ТЕОРИЯ КАТЕГОРИЙ ДЛЯ ПРОГРАММИСТОВ

Бартош Милевски

Излагаются основы теории категорий с точки зрения программирования, прежде всего, на языках функционального типа. Доказательства категорных свойств, как правило, не приводятся, но объясняются программными конструкциями. Такая возможность позволяет осветить значительный объем сведений по теории категорий, имеющих полезные применения в прикладных информационных областях, вообще, и в программировании, в частности.

Редактор перевода:
Геннадий Чернышев
(https://henrychern.wordpress.com/)

Category Theory for Programmers

by **Bartosz Milewski**

compiled and edited by
Igal Tabachnik

Category Theory for Programmers

Bartosz Milewski

Version v36-98b71ac, January 30, 2023

This work is licensed under a Creative Commons

Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0).

Converted to LaTeX from a series of blog posts by Bartosz Milewski.

PDF and book compiled by Igal Tabachnik.

LaTeX source code is available on GitHub:
https://github.com/hmemcpy/milewski-ctfp-pdf

От редактора перевода

Перевод соответствует содержимому книги "Bartosz Milewski. Category Theory for Programmers" (версия **v36-98b71ac** от 30 января 2023), которая была скомпонована (Бартош Милевски) путем преобразования исходного текста в формат LaTeX и (Игаль Табачник) предварительного извлечения исходных сообщений блога WordPress с помощью Mercury Web Parser
(https://mercury.postlight.com/web-parser/) для получения чистого HTML-содержимого, изменения и настройки с помощью Beautiful Soup (https://www.crummy.com/software/BeautifulSoup/) и, наконец, преобразования в LaTeX с помощью Pandoc
(https://pandoc.org/). Книга защищена международной лицензия Creative Commons Attribution-ShareAlike 4.0 (cc by-sa 4.0).

Текущая авторская версия книги, на самом деле, имеет несколько реализаций, различающиеся использованием в примерах разных функциональных языков, из числа: OCaml, ReasonML, Scala, Haskell (при этом, три реализации содержат код на Haskell, за которым следует код этого же фрагмента на другом языке).

В авторском тексте приведены следующие ссылки на вспомогательный материал (видео Catsters):
https://www.youtube.com/watch?v=upCSDIO9pjc (о произведениях и ко-произведениях, для главы 5),
https://www.youtube.com/watch?v=4QgjKUzyrhM (о представимых функторах, для главы 14),
https://www.youtube.com/watch?v=TLMxHB19khE (о представимых функторах и лемме Йонеды, для главы 15).

В конце некоторых глав приведена библиография, относящаяся к текущей главе.

В переводе использована «чистая» Haskell-реализация (читателю, при необходимости, придется обратится к соответствующей исходной реализации книги для просмотра программного кода на других языках). Перевод подготовлен полностью на LyX.

Основная особенность оформления перевода, отличающая текст от авторского, заключается в цветовом выделении формальных фрагментов.

Синим цветом выделены математические формулы и символы, коричневым — программные фрагменты, набранные моноширинным шрифтом, на псевдо Haskell. Пурпурный цвет выделяет программные фрагменты на C++ и Haskell. Помимо этого везде черным полужирным начертанием выделены обозначения категорий.

Такое «цветовое» решение выбрано с целью облегчения зрительного разделения фрагментов содержимого на формальное (в основном, это теоретико-категорное описание) и прикладное (реализации на Haskell), что позволяет читать книгу выборочно по этим разрезам. Последнее полезно для читателей, желающих ознакомиться с категорными понятиями и подходом, не углубляясь в детали программирования на функциональном языке.

В перевод включен список дополнительной литературы по теории категорий, языку функционального программирования и сопутствующим разделам математики и компьютерных наук. Список далеко не является полным и включает источники для дальнейшего изучения изложенных в книге вопросов или получения сведений справочного характера. Так в [2], помимо введения в функциональное программирования на Haskell, приведены ссылки на важные ресурсы по этому языку.

В книгах [5, 6, 7, 8, 9] и журналах [23] приводится множество полезных сведений по основам и расширениям языков функционального программирования в теоретическом и прикладном аспектах. В [24] рассмотрены проблемы аппликативного программирования, особое внимание уделено вопросам реализации функциональных языков, основанных на λ-исчислении (на базе языка Hope).

Из источников по теории категорий следует рекомендовать [4], содержащий весьма доступное введение в теорию категорий и теорию топосов. Книги [20] и [25] содержат описания прикладных аспектов теории категорий в области баз данных.

Перечисленные и другие вопросы, связанные с обсуждаемыми в данной книге, можно найти на сайте редактора перевода[1].

