояния. Вместо применения нескольких изменений к глобально видимому объекту — вычисление большого изменения в локальной области видимости и его атомарное применение. Этот прием уже был рассмотрен выше в разделе «Концентрация изменений во времени».

Навязывание эквивалентности трасс. Кардинально иной подход к уменьшению числа различных трасс — стирание различий между некоторыми из них. Для этого можно использовать следующие два алгебраических свойства некоторых операций (а если операции ими не обладают — попытаться перепроектировать их так, чтобы обладали):

- **Идемпотентность:** Операция, будучи выполненной несколько раз, имеет тот же эффект, что и при однократном выполнении.
- **Коммутативность:** Порядок последовательности из нескольких операций различного типа или с различными аргументами не имеет значения.

Такие операции особенно важны в распределенном и асинхронном программировании, когда синхронизация может быть крайне неэффективна или просто-напросто невозможна.

Пример: Покупка в интернет-магазине. Рассмотрим простой интерфейс оформления покупки в интернет-магазине: пользователь логинится, выбирает продукт, количество, и жмет кнопку «купить». При этом на сервере вызывается следующий API:

```
class Shop {
    void buy(Request request, int productId, int quantity) {
        Session session = request.getSession();
        Order order = new Order(
            session.getCustomer(), productId, quantity);
        database.saveHistory(order);
        billing.bill(order);
    }
}
```

Представим себе ситуацию, когда закупиться хочет пользователь с нестабильным соединением с интернетом. Он нажимает кнопку «Купить», однако ответный TCP-пакет от сервера теряется и браузер в течение 5 минут показывает «Идет соединение…». Наконец, пользователю надоедает ждать и он нажимает кнопку «Купить» еще раз. На этот раз все проходит нормально, однако через неделю пользователю приходит два экземпляра товара и он с удивлением обнаруживает, что деньги с его кредитной карты также сняты два раза.

Это произошло потому, что операция `buy` не была идемпотентна — многократное ее выполнение не было равносильно однократному. Конечно, нельзя говорить об идемпотентности операции покупки товара — ведь купить телевизор два раза — не то же самое, что купить его один раз! Можно, однако, говорить об идемпотентности операции нажатия кнопки «Купить» на некоторой странице.

```
class Session {
    int stateToken;

    void advance() {
        ++stateToken;
    }
    int getStateToken() {
        return stateToken;
    }
}
class Shop {
    Page generateOrderPage(Request request) {
        Session session = request.getSession();
        session.advance();
        ...<INPUT type='hidden'
                  value='"+session.getStateToken()+"'>...
    }

    void buy(Request request, int productId, int quantity) {
        Session session = request.getSession();
```

```
            if(session.currentStateToken() !=
               request.getParamAsLong("stateToken"))
            {
                return;
            }
            Order order = new Order(
                session.getCustomer(), productId, quantity);
            database.saveHistory(order);
            billing.bill(order);

            session.advance();
        }
    }
```

Таким образом, каждая страница заказа помечается целым числом; при каждой загрузке страницы заказа или покупке это число увеличивается. Нажать кнопку «Купить» несколько раз с одной страницы нельзя — это обеспечивается тем, что после того, как покупка уже была произведена, `stateToken` у сессии увеличивается и перестает совпадать со `stateToken` в запросе.

Операция стала идемпотентной, а ситуации, подобные описанной, невозможными.

3.5. Заключение

Тот факт, что обеспечение корректности программ, использующих изменяемое состояние, особенно многопоточно — нелегкая задача, осознают почти все программисты. Программистский фольклор включает множество описанных и неописанных приемов рефакторинга таких программ в более простую и корректную форму. В этой статье автор предпринял попытку собрать вместе и описать эти приемы, а также выделить их общие идеи.

Идеи, как оказалось, сводятся к избавлению от изменяемого состояния, приведению кода в более декларативную и близкую к предметной области форму, инкапсуляции и локализации объектов с состоянием, налаганию на код определенных алгебраических свойств — все то, что знакомо функциональным программистам, ценящим возможность легко рассуждать о свойствах программ математически.

Автор выражает надежду, что данная статья послужит как полезным справочником для программистов на объектно-ориентированных и процедурных языках, так и мотивирует их погрузиться в интереснейший мир функциональных языков, откуда многие из описанных идей почерпнуты и где изменяемое состояние — скорее исключение, чем правило, благодаря чему код легко тестируем и настолько корректен, что бытует лишь наполовину шуточное утверждение «компилируется — значит работает».

Литература

[1] Effect system. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Effect_system (дата обращения: 20 июля 2009 г.).

[2] *Johnson S.* — Комментарий на форуме Lambda-the-Ultimate, URL: http://lambda-the-ultimate.org/node/724#comment-6621 (дата обращения: 20 июля 2009 г.).

[3] *Liskov B. H., Wing J. M.* Behavioural subtyping using invariants and constraints // Formal methods for distributed processing: a survey of object-oriented approaches. — New York, NY, USA: Cambridge University Press, 2001. — Pp. 254–280.

[4] *Noble J.* Arguments and results // In PLOP Proceedings. — 1997.

[5] *Okasaki C.* Purely Functional Data Structures. — Cambridge University Press, 1998.

Давно не брал я в руки шашек

Дмитрий Астапов
adept@fprog.ru

Аннотация

Функциональные языки обладают рядом полезных свойств, позволяющих ускорить прототипирование и разработку программ. В частности, функциональные языки существенно облегчают разработку программ методом «сверху вниз», позволяя программисту сосредоточиться на высокоуровневом проектировании системы до того, как он углубится в детали реализации.

Данная статья на практическом примере демонстрирует процесс проектирования и разработки «сверху вниз» программ на языке Haskell. Она рассчитана на программистов, которые уже начали знакомство с функциональным программированием и языком Haskell, но испытывают нехватку практических примеров того, как проектируются и разрабатываются программы на Haskell и функциональных языках.

4.1. Введение

Начиная реализацию нового проекта, программист зачастую уже обладает определенным набором библиотек, которые он хочет (или должен) использовать. Очень часто можно наблюдать, как работа при этом строится «снизу вверх» — отталкиваясь от предлагаемых библиотеками функций, программист пишет свой код, постепенно повышая уровень абстракции и переходя ко все более высокоуровневым компонентам системы.

В то же время, существует целый ряд причин, по которым подход к разработке в противоположном направлении — «сверху вниз» — может оказаться предпочтительнее:

Контекст всегда понятен. Проектирование высокоуровневой структуры системы дает контекст для проектирования более мелких компонентов — будь то подсистемы, классы или функции — повышая вероятность того, что они будут правильно абстрагированы, а интерфейсы между ними будут качественно построены. При проектировании «снизу вверх», напротив, легко потерять общий контекст и упустить из виду какой-то аспект решаемой задачи.

Идентификация рисков. Построение системы «от общего к частному» дает дополнительные возможности по идентификации рисков. После того, как основные подсистемы и компоненты реализованы (пусть даже в виде «пустышек»), появляется возможность оценить сложность их реализации и заранее предотвратить возможные риски, например, изменив дизайн системы.

Управление временем. Когда основные модули системы уже написаны (или спроектированы), у разработчика появляется возможность примерно оценить сложность и время, требуемое на разработку того или иного компонента. В дальнейшем разработчик, скорее всего, будет стремиться рационально распределять свои усилия и не тратить слишком много времени на углубление в какой-то один аспект системы. В то же время, при разработке «снизу вверх» велик риск потратить слишком много времени на «вылизывание» одного мелкого участка в ущерб общему качеству продукта.

Функциональные языки способствуют тому, чтобы использовать подход к проектированию «сверху вниз» в повседневной работе. Развитые возможности композиции сложных программных функций из более простых и статическая типизация с автоматическим выводом типов позволяют программисту писать код итеративно, постепенно уменьшая уровень абстракции и продвигаясь от прототипа к работающей реализации. При этом программист может опираться на компилятор или интерпретатор функционального языка как на техническое средство проверки внутренней непротиворечивости уже написанного кода.[1]

[1]Для этого язык должен быть статически типизирован, а его компилятор — обладать возможностью выведения типов выражений. Примеры подобных языков: Haskell, OCaml.

Впрочем, лучше один раз увидеть, чем сто раз услышать. Данная статья покажет на практическом примере, как выполняется дизайн и пишется «сверху вниз» программный код на функциональном языке программирования Haskell.

Предполагается, что читатель уже ознакомился с синтаксисом языка Haskell при помощи книг (например, [7] и [11]) или учебников (например, [12], [10] или [4]). Характерные приемы написания кода, использованные в примерах, будут указываться в сносках, начинающихся со слова Haskell. Предполагается, что читатель сам сможет углубленно ознакомиться с указанными темами, используя свободно доступные в сети ресурсы.

4.2. Постановка задачи и функция main

Практической задачей, которая будет решена в рамках этой статьи, будет реализация программы, позволяющей играть в шашки. Определим высокоуровневые требования таким образом: программа должна позволять играть в шашки (русские или международные) двум игрокам, причем в роли каждого игрока может быть человек или «искусственный интеллект».

Таким образом, программа должна сначала определять конфигурацию партии (при помощи файла с настройками, через пользовательский интерфейс или как-то иначе), затем проводить партию и по ее завершении прекращать работу.

Соответствующий код на Haskell выглядит так[2]:

```
main = do
  (newBoard, playerA, playerB) ← getConfig
  play newBoard playerA playerB
```

Поскольку устройство функций `getConfig` и `play` пока еще не рассматривается, их определение временно будет состоять из вызова функции **`undefined`**:

```
getConfig = undefined
play = undefined
```

Вызов функции **`undefined`** приводит к генерации ошибки во время исполнения программы и выводу сообщения «Prelude: undefined». Однако у этой функции есть одно примечательное свойство, позволяющее использовать ее в качестве универсального маркера «TODO» при написании кода: компилятор считает, что тип результата этой функции — это самый общий, самый глобальный тип, который только может быть. Все остальные типы, с точки зрения компилятора, являются подтипами этого общего типа. Некоторой аналогией подобной концепции может служить тип `Object` в языке Java или `void *` в языке C.[3]

На практике это означает, что программист может осуществлять проектирование «сверху вниз» планомерно, не углубляясь в детали реализации вводимых функций

[2] Haskell: в функции `main` используется «синтаксический сахар» под названием «**do**-нотация» для упрощения записи операций ввода/вывода. См. [3] и [7, глава 7].

[3] Детальное описание этого примечательного факта см. в [1].

сразу же после первого их упоминания, и при этом поддерживать код в компилируемом состоянии. Да, запуск подобной программы будет невозможен, но в ходе компиляции будет выполнена проверка типов в уже написанном коде, и программист будет уведомлен об обнаруженных ошибках. Поскольку в случае Haskell успешно скомпилированный код с большой вероятностью будет корректно работать, это очень ценный результат.

Безусловно, в программах на Java/C/C++ тоже можно описывать методы или функции с пустым телом для достижения подобной цели. Однако при этом программисту необходимо будет сразу указать полную сигнатуру метода или функции: количество и тип аргументов, тип возвращаемого результата. Если же в ходе проектирования высокоуровневые функции или методы будут изменены, программисту с большой вероятностью потребуется изменить сигнатуры многих пустых методов, прежде чем код сможет снова быть скомпилирован. Charles Petzold[4] считает [9], что подобные особенности императивных языков, вкупе с технологиями интеллектуального дополнения кода (Intellisense), опирающимися на информацию о типах объявленных функций и методов, приводят к практической невозможности заниматься разработкой «сверху вниз», вынуждая программиста работать «снизу вверх».

Чтобы убедиться, что написанный код на Haskell действительно компилируется, его можно сохранить в файл (допустим, «checkers01.hs») и загрузить в интерпретатор Haskell[5]:

```
% ghci checkers01.hs
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( checkers01.hs, interpreted )
Ok, modules loaded: Main.
```

4.3. Формализация правил игры

Поскольку способ настройки программы и конкретный вид настроек сильно зависят от деталей реализации игры, разумно временно забыть про функцию `getConfig` и сосредоточить внимание на функции `play`, отвечающей за проведение одной игровой партии.

Как известно, партия в шашки начинается с расстановки фигур, после чего стороны поочередно делают ход до тех пор, пока один из игроков не устранит с доски все фигуры противника, либо пока игра не завершится ничьей. Чтобы упростить код

[4] Автор множества книг по программированию под Windows.

[5] В статье демонстрируются примеры с использованием интерпретатора `ghci` под OS Linux, но с таким же успехом можно использовать и любой другой интерпретатор — например, `hugs` — в привычной операционной среде.

и описание, будем считать, что партия может завершиться только победой одной из сторон.

Данное описание фактически дословно переводится в код на Haskell[6]:

```
play newBoard playerA playerB = do
  displayBoard newBoard
  makeMove newBoard White
  where
    makeMove board side = do
      nextBoard ← undefined
      displayBoard nextBoard
      if isVictory side nextBoard
        then return ("Winner is " ++ show side)
        else makeMove nextBoard (otherSide side)

    isVictory = undefined

data Side = White | Black deriving Show
otherSide White = Black
otherSide Black = White

displayBoard = undefined
```

Разумно предположить, что сведения об игроке/цвете фигур потребуются не только в этой части кода — например, для описания расположения фигур на доске. До проектирования этих частей системы еще далеко, но уже сейчас можно ввести отдельный тип данных `Side` для кодирования сведений об игроке.

Новое расширенное определение функции `play` включает в себя две «заглушки»: функцию `isVictory`, используемую для определения конца партии, и код генерации нового состояния доски `newBoard` на основании хода игрока. Поскольку правила игры в шашки жестко регламентируют приоритет и очередность всех действий игрока в течении хода, их можно (и нужно) запрограммировать.

Во-первых, если игрок имеет возможность взять фигуру противника, он обязан это сделать, в противном случае он может сделать ход любой (имеющей на то возможность) фигурой. Если игрок может взять несколько фигур подряд — он также обязан это сделать. Для упрощения кода и изложения будем считать, что шашки ходят и берут исключительно вперед (не назад)[7], и игрок не обязан делать ход именно той фигурой, которая способна взять максимальное количество фигур противника.

---

[6] Haskell: функция `makeMove` производит обновление состояния доски без использования переменных — новое состояние передается в рекурсивный вызов функции `makeMove`.

Объявление **`deriving Show`** просит компилятор автоматически реализовать для указанного типа данных функцию **`show`**, используемую при объявлении победителя.

Так как функция `displayBoard` производит вывод на экран, для записи функции `makeMove` также используется **`do`**-нотация.

[7] Подобные соглашения приняты, в частности, в английском варианте игры в шашки.

Если по окончании хода шашка игрока оказалась на самом дальнем от него ряду игрового поля, она превращается в «дамку»[8] и получает возможность ходить (и бить) на произвольное количество полей по диагонали как вперед, так и назад. Расширенное определение локальной функции `makeMove`, реализующей эти правила, выглядит так:

```
makeMove board side = do
  afterMove ← if canAttack board side
                then undefined -- do attack
                else undefined -- do move
  let nextBoard = upgradeToKings afterMove
  displayBoard nextBoard
  if isVictory side nextBoard
    then ...

upgradeToKings = undefined
canAttack      = undefined
```

Выбор фигуры, которая будет ходить или выполнять взятие, должен выполняться функцией `playerA` (или `playerB`, в зависимости от того, чей ход). В качестве результата эти функции должны возвращать обновленное состояние доски после завершения хода.

Очевидно, что для принятия правильного решения эти функции должны получить, как минимум, информацию о:

1) текущем состоянии доски (фигур);

2) цвете фигур, которыми играет игрок;

3) том, какого рода ход надо сделать — простой или со взятием.

Для кодирования типа хода будет введен тип данных `MoveType`:

```
data MoveType = Move | Attack
```

Теперь пробелы в определении функции `makeMove` могут быть заполнены таким образом[9]:

```
makeMove board side = do
  let player = getPlayer side
  afterMove ← if canAttack board side
                then attackLoop player board side
                else player Move board side
  ...
getPlayer = undefined
```

[8] По-английски «дамка» называется «king», и именно так она будет именоваться в коде программы.

[9] Haskell: этот пример хорошо иллюстрирует отсутствие какого-либо особого статуса у функций по сравнению со всеми прочими типами данных. Видно, что `player` — это результат работы функции `getPlayer`. При этом `player` используется в выражении `player Move board side`. То есть, `player` — это функция; как функция, она может быть возвращена в качестве результата работы других функций. В то же время, `player` передается в качестве аргумента в функцию `attackLoop`.