Обнаруженные в ходе работы над переводом неточности и опечатки были устранены благодаря читателям и при непосредственном участии ав-

[1] http://henrychern.wordpress.com

тора книги. С благодарностью принимаются любые замечания по переводу и оформлению текста.

Распределение работы между переводчиками было следующим: Максим Страхов — часть I, главы 1-4; Алексей Бирюков — часть I, главы 5-7; Геннадий Чернышев — часть I, главы 8-10, части II и III.

Г. Чернышев

Предисловие автора

> В течение довольно продолжительного времени я обдумывал идею: написать книгу по теории категорий для программистов. Заметьте, не для компьютерных теоретиков, а для программистов — скорее инженеров, чем ученых. Я знаю, что это звучит безумно, и я сам был достаточно напуган. Я знаю, что есть огромная разница между наукой и техникой, потому, что я работал по обе стороны баррикад. Но у меня всегда был очень сильный порыв объяснять сложные вещи простым языком. Я восхищаюсь Ричардом Фейнманом, который был мастером простых объяснений. Я знаю, я не Фейнман, но я стараюсь и буду стараться с полной самоотдачей. Я начинаю с публикации этого предисловия, которое должно мотивировать читателя изучать теорию категорий, и надеюсь на начало дискуссии и обратную связь[2].

Я постараюсь в нескольких параграфах убедить вас, что эта книга написана для вас, и развеять все ваши сомнения в необходимости изучения этой, одной из самых абстрактных областей математики, в свое драгоценное свободное время.

Мой оптимизм основан на нескольких наблюдениях. Во-первых, теория категорий — сокровищница чрезвычайно полезных идей программирования. Haskell-программисты черпают из нее эти идеи уже долгое время, но эти идеи медленно просачиваются в другие языки, этот процесс идет слишком медленно. Необходимо его ускорить.

Во-вторых, есть много различных направлений в математике, и все они предназначены для разных аудиторий. У вас может быть аллергия на математический анализ или алгебру, но это не означает, что вам не понравится теория категорий. Не побоюсь утверждать, что теория категорий — это именно та математика, которая особенно хорошо подходит для мышления программистов. Это потому, что теория категорий вместо того, чтобы иметь дело с деталями, оперирует структурой. Она оперирует такими понятиями, которые делают программы компонуемыми.

[2] Вы также можете ознакомиться с тем, как я преподаю этот материал вживую, по ссылке `https://goo.gl/GT2UWU` (или выполните поиск "bartosz milewski category theory" на YouTube).

Композиция в самой основе теории категорий, она — часть самого определения категории. И я утверждаю, что композиция — суть программирования. Мы комбинировали разные сущности уже очень давно, задолго до того, как какой-то великий инженер придумал подпрограммы. Некоторое время назад принципы структурного программирования произвели революцию в программировании, — они сделали блоки кода комбинируемыми. Потом пришло объектно-ориентированное программирование, суть которого заключается в комбинировании объектов. Функциональное программирование имеет дело не только с комбинированием функций и алгебраических структур данных, еще оно делает параллелизм компонуемым, что практически невозможно в других парадигмах.

В-третьих, у меня есть секретное оружие, нож мясника, которым я буду кромсать математику, чтобы сделать ее понятнее для программистов. Когда вы профессиональный математик, вы должны быть очень осторожны, чтобы определить все ваши предположения точно, выписать каждое выражение должным образом, и строить все свои доказательства строго. Это делает математические статьи и книги чрезвычайно трудными для чтения непосвященными. Я по образованию физик, и в физике мы добились удивительных успехов, используя неформальные рассуждения. Математики смеялись над дельта-функцией Дирака, которая была введена великим физиком, чтобы решить некоторые дифференциальные уравнения. Они перестали смеяться, когда была придумана совершенно новая область анализа, формализующая идеи Дирака, и названная теорией распределений.

Конечно, размахивая руками, вы рискуете сказать что-то откровенно неверное, поэтому я постараюсь убедить читателя, что за неформальными аргументами имеется твердая математическая теория (у меня в качестве настольной содержится потертая копия книги Сандерса МакЛейна «Категории для работающего математика»).

Поскольку книга, которую вы читаете, о теории категорий для программистов, я буду иллюстрировать все основные понятия, используя компьютерные программы. Вы, наверное, знаете, что функциональные языки ближе к математике, чем более распространенные императивные языки. В них так же можно создавать более мощные абстракции. У меня было естественное искушение сказать: вы должны научиться Haskell, прежде чем теория категорий станет вам доступна, но это означало бы, что теория категорий не имеет никакого применения за пределами функци-

онального программирования, а это просто неправда. Так что, я приведу много примеров на C++. Конечно, придется преодолеть уродливый синтаксис, и увидеть шаблоны будет сложнее из-за многословия, а вам, возможно, придется делать много копирований-вставок, вместо использования абстракций высшего порядка, но это и так большая часть жизни C++-программиста.

Однако, не все так просто с Haskell. Не требуется становиться программистом на нем, но придется его освоить в качестве языка для зарисовок идей, которые позже будут реализованы на C++. Именно так я и начал программировать на Haskell. Его краткий синтаксис и мощная система типов очень помогли мне с пониманием и реализацией C++-шаблонов, структур данных и алгоритмов. Однако, так как я не могу ожидать, что читатели уже знают Haskell, я введу его постепенно и объясню все на ходу.

Если вы опытный программист, то можете парировать: я давно программирую и не беспокоюсь ни о какой теории категорий или функциональных методах, зачем же что-то менять? Конечно, вы не могли не заметить, что в императивных языках существует устойчивый тренд на добавление новых функциональных возможностей. Даже Java, бастион объектно-ориентированного программирования, добавила лямбды. C++ недавно начал развиваться бешеными темпами, — новые стандарты каждые несколько лет, — в попытке догнать меняющийся мир. Вся эта деятельность — подготовка к разрушительным изменениям или, как мы, физики, называем это — смена фазы. Если вы нагреваете воду, она, в конце концов, закипит. Сейчас мы находимся в положении лягушки, которая должна решить, продолжать ли плавание в нагревающейся воде, или начать искать альтернативы.

Одна из движущих сил для больших изменений — многоядерная ре-

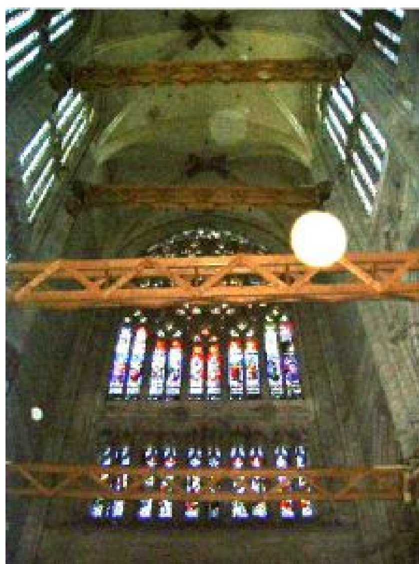
волюция в компьютерной архитектуре. Преобладающая парадигма программирования, — объектно-ориентированное программирование, не дает вам ничего в области параллелизма, а вместо этого поощряет опасный и подверженный ошибкам дизайн. Основные принципы объектной ориентированности: сокрытие, совместное использование и мутация данных — идеальная среда для гонок данных. Идея объединения данных с взаимной блокировкой, которая их защищает, хороша, но, к сожалению, блокировки не компонуемы, а сокрытие блокировок делает тупиковые ситуации более вероятными и менее отлаживаемыми.

Даже при отсутствии параллелизма, растущая сложность программных систем испытывает пределы масштабируемости императивного программирования. Проще говоря, побочные эффекты выходят из-под контроля. Функции с побочными эффектами легко и удобно писать. Побочные эффекты могут, в принципе, быть закодированы в их названиях и комментариях. Функция с названием `SetPassword` или `WriteFile`, очевидно, изменяет некое состояние и имеет побочные эффекты, но мы к этому привыкли. И только тогда, когда мы начинаем писать функции, имеющие побочные эффекты, работающие с другими функциями, имеющими побочные эффекты, и так далее, тогда подобные конструкции начинают становиться сложными. Не то чтобы побочные эффекты изначально плохи, плохо то, что они скрыты от наблюдения. Из-за этого, в более крупных масштабах, становится невозможно ими управлять. Побочные эффекты не масштабируемы, а императивное программирование полностью построено на побочных эффектах.

Изменения в железе и растущая сложность программного обеспечения заставляют нас переосмыслить основы программирования. Так же, как строители великих готических соборов Европы, мы, в нашем ремесле, подходим к пределам материалов и структуры. В Бове, во Франции, есть недостроенный готический собор[3], который является свидетелем этой человеческой борьбы с естественными ограничениями. Задумывалось, что он побьет все предыдущие рекорды высоты и легкости, но в процессе постройки произошел ряд обрушений. От полного разрушения его защищают «костыли»: железные прутья и деревянные опоры. С современной точки зрения, чудо, что так много готических сооружений было успешно завершено без помощи современного материаловедения, компьютерного моделирования, анализа методом конечных элементов и

[3] http://en.wikipedia.org/wiki/Beauvais_Cathedral

общей математики и физики. Я надеюсь, что будущие поколения будут так же восхищены навыками программирования, которые мы демонстрируем, строя сложные операционные системы, веб-сервера и интернет-инфраструктуру. И, честно говоря, мы сделали все это на очень хлипкой теоретической основе. Наша задача — улучшить эти основы, если мы хотим двигаться вперед.