Функция `getPlayer`, выбирающая правильную функцию-реализацию поведения игрока в зависимости от того, чей сейчас ход, выглядит при этом так[10]:

```
getPlayer White = playerA
getPlayer Black = playerB
```

Осталось описать вспомогательную функцию `attackLoop`, реализующую взятие фигур до тех пор, пока это возможно:

```
attackLoop player board side = do
  board' ← player Attack board side
  if canAttack board' side
    then attackLoop player board' side
    else return board'
```

Полная версия кода, получившегося в результате, приведена на рисунке 4.1. Она также находится в файле «checkers02.hs», расположенном в архиве с примерами кода.

4.4. Компьютерный игрок с примитивной стратегией

Очевидно, что код программы должен включать в себя некоторый API для доступа к состоянию доски и его модификации. Этим API будут пользоваться, к примеру, функции, реализующие поведение компьютерных игроков.

В рамках подхода к проектированию «сверху вниз» это означает, что сначала будет написан код, реализующий поведение компьютерных игроков, а в процессе его написания будут выработаны требования к тому, какие функции должны входить в это API, и определена их функциональность.

Простейший компьютерный игрок — это игрок со случайным поведением. На каждом ходе он выбирает один из доступных ходов, без всякой стратегии, и делает его. Чтобы иметь возможность выбрать случайный ход из списка доступных, надо каким-то образом этот список получить. В принципе, возможность получить список допустимых ходов и взятий черезвычайно полезна для реализации как компьютерных игроков, так и интерфейса с игроком-человеком.

Таким образом, можно сформулировать следующую подзадачу: необходимо реализовать функции, возвращающие список доступных ходов и взятий, а потом на их основе построить реализацию компьютерного игрока. Естественно, что поведение этих функций (назовем их `availableMoves` и `availableAttacks` соответственно) должно зависеть от текущего состояния доски и того, какая сторона делает ход.

Код, реализующий выбор хода компьютерным игроком, может быть записан с использованием функции `availableMoves` так[11]:

[10] Haskell: поскольку функция `getPlayer` определена локально в where-блоке функции `play`, ей непосредственно доступны все аргументы функции `play` и их не нужно явно передавать при вызове `getPlayer`.

[11] Haskell: функция `$` может использоваться для уменьшения количества круглых скобок в коде. `(print (process (parse (read x))))` — то же самое, что и

```
module Main where

main = do
  (newBoard, playerA, playerB) ← getConfig
  play newBoard playerA playerB

play newBoard playerA playerB = do
  let board = newBoard
  displayBoard board
  makeMove board White
  where
    makeMove board side = do
      let player = getPlayer side
      afterMove ← if canAttack board side
                    then attackLoop player board side
                    else player Move board side
      let nextBoard = upgradeToKings afterMove side
      displayBoard nextBoard
      if isVictory side nextBoard
        then return ("Winner is " ++ show side)
        else makeMove nextBoard (otherSide side)

    attackLoop player board side = do
      board' ← player Attack board side
      if canAttack board' side
        then attackLoop player board' side
        else return board'

    getPlayer White = playerA
    getPlayer Black = playerB

    isVictory  = undefined
    canAttack  = undefined

data Side = White | Black deriving Show
otherSide White = Black
otherSide Black = White

data MoveType = Move | Attack

getConfig      = undefined
displayBoard   = undefined
upgradeToKings = undefined
```

Рис. 4.1. Листинг checkers02.hs

```
randomComputer Move board side =
  case availableMoves board side of
    []       → return board
    variants → do v ← getRandom variants
                  return $ move board side v
```

Тут функция `move` — это (еще не реализованная) функция из API работы с доской, выполняющая манипуляции по перемещению фигуры в результате хода.

По аналогии, функция выбора атакующего кода будет записана так:

```
randomComputer Attack board side = do
  v ← getRandom (availableAttacks board side)
  return $ attack board side v
```

Компьютерный игрок будет выполнять выбор атакующего кода только в том случае, если функция `play` проверила потенциальную возможность выполнения взятия этим игроком. Соответственно, нет необходимости проверять, вернула ли функция `availableAttacks` хотя бы один вариант хода — его наличие гарантируется. С другой стороны, функция `availableMoves` вполне может возвратить пустое множество доступных ходов — в этом случае компьютерный игрок пропускает ход (возвращает состояние доски без изменений).

Несмотря на довольно большой список нереализованных функций (`isVictory`, `getConfig`, `displayBoard`, `canAttack`, `upgradeToKings`, `availableMoves`, `availableAttacks`, `move`, `attack`, `getRandom`), написанный код компилируется — то есть он синтаксически корректен.

Читатели, самостоятельно пишущие код в процессе чтения статьи, могут сравнить свой вариант с файлом «checkers03.hs», расположенным в архиве с примерами кода.

4.5. Сбор информации о доступных ходах со взятием

Чтобы выяснить, какие еще функции должны входить в API работы с состоянием доски, приступим к реализации функции `availableAttacks`, которой потребуется всесторонне анализировать информацию о расстановке фигур обоих игроков.

Как известно, шашки могут «бить» вражеские фигуры, перепрыгивая через них на следующую по диагонали клетку, если она свободна. При этом шашка может двигаться только вперед. Соответственно, перебрав все шашки игрока и проверив для каждой из них возможность атаковать по обоим диагональным направлениям, можно собрать информацию о всех доступных игроку атакующих ходах.

Диагонали, проходящие через клетку, на которой находится шашка, можно задать, изменяя номер ряда и столбца клетки одновременно на единицу (с нужным знаком). В принципе, уже можно приступать к написанию кода `availableAttacks`[12]:

(**print** $ process $ parse $ **read** x).

[12]Haskell: Из определения функции `checkDiagonals` следует, что она принимает три аргумента. В то же время, в строке **`concatMap`** `(checkDiagonals board side)` эта функция применена к двум аргу-

```
availableAttacks board side =
  concatMap (checkDiagonals board side) (pieces side board)
  where
    checkDiagonals board side (piece,coords) =
      if isChecker piece
        then checkerAttack coords forward left
             ++ checkerAttack coords forward right
        else kingAttack coords forward left
             ++ kingAttack coords forward right
             ++ kingAttack coords (-forward) left
             ++ kingAttack coords (-forward) right

    (forward,left,right) =
      if side == White then (-1, -1, 1) else (1,1,-1)

    checkerAttack coords deltaRow deltaColumn
      | hasTargetAndEmptySquareBehind squares = undefined
      | otherwise                             = []
      where
        squares = diagonal board coords deltaRow deltaColumn

    kingAttack coords deltaRow deltaColumn
      | emptySqrsThenTargetAndEmptySqrBehind squares = undefined
      | otherwise                                    = []
      where
        squares = diagonal board coords deltaRow deltaColumn

isChecker = undefined
diagonal  = undefined
hasTargetAndEmptySquareBehind = undefined
emptySqrsThenTargetAndEmptySqrBehind = undefined
```

Предполагается, что функция `diagonal` возвращает список координат клеток, лежащих по диагонали от клетки с координатами `coords` в направлении, указанном при помощи `(deltaRow, deltaColumn)`.

Функция `availableAttacks`, даже не будучи реализованной в полном объеме, получается достаточно громоздкой. Кроме того, если задуматься о реализации функции `availableMoves`, то станет понятно, что ее код будет практически полностью повторять код функции `availableAttacks`. Отличие будет заключаться в том, что вместо `hasTargetAndEmptySquareBehind` будет использована другая функция,

---

ментам. Результатом такого частичного применения (partial application, см. [8]) является функция одного аргумента, принимающая на вход пару `(piece,coords)`. Именно такая функция требуется для обработки списка фигур с помощью функции **`concatMap`**.

В определении функций `checkerAttack` и `kingAttack` использованы охранные выражения (guards, см. [2]), позволяющие выбирать тот или иной вариант поведения в зависимости от истинности указанных условий.

проверяющая наличие перед шашкой пустой клетки, на которую можно будет сделать ход.

Аналогичное изменение будет сделано для обработки «дамок», но весь остальной код, осуществляющий последовательный перебор фигур, объединение результатов работы `checkDiagonals`, генерацию списков клеток, лежащих на диагонали и т. п., останется без изменений.

Разумно вынести общий алгоритм работы `availableMoves` и `availableAttacks` в отдельную функцию `collectOpportunities`. Эта функция будет заниматься сбором информации о «возможностях», доступных игроку в текущем игровом состоянии, где под «возможностями» понимаются ходы со взятием или без него. В качестве аргумента ей будет передаваться функция, проверяющая, подходит ли данное диагональное направление для того, чтобы осуществить данной фигурой ход нужного типа (со взятием или без). Кроме того, аргументом `collectOpportunities` должна быть еще одна функция, которая сформирует данные о «возможности» (возможном ходе) и вернет их в качестве результата работы `checkerAttack` или `kingAttack`.

4.6. Сбор информации о доступных ходах: вторая попытка

В процессе написания функции `collectOpportunities` можно сделать еще несколько упрощений имеющегося кода.

Поскольку, согласно оговоренным правилам, шашки ходят и бьют только вперед-влево и вперед-вправо, а «дамки» — еще и назад-влево и назад-вправо, можно создать функцию `validDirections`, которая будет возвращать список допустимых направления движения для указанной фигуры.

Кроме того, можно объединить функции `checkerAttack` и `kingAttack` в одну, вынеся функциональность, реализующую разное поведение фигуры в зависимости от ее типа за пределы `collectOpportunities`.

Теперь сама функция `collectOpportunities` может быть записана следующим образом:

```
collectOpportunities board side canAct describeAction =
  concatMap checkDiagonals (pieces side board)
  where
    checkDiagonals (coords,piece) =
      concatMap (checkDirection coords piece) (validDirections piece)

    checkDirection coords piece direction
      | canAct piece squares = describeAction piece coords squares
      | otherwise            = []
      where squares = diagonal coords direction
```

```
      validDirections = undefined
      diagonal = undefined

pieces = undefined
```

Тут функция `checkDirection` выполняет проверку хода фигуры `piece` в направлении `direction`. Если набор клеток на нужной диагонали будет признан подходящим функцией `canAct`, то будет вызвана функция `describeAction` для генерации информации о возможном ходе фигуры в этом направлении.

Функция `checkDiagonals` выполняет обработку всех возможных направлений движения одной фигуры, а функция `collectOpportunities` просто применяет `checkDiagonals` ко всем фигурам конкретного игрока. Реализация функции `pieces`, возвращающей список фигур, отложена на будущее.

Непосредственная работа с направлениями будет реализована в функциях `validDirections` и `diagonal`, которые, скорее всего, будут содержать код, похожий на использованный в первом варианте `availableAttacks`. Но так как функция `collectOpportunities` получилась более высокоуровневой, можно сейчас не углубляться в детали их реализации.

Теперь можно свести функции `availableAttacks` и `availableMoves` к вызову `collectOpportunities` с правильными аргументами:

```
availableAttacks board side =
  collectOpportunities board side canTake genAttackInfo
  where
    canTake       = undefined
    genAttackInfo = undefined

availableMoves board side =
  collectOpportunities board side canMove genMoveInfo
  where
    canMove     = undefined
    genMoveInfo = undefined
```

Получившийся код находится в файле «checkers04.hs» в архиве с примерами кода. На рисунке 4.2 можно видеть иерархию функций, описанных в этом файле, пунктиром обозначены еще не реализованные функции.

4.7. Передвижение фигур

К настоящему моменту в коде насчитывается пятнадцать функций-«заглушек», определенных как **`undefined`**. Несмотря на это, компилятор уже может делать определенные выводы о том, какие значения будут принимать и возвращать описанные функции. Если загрузить код в интерпретатор Haskell и поинтересоваться, какой тип был автоматически определен для функции `availableAttacks`, можно получить вполне конкретный ответ:

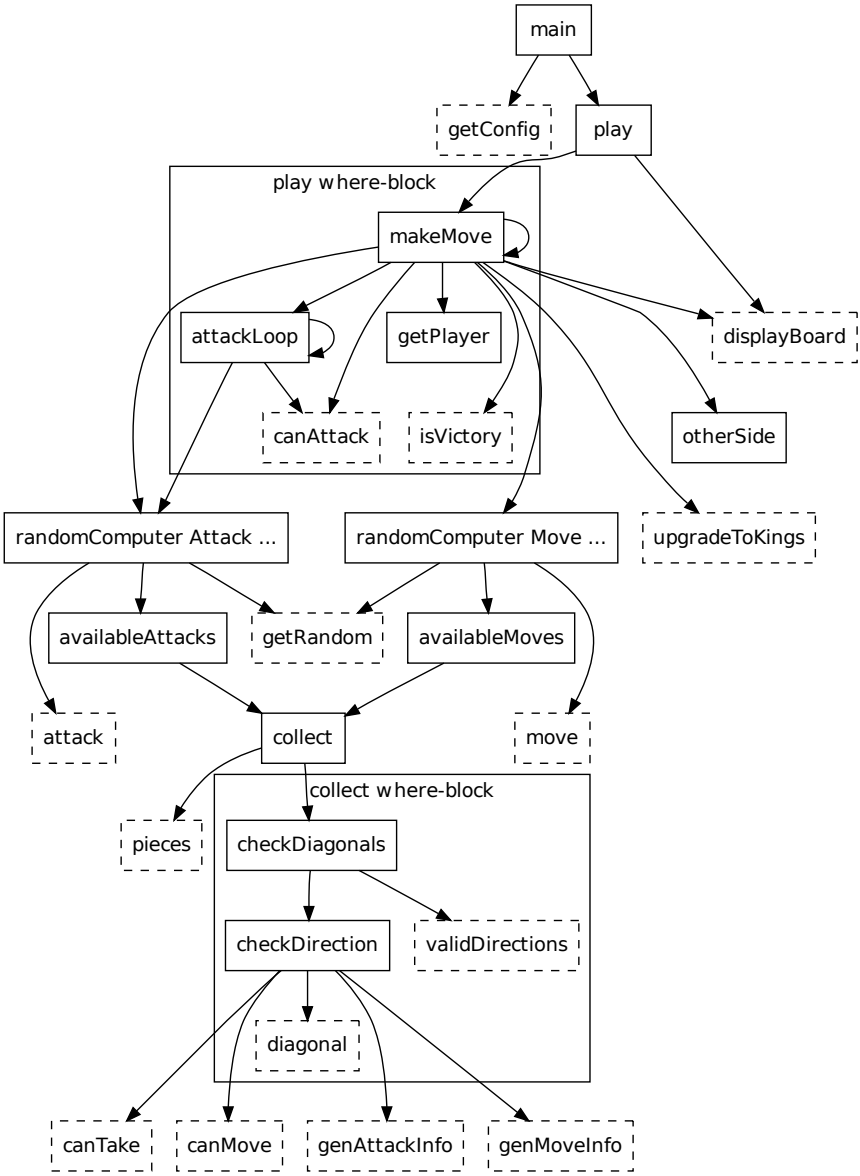


Рис. 4.2. Дерево вызовов функций в файле checkers04.hs

```
*Main> :load checkers04.hs
[1 of 1] Compiling Main             ( checkers04.hs, interpreted )
Ok, modules loaded: Main.

*Main> :type availableAttacks
availableAttacks :: t -> t1 -> [alpha]
```

Хотя компилятор еще не знает, какие конкретно типы данных будут использованы для описания игровой доски и допустимых ходов, он уже может сделать вывод о том, что функция `availableAttacks` будет возвращать список значений.

Список будет пуст, если допустимых ходов со взятием нет. Зная это, можно реализовать одну из функций-«заглушек»:

```
canAttack board side = not $ null $ availableAttacks board side
```

Теперь самое время углубится в детали, которые помогут связать друг с другом разные части кода. По дизайну, компьютерный игрок `randomComputer` будет выбирать один из элементов списка, возвращаемого функциями `availableAttacks` или `availableMoves`, и без изменений передавать его в функции `attack` или `move`.

Какого же типа должны быть эти значения, и какие данные они должны содержать? Чтобы функция `move` могла обновить информацию о состоянии доски, ей потребуются сведения о том:

1) что за фигура (шашка или «дамка») ходит;

2) с какой клетки;

3) на какую клетку.