«Костыли», защищающие собор в Бове от разрушения

Оглавление

От редактора перевода v

Предисловие **viii**

Часть I **1**

1 Категория: суть композиции **3**

1.1 Стрелки как функции 3

1.2 Свойства композиции 6

1.3 Композиция — суть программирования 8

2 Типы и функции **11**

2.1 Кому нужны типы? 11

2.2 Типы нужны для компонуемости 12

2.3 Что такое типы? 14

2.4 Зачем нам математическая модель? 16

2.5 Чистые и грязные функции 19

2.6 Примеры типов 19

3 Категории, большие и малые **25**

3.1 Без объектов 25

3.2 Простые графы 25
3.3 Порядки 26
3.4 Моноид как множество 27
3.5 Моноид как категория 31

4 Категории Клейсли 35
4.1 Композиция журналов 35
4.2 Категория Writer 39
4.3 Writer на Haskell 43
4.4 Категории Клейсли 45

5 Произведения и копроизведения 47
5.1 Инициальный объект 48
5.2 Терминальный объект 49
5.3 Двойственность 51
5.4 Изоморфизмы 51
5.5 Произведения 53
5.6 Копроизведения 58
5.7 Асимметрия 61

6 Простые алгебраические типы 65
6.1 Тип-произведение 65
6.2 Записи 69
6.3 Тип-сумма 71
6.4 Алгебра типов 75

7 Функторы 81
7.1 Функторы в программировании 84
7.2 Функторы как контейнеры 96
7.3 Композиция функторов 99

8 Функториальность **103**
8.1 Бифункторы 103
8.2 Произведение и копроизведение 106
8.3 Алгебраические типы данных 108
8.4 Функторы в C++ 111
8.5 Функтор Writer 114
8.6 Ко- и контра-вариантныефункторы 115
8.7 Профункторы 119
8.8 **hom**-функтор 121

9 Функциональные типы **123**
9.1 Универсальная конструкция 124
9.2 Карринг 129
9.3 Экспоненциалы 132
9.4 Декартово замкнутые категории 133
9.5 Экспоненциалы и АТД 134
9.6 Изоморфизм Карри-Говарда 137

10 Естественные преобразования **141**
10.1 Полиморфные функции 145
10.2 За пределами естественности 151
10.3 Категория функторов 153
10.4 2-категории 156
10.5 Заключение 160

Часть II **162**

11 Декларативное программирование **165**

12 Пределы и копределы **173**
12.1 Предел как естественный изоморфизм 178
12.2 Примеры пределов 183
12.3 Копределы 189
12.4 Непрерывность 190

13 Свободные моноиды **195**
13.1 Свободный моноид в Haskell 197
13.2 Универсальная конструкция 198

14 Представимые функторы **203**
14.1 hom-функтор 204
14.2 Представимые функторы 207

15 Лемма Йонеды **213**
15.1 Йонеда в Haskell 219
15.2 Ко-Йонеда 221

16 Вложение Йонеды **223**
16.1 Вложение 225
16.2 Применимость в Haskell 226
16.3 Пример с предпорядком 227
16.4 Естественность 229

Часть III **231**

17 Все о морфизмах **233**
17.1 Функторы 233
17.2 Коммутативные диаграммы 234

17.3 Естественные преобразования . . . 234
17.4 Естественные изоморфизмы . . . 236
17.5 hom-множества . . . 237
17.6 Изоморфизмы hom-множества . . . 237
17.7 Асимметрия hom-множеств . . . 238

18 Сопряжения **241**
18.1 Сопряжение и пара «единица/коединица» . . . 242
18.2 Сопряжения и hom-множества . . . 247
18.3 Произведение из сопряжения . . . 251
18.4 Экспоненциал из сопряжения . . . 255

19 Свободные/забывающие сопряжения **257**
19.1 Свободный моноид из сопряжения . . . 257
19.2 Немного интуиции . . . 260

20 Монады: определение программиста **265**
20.1 Категория Клейсли . . . 267
20.2 Анатомия >=> . . . 269
20.3 do-нотация . . . 272