Для функции `attack` перечень будет таким же, плюс дополнительно потребуются сведения о том, на какой клетке находится «битая» фигура. Теперь можно описать соответствующий тип данных и уточнить реализацию функций `attack` и `move`[13]:

```
data MoveInfo = MoveInfo { piece    :: Piece
                         , from     :: Coords
                         , to       :: Coords
                         }
              | AttackInfo { piece  :: Piece
                           , from   :: Coords
                           , victim :: Coords
                           , to     :: Coords
                           }
```

[13]Haskell: описание типа `MoveInfo` выполнено с использованием record syntax, чтобы компилятор автоматически генерировал функции `piece :: MoveInfo → Coords`, `from :: MoveInfo → Coords` и т. п. для доступа к компонентам данных.

Имя типа данных `MoveInfo` совпадает с именем одного из его конструкторов. Это довольно распространенная практика. При употреблении `MoveInfo` компилятор (и программист) из контекста в любой момент может легко понять, что именно имеется в виду.

Детали реализации типов `Piece` и `Coords` пока обсуждать рано, можно временно описать их как «типы без данных» (аналог трюка с **`undefined`** для описания типов данных):

```
data Piece = Piece
data Coords = Coords
```

Модификация доски при ходе фигуры `piece` с клетки `from` на клетку `to` заключается в том, чтобы убрать фигуру указанного цвета со старых координат и разместить ее на новых:

```
move board side (MoveInfo piece from to) =
  replace (from,piece) (to,piece) side board

replace = undefined
```

При ходе со взятием фигура `piece` выполняет передвижение с клетки `from` на клетку `to`, перепрыгивая через «жертву», стоящую на клетке `victim`. По окончании хода «жертва» убирается с доски:

```
attack board side (AttackInfo piece from victim to) =
  remove victim (otherSide side) $ replace (from,piece) (to,piece) side board

remove = undefined
```

Получившийся код находится в файле «checkers05.hs» в архиве с примерами кода.

4.8. Доска, фигуры, координаты

Функции `replace` и `remove` будут работать со списком фигур, хранящимся в описании игрового поля. Функция `pieces` для получения текущего значения этого списка уже упоминалась при описании функции `collectOpportunities` (см. 4.6). Если предположить, что для обновления информации о списке фигур имеется парная функция `setPieces`, то можно реализовать функции `replace` и `remove` с помощью стандартных операций работы со списками:

```
replace (from,piece) (to,piece') side board =
  let pcs = pieces side board
      pcs' = (to,piece'):(filter (≠(from,piece)) pcs)
      in setPieces side board pcs'

remove coords side board =
  let pcs = pieces side board
      pcs' = filter ((≠coords).getCoords) pcs
      in setPieces side board pcs'
```

В принципе, для описания типа данных, хранящего информацию о состоянии доски, уже все готово. Информация о фигурах может храниться либо в виде списка троек (координата, фигура, цвет), либо в виде двух отдельных списков пар (координаты, фигура): в одном списке будет информация о черных фигурах, во втором — о белых. Вариант с двумя списками лучше соответствует уже написанному коду — до сих пор везде происходила обработка фигур одного конкретного цвета, и все написанные функции рассчитывают, что функция `pieces` будет возвращать именно список пар (координаты, фигура). А при использовании одного списка для хранения всех фигур функции `pieces` придется при каждом вызове отфильтровывать из него информацию о фигурах ненужного цвета.

Осталось решить вопрос с тем, как хранить информацию о координатах клеток доски. Самый простой способ — пронумеровать строки и столбцы доски, и хранить координаты в виде пар (строка, столбец):

```
type Coords = (Int,Int)
```

Тип данных, описывающий фигуру — это простое перечисление (набор констант) из двух значений: «шашка» и «дамка»:

```
data Piece = Checker | King
```

Теперь описание доски будет выглядеть так[14]:

```
data Board = Board { height :: Int
                   , width  :: Int
                   , whites :: [(Coords,Piece)]
                   , blacks :: [(Coords,Piece)]
                   }
```

Для удобства работы с компонентами типа `Board` можно определить вспомогательные функции `pieces`, `setPieces` и `getCoords`:

```
pieces White = whites
pieces Black = blacks
setPieces White board ws = board { whites=ws }
setPieces Black board bs = board { blacks=bs }

getCoords (coords,p) = coords
```

Получившийся код находится в файле «checkers06.hs» в архиве с примерами кода.

4.9. Выход в «дамки» и отображение состояния доски

Теперь можно реализовать часть функций-«заглушек», которым требуется доступ к деталям реализации состояния о доске. Например, превращение шашек в «дамки»

[14]Автору неизвестны варианты игры в шашки на прямоугольных досках, но тем не менее возможность создания прямоугольной доски была оставлена.

по окончании хода. Необходимо проверить, нет ли на доске шашек, стоящих на самом дальнем от игрока ряду. Для белых это ряд номер один, для черных — ряд с номером, равным высоте доски. Если такие шашки найдены — они меняются на «дамки» при помощи функции `replace`[15]:

```
upgradeToKings board side = newBoard
  where
    kingRow White = 1
    kingRow Black = height board
    newBoard =
          in case filter (upgradeableChecker (kingRow side)) pcs of
             (coords,Checker):_ →
               replace (coords,Checker) (coords,King) side board
             _ →
               board
    upgradeableChecker targetRow ((row,col),Checker) = row == targetRow
    upgradeableChecker _ _ = False
```

Еще одна функция, реализация которой требует информации о ширине и высоте доски — это `displayBoard`. Доска будет отображаться алфавитно-цифровыми символами, в стиле фильмов о компьютерах конца прошлого века. Чтобы таким образом отобразить всю доску, надо отобразить каждый её ряд на отдельной строке. Чтобы отобразить ряд клеток доски, надо разобраться, занята ли клетка фигурой, и вывести либо обозначение фигуры, либо какой-то символ, обозначающий цвет этой клетки[16]:

```
displayBoard board = putStrLn (boardToString board)
  where
    boardToString board = unlines rows
    rows = [ row r | r ← [1..height board ] ]
    row r = [ square r c | c ← [1..width board] ]
    square r c = case lookup (r,c) allPieces of
                   Just (Checker,White) → 'w'
                   Just (King,White)    → 'W'
                   Just (Checker,Black) → 'b'
                   Just (King,Black)    → 'B'
                   Nothing → if isBlackSquare r c
                 then '.'
                 else ' '
    allPieces =
      [ (coords,(piece,side)) | side ← [ Black, White ]
                              , (coords,piece) ← pieces side board ]
```

[15]Haskell: В выражении **case** использован шаблон _, сопоставляемый с произвольным выражением; таким образом реализуется семантика «для всех прочих выражений — результат такой-то». Подобный прием можно видеть и в определении функции `upgradeableChecker`.

[16]Haskell: В определениях `rows`, `row` и `allPieces` используется «синтаксический сахар» для определения списков, позволяющий обойтись без циклов `for` (см. [5], раздел «list comprehensions»).

```
isBlackSquare r c = odd (r+c)
```

Получившийся код находится в файле «checkers08.hs» в архиве с примерами кода. Загрузив его в интерпретатор Haskell, можно убедиться, что функция `displayBoard` может отобразить доску без расставленных фигур:

```
Prelude> :l ./checkers08.hs
[1 of 1] Compiling Main             ( checkers08.hs, interpreted )
Ok, modules loaded: Main.
*Main> displayBoard (Board 8 8 [] [])
 . . . .
. . . .
 . . . .
. . . .
 . . . .
. . . .
 . . . .
. . . .

*Main>
```

4.10. Создание доски

Прежде чем будет возможно проверить работу `displayBoard` для доски с расставленными фигурами, необходимо реализовать функцию создания описания доски с расставленными фигурами. Точнее, потребуется несколько функций, ведь по требованиям программа должна реализовывать игру как в русские, так и в международные шашки.

Варианты игры в шашки отличаются как размером доски, так и количеством шашек у каждого игрока. Доска для классических (русских) шашек — размером 8×8, у каждого игрока по 12 фигур:

```
checkers8x8 = newBoard 8 8 12
```

Доска для международных шашек — 10×10, у каждого игрока по 20 фигур:

```
checkers10x10 = newBoard 10 10 20
```

Фигуры расставляются на черных клетках доски. Если иметь упорядоченный в порядке возрастания рядов список координат черных клеток, то можно сказать, что черные фигуры занимают 12 (или 20) первых клеток из этого списка, а белые — такое же количество последних[17]:

[17]Haskell: определение `coords` использует «конструктор списка» (list comprehension, см. [5]) с предикатом, при этом в результат попадают только пары `(row,column)`, удовлетворяющие условию `isBlackSquare`.

Хотя функция **`repeat`** и генерирует бесконечный список значений, результат работы **`zip`** будет ограничен длиной более короткого аргумента, и будет иметь длину `numPieces`.

```
newBoard h w numPieces = Board h w whitePieces blackPieces
  where
    coords = [ (row,column) | row ← [1..h], column ← [1..w]
                            , isBlackSquare row column ]
    blackPieces = zip (take numPieces coords)           (repeat Checker)
    whitePieces = zip (take numPieces (reverse coords)) (repeat Checker)
```

Для упрощения реализации конфигурация программы будет описываться прямо в коде. Читатель может реализовать ввод конфигурации с клавиатуры или из внешнего файла в качестве самостоятельного упражнения:

```
getConfig = return (checkers8x8, randomComputer, randomComputer)
```

Получившийся код (с некоторыми сокращениями) приведен на рисунке 4.3. Полный текст находится в файле «checkers09.hs» в архиве с примерами кода.

Теперь можно убедиться, что функция `displayBoard` корректно работает для доски с расставленными фигурами:

```
[1 of 1] Compiling Main             ( checkers09.hs, interpreted )
Ok, modules loaded: Main.
*Main> displayBoard ( checkers8x8 )
 b b b b
b b b b
 b b b b
. . . .
 . . . .
w w w w
 w w w w
w w w w

*Main>
```

4.11. Диагонали

Судя по всему, дизайн решения можно считать завершенным. Для того чтобы программа заработала, необходимо всего лишь заменить оставшиеся **`undefined`** подходящими функциями.

Так, чтобы завершить описание функции `collectOpportunities`, определим функции `validDirections` и `diagonal`.

Как уже говорилось ранее (см. 4.5), функция `diagonal` должна возвращать список клеток, лежащих по диагонали от данной в указанном направлении. Направление можно задавать, указывая, как именно изменяются номера строк и столбцов клеток, лежащих на диагонали — они либо уменьшаются, либо увеличиваются на единицу для каждой следующей клетки. Другими словами, диагональ, начинающаяся в клетке с координатами `(row,column)` и идущая в направлении

```
module Main where

main = do
  (newBoard, playerA, playerB) ← getConfig
  play newBoard playerA playerB

play newBoard playerA playerB = ...

data Side = White | Black
otherSide White = Black
otherSide Black = White

data MoveType = Move | Attack

randomComputer Move board side = ...

randomComputer Attack board side = ...

availableAttacks board side =
  collectOpportunities board side canTake genAttackInfo
  where
    canTake       = undefined
    genAttackInfo = undefined

availableMoves board side =
  collectOpportunities board side canMove genMoveInfo
  where
    canMove     = undefined
    genMoveInfo = undefined

collectOpportunities board side canAct describeAction =
  concatMap checkDiagonals (pieces side board)
  where
    checkDiagonals (coords,piece) =
      concatMap (checkDirection coords piece) (validDirections piece)

    checkDirection coords piece direction
      | canAct piece squares = describeAction piece coords squares
      | otherwise            = []
      where squares = diagonal coords direction

    validDirections = undefined
    diagonal = undefined

move board side (MoveInfo piece from to) =
  replace (from,piece) (to,piece) side board

attack board side (AttackInfo piece from victim to) =
  remove victim (otherSide side) $ replace (from,piece) (to,piece) side board

data MoveInfo = MoveInfo { piece::Piece, from::Coords, to::Coords }
              | AttackInfo { piece::Piece, from::Coords, victim::Coords, to::Coords}

replace (from,piece) (to,piece') side board =
  let pcs = pieces side board
      pcs' = (to,piece'):(filter (≠(from,piece)) pcs)
      in setPieces side board pcs'

remove coords side board =
  let pcs = pieces side board
      pcs' = filter ((≠coords).getCoords) pcs
      in setPieces side board pcs'

data Piece = Checker | King deriving Eq

type Coords = (Int,Int)
data Board = Board { height :: Int
                   , width  :: Int
                   , whites :: [(Coords,Piece)]
                   , blacks :: [(Coords,Piece)]
                   }

pieces White = whites
pieces Black = blacks
setPieces White board ws = board { whites=ws }
setPieces Black board bs = board { blacks=bs }

getCoords (coords,p) = coords

upgradeToKings board side = ...

displayBoard board = ....

isBlackSquare r c = odd (r+c)

getConfig = return (checkers8x8, randomComputer, randomComputer)

checkers8x8   = newBoard  8  8 12

checkers10x10 = newBoard 10 10 20

newBoard h w numPieces = Board h w whitePieces blackPieces
  where
    coords = [ (row,column) | row ← [1..h], column ← [1..w]
                            , isBlackSquare row column ]
    blackPieces = zip (take numPieces coords)           (repeat Checker)
    whitePieces = zip (take numPieces (reverse coords)) (repeat Checker)

getRandom = undefined
```

Рис. 4.3. Листинг checkers09.hs

`(dr, dc)` — это последовательность клеток с координатами `(row+dr,column+dc)`, `(row+dr+dr,column+dc+dc)`, и так далее, не выходящая за пределы доски[18]:

```
diagonal (row,column) (dr,dc) =
  [ (r,c) | (r,c) ← zip rows columns
          , inside board r c ]
  where
    rows    = take (height board) [row+dr,row+dr+dr..]
    columns = take (width board) [column+dc,column+dc+dc..]
    inside board r c =
      r ≥ 1 && c ≥ 1
      && r ≤ height board && c ≤ width board
```

Диагональные направления, которые будут обрабатываться при помощи функции `diagonal`, генерируются функцией `validDirections`. Теперь уже видно, что функция `validDirections` должна просто возвращать список вида `[(−1,1),(−1,−1),...]`, в зависимости от того, о какой фигуре идет речь, и какая сторона (черные или белые) делает ход.

Допустимые направления движения и взятия для шашки — вперед-влево или вперед-вправо:

```
validDirections Checker = [ (forward,left), (forward,right) ]
```

Для «дамки» добавляются еще и аналогичные движения назад-влево и назад-вправо:

```
validDirections King = [ (forward,left), (forward,right)
                       , (backward, left), (backward, right) ]
```

Конкретные значения `forward`, `backward`, `left` и `right` изменяются в зависимости от того, с какой стороны доски смотреть на игровую ситуацию:

```
(forward,backward,left,right) =
   if side == White
     then (−1, 1, −1, 1)
     else ( 1,−1,  1,−1)
```

Получившийся код находится в файле «checkers10.hs» в архиве с примерами кода.

4.12. Реализация *availableAttacks* и *availableMoves*

Было бы неплохо проверить работоспособность функции `collectOpportunities` в интерпретаторе Haskell так же, как это дела-

[18]Haskell: определения `rows` и `columns` используют «синтаксический сахар» `(..)` для генерации координат диагонали. Получившиеся списки — бесконечные. Чтобы гарантировать завершение функции `diagonal`, списки ограничиваются при помощи функции **`take`**.

В генераторах списков (list comprehensions, см. [5]) возможно использование не только явно заданных списков, но и результатов применения функций. Например, пары координат в определении `diagonal` берутся из результата работы функции **`zip`**.

лось для функции `displayBoard`. Однако в своем нынешнем виде функция `collectOpportunities` неработоспособна — для генерации результата она вызывает функции, передаваемые из `availableAttacks` и `availableMoves` в качестве аргументов `canAct` и `describeAction`. Необходимо завершить реализацию функций `availableAttacks` и `availableMoves`, чтобы в `collectOpportunities` передавались не «заглушки» **`undefined`**, а нечто более осмысленное.

Начать можно с функций, передаваемых в качестве параметра `canAct` — это функции `canMove` и `canTake`. Правила ходов (со взятием и без) описываются в терминах свободных и занятых клеток доски. Предположив, что для проверки занятости конкретной клетки реализованы функции `empty` и `hasPiece`, можно приступать к написанию кода.

Шашка может сделать ход только в том случае, если непосредственно перед ней есть пустая клетка:

```
canMove Checker (inFront:_) = empty board inFront
```

«Дамка» может сделать ход, если по диагонали с ней соседствуют одна или более пустых клеток:

```
canMove King diagonal = not $ null $ squaresToObstacle diagonal

squaresToObstacle diagonal = takeWhile (empty board) diagonal
```

Во всех прочих случаях ход фигуры невозможен[19]:

```
canMove _ _ = False
```

Правила выполнения ходов со взятием описываются так же просто. Шашка может выполнить взятие, если перед ней стоит фигура противника, а за этой фигурой находится пустая клетка:

```
canTake Checker (inFront:behindIt:_) =
  hasPiece board (otherSide side) inFront && empty board behindIt
```

«Дамка» может выполнить взятие, если перед ней 0..n пустых клеток, за которыми находится вражеская фигура, а за ней — пустая клетка. Фактически, если пропустить первые пустые клетки, можно повторно использовать правило для шашек:

```
canTake King diagonal = canTake Checker nearestPiece
  where nearestPiece = dropWhile (empty board) diagonal
```

Во всех прочих случаях ход со взятием невозможен:

```
canTake _ _ = False
```

[19]Haskell: в объявлении функции `canMove` использовано сопоставление с образцом `_`, который совпадает с любым значением аргумента. Таким образом реализуется поведение «во всех прочих случаях — делать так-то».

После того, как возможность совершить ход (со взятием или без) подтверждена, необходимо составить описание этого хода в виде данных типа `MoveInfo`. Для этого в свое время были предусмотрены функции `genMoveInfo` и `genAttackInfo`.

Шашка может пойти только на ближайшее пустое поле вдоль диагонали[20]:

```
genMoveInfo Checker coords diagonal =
  [ MoveInfo { piece=Checker, from=coords, to=(head diagonal) } ]
```

«Дамка» может пойти на любое свободное поле диагонали, вплоть до ближайшего препятствия. Таким образом, для «дамки» необходимо генерировать список элементов `MoveInfo`, по одному на каждый возможный ход:

```
genMoveInfo King coords diagonal = map moveTo $ squaresToObstacle
diagonal
  where moveTo square = MoveInfo { piece=King, from=coords, to=square }
```

Похожим образом записывается и функция `genAttackInfo`. Шашка выполняет взятие, перепрыгивая через ближайшую по диагонали клетку на следующую за ней:

```
genAttackInfo Checker coords diagonal =
  [ AttackInfo { piece=Checker,         from=coords,
                 victim=(diagonal!!0),  to=(diagonal!!1) } ]
```

«Дамка» выполняет взятие, перепрыгивая через ближайшую по диагонали фигуру на любую следующую за ней пустую клетку[21]:

```
genAttackInfo King coords diagonal = map leapOverNearestPiece landingPlaces
  where
    leapOverNearestPiece square =
      AttackInfo { piece=King,           from=coords,
                   victim=nearestPiece,  to=square }
    (nearestPiece:behindNearestPiece) = dropWhile (empty board) diagonal
    landingPlaces = takeWhile (empty board) behindNearestPiece
```

Получившийся код находится в файле «checkers11.hs» в архиве с примерами кода. Чтобы функции `availableAttacks` и `availableMoves` заработали, необходимо дать определение функций `empty` и `hasPiece`, но уже сейчас можно проверить, что компилятор правильно вывел типы:

```
Prelude> :load checkers11.hs
[1 of 1] Compiling Main             ( checkers11.hs, interpreted )
Ok, modules loaded: Main.

*Main> :type availableMoves
availableMoves :: Board -> Side -> [MoveInfo]
```

[20] Haskell: использование структур с именованными полями (record syntax, см. [6]) повышает читаемость кода.

[21] Haskell: в where-блоке используется сопоставление с образцом, чтобы сразу разбить результат работы **`dropWhile`** на первый элемент списка `nearestPiece` и «хвост» `behindNearestPiece`.

```
*Main> :type availableAttacks
availableAttacks :: Board -> Side -> [MoveInfo]
```

4.13. Финальные штрихи

Чтобы завершить написание программы игры в шашки, осталось определить четыре функции: `hasPiece`, `empty`, `isVictory` и `getRandom`.

Проверка наличия фигуры указанного цвета в точке с указанными координатами реализуется просто. Если в списке фигур игрока есть запись с такими координатами, значит есть и фигура:

```
hasPiece board side coords =
  case lookup coords (pieces side board) of
    Nothing → False
    _       → True
```

Клетка доски считается пустой, если она не занята фигурой ни одного из игроков:

```
empty board coords =
  not (hasPiece board White coords || hasPiece board Black coords)
```

Игрок считается победителем, если на доске не осталось фигур противника:

```
  isVictory side board = null $ pieces (otherSide side) board
```

И, наконец, «интеллект» компьютерного игрока — функция, выбирающая случайный элемент из списка:

```
import System.Random

getRandom lst = do
  idx ← randomRIO (0,length lst−1)
  return (lst!!idx)
```

Получившийся код находится в файле «checkers11.hs» в архиве с примерами кода.

Теперь можно проверить правильность работы функций `availableAttacks` и `availableMoves`. Из начального положения доски для игры в русски шашки игроку доступно семь обычных ходов и ни одного хода со взятием:

```
Prelude> :l checkers12.hs
[1 of 1] Compiling Main             ( checkers12.hs, interpreted )
Ok, modules loaded: Main.

*Main> length (availableMoves checkers8x8 White)
7

*Main> availableAttacks checkers8x8 White
[]
```

Запустив программу при помощи команды «runhaskell checkers12.hs», можно увидеть сражение двух компьютерных игроков друг с другом.

На рисунке 4.4 можно увидеть, как взаимодействуют друг с другом все высокоуровневые функции финальной программы. Функции, определенные локально (в let- и where-блоках) не включены в диаграмму, чтобы повысить ее читаемость.

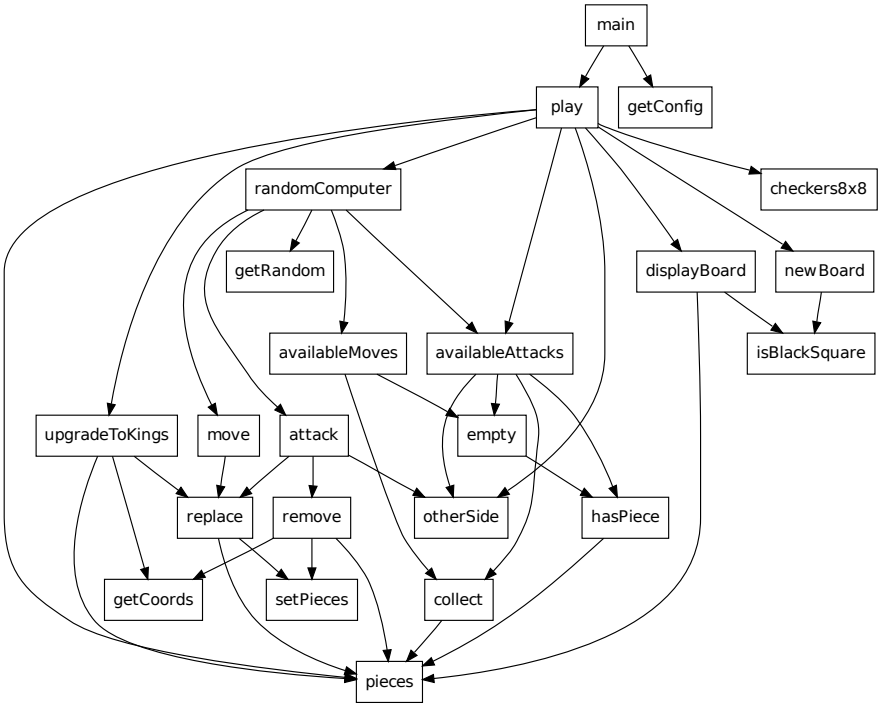


Рис. 4.4. Дерево вызовов функций программы checkers12.hs

4.14. Разбиение программного кода на модули

Изучив схему взаимодействия функций получившегося кода, можно прийти к выводу, что читаемость программы и удобство ее развития могут быть существенно улучшены путем разбиения кода на независимые модули.

Например, можно отделить описания всех используемых структур данных и раз-

местить их в отдельном модуле `Types`. Можно выделить в отдельный модуль `Board` все функции, работающие с состоянием доски, а в модуль `Checkers` — реализацию собственно игры в шашки. Если дополнительно вычленить реализацию компьютерного игрока в модуль `RandomComputer`, то окажется, что главный модуль программы состоит буквально из нескольких строк:

```
module Main where

import Board (checkers8x8)
import Checkers (play)
import RandomComputer (randomComputer)

main = do
  (newBoard, playerA, playerB) ← getConfig
  play newBoard playerA playerB

getConfig = return (checkers8x8, randomComputer, randomComputer)
```

Подобное разбиение на модули позволит скрыть часть деталей реализации. Функции `collectOpportunities`, `hasPiece`, `empty` и им подобные можно не экспортировать за пределы модулей, в которых они определяются и используются. На рисунке 4.5 видно, как сильно упрощается в результате схема взаимодействия разных частей программы.

Окончательный код программы находится в директории «checkers13» в архиве с примерами кода. Взяв его за основу, читатель может попробовать в качестве самостоятельного упражнения решить такие задачи:

- Модифицировать `displayBoard` так, чтобы рядом с изображением доски указывались номера строк и столбцов.
- Определить функцию, которая будет запрашивать с клавиатуры координаты передвигаемой фигуры, валидировать их и делать ход; затем использовать ее вместо `randomComputer` в вызове функции `play` для игры человека против компьютера.
- Для варианта игры «человек против компьютера» реализовать «undo» — возможность вернуть состояние на один или несколько ходов назад.
- Написать более интеллектуального компьютерного игрока, который предпочитает делать ходы, не подставляющие свои фигуры под ответный удар.

4.15. Заключение

В начале статьи было сделано утверждение о том, что функциональные языки со статической типизацией хорошо подходят для прототипирования программ и разра-

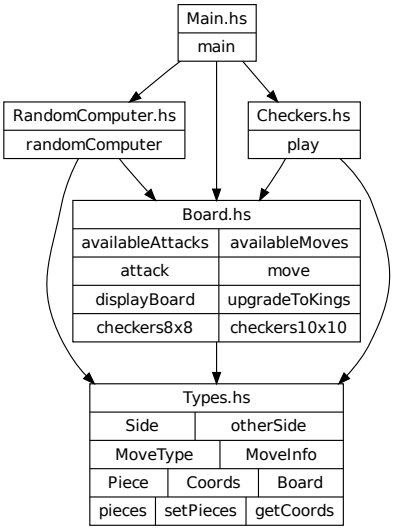


Рис. 4.5. Зависимости между модулями и перечень экспортируемых функций программы checkers13

ботки методом «сверху вниз». Примеры, приведенные в статье, должны убедительно проиллюстрировать это.

В самом деле, легко видеть, что словесное описание функций всегда было более объемным, чем соответствующая реализация на Haskell, даже в случае сложных функций вроде `collectOpportunities`. Использование функций высших порядков и возможность передавать функции в качестве аргументов позволили радикально сократить количество вспомогательного кода и сосредоточиться на реализации нужной функциональности.

Статическая типизированность языка Haskell, требующая, чтобы типы всех выражений были известны на этапе компиляции, не требовала от программиста дополнительного приложения сил. Скорее наоборот: автоматический вывод типов позволил не тратить время на проработку и описание сигнатур функций, а использование функции **`undefined`** позволило быстро получить пусть и не полностью реализованный, но уже компилируемый вариант кода и постепенно его развивать.

Литература

[1] Bottom. — Статья в Haskell Wiki, URL: http://www.haskell.org/haskellwiki/Bottom (дата обращения: 20 июля 2009 г.).

[2] Control structures. — Статья в Wiki-книге Haskell, URL: http://en.wikibooks.org/wiki/Haskell/Control_structures (дата обращения: 20 июля 2009 г.).

[3] Introduction to io. — Статья в Haskell Wiki, URL: http://www.haskell.org/haskellwiki/Introduction_to_IO (дата обращения: 20 июля 2009 г.).

[4] Learn you a haskell for great good. — Учебник, URL: http://learnyouahaskell.com/ (дата обращения: 20 июля 2009 г.).

[5] More about lists. — Статья в Wiki-книге Haskell, URL: http://en.wikibooks.org/wiki/Haskell/More_about_lists (дата обращения: 20 июля 2009 г.).

[6] More on datatypes. — Статья в Wiki-книге Haskell, URL: http://en.wikibooks.org/wiki/Haskell/More_on_datatypes (дата обращения: 20 июля 2009 г.).

[7] *O'Sullivan B., Stewart D., Goerzen J.* Real World Haskell. — O'Reilly Media, Inc., 2008. http://book.realworldhaskell.org/read/.

[8] Partial application. — Статья в Haskell Wiki, URL: http://www.haskell.org/haskellwiki/Partial_application (дата обращения: 20 июля 2009 г.).

[9] *Petzold C.* Does visual studio rot the mind? — Статья в блоге, URL: http://www.charlespetzold.com/etc/DoesVisualStudioRotTheMind.html (дата обращения: 20 июля 2009 г.).

[10] Yet another haskell tutorial. — Учебник, URL: http://darcs.haskell.org/yaht/yaht.pdf (дата обращения: 20 июля 2009 г.).

[11] *Душкин Р. В.* Функциональное программирование на языке Haskell. — М.: ДМК Пресс, 2007.

[12] Мягкое введение в haskell. — Учебник, URL: http://www.rsdn.ru/article/haskell/haskell_part1.xml (дата обращения: 20 июля 2009 г.).

Моноиды в Haskell и их использование

(Haskell Monoids and their Uses [2])

Dan Piponi

(Перевод Кирилла Заборского)

Аннотация

Haskell — замечательный язык для модульного конструирования кода из небольших ортогональных друг другу блоков. Одним из таких блоков является моноид. Несмотря на то, что моноиды родом из математики (точнее, из алгебры), они применяются повсюду в программировании. На каком бы языке вы ни программировали, вы почти наверняка неявно используете один или два моноида в каждой строке кода, сами о том пока не подозревая. Используя их явно, мы находим новые способы построения кода — в частности, способы, позволяющие облегчить написание и улучшить читаемость кода.

Предлагаемая статья является введением в моноиды на Haskell. Предполагается знакомство читателя с классами типов, так как моноиды в Haskell являются классом типов. Также предполагается хотя бы поверхностное знакомство с монадами.

5.1. Определение моноидов

Моноид в Haskell — это тип, для которого задано правило комбинирования двух элементов этого типа для получения нового элемента этого же типа. Для задания моноида необходимо также определить нейтральный элемент, комбинирование с которым любого другого элемента даёт результат, равный этому другому элементу.

Замечательным примером являются списки. Два списка — предположим `[1,2]` и `[3,4]` — могут быть объединены оператором `++` в единый список `[1,2,3,4]`. Существует также пустой список `[]`, при комбинировании с которым мы получаем второй список в неизменном виде — например, `[]++[1,2,3,4]==[1,2,3,4]`.

Другим примером является тип целых чисел **`Integer`**. Два элемента — например, 3 и 4 — могут быть скомбинированы оператором `+`, давая в результате сумму — 7. У нас также есть элемент 0, при сложении с которым любое целое число остаётся неизменным.

Вот пример определения класса типов `Monoid`:

```
class Monoid m where
    mappend :: m → m → m
    mempty :: m
```

Функция `mappend` используется для комбинирования пар элементов, а `mempty` представляет собой нейтральный элемент. Мы можем сделать списки моноидами, включив их в этот класс:

```
instance Monoid [α] where
    mappend = (++)
    mempty = []
```

Поскольку мы хотим, чтобы `mempty` не модифицировал комбинируемый с ним элемент, мы требуем, чтобы моноиды удовлетворяли следующей паре правил:

```
a `mappend` mempty = a
```

и

```
mempty `mappend` a = a.
```

Заметьте, что существует два способа скомбинировать `a` и `b` с использованием `mappend`. Мы можем написать `a `mappend` b` или `b `mappend` a`. От моноида не требуется совпадения результатов этих двух операций (эта тема обсуждается далее в статье), в то же время, моноиды должны обладать еще одним свойством. Предположим, у нас имеется список `[3,4]`. Мы хотим объединить его со списком `[1,2]` слева и со списком `[5,6]` справа. Мы можем выполнить объединение слева и получить `[1,2]++[3,4]`, а затем сформировать `([1,2]++[3,4])++[5,6]`. Мы можем также начать справа и получить `[1,2]++([3,4]++[5,6])`. Поскольку мы присоединяем списки с разных концов, эти операции не влияют друг на друга, и не имеет значения, которая из них выполнится первой. Это приводит нас к третьему и последнему требованию, которому должны удовлетворять моноиды:

```
(a `mappend` b) `mappend` c == a `mappend` (b `mappend` c)
```

Сформулируем это требование: «комбинирование слева и справа не мешают друг другу». Обратите внимание, что целые числа, комбинируемые операцией +, также удовлетворяют этому требованию. Это очень полезное свойство называется «ассоциативность».