21 Монады и эффекты **275**
21.1 Проблемы . . . 275
21.2 Решение . . . 276
21.3 Заключение . . . 290

22 Монады с точки зрения категорий **291**
22.1 Моноидальные категории . . . 295
22.2 Моноид в моноидальной категории . . . 300
22.3 Монады как моноиды . . . 302
22.4 Монады из сопряжений . . . 304

23 Комонады **307**
23.1 Программирование с помощью комонад 308
23.2 Комонада `Product` 309
23.3 Анализ композиции 310
23.4 Комонада `Stream` 312
23.5 Комонада с категорной точки зрения 314
23.6 Комонада `Store` 317

24 F-алгебры **321**
24.1 Рекурсия 325
24.2 Категория F-алгебр 327
24.3 Натуральные числа 330
24.4 Катаморфизмы 331
24.5 Свертки 333
24.6 Коалгебры 334

25 Алгебры для монад **339**
25.1 T-алгебры 341
25.2 Категория Клейсли 345
25.3 Коалгебры для комонад 347
25.4 Линзы 347

26 Концы и ко-концы **351**
26.1 Диестественные преобразования 353
26.2 Концы 354
26.3 Концы как уравнители 357
26.4 Естественные преобразования как концы 358
26.5 Ко-концы 360
26.6 Лемма ниндзя Йонеды 363
26.7 Композиция профункторов 364

27 Расширения Кана 367
27.1 Правое расширение Кана 369
27.2 Расширение Кана как сопряжение 371
27.3 Левое расширение Кана 373
27.4 Расширения Кана как концы 376
27.5 Расширения Кана на Haskell 379
27.6 Свободный функтор 381

28 Обогащенные категории 385
28.1 Почему моноидальная категория? 386
28.2 Моноидальная категория 387
28.3 Обогащенная категория 389
28.4 Предпорядки 391
28.5 Метрические пространства 392
28.6 Обогащенные функторы 393
28.7 Самообогащение 394
28.8 Связь с 2-категориями 396

29 Топосы 397
29.1 Классификатор подобъектов 398
29.2 Топос 402
29.3 Топосы и логика 403

30 Теории Ловера 405
30.1 Универсальная алгебра 405
30.2 Теории Ловера 407
30.3 Модели теорий Ловера 410
30.4 Теория моноидов 412
30.5 Теории Ловера и монады 413
30.6 Монады как ко-концы 416
30.7 Теория Ловера побочных эффектов 419

31 Монады, моноиды и категории **423**
31.1 Бикатегории 423
31.2 Монады 428

Литература **432**

Предметный указатель **437**

Благодарности **440**

Часть I

Глава 1

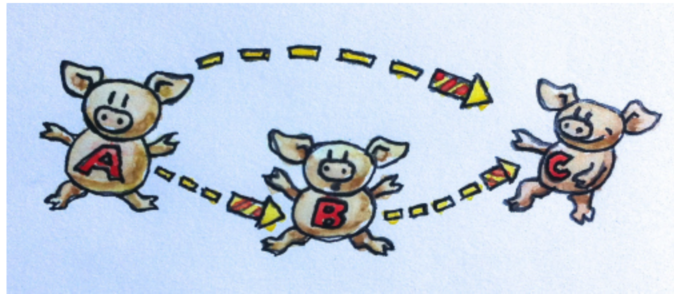
Категория: суть композиции

Категория — очень простая концепция.

Категория состоит из *объектов* и *морфизмов*, которые, как мостики, соединяют объекты. Поэтому категории так легко представить графически. Объект можно изобразить в виде круга или точки, а морфизмы — это просто *стрелки* между ними (для разнообразия, я буду время от времени изображать объекты в виде знакомых сущностей, а стрелки — как фейерверки). Но суть категории — композиция. Или, если вам больше нравится, суть композиции — категория. Стрелки компонуются так, что если имеется стрелка от объекта A к объекту B, и еще одна стрелка от объекта B к C, то должна быть и стрелка — их композиция, от A к C.

1.1 Стрелки как функции

Уже слишком много абстрактной ерунды? Не отчаивайтесь. Давайте посмотрим на примеры. Думайте о стрелках, которые еще называются морфизмами, как о функциях. У вас есть функция f, которая принимает аргумент типа A и возвращает значение типа B. Еще есть другая функция g, которая принимает B и возвращает C. Вы можете скомпоновать их, передавая результат из f в g. Таким образом, мы только что определили новую функцию, которая принимает A и возвращает C.



В категории, если есть стрелка от A к B и стрелка от B к C, то должна быть стрелка от A до C, которая называется их *композицией*. Эта схема — не полное определение категории потому, что не хватает тождественных морфизмов (см. ниже).