Таково полное определение моноида. Haskell не принуждает к соблюдению приведенных трёх правил, но читая код, в котором присутствует моноид, мы всегда ожидаем, что эти правила соблюдены.

5.2. Некоторые применения моноидов

Почему нам может понадобиться использовать `mappend`, когда мы уже имеем в наличии такие функции, как ++ и +?

Одна из причин заключается в том, что моноиды автоматически предоставляют еще одну функцию — `mconcat`. Эта функция принимает на вход список значений в моноиде и комбинирует их вместе. Например, `mconcat [a,b,c]` будет эквивалентно `a ` mappend` (b ` mappend` c)`. Таким образом, в любом моноиде существует легкий способ скомбинировать вместе целый список. Стоит отметить, что в идее `mconcat` заключается некоторая двусмысленность. Какой порядок должен быть выбран, чтобы вычислить `mconcat [a,b,...,c,d]`? Должны ли мы выполнять операции слева направо, или начать с `c ` mappend` d`? Здесь вступает в силу правило ассоциативности: порядок не имеет значения.

Моноиды также будут к месту, если вам необходимо, чтобы ваш код был применим вне зависимости от способа комбинирования элементов. Вы можете написать код, который подобно `mconcat` будет работать с любым моноидом.

Явное использование класса типов `Monoid` в сигнатуре функции помогает читающему код понять замысел автора. Если функция имеет сигнатуру типа `[`α`]` $\rightarrow$ β, мы знаем о ней только то, что она принимает на вход список и создаёт из него объект типа β. Внутри она может делать со списком все, что угодно. Если же мы видим функцию типа `(Monoid `α`)` $\Rightarrow$ $\alpha \rightarrow \beta$, то даже если она применяется исключительно к спискам, мы имеем примерное представление о том, что происходит со списком внутри функции. Например, мы знаем, что функция может добавить новые элементы к списку, но не удалить элемент из списка.

Один и от же тип данных может служить основой разных моноидов. Например, как я уже упоминал, целые числа могут образовывать моноид, который определяется так:

```
instance Monoid Integer where
    mappend = (+)
    mempty = 0
```

В то же время, существует и другой естественный способ сделать моноид из целых чисел:

```
instance Monoid Integer where
    mappend = (×)
    mempty = 1
```

Мы не можем использовать оба эти определения одновременно. Поэтому библиотека `Data.Monoid` не создаёт моноид напрямую из **`Integer`**. Вместо этого, она оборачивает его в `Sum` (сумму) и `Product` (произведение). К тому же, библиотека делает это в более общем виде, позволяя превратить любые типы из класса **`Num`** в один из двух видов моноидов:

```
Num α ⇒ Monoid (Sum α)
```

и

```
Num α ⇒ Monoid (Product α)
```

Чтобы воспользоваться описанными функциями моноида, необходимо представить наши данные соответствующим образом. Например, `mconcat [Sum 2, Sum 3, Sum 4]` — это `Sum 9`, в то время как `mconcat [Product 2, Product 3, Product 4]` — это `Product 24`.

Использование `Sum` и `Product` кажется явным усложнением обычных операций сложения и умножения. Зачем же так делать?

5.3. Монада Writer

Моноиды можно представлять как накопители. Имея промежуточную сумму `n` и текущее значение `a`, мы можем получить новую промежуточную сумму `n' = n `mappend` a`. Накопление итогов часто применяется в программировании, поэтому остановимся на этой идее подробнее. Монада `Writer` предназначена специально для этого. Мы можем написать монадический код, который в качестве «побочного эффекта»будет накапливать некоторые значения. Функция, выполняющая накопление, называется несколько странно — `tell`. Следующий пример демонстрирует реализацию трассировки совершаемых действий.

```
1  import Data.Monoid
2  import Data.Foldable
3  import Control.Monad.Writer
4  import Control.Monad.State
5
6  fact1 :: Integer → Writer String Integer
7  fact1 0 = return 1
8  fact1 n = do
9    let n' = n−1
10   tell $ "We've taken one away from " ++ show n ++ "\n"
11   m ← fact1 n'
12   tell $ "We've called f " ++ show m ++ "\n"
13   let r = n × m
```

```
14    tell $ "We've multiplied " ++ show n
15                    ++ " and " ++ show m ++ "\n"
16    return r
```

В этом примере реализована функция вычисления факториала, сообщающая нам о выполняемых вычислениях. Каждый раз, когда вызывается функция `tell`, мы комбинируем её аргумент с промежуточным журналом вычислений, который был накоплен в результате предыдущих вызовов этой функции. Для извлечения журнала мы используем `runWriter`. Запустив следующий код:

```
17  ex1 = runWriter (fact1 10)
```

мы получаем значение `10!`, а вместе с ним — список шагов, которые потребовались для вычисления этого значения.

`Writer` позволяет накапливать не только строки. Мы можем использовать эту монаду с любым моноидом. Например, мы можем использовать её для подсчета количества операций сложения и умножения, необходимых для вычисления факториала числа. Для этого мы должны передать в `tell` значение соответствующего типа. В данном случае мы будем складывать значения, воспользовавшись моноидом для сложения — `Sum`. Мы можем написать:

```
18  fact2 :: Integer → Writer (Sum Integer) Integer
19  fact2 0 = return 1
20  fact2 n = do
21    let n' = n-1
22    tell $ Sum 1
23    m ← fact2 n'
24    let r = n * m
25    tell $ Sum 1
26    return r
27
28  ex2 = runWriter (fact2 10)
```

Эту задачу мы могли бы выполнить другим способом, применениив монаду `State`:

```
29  fact3 :: Integer → State Integer Integer
30  fact3 0 = return 1
31  fact3 n = do
32    let n' = n-1
33    modify (+1)
34    m ← fact3 n'
35    let r = n * m
36    modify (+1)
37    return r
38
39  ex3 = runState (fact3 10) 0
```

Результат такой реализации идентичен предыдущему, но версия с использованием `Writer` имеет большое преимущество. Из её типа —

`f :: Integer → Writer (Sum Integer) Integer` — можно сразу понять, что наша функция имеет побочный эффект, заключающийся в аддитивном накопении некоторого целого числа. Мы точно знаем, что она не перемножает накапливаемые значения. Не прочитав ни одной строчки кода функции, мы можем понять, что происходит у неё внутри, благодаря информации, содержащейся в её типе. Версия реализации функции, использующая `State`, вольна делать с накапливаемым значением все, что угодно, поэтому её назначение понять сложнее.

`Data.Monoid` также даёт нам моноид `Any`. Это тип **`Bool`** с заданной на нем операцией дизъюнкции, более известной как `||`. Такое название дано специально, чтобы показать, что при комбинировании любого набора элементов типа `Any`, наличие в наборе элемента со значением «истина» (`Any True`) даёт результат, выражающийся как «некоторый элемент является истинным» (`Any True`). Таким образом, мы получаем своего рода односторонний переключатель. Мы начинаем накопление с `mempty`, то есть `Any False`, что соответствует выключенному положению переключателя. Как только к нашему промежуточному результату добавляется значение `Any True`, переключатель переводится во включенное состояние. Вне зависимости от того, какие значения будут добавлены позже, переключатель уже не выключится. Этот процесс соответствует часто используемому в программировании шаблону: флаг, который в качестве побочного эффекта включается в случае выполнения некоторого условия.

```
40  fact4 :: Integer → Writer Any Integer
41  fact4 0 = return 1
42  fact4 n = do
43    let n' = n−1
44    m ← fact4 n'
45    let r = n × m
46    tell (Any (r==120))
47    return r
48
49  ex4 = runWriter (fact4 10)
```

В приведенном выше примере по окончании вычисления мы получаем значение `n!`, при этом нам также сообщается, если в процессе вычисления было выполнено умножение, результат которого был равен 120. Вызов функции `tell` практически повторяет словесное описание задачи на английском языке: «сообщи вызвавшему меня, если значение `r` когда-либо станет равно 120». Помимо того, что реализация этого флага требует минимального количества кода, существует еще одно перимущество. Достаточно взглянуть на тип этой версии функции, чтобы понять, что в ней происходит. Мы сразу видим, что эта функция в качестве побочного эффекта вычисляет флаг, который может включиться, но не может быть выключен. Для сигнатуры типа это большой объем информации. Во многих других языках программирования мы можем встретить булевый тип в сигнатуре, но нам придётся читать код, чтобы понять, как именно он используется.

5.4. Коммутативные моноиды, некоммутативные моноиды и дуальные моноиды

Говорят, что два элемента моноида `x` и `y` можно поменять местами, если `x `mappend` y == y `mappend` x`. Моноид называется коммутативным, если все его элементы можно менять местами. Хорошим примером коммутативного моноида является тип целых чисел. Для любой пары целых чисел `a+b == b+a`.

Если моноид не является коммутативным, то его называют некоммутативным. Если он некоммутативен, то существует пара элементов `x` и `y`, для которой `x `mappend` y` не равно `y `mappend` x`, и следовательно функции `mappend` и **`flip`** `mappend` не равнозначны. Например, `[1,2]++[3,4]` отличается от `[3,4]++[1,2]`. Интересным следствием этой особенности является то, что мы можем создать другой моноид, в котором функцией комбинирования будет **`flip`** `mappend`. Мы по-прежнему можем использовать тот же элемент `mempty`, таким образом два первых правила для моноидов будут соблюдаться. Будет неплохим упражнением доказать, что и третье правило при этом также будет выполнено. Такой «перевёрнутый» моноид называется дуальным, и `Data.Monoid` предоставляет конструктор типов `Dual` для построения дуальных моноидов. Он может быть использован для инвертирования порядка накопления данных монадой `Writer`. К примеру, следующий код собирает трассировку выполненных действий в обратном порядке:

```
50  fact5 :: Integer → Writer (Dual String) Integer
51  fact5 0 = return 1
52  fact5 n = do
53    let n' = n−1
54    tell $ Dual $ "We've taken one away from " ++ show n ++ "\n"
55    m ← fact5 n'
56    tell $ Dual $ "We've called f " ++ show m ++ "\n"
57    let r = n × m
58    tell $ Dual $ "We've multiplied " ++ show n
59                            ++ " and " ++ show m ++ "\n"
60    return r
61
62  ex5 = runWriter (fact5 10)
```

5.5. Произведение моноидов

Предположим, что мы хотим получать два побочных эффекта одновременно. Например, мы хотим вести подсчет числа выполняемых инструкций, а также получать словесную трассировку всех вычислений. Для комбинирования двух монад `Writer` мы могли бы воспользоваться преобразователями монад, но есть способ проще — мы можем скомбинировать два моноида в «произведение» моноидов. Определяется оно следующим образом:

```
instance (Monoid α, Monoid β) ⇒ Monoid (α, β) where
    mempty = (mempty, mempty)
    mappend (u,v) (w,x) = (u `mappend` w, v `mappend` x)
```

Каждый раз, применяя `mappend` к произведению, мы на самом деле применяем пару `mappend` отдельно к каждому элементу пары. С помощью следующих вспомогательных функций:

```
63 tellFst a = tell $ (a, mempty)
64 tellSnd b = tell $ (mempty, b)
```

мы можем использовать два моноида одновременно:

```
65 tellFst a = tell $ (a, mempty)
66 fact6 :: Integer → Writer (String, Sum Integer) Integer
67 fact6 0 = return 1
68 fact6 n = do
69   let n' = n−1
70   tellSnd (Sum 1)
71   tellFst $ "We've taken one away from " ++ show n ++ "\n"
72   m ← fact6 n'
73   let r = n × m
74   tellSnd (Sum 1)
75   tellFst $ "We've multiplied " ++ show n
76                       ++ " and " ++ show m ++ "\n"
77   return r
78
79 ex6 = runWriter (fact6 5)
```

Если бы мы имплементировали наш код с использованием одного специфического моноида, скажем, моноида для списков, применимость нашего кода была бы очень ограничена. Используя же обобщённый класс типов `Monoid`, мы обеспечиваем возможность повторного использования не только отдельных моноидов из нашего кода, но и наборов моноидов. Это способствует эффективности кода, поскольку мы можем собирать различные значения за один обход структуры данных. При этом мы обеспечиваем читаемость кода — наши алгоритмы легко читаются, поскольку код использует интерфейс к единственному моноиду.

5.6. «Сворачиваемые» данные

Последним примером применения моноидов в данной статье будет библиотека `Data.Foldable`. Она предоставляет обобщённый подход к обходу структур данных и сборке необходимых значений в процессе. Функция `foldMap` применяет соответствующую функцию к каждому элементу структуры и собирает результаты. Ниже следует пример реализации `foldMap` для деревьев:

```
80  data Tree α = Empty | Leaf α | Node (Tree α) α (Tree α)
81
82  instance Foldable Tree where
83    foldMap f Empty = mempty
84    foldMap f (Leaf x) = f x
85    foldMap f (Node l k r) = foldMap f l
86                             `mappend` f k
87                             `mappend` foldMap f r
```

Теперь мы можем использовать любой из рассмотренных выше моноидов для вычисления свойств деревьев. Например, мы можем использовать функцию `(==1)` для проверки равенства каждого элемента единице или использовать моноид `Any`, чтобы выяснить, существует ли в дереве элемент, равный единице. Вот пара примеров: один выясняет, существует ли в дереве элемент, равный 1, а другой — проверяет, каждый ли элемент дерева имеет значение больше 5.

```
88  tree = Node (Leaf 1) 7 (Leaf 2)
89
90  ex7 = foldMap (Any · (== 1)) tree
91  ex8 = foldMap (All · (> 5)) tree
```

Заметьте, что эти выражения без изменений могут быть использованы с любым сворачиваемым типом, не только с деревьями.

Надеюсь, вы согласитесь, что наши намерения представлены в коде в удобной для прочтения форме.

Тут же напрашивается еще одно упражнение: напишите подобный код для нахождения минимального и максимального элемента дерева. Для этого вам может понадобиться сконструировать новый моноид наподобие `Any` или `All`. Попробуйте найти оба элемента за один обход дерева, используя произведение моноидов.

Пример со «сворачиваемыми» данными иллюстрирует еще один момент. Программисту, реализующему `foldMap` для дерева, нет нужды заботиться о том, должно ли левое поддерево присоединяться к центральному элементу до правого или после. Ассоциативность гарантирует, что функция даст одинаковый результат вне зависимости от способа.

5.7. Заключение

Моноиды предоставляют общий подход к комбинированию и сбору значений. Они позволяют нам писать такой код, для которого неважно, каким образом мы комбинируем значения, что делает его более удобным для повторного использования. Используя именованные моноиды, мы можем указывать сингатуры типов так, чтобы читающим код были понятны наши намерения: например, используя `Any` вместо **`Bool`**, мы поясняем, как именно будет использовано булево значение. Мы можем комбинировать основанные на моноидах блоки, предоставляемые библиотеками Haskell, для построения полезных и легко читаемых алгоритмов с минимумом усилий.

Заключительные заметки: математики часто называют `mappend` «бинарным оператором» или «умножением». Так же как и в обычной алгебре, его часто записывают знаком умножения ($a \times b$) или даже слитно (ab). Подробнее о моноидах можно прочитать на Википедии [6]. К сожалению, у меня не хватает времени написать о морфизмах моноидов, о том, почему списочные моноиды являются свободными[1] (и какие возможности это даёт при написании кода), а также как альфа-композиция изображений выражается в моноидах, и о многом другом.

Литература

[1] Control.Monad.Writer.Lazy. — Online документация к GHC, URL: http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Writer-Lazy.html (дата обращения: 20 июля 2009 г.).

[2] *Dan Piponi*. Haskell Monoids and their Uses. — Запись в блоге: URL: http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html (дата обращения: 20 июля 2009 г.). — 2009.

[3] Data.Monoid. — Online документация к GHC, URL: http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Monoid.html (дата обращения: 20 июля 2009 г.).

[4] *Hurt B.* Random thoughts on haskell. — Запись в блоге, URL: http://enfranchisedmind.com/blog/posts/random-thoughts-on-haskell/ (дата обращения: 20 июля 2009 г.).

[5] *Jones M. P.* Functional programming with overloading and higher-order polymorphism. — 1995.

[6] Monoid. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Monoid (дата обращения: 20 июля 2009 г.).

[1] О том, что такое свободные моноиды, смотрите определение в Википедии: http://en.wikipedia.org/wiki/Free_monoid *Прим. пер.*

Обзор литературы о функциональном программировании

Алексей Отт
alexott@fprog.ru

Аннотация

За долгую историю развития функционального и декларативного программирования в мире было издано большое количество книг, посвященных теоретическим и практическим аспектам этих тематик, включая описания конкретных языков программирования. Достаточно много книг было издано также и на русском языке.

В данной статье сделана попытка провести обзор имеющейся русскоязычной литературы. Кроме того, представлен небольшой обзор существующей англоязычной литературы, имеющей отношение к данным темам. В конце статьи приводится список рекомендуемой литературы как по теоретическим вопросам ФП, так и по конкретным языкам программирования.

6.1. Литература на русском языке

В 70—80-е гг. в СССР было выпущено достаточно большое количество литературы, касающейся функционального и декларативного программирования. Список книг включает не только переводные книги, но и книги и учебники отечественных авторов, работавших в данных областях. В 90-е годы издание такой литературы практически сошло на нет, но в последние годы эта ситуация стала исправляться — появились переводы хороших зарубежных книг[1], а также вышло несколько книг русскоязычных авторов, в том числе и учебники, разработанные специально для вузов[2].

Важно отметить, что большая часть описанных ниже старых книг доступна в электронном виде, что облегчает возможность использования их при изучении соответствующих языков программирования.

[1] Очень часто они переводились силами энтузиастов функционального программирования.

[2] Тут необходимо отметить серию учебников и учебных курсов проекта Интуит, описанных ниже.

6.1.1. Общие вопросы ФП

В данном разделе рассматриваются книги и учебники, не посвященные конкретным языкам программирования, но дающие читателю возможность получить представление о функциональном программировании, его теоретических основах, и часто — о реализации языков.

«Функциональное программирование» (Харрисон/Филд)

В 1993 году издательство «Мир» выпустило перевод достаточно известной книги Functional Programming [16], написанной Петером Харрисоном (Peter G. Harrison) и Антони Филдом (Anthony J. Field) в 1988 году. На русском языке она называется «Функциональное программирование» [92].

Данная книга начинается с рассмотрения функций как таковых и использования функций высшего порядка, а также рассматривает виды вычислений, используемые при функциональном стиле программирования. Для демонстрации приемов программирования в книге вводится язык Hope. Помимо Hope, кратко описываются и другие языки программирования: Lisp, Miranda, FP.

За введением в ФП (функциональное программирование) следует основная часть книги, посвященная вопросам реализации языков программирования, начиная с основ лямбда-исчисления, системы вывода и проверки типов, вопросов интерпретации и компиляции кода, представления данных, сборки мусора, и заканчивая вопросами оптимизации программ (преобразование кода во время компиляции, оптимизация ленивого вычисления данных и т. п.).

Эту книгу можно рекомендовать всем тем, кто хочет не только досконально освоить ФП, но и разобраться во внутреннем устройстве языков программирования.

«Введение в функциональное программирование» (Харрисон)

Данный проект является переводом курса Introduction to Functional Programming [25] Джона Харрисона (John Harrison). Этот курс может использоваться для быстрого ознакомления с основами ФП и семейством языков ML. Он содержит в себе как описание теоретических основ ФП (от лямбда-исчисления до систем типов), так и примеры применения парадигм ФП для решения конкретных задач.

В данном курсе используется язык программирования Caml Light[3], входящий в семейство языков ML. По мере прохождения данного курса читатель получает набор знаний, необходимый для освоения данного языка и написания на нем достаточно сложных программ.

[3] Хочется отметить, что ведется работа над версией курса лекций, адаптированной для языка OCaml, который является развитием Caml Light, но не полностью совместим с ним.

Перевод может использоваться как основа курса лекций по ФП — помимо конспектов лекций (lecture notes) в нем содержатся переводы всех сопутствующих слайдов. Последняя версия перевода может быть загружена с сайта проекта.

«Структура и интерпретация компьютерных программ»

В 2006 году был выпущен перевод на русский язык классического учебника MIT по основам программирования «Структура и интерпретация компьютерных программ» [68] (Structure & Interpretation of Computer Programs, SICP[1]). Перевод был выполнен Георгием Бронниковым.

Данная книга содержит материалы по основам программирования; показывает, как с помощью композиции несложных процедур программист может строить сложные программные системы. Особый упор делается на показ преимуществ использования абстракций и модульности программ, а в качестве примеров рассматриваются построение языка программирования, включая компиляцию, обработка символьных данных, бесконечные потоки данных и т. п.

Книга отличается от других учебников тем, что в ней описываются разные подходы к композиции программ, демонстрируются преимущества функционального подхода к построению программ, использование функций высшего порядка и т. п., а в качестве основного языка программирования используется язык Scheme.

Качество перевода книги очень высокое, однако имеются недостатки, связанные с изданием самой книги: она вышла в мягком переплете, и ее не очень удобно читать, имеются проблемы верстки и опечатки, а главное — малый тираж (всего 1000 экземпляров), в связи с чем книгу уже тяжело найти в магазинах. В то же время, ее можно найти в электронном виде.

Учебные курсы проекта «Интуит»

Среди учебных курсов проекта «Интуит» имеется несколько курсов, которые посвящены вопросам функционального и декларативного программирования. Некоторые из них предлагают теоретическое изложение принципов ФП, другие посвящены конкретным языкам программирования. Эти курсы могут стать хорошим подспорьем при изучении ФП, поскольку материал рассчитан на людей, только начинающих знакомиться с соответствующими темами. Практически все курсы содержат задачи и упражнения, выполняя которые можно приобрести практический опыт применения полученных знаний.

В настоящее время опубликованы следующие курсы (материалы некоторых курсов доступны также в печатном виде — книги можно заказать с сайта проекта):

- «Стили и методы программирования» [88] — учебный курс, в котором систематически излагаются сведения о стилях программирования и их методах, а также обсуждаются вопросы сочетаемости различных методов в разработке программ.
- «Язык и библиотеки Haskell 98» — перевод известного учебника A Gentle Introduction To Haskell, описанного ниже, в разделе про Haskell (6.1.2).

• «Введение в программирование на Лиспе» [75] — вводный курс по программированию на языке Lisp с примерами решения задач на этом языке.

• «Основы функционального программирования» [73] — учебник по практическому программированию на языке Lisp.

• «Парадигмы программирования» [74] — курс, рассматривающий различные парадигмы программирования — функциональное, объектно-ориентированное, императивное и другие.

• «Введение в теорию программирования. Функциональный подход» [80] — еще один учебник по ФП. Здесь для примеров используется язык Standard ML.

• «Основы программирования на языке Пролог» [97] — учебный курс по логическому программированию и языку Пролог.

Стоит отметить, что материалы некоторых курсов пересекаются между собой, и некоторые курсы написаны достаточно сложно для самостоятельного изучения.[4]

«Типы в языках программирования» (Пирс)

Эта книга является переводом известной книги Types and Programming Languages Бенджамина Пирса (Benjamin C. Pierce) [55]. В книге рассматриваются различные аспекты использования типов в языках программирования: математические основы, различные типовые системы, вывод типов и т. д.

Этот перевод, также как и SICP, осуществляется Георгием Бронниковым. Бета-версии книги доступны в электронном виде, текущую версию вы можете найти на сайте проекта. Выход книги в печатном виде планируется после завершения работы над переводом, скорее всего в следующем году.

Другие книги, имеющие отношение к ФП

Помимо описанных выше, на русском языке было издано еще некоторое количество книг, имеющих отношение к функциональному программированию — о математических основах ФП, реализации языков и т. д. Ниже приведен краткий (и, вероятно, неполный) их список:

• В 1992 году был выпущен перевод известной книги Implementing functional languages: a tutorial [38], написанной Simon Peyton Jones & David Lester. На русском языке она называется «Реализация функциональных языков» [76]. Книга посвящена практическим вопросам реализации функциональных языков программирования. К сожалению, в настоящее время найти эту книгу ни в электронном, ни в бумажном виде не удаётся, поэтому доступным остаётся только английский оригинал.

• Перевод книги Питера Хендерсона «Функциональное программирование. Применение и реализация» [93] (Functional Programming: Application and Implementation [28]), вышедший в 1983 году. Книга не только знакомит с основами ФП, но и охватывает более сложные темы, включая тонкости реализации языков программирования (сборка мусора, компиляция кода и т. д.).

[4] Это, к сожалению, беда многих советских и российских учебников.

• В 1985 году был выпущен перевод книги Х. Барендрегта «Ламбда-исчисление: его синтаксис и семантика» [70] (The Lambda Calculus. Its Syntax and Semantics [3]). Книга посвящена теоретическим аспектам лямбда-исчисления, в ней рассматриваются классическое лямбда-исчисление, различные виды редукций и связанные с ними темы.

• Книга С. Маклейна «Категории для работающего математика» [86] (Categories for the Working Mathematician [44]), выпущенная в 2004 году, посвящена теории категорий, в рамках которой дается определение монад и других понятий и абстракций, нашедших применение в ФП. В книге всесторонне рассматриваются положения и концепции теории категорий.

• Учебное пособие В.М. Зюзькова «Математическое введение в декларативное программирование» [81] рассматривает математические основы декларативного и функционального программирования, лямбда-исчисление и методы доказательства теорем. Для примеров используются языки Prolog и Haskell.

6.1.2. О конкретных языках

Наряду с книгами, описывающими общие вопросы программирования на функциональных языках и математические основы лямбда-исчисления, в СССР и России издавались и книги по конкретным функциональным и декларативным языкам программирования. Достаточно широко представлена информация о языках Lisp, Haskell и Prolog, но к сожалению практически отсутствует литература по языку Erlang.

Lisp

Языку Lisp, являющемуся самым старым функциональным языком, в СССР было посвящено несколько публикаций (хотя их не так много, по сравнению с языком Пролог).

В 70-х гг. было выпущено сразу несколько книг по Лиспу:

• В 1976 году вышел перевод книги У. Маурера «Введение в программирование на языке ЛИСП» (Maurer W. D., The Programmer's Introduction to LISP [47]), содержавшей описания языка Лисп и множество примеров и задач на его использование.

• Через год Московским Энергетическим Институтом было издано учебное пособие по программированию на языке Lisp 1.5, написанное Е. Семеновой. Пособие содержит описание языка Lisp 1.5 и примеры его использования для решения учебных задач.

• И в 1978 году была выпущена книга С.С. Лаврова и Г.С. Силагадзе «Автоматическая обработка данных. Язык ЛИСП и его реализация» [84], описывающая язык Лисп и рассматривающая вопросы реализации этого языка.

В 1990 году вышел в свет широко известный двухтомник «Мир Лиспа» [95], являющийся переводом одноименной книги финских авторов Э. Хювёнен и И. Сеппянен. В первом томе содержится описание языка Common Lisp, включая типы данных

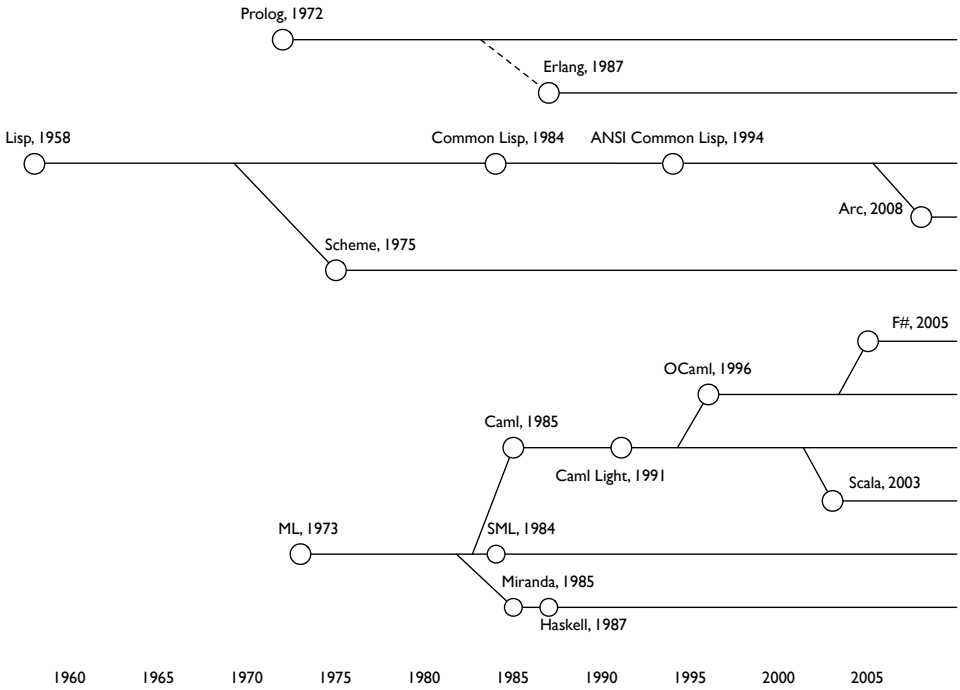


Рис. 6.1. Генеалогическое дерево семейств функциональных и декларативных языков

и наиболее часто используемые функции, ввод и вывод данных, обработку символьных данных и т. п. Кроме того, часть первого тома посвящена введению в методы ФП: использование рекурсии, функций высшего порядка, замыканий и макросов. Второй том содержит введение в другие методы программирования — логическое и объектное, описание среды программирования Лисп, а также большое количество примеров программ на этом языке, включая простой интерпретатор Лиспа.

Логическое программирование и язык Пролог

За последние тридцать лет в СССР (а затем и в России) было выпущено достаточно большое количество книг на темы логического программирования и искусственного интеллекта вообще и языка Пролог в частности (особенно много их было издано в 80-х гг.). Этот далеко не полный список включает следующие книги:

• *Иван Братко. «Программирование на языке Пролог для искусственного интеллекта»*. Первое издание на русском языке вышло в 1990 году [71] . В настоящее время в магазинах доступно третье издание этой книги [72], выпущенное в 2004 году. Первая часть книги полностью посвящена языку Пролог и методам работы с ним, а во второй части рассматриваются прикладные вопросы использования данного языка: построение экспертных систем, решение задач поиска, обучение машин, обработка

лингвистической информации и т. п.

- *Клоксин У., Меллиш К. «Программирование на языке пролог»* [83]. Эта книга, изданная в 1987 году, содержит только описание языка Пролог и особенностей его использования.
- *А. Адаменко, А. Кучуков. «Логическое программирование и Visual Prolog»* [69]. Книга издана в 2003 году и содержит небольшое введение в логическое программирование, в то время как основная часть книги посвящена вопросам программирования на Прологе с учетом особенностей Visual Prolog.
- *Дж. Малпас. «Реляционный язык Пролог и его применение»* [87]. Данная книга является подробным описанием языка Пролог и различных приемов программирования на этом языке (обработка текстов, представление знаний); содержит много примеров.
- *С. Чери, Г. Готлоб, Л. Танка «Логическое программирование и базы данных»* [96]. Книга рассматривает вопросы организации баз данных логических высказываний, включая краткое описание основ логического программирования и языков Пролог и Дейталог.
- *Л. Стерлинг, Э. Шапиро. «Искусство программирования на языке Пролог»* [90] (The Art of Prolog: Advanced Programming Techniques [62]). Выпущенная в 1990 году, книга английских ученых содержит материалы по теории логического программирования, достаточно подробно описывает язык Пролог и содержит большое количество примеров программирования на этом языке, включая систему для решения уравнений и компилятор простого языка программирования.
- *Ц. Ин, Д. Соломон. «Использование турбо-пролога»* [82]. Книга содержит описание принципов работы со средой программирования Турбо-Пролог, включая такие вопросы как использование машинной графики, создание многооконного интерфейса и т. п.
- *Дж. Макаллистер. «Искусственный интеллект и Пролог на микроЭВМ»* [85]. Книга в краткой форме содержит сведения по языку Пролог, логике, базам знаний и экспертным системам. В первую очередь предназначалась для владельцев небольших компьютеров серии Спектрум и т. п.
- *Дж. Стобо. «Язык программирования Пролог»* [91]. Данная книга является переводом книги «Problem Solving with Prolog» [63] () и описывает язык Пролог и его применение для решения различных задач — построения баз знаний, системы решения задач и других.
- *Дж. Доорс, А.Р. Рейблейн, С. Вадера. «Пролог — язык программирования будущего»* [77]. Книга содержит краткое описание языка Пролог, практических приемов работы с ним, а также решения некоторых логических задач.