В математике, такая композиция обозначается небольшим кружком между обозначениями функций: $g \circ f$. Обратите внимание на порядок композиции, справа налево. Некоторых это сбивает с толку. Вы можете увидеть сходство с обозначениями пайпов в Unix, например:

```
lsof | grep Chrome
```

или с композицией `>>` в F#, и те и другие идут слева направо. Но в математике и в функциях Haskell композиция направлена справа налево. Это помогает, если воспринимать $g \circ f$ как «g после f».

Покажем это еще более явно кодом на языке C. Пусть имеется функция `f`, которая принимает аргумент типа `A` и возвращает значение типа `B`:

```
B f(A a);
```

и другая функция:

```
C g(B b);
```

Их комбинацией будет:

```
C g_after_f(A a)
{  return g(f(a));  }
```

Тут вы снова видите, теперь на C, композицию справа налево: `g(f(a))`.

Хотелось бы сказать, что существует шаблон в стандартной библиотеке C++, который принимает две функции и возвращает их композицию, но такого нет. Так что, попробуем Haskell, для разнообразия. Вот объявление функции (стрелки) от `A` к `B`:

```
f :: A -> B
```

Аналогично:

```
g :: B -> C
```

Их комозиция есть:

```
g . f
```

Как только вы видите, насколько просто это на Haskell, неспособность выразить простые функциональные концепции в C++ немного смущает. Haskell даже позволяет использовать Unicode символы, так что вы можете записать композицию так:

```
g ∘ f
```

или, даже можете использовать двойные двоеточия и стрелки из Unicode:

```
f ∷ A → B
```

Вот и первый Haskell-урок: двойное двоеточие означает «имеет тип ...». Сам тип функции создается путем расположения стрелки между двумя типами. Композиция двух функций записывается точкой между ними (или кружочком из Unicode).

1.2 Свойства композиции

Есть два очень важных свойства, которым композиция должны удовлетворять в любой категории.

1. Композиция ассоциативна. Если у вас есть три морфизма, f, g и h, которые могут быть скомпонованы (то есть, их типы согласованы друг с другом), вам не нужны скобки, чтобы составить их. Математически это записывается так:

   $$h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$$

   В (псевдо) Haskell:

   ```
   f :: A -> B
   g :: B -> C
   h :: C -> D
   h . (g . f) == (h . g) . f == h . g . f
   ```

   Здесь использовано «псевдо» потому, что сравнение не определено для функций. Ассоциативность довольно очевидна, когда речь идет о функциях, но может быть не так очевидна в других случаях.

2. Для каждого объекта A есть стрелка, которая будет единицей композиции. Эта стрелка от объекта к самому себе. Быть единицей композиции — означает, что при композиции единицы с любой стрелкой, которая, либо начинается на A, либо заканчивается на A соответственно, композиция возвращает ту же стрелку. Единичная стрелка объекта A обозначается id_A (*тождественность* на A). В математической нотации, если f направлена от A к B, то

   $$f \circ \mathrm{id}_A = f$$

   и

   $$\mathrm{id}_B \circ f = f$$

При работе с функциями, единичная стрелка реализована в виде тождественной функции, которая просто возвращает свой аргумент. Реализация одинакова для каждого типа, что означает, эта функция является универсально полиморфной. В C++ можно написать ее в виде шаблона:

```
template<class T> T id(T x) { return x; }
```

Примечание: по-хорошему, тождественную функцию стоит определить так:

```
template<class T> auto id(T&& x)
                  { return std::forward<T>(x); }
```

Конечно, в C++ ничто не бывает так просто, потому что вы должны принимать во внимание не только то, что вы передаете, но и как (то есть, по значению, по ссылке, по константной ссылке, по перемещению, а может быть, и еще как-то).

На Haskell, тождественная функция — часть стандартной библиотеки (называемой Prelude). Вот ее объявление и определение:

```
id  :: a -> a
id x = x
```

Можно увидеть, что полиморфные функции в Haskell очень просты. В декларации вы просто заменяете конкретный тип переменной типа. При этом нужно помнить, что имена конкретных типов всегда начинаются с заглавной буквы, имена ти́повых переменных — *переменных типа*, начинаются с маленькой буквы.

Определение функции Haskell состоит из имени функции с последующим формальным параметрам — здесь только один: `x`. Тело функции следует за знаком равенства. Эта лаконичность часто шокирует новичков, но вы быстро увидите, что это отличный синтаксис. Определение функции и ее вызов — основные инструменты функционального программирования, так что их синтаксис сведен к минимуму. Нет ни скобок вокруг аргументов, ни запятых между ними (вы увидите это позже, когда будут определять функции с несколькими аргументами).

Тело функции всегда выражение — нет никаких объявлений. Результат функции — это выражение — здесь, просто `x`.

На этом мы завершаем наш второй урок по Haskell.

Условия на композицию с единичной стрелкой могут быть записаны (опять же, на псевдо-Haskell), так:

```
f . id == f
id . f == f
```

Вы можете задавать себе вопрос: зачем кому-то возиться с тождественной функцей — функцией, которая ничего не делает? Опять же, почему мы возимся с числом ноль? Ноль — символ пустоты. Древние римляне не имели нуля в системе исчисления, и они строили отличные дороги и акведуки, некоторые из которых сохранились и по сей день.

Нейтральные значения, такие как ноль или id, крайне полезны при работе с символьными переменными. Тождественная функция очень удобна в качестве аргумента, или возвращаемого значения из функции высшего порядка. Функции высшего порядка — это то, что делает символьные преобразования функций возможными. Они — алгебра функций.

Подведем итог: категория состоит из объектов и стрелок (морфизмов). Стрелки могут быть скомпонованы, их композиция ассоциативна. Каждый объект имеет единичную стрелку, которая служит в качестве единицы композиции.

1.3 Композиция — суть программирования

Функциональные программисты имеют своеобразный способ подхода к проблемам. Они начинают с задания весьма дзен-подобных вопросов. Например, при проектировании интерактивной программы, они спросят: что такое интерактивность? При реализации игры Жизнь, они, вероятно, задумаются о смысле жизни. На этой волне я собираюсь спросить: что такое программирование? На самом базовом уровне, программирование — говорить компьютеру, что ему делать. «Возьмите содержимое памяти по адресу x и добавьте его к содержимому регистра EAX». Но, даже когда мы программируем на ассемблере, инструкции, которые мы даем компьютеру — выражение чего-то более важного. Мы решаем нетривиальную задачу (если бы она была тривиальной, мы не нуждались бы в помощи компьютера). И как мы решаем задачи? Раскладываем большие задачи на более мелкие. Если мелкие все еще слишком большие, то их тоже раскладываем, и так далее. Наконец, мы пишем код, который решает все мелкие задачи. И вот тут проявляется суть программирования:

мы составляем эти куски кода для создания решений более серьезных задач. Разложение не имело бы смысла, если мы не смогли сложить кусочки вместе.

Этот процесс иерархической декомпозиции и рекомпозиции не навязывается нам компьютером. Он отражает ограничения человеческого ума. Наш мозг может иметь дело только с небольшим количеством концепций одновременно. В одной из наиболее цитируемых работ в области психологии, «Магическое число семь плюс-минус два»[1], постулируется, что мы можем держать только 7 ± 2 «кусков» информации в наших умах. Детали нашего понимания кратковременной человеческой памяти могут меняться, но мы точно знаем, что она ограничена. Суть в том, что мы не в состоянии справиться с супом объектов или спагетти кода. Мы должны структурировать программы не потому, что так они приятны на вид, а потому, что в противном случае наши мозги не смогут их обработать. Мы часто называем кусок кода элегантным или красивым, но на самом деле мы имеем ввиду, что его легко понимать нашим ограниченным умом. Элегантный код состоит из кусков именно того, правильного, для усвоения с помощью наших умственных сил, размера.

Так какие куски правильны для составления программ? Площадь их поверхности, должна быть меньше, чем их объем (я люблю эту аналогию потому, что площадь поверхности геометрического объекта растет пропорционально квадрату ее размера, — но медленнее, чем объем, который растет пропорционально кубу ее размера). Площадь поверхности — это информация, которая нужна нам для того, чтобы комбинировать куски. Объем — это информация, которая нужна для того, чтобы их реализовать. Идея заключается в том, что, как только какой-то фрагмент будет реализован, мы можем забыть о деталях его реализации и сосредоточиться на том, как он взаимодействует с другими фрагментами. В объектно-ориентированном программировании, поверхность — это декларация класса или его абстрактный интерфейс. В функциональном программировании — это объявление функции. Я немного упрощаю, но суть именно в этом.

Теория категорий — крайний случай в том смысле, что она активно мешает нам заглянуть внутрь объектов. Объект в теории категорий яв-

[1] http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

ляется абстрактной туманной сущностью. Все, что вы можете знать о нем — это как он относится к другому объекту — как он компонуется с ними при помощью стрелок. Именно так поисковые машины ранжируют веб-сайты, анализируя входящие и исходящие ссылки (кроме случаев, когда они хитрят).