Haskell

В настоящее время количество русскоязычных материалов по языку Haskell невелико. Только в последние годы стали появляться книги об этом языке (упомянутые

далее в статье книги Р. Душкина и Н. Рогановой, курсы проекта «Интуит») и появились энтузиасты, работающие над переводом англоязычных книг и статей на русский язык в целях популяризации Haskell среди русскоязычных программистов.

Книги о Haskell Романа Душкина В 2006—2007 гг. Роман Душкин, читавший в МИФИ в 2001—2006 гг. курсы по ФП, выпустил две книги, посвященные языку программирования Haskell.

Первая из них называется «Функциональное программирование на языке Haskell» [78] и является учебником по ФП, с примерами на языке Haskell, и используется в ряде вузов в качестве учебного пособия по ФП. В книге рассматриваются основы лямбда-исчисления, принципы построения программ на функциональных языках, а также описывается круг типовых задач, для которых использование функциональных языков является целесообразным. Использование монад, ввод/вывод данных, классы типов (включая стандартные классы языка Haskell) и другие вопросы иллюстрируются примерами на языке Haskell. В последних двух главах рассматриваются вопросы построения трансляторов и имеющиеся в Haskell средства для этого, а также обсуждаются подходы к решению некоторых задач искусственного интеллекта на языке Haskell.

Стоит отметить, что книга содержит достаточно большое количество математики и написана суховатым языком, что делает ее излишне теоретизированой с точки зрения программиста-практика и затрудняет восприятие. Кроме того, в книге не так много примеров, которые показывали бы применимость языка в повседневной разработке (если сравнивать с книгой Real World Haskell, которая является хорошим образцом в этом деле). Еще одной вещью, затрудняющей чтение книги является качество издания — верстки самой книги и бумаги, на которой она напечатана.

Вторая книга этого же автора называется «Справочник по языку Haskell» [79] и является дополнением к первой. Книга предназначена для читателей, уже знакомых с основами языка Haskell, поэтому она не должна рассматриваться как учебник по этому языку. Она содержит краткое описание синтаксиса языка Haskell, основных типов данных, а также (что важно!) основные приемы программирования на этом языке — использование различных видов рекурсии, функций высшего порядка и анонимных функций, защитных выражений и т. д.

Основная часть книги посвящена стандартным библиотекам, входящим в состав Hugs98 & GHC: начиная с Prelude и включая основные библиотеки (Control, System, Data, Text). Для каждой библиотеки приводится описание определенных в ней типов, классов и функций. Приводимые в справочнике определения функций могут использоваться в качестве примеров по написанию «правильного» кода на Haskell и являются хорошим подспорьем в работе.

«Функциональное программирование» (Роганова) В 2002 году Институт ИНФО издал учебное пособие Н.А. Рогановой под названием «Функциональное программирование» [89]. В данном пособии основной упор делается на практическое примене-

ние ФП для решения конкретных задач (автор выбрала задачи обработки структур данных и различные математические задачи). В нем практически нет теории, изобилующей математикой, что отличает его от других учебников по ФП. Все вводимые понятия иллюстрируются примерами на языке Haskell, который описан достаточно подробно, поэтому данное учебное пособие можно рассматривать в качестве начального по данному языку.

К недостаткам пособия можно отнести то, что отсутствие материала по теоретическим основам ФП (лямбда-исчисление и т. п.) требует изучения дополнительных материалов (которые, к сожалению, не указаны в списке литературы). Кроме того, в части описания языка Haskell мало внимания уделено таким вопросам, как ввод/вывод данных, разбор данных и т. п.

Переводы документации М. Ландина и В. Роганов в 2005 году выполнили перевод The Haskell 98 Report [39] — основного документа, который определяет синтаксис языка Haskell, а также состав основных библиотек этого языка. Перевод этого документа доступен с сервера haskell.ru как в варианте для печати, так и в online-версии.

Еще одна группа энтузиастов выполнила перевод на русский язык хорошо известного учебника по языку Haskell — Gentle Introduction To Haskell [33]. Данный учебник описывает основные возможности языка Haskell и наиболее часто используемые функции стандартных библиотек, включая ввод и вывод, и может использоваться для изучения основ языка. Перевод учебника доступен с сервера RSDN [94] и состоит из двух частей — часть 1 и часть 2.

Семейство языков ML

О семействе языков ML (Standard ML, Objective Caml, Caml Light) на русском языке существует сравнительно немного литературы. В качестве небольшого введения в программирование на языке Caml Light можно использовать курс лекций «Введение в функциональное программирование», описанный выше (6.1.1).

Кроме того, существует незаконченный перевод книги Developing Applications With Objective Caml [7] — переведено 11 глав, описывающих сам язык OCaml и базовые библиотеки, их можно использовать в качестве учебника по данному языку.

6.1.3. Планируется выпустить

Книга Сергиевского и Волчёнкова «Декларативное программирование» в настоящее время находится в процессе издания и должна появиться к концу этого года. Книга предназначена для использования в учебных заведениях. Она рассматривает вопросы функционального и логического программирования, включая теоретические вопросы ФП, доказательство свойств программ и т. д. Для примеров используются языки Lisp и Haskell. Отдельная часть учебника посвящена вопросам логического программирования с использованием языка Prolog.

Другие авторы также ведут работу над несколькими книгами, посвященными Haskell. Одна из них касается вопросов создания специализированных языков программирования (DSL) средствами языка Haskell, включая создание синтаксических анализаторов, а также ряда связаных с этим тем. Еще одна книга будет посвящена практическим аспектам использования Haskell с целью показать применимость языка Haskell для решения «реальных» задач.

Также в последнее время ведется работа над переводом на русский язык книги Practical Common Lisp. Книга содержит достаточно подробное введение в язык Common Lisp и содержит большое количество практических примеров, которые помогают начать использование этого языка в повседневной работе. Работа над переводом находится в заключительной стадии, а переведенный материал доступен на сайте проекта.

6.2. Англоязычная литература

На английском языке издано большое количество книг по ФП, его теоретическим основам, а также функциональным языкам программирования. Хотя некоторые книги и были переведены на русский, количество публикаций на английском языке гораздо больше. Краткие рецензии на некоторые из них приведены в этом разделе.

6.2.1. Общие вопросы ФП

В данном списке собраны книги, посвященные общим вопросам разработки ПО на функциональных языках, а также теоретическим вопросам ФП:

- Книга Programming Languages: Application and Interpretation [42] является учебником для курса «Языки программирования». В ней рассматриваются различные аспекты проектирования и разработки языков программирования. Для примеров используется язык Scheme.
- Purely Functional Data Structures [50] — отличная книга Криса Окасаки (Chris Okasaki) в которой описываются методы работы со сложными структурами данных в «чистых» функциональных языках.
- Книга The Functional Approach to Programming (Guy Cousineau, Michel Mauny) [11], описывающая все основные вопросы ФП, может использоваться в качестве учебника по ФП. Для примеров используется язык Caml.
- В книге Algorithms: A Functional Programming Approach [57] рассматриваются вопросы реализации различных алгоритмов на «чистых» функциональных языках, включая некоторые темы, описанные в книге «Purely Functional Data Structures». Для примеров используется Haskell.
- Книга Advanced Programming Language Design [17] (online-версия) содержит информацию о разных подходах к программированию, в том числе и несколько глав о функциональном и логическом программировании.

• Книга How to Design Programs: An Introduction to Programming and Computing [30] (имеющаяся в свободном доступе и поставляемая вместе с PLT Scheme), является учебником по программированию, демонстрирующим различные подходы к разработке программ. Для примеров в книге используется язык Scheme.

• Basic Category Theory for Computer Scientists [54] — данная книга рассматривает теорию категорий, лежащую в основе некоторых приемов, используемых в ФП (в частности, монад в языке Haskell).

6.2.2. Реализация языков программирования

Вопросы реализации функциональных языков программирования рассматриваются в некоторых описанных выше книгах, посвященных теории ФП, но кроме этого, существуют книги, посвященные исключительно вопросам реализации таких языков программирования:

• Книга Design Concepts in Programming Languages [66] посвящена теоретическим и практическим аспектам разработки языков программирования.

• Книга The Implementation of Functional Programming Languages [37], написанная Simon Peyton Jones и изданная в 1987 году, описывает такие темы, как лямбда-исчисление, вывод и проверка типов, сопоставление с образцом (pattern-matching), и использование этих приемов при реализации функциональных языков программирования.

• Книга Implementing functional languages: a tutorial [38], написанная Simon Peyton Jones & David Lester и изданная в 1992 году, рассматривает вопросы реализации функциональных языков программирования на примере реализации простого языка.

• Книга Garbage Collection: Algorithms for Automatic Dynamic Memory Management [36] посвящена описанию применяемых в функциональных языках программирования технологий «сборки мусора».

6.2.3. Конкретные языки ФП

Ниже перечислены наиболее интересные книги на английском языке, посвященные конкретным функциональным языкам программирования.

Haskell

Среди публикаций, посвященных языку Haskell, я хотел бы отметить следующие:

• Introduction to Functional Programming using Haskell Ричарда Бёрда [4] является учебником ФП, использующим Haskell в качестве основного языка. В нем рассмотрены базовые концепции ФП и их применение в Haskell. Книга содержит много примеров и упражнений для самостоятельного решения.

• Real World Haskell [51] является отличной книгой по языку Haskell, поскольку, кроме описания самого языка, содержит множество примеров, показывающих применение Haskell в реальной жизни: программирование баз данных и графических ин-

терфейсов, разбор данных, тестирование приложений и многое другое. Эта книга находится в свободном доступе на официальном сайте.

- The Haskell Road To Logic, Maths And Programming [13] показывает применение Haskell в математике и логике.
- Programming in Haskell [34], написанная Graham Hutton, описывает язык Haskell немного суховато, но может использоваться в качестве справочника теми, кто уже знаком с этим или другими функциональными языками, например, OCaml или Standard ML.
- Книга Haskell: The Craft of Functional Programming [65] посвящена описанию языка Haskell и принципов программирования на нем и включает отдельные главы по работе с типами данных, классами типов и т. п.
- The Haskell School of Expression: Learning Functional Programming through Multimedia [32] показывает практические аспекты применения Haskell, при этом описывает достаточно сложные темы, такие как взаимодействие с внешним миром, проектирование программ на Haskell и т. д.

Кроме напечатанных книг и учебников, имеются и материалы, доступные online. К наиболее интересным можно отнести:

- Раздел на сайте проекта Wikibooks, посвященный Haskell, содержит очень большое количество материалов различной степени сложности.
- A Gentle Introduction to Haskell 98 [33] — учебник по языку Haskell 98.
- Yet Another Haskell Tutorial [35] — еще один учебник по Haskell, содержащий примеры использования языка и упражнения для самостоятельного решения.
- Write Yourself a Scheme in 48 Hours — данный учебник позволяет получить навыки программирования на Haskell на практическом примере написания интерпретатора языка Scheme.
- All About Monads — учебник, посвященный теории и вопросам практического применения монад в Haskell.

Erlang

Книга Programming Erlang. Software for a Concurrent World [2], написанная Джо Армстронгом (Joe Armstrong), является практически единственным доступным печатным изданием, посвященным языку Erlang, поскольку выпущенная ранее книга «Concurrent Programming in Erlang» [10] стала уже библиографической редкостью (в интернете можно найти первую часть этой книги). «Programming Erlang» описывает язык простым языком и знакомит читателя с его основным функционалом. Кроме самого языка, книга описывает более сложные темы: базы данных, использование OTP и т. п.

Кроме того, в этом году планируется выпуск следующих книг, посвященных как самому языку Erlang, так и применению его в конкретных задачах:

- Erlang Programming [6],
- Concurrent Programming with Erlang/OTP [46],
- Erlang Web Applications: Problem-Design-Solution [22].

Caml & Objective Caml

Вопросам программирования на языке Objective Caml (OCaml) посвящено несколько книг.

Наиболее известной является свободно доступная книга Developing Applications with Objective Caml [7], которая не только описывает сам язык OCaml, но и рассматривает различные вопросы программирования с его использованием.

Недавно также появилась свободно распространяемая книга Introduction to Objective Caml [29], которая содержит достаточно подробное описание языка и примеры его применения.

Книга OCaml for Scientists [26] посвящена вопросам использования OCaml для «научного программирования» — обработки данных, математических вычислений, визуализации данных и оптимизации кода для лучшей производительности.

Еще одна книга — Practical OCaml [60], описывает язык OCaml и приемы программирования на нем. К сожалению, по многочисленными отзывами читателей, книга написана не очень хорошо.

Технический отчет The ZINC experiment: an economical implementation of the ML language [45], написанный Xavier Leroy в 1990 году, представляет собой подробное описание реализации языка ML и может быть интересен тем, кто интересуется внутренним устройством Caml & OCaml.

F#

В настоящее время по языку F# написана серия книг.

Foundations of F# [53] описывает основы языка и показывает разные методы программирования на нем, включая создание пользовательских интерфейсов и работу с базами данных.

Книга Expert F# [64] в свою очередь посвящена более сложным вопросам применения F# для разработки программ, таким как взаимодействие с кодом, написанным на других языках, использование библиотек .Net, разбор данных, асинхронное программирование и т. д.

F# for Scientists [27] является версией книги «OCaml for Scientists», адаптированной для языка F#, и содержит информацию по разным аспектам применения F# в «научном программировании» — визуализации данных, работе с базами данных, обработке данных и т. д.

Также в скором времени планируется выпуск еще нескольких книг, посвященных программированию на языке F# : Beginning F#, The Definitive Guide to F# и Functional Programming for the Real World: With Examples in F# and C#.

Standard ML

По языку Standard ML также выпущено достаточно большое количество книг.

Книга ML for the Working Programmer [52] является практическим введением в этот язык, описывающим сам язык и демонстрирующим некоторые приемы программирования на нем.

Книга The Little MLer [15] является кратким справочником по языку с примерами программ.

Книга «Unix System programming with Standard ML» [59] посвящена демонстрации применимости функциональных языков в повседневной работе.

Книга Elements of ML Programming, ML97 Edition [67], также описывающая сам язык и методы программирования на нем, может использоваться как введение в язык Standard ML.

Несколько книг посвящены изложению стандарта языка. К ним можно отнести книги The Definition of Standard ML [48] и The Standard ML Basis Library [21], которые содержат подробную информацию о языке и стандартной библиотеке.

Lisp

Кроме описанных ранее русскоязычных книг по языку Lisp, существует большое количество книг на английском языке, посвященных Lisp и его диалектам:

- Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP [49] — классическая книга Питера Норвига (Peter Norvig), посвященная вопросам искусственного интеллекта, показывает применение языка Common Lisp для решения некоторых задач искусственного интеллекта.
- ANSI Common Lisp [24], написанная Полом Гремом (Paul Graham), предназначена для начинающих программировать на Common Lisp. Книга содержит описание языка и примеры его использования.
- On Lisp [23], также написанная Полом Гремом, раскрывает более сложные вопросы программирования на Common Lisp: создание макросов, использование макросов для построения domain-specific languages (DSL) и т. п.
- Книги Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS [40] и The Art of Metaobject Protocol [41] содержат подробную информацию о программировании с использованием Common Lisp Object System. При этом, вторая книга в большей степени посвящена вопросам реализации Metaobject Protocol, лежащего в основе CLOS, и рекомендуется всем, кто интересуется вопросами объектно-ориентированного программирования (ООП).
- Книга Let Over Lambda [31] посвящена рассмотрению сложных тем программирования на Common Lisp — созданию и использованию макросов, правильному проектированию программ и т. п.
- Книга Common Lisp: The Language, 2ed [61] (также доступна online) является полным справочником по языку Common Lisp.
- Successful Lisp: How to Understand and Use Common Lisp [43] — еще одна книга для начинающих программировать на Lisp'е. Книга также имеет online версию.
- Lisp in Small Pieces [56] — достаточно известная книга по Lisp. В ней рассматриваются реализации языков Lisp и Scheme, включая программирование с использо-

ванием продолжений[5], построение интерпретатора и компилятора этих языков, поддержку макросов и многое другое.

Scheme

По языку программирования Scheme также выпущено несколько книг, в настоящее время можно купить следующие из них:

- The Little Schemer [20],
- The Reasoned Schemer [18],
- The Seasoned Schemer [19],
- The Scheme Programming Language, 3ed [14].

Книги описывают как сам язык, так и различные аспекты его использования. Эти книги могут использоваться как справочники по языку и являются хорошим дополнением к книгам Structure and Interpretation of Computer Programs [1] и How to Design Programs [30], в которых язык Scheme использован для примеров.

Prolog

Количество англоязычных книг по Прологу достаточно велико. Ниже приведен лишь небольшой список имеющейся литературы — я старался отобрать наиболее интересные книги:

- Logic Programming with Prolog [5] — хорошая книга по Prolog начального уровня. В ней описываются основные принципы программирования на Prolog вместе с примерами решения конкретных задач.
- Книга The Art of Prolog, Second Edition: Advanced Programming Techniques [62] посвящена вопросам использования языка, которые обычно не рассматриваются в книгах, предназначенных для изучения самого языка: построение интерпретаторов и компиляторов, преобразование программ, теории логического программирования.
- Programming in Prolog: Using the ISO Standard [8] — еще один учебник по Прологу, демонстрирующий основные принципы программирования на этом языке.
- Clause and Effect: Prolog Programming for the Working Programmer [9] является небольшим введением в Пролог для программистов, владеющих другими языками.
- Prolog Programming in Depth [12] — еще одна книга, посвященная «сложным» аспектам применения Пролога: взаимодействию с внешним миром, императивному программированию на Прологе, построению экспертных систем и т. п.

6.3. Рекомендации

Если вы хотите познакомиться с принципами создания функциональных языков программирования, то на русском языке базовую информацию вы почерпнете

[5] Продолжение — continutation.

из книг «Функциональное программирование» [92], «Функциональное программирование. Применение и реализация» (Хендерсон) и «Реализация функциональных языков». Из книг на английском языке я могу порекомендовать книги, перечисленные в разделе «Реализация функциональных языков программирования (6.2.1)».

Заинтересовавшиеся Common Lisp могут начать его изучение с книги Practical Common Lisp [58] (существующей и на русском языке), которая даст информацию по основным аспектам языка. Более сложные аспекты работы с Lisp описаны в On Lisp [23], The Art of Metaobject Protocol [41], Let Over Lambda [31], Lisp in Small Pieces [56] и других англоязычных книгах (6.2.3).

Для обучения функциональному программированию на языке Haskell можно порекомендовать книгу «Introduction to Functional Programming using Haskell» Ричарда Бёрда [4]. Для желающих узнать о практическом применении Haskell хорошим выбором будет книга Real World Haskell [51], в которой приводятся практические примеры использования Haskell. Среди учебников можно отметить Yet another Haskell tutorial [35] и A Gentle Introduction to Haskell 98 [33] (также доступный на русском языке), ну и конечно раздел о Haskell в проекте Wikibooks.

В настоящее время по языку Erlang доступно не так уж много литературы — только книга Programming Erlang. Software for a Concurrent World [2] и официальная документация. Книга может быть использована для ознакомления с языком и концепциями, лежащими в основе OTP, после чего можно переходить к изучению библиотек и фреймворков, входящих в состав дистрибутива языка. Хочется надеяться, что ситуация с литературой по данному языку улучшится с выходом новых книг (6.2.3).

Для ознакомления с языками семейства ML существует достаточно много литературы. Выбравшим OCaml лучше начать с книги Introduction to Objective Caml [29], используя её вместе со справочником по языку, а потом переходить к Developing Applications with Objective Caml [7] и другим книгам из списка выше (6.2.3). А изучение F# стоит начать с Foundations of F# [53] и продолжить чтением Expert F# [64] и F# for Scientists [27].

Для Prologa выбор книг достаточно велик — начать можно с книги Братко «Программирование на языке Пролог для искусственного интеллекта» [72], а затем переходить к книгам на английском языке, перечисленным выше (6.2.3).

6.4. Заключение

Хотелось бы отметить, что появившаяся тенденция к изданию на русском языке книг по тематике функционального/декларативного программирования не может не радовать. В печати появляются как переводы отличных зарубежных книг, так и публикации отечественных авторов. Некоторые книги зарубежных авторов переводятся силами энтузиастов, что часто позволяет получить очень качественный с технической точки зрения перевод.

Литература

[1] *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs, 2nd Edition. — The MIT Press, 1996. http://mitpress.mit.edu/sicp/.

[2] *Armstrong J.* Programming Erlang: Software for a Concurrent World. — Pragmatic Programmers, 2007.

[3] *Barendregt H. P.* The Lambda Calculus: its Syntax and Semantics. — North-Holland, 1981.

[4] *Bird R. S.* Introduction to Functional Programming Using Haskell, 2nd Edition. — 2nd edition. — Prentice-Hall, 1998.

[5] *Bramer M.* Logic Programming with Prolog. — Springer, 2005.

[6] *Cesarini F., Thompson S.* Erlang Programming. — O'Reilly, 2009.

[7] *Chailloux E., Manoury P., Pagano B.* Developing Applications With Objective Caml. — O'Reilly, 2000. — 757 pp. http://caml.inria.fr/pub/docs/oreilly-book/.

[8] *Clocksin W., Mellish C.* Programming in Prolog: Using the ISO Standard, 5th Edition. — Springer, 2003.

[9] *Clocksin W. F.* Clause and Effect: Prolog Programming for the Working Programmer. — Springer, 2003.

[10] Concurrent Programming in Erlang, Second Edition / J. Armstrong, R. Virding, C. Wikström, M. Williams. — Prentice-Hall, 1996.

[11] *Cousineau G., Mauny M.* The Functional Approach to Programming. — Cambridge University Press, 1998.

[12] *Covington M. A., Nute D., Vellino A.* Prolog Programming in Depth. — Prentice Hall, 1996.

[13] *Doets K., van Eijck J.* The Haskell Road to Logic, Maths and Programming. — College Publications, 2004.

[14] *Dybvig R.* The Scheme Programming Language. — 3rd edition. — The MIT Press, 2003.

[15] *Felleisen M., Friedman D. P.* The Little MLer. — The MIT Press, 1997.

[16] *Field A. J., Harrison P. G.* Functional Programming. — Addison-Wesley, 1988.

[17] *Finkel R.* Advanced Programming Language Design. — Addison Wesley, 1995.

[18] *Friedman D. P., Byrd W. E., Kiselyov O.* The Reasoned Schemer. — The MIT Press, 2005.

[19] *Friedman D. P., Felleisen M.* The Seasoned Schemer. — The MIT Press, 1995.

[20] *Friedman D. P., Felleisen M., Sussman G. J.* The Little Schemer, 4th Edition. — The MIT Press, 1995.

[21] *Gansner E. R., Reppy J. H.* The Standard ML Basis Library. — Cambridge University Press, 2002.

[22] *Gerakines N.* Erlang Web Applications: Problem-Design-Solution. — John Wiley and Sons, 2009.

[23] *Graham P.* On Lisp. — Prentice Hall, 1993. http://www.paulgraham.com/onlisp.html.

[24] *Graham P.* ANSI Common LISP. — Prentice Hall, 1995.

[25] *Harrison J.* Introduction to functional programming. — Lecture notes. — 1997. http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/.

[26] *Harrop J.* OCaml for Scientists. — 2007.

[27] *Harrop J.* F# for Scientists. — Wiley-Interscience, 2008.

[28] *Henderson P.* Functional Programming: Application and Implementation. — Prentice-Hall, 1980.

[29] *Hickey J.* Introduction to objective caml. — 2008. http://www.freetechbooks.com/introduction-to-objective-caml-t698.html.

[30] How to Design Programs: An Introduction to Programming and Computing / M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi. — The MIT Press, 2001. http://htdp.org/.

[31] *Hoyte D.* Let Over Lambda. — Lulu.com, 2008.

[32] *Hudak P.* The Haskell School of Expression: Learning Functional Programming through Multimedia. — Cambridge University Press, 2000.

[33] *Hudak P., Peterson J., Fasel J.* A gentle introduction to haskell, version 98. http://haskell.cs.yale.edu/tutorial/.

[34] *Hutton G.* Programming in Haskell. — Cambridge University Press, 2007.

[35] *III H. D.* Yet another haskell tutorial. — Учебник, URL: http://darcs.haskell.org/yaht/yaht.pdf (дата обращения: 20 июля 2009 г.). http://darcs.haskell.org/yaht/yaht.pdf.

[36] *Jones R., Lins R.* Garbage Collection: Algorithms for Automatic Dynamic Memory Management. — Wiley, 1996.

[37] *Jones S. L. P.* The Implementation of Functional Programming Languages. Computer Science. — Prentice-Hall, 1987. http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/.

[38] *Jones S. L. P., Lester D.* Implementing functional languages: a tutorial. — 1992. http://research.microsoft.com/en-us/um/people/simonpj/papers/pj-lester-book/.

[39] *Jones S. P.* Haskell 98 language and libraries. the revised report. — 2002. http://haskell.org/haskellwiki/Definition.

[40] *Keene S. E.* Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. — Addison-Wesley Professional, 1989.

[41] *Kiczales G., des Rivieres J., Bobrow D. G.* The Art of the Metaobject Protocol. — The MIT Press, 1991.

[42] *Krishnamurthi S.* Programming Languages: Application and Interpretation. — 2003. http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/.

[43] *Lamkins D. B.* Successful Lisp: How to Understand and Use Common Lisp. — bookfix.com, 2004. http://psg.com/~dlamkins/sl/contents.html.

[44] *Lane S. M.* Categories for the Working Mathematician. — Springer Verlag, 1998.

[45] *Leroy X.* The zinc experiment: an economical implementation of the ml language: Technical report 117: INRIA, 1990. http://gallium.inria.fr/~xleroy/publi/ZINC.pdf.

[46] *Logan M., Merritt E., Carlsson R.* Concurrent Programming with Erlang/OTP. — Manning, 2009.

[47] *Maurer W. D.* The programmer's introduction to LISP. — London, Macdonald, 1972.

[48] *Milner R., Tofte M., Harper B.* The Definition of Standard ML. — MIT Press, 1990.

[49] *Norvig P.* Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. — Morgan Kaufmann, 1991.

[50] *Okasaki C.* Purely Functional Data Structures. — Cambridge University Press, 1998.

[51] *O'Sullivan B., Stewart D., Goerzen J.* Real World Haskell. — O'Reilly Media, Inc., 2008. http://book.realworldhaskell.org/read/.

[52] *Paulson L. C.* ML for the Working Programmer, 2ed. — Cambridge University Press, 1996.

[53] *Pickering R.* Foundations of F#. — Apress, 2007.

[54] *Pierce B. C.* Basic Category Theory for Computer Scientists. — The MIT Press, 1991.

[55] *Pierce B. C.* Types and Programming Languages. — MIT Press, 2002. http://www.cis.upenn.edu/~bcpierce/tapl.

[56] *Queinnec C.* Lisp in Small Pieces. — Cambridge University Press, 2003.

[57] *Rabhi F. A., Lapalme G.* Algorithms: A Functional Programming Approach. — Addison Wesley, 1999.

[58] *Seibel P.* Practical Common Lisp. — Apress, 2005. http://www.gigamonkeys.com/book/.

[59] *Shipman A. L.* Unix System Programming with Standard ML. — 2001. http://web.archive.org/web/20030302003837/http://web.access.net.au/felixadv/files/output/book/.

[60] *Smith J. B.* Practical OCaml. — Apress, 2006.

[61] *Steele G.* Common LISP. The Language, 2ed. — Digital Press, 1990. http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html.

[62] *Sterling L., Shapiro E.* The Art of Prolog: Advanced Programming Techniques. — The MIT Press, 1986.

[63] *Stobo J.* Problem Solving with Prolog. — Pitman, 1989.

[64] *Syme D., Granicz A., Cisternino A.* Expert F#. — Apress, 2007.

[65] *Thompson S.* Haskell: The Craft of Functional Programming, 2nd Edition. — Addison-Wesley, 1999.

[66] *Turbak F. A., Gifford D. K.* Design Concepts in Programming Languages. — The MIT Press, 2008.

[67] *Ullman J. D.* Elements of ML Programming, ML97 Edition, 2ed. — Prentice Hall, 1998.

[68] *Харольд Абельсон, Джеральд Джей Сассман.* Структура и интерпретация компьютерных программ. — М.: Добросвет, 2006.

[69] *А. Адаменко, А. Кучуков.* Логическое программирование и Visual Prolog. — БХВ-Петербург, 2003.

[70] *Х. Барендрегт.* Ламбда-исчисление. Его синтаксис и семантика. — М.: Мир, 1985.

[71] *И. Братко.* Программирование на языке PROLOG для искусственного интеллекта. — М.: Мир, 1990.

[72] *И. Братко*. Алгоритмы искусственного интеллекта на языке Prolog. — Вильямс, 2004.

[73] *Л. В. Городняя*. Основы функционального программирования. http://www.intuit.ru/department/pl/funcpl/.

[74] *Л. В. Городняя*. Парадигмы программирования. http://www.intuit.ru/department/se/paradigms/.

[75] *Л. В. Городняя, Н.А. Березин*. Введение в программирование на Лиспе. http://www.intuit.ru/department/pl/lisp/.

[76] *С. П. Джонс, Д. Лестер*. Реализация функциональных языков. — 1992.

[77] *Дж. Доорс, А. Р. Рейблейн, С. Вадера*. Пролог - язык программирования будущего. — М.: Финансы и статистика, 1990.

[78] *Р. В. Душкин*. Функциональное программирование на языке Haskell. — М.: ДМК Пресс, 2007.

[79] *Р. В. Душкин*. Справочник по языку Haskell. — М.: ДМК Пресс, 2008.

[80] *С. В. Зыков*. Введение в теорию программирования. Функциональный подход. http://www.intuit.ru/department/se/tppfunc/.

[81] *В. М. Зюзьков*. Математическое введение в декларативное программирование. — 2003. http://window.edu.ru/window/library?p_rid=46691.

[82] *Ц. Ин, Д. Соломон*. Использование Турбо-Пролога. — М.: Мир, 1990.

[83] *Клоксин У. and Меллиш К.* Программирование на языке пролог. — М.: Мир, 1987.

[84] *С. С. Лаврова, Г. С. Силагадзе*. Автоматическая обработка данных. Язык ЛИСП и его реализация. — М.: Наука, 1978.

[85] *Дж. Макаллистер*. Искусственный интеллект и Пролог на микроЭВМ. — М.: Машиностроение, 1990.

[86] *С. Маклейн*. Категории для работающего математика. — Физматлит, 2004.

[87] *Дж. Малпас*. Реляционный язык Пролог и его применение. — М.: Наука, 1990.

[88] *Н. Н. Непейвода*. Стили и методы программирования. http://www.intuit.ru/department/se/progstyles/.

[89] *Н. А. Роганова*. Функциональное программирование. — 2002.

[90] *Л. Стерлинг, Э. Шапиро*. Искусство программирования на языке Пролог. — М.: Мир, 1990.

[91] *Дж. Стобо*. Язык программирования Пролог. — М.: Радио и связь, 1993.

[92] *А. Филд, П. Харрисон*. Функциональное программирование. — М.: Мир, 1993.

[93] *П. Хендерсон*. Функциональное программирование. Применение и реализация. — М.: Мир, 1983.

[94] *Пол Хьюдак, Джон Петерсон, Джозеф Фасел*. Мягкое введение в haskell. — Учебник. http://www.rsdn.ru/article/haskell/haskell_part1.xml.

[95] *Э. Хювёнен, И. Сеппянен*. Мир Лиспа. — М.: Мир, 1990.

[96] *С. Чери, Г. Готлоб, Л. Танка*. Логическое программирование и базы данных. — М.: Мир, 1992.

[97] *П. А. Шрайнер*. Основы программирования на языке Пролог. http://www.intuit.ru/department/pl/plprolog/.