Примечание: на самом деле, не совсем так. В объектно-ориентированном программировании идеализированный объект виден только через абстрактный интерфейс (только поверхность, без объема), с методами, играющими роль стрелок. Как только вам нужно посмотреть реализацию объекта, чтобы понять, как компоновать его с другими объектами, вы теряете достоинства ООП.

Упражнения

1. Постарайтесь реализовать тождественную функцию в вашем основном языке программирования (или во втором по значимости, если основным языком яляется Haskell).

2. Реализуйте функцию композиции в вашем основном языке. Она должна принимать две функции в качестве аргументов и возвращать функцию, которая является их композицией.

3. Напишите программу, которая проверяет, что ваша функция композиции учитывает идентичность.

4. Является ли всемирная паутина категорией, в каком-либо смысле? Являются ли ссылки морфизмами?

5. Является ли Facebook категорией, с людьми как с объектами и их дружбой в качестве морфизмов?

6. Когда ориентированный граф является категорией?

Глава 2

Типы и функции

Категория типов и функций играет важную роль в программировании, так что давайте поговорим о том, что такое типы, и зачем они нам нужны.

2.1 Кому нужны типы?

В сообществе программистов есть некоторые разногласия о преимуществах статической типизации перед динамической и сильной типизации перед слабой. Позвольте мне проиллюстрировать выбор типизации с помощью мысленного эксперимента. Представьте себе миллионы обезьян с клавиатурами, радостно жмущих случайные клавиши, которые пишут, компилируют и запускают программы.

Любая комбинация байтов производимая обезьянами с машинным языком будет принята и запущена. Но в высокоуровневых языках высоко ценится то, что их компиляторы способны обнаружить лексические и грамматические ошибки. Многие программы будут просто отвергнуты, а обезьяны останутся без бананов, зато остальные будут иметь больше шансов быть осмысленными. Проверка типов обеспечивает еще один барьер против бессмысленных программ. Кроме того, в то время как в динамически типизированных языках несоответствия типов будут обнаружены только во время выполнения, в строго типизированных статически проверяемых языках несоответствия типов обнаруживаются во время компиляции, что отсеивает множество некорректных программ, прежде чем у них появится шанс быть запущенными.

Итак, вопрос в том, хотим ли мы, чтобы обезьяны были счастливы, или создавать корректные программы?

Обычно цель мысленного эксперимента с печатающими обезьянами — создание полного собрания сочинений Шекспира. Проверка орфографии и грамматики в цикле резко увеличит шансы на успех. Аналог проверки типов пойдет еще дальше: после того, как Ромео объявлен человеком, проверка типов убедится, что на нем не растут листья и что он не ловит фотоны своим мощным гравитационным полем.

2.2 Типы нужны для компонуемости

Теория категорий изучает композиции стрелок. Не любые две стрелки могут быть скомпонованы: целевой объект одной стрелки должен совпадать с исходным обьектом следующей. В программировании мы передаем результаты из одной функции в другую. Программа не будет работать, если вторая функция не может правильно интерпретировать данные, полученные с помощью первой. Обе функции должны подходить друг к другу, чтобы их композиция заработала. Чем сильнее система типов языка, тем лучше это соответствие можно описать и автоматически проверить.

Единственный серьезный аргумент, который я слышу против строгой статической типизации: она может отвергнуть некоторые программы, которые семантически верны. На практике это случается крайне редко и, в

любом случае, каждый язык содержит какой-то черный ход, чтобы обойти систему типов, когда это действительно необходимо. Даже Haskell имеет `unsafeCoerce`. Но такие конструкции должны использоваться разумно. Персонаж Франца Кафки, Грегор Замза, нарушает систему типов, когда он превращается в гигантского жука, и мы все знаем, чем это закончилось.

Другой аргумент, который я часто слышу, заключается в том, что строгая типизация накладывает слишком много нагрузки на программиста. Я могу посочувствовать этой проблеме, так как сам написал несколько обьявлений итераторов в C++, только вот имеется технология, *вывод типов*, которая позволяет компилятору вывести большинство типов из контекста, в котором они используются. В C++ вы можете объявить переменную `auto`, и компилятор выведет тип за вас.

В Haskell, за исключением редких случаев, аннотации типа являются опциональными. Программисты, как правило, все равно их используют, потому что типы могут многое рассказать о семантике кода, и обьявления типов помогают понимать ошибки компиляции. Обычная практика в Haskell — начинать проект с разработки типов. Позже, аннотации типов являются основой для реализации и становятся гарантированными компилятором комментариями.

Строгая статическая типизация часто используется в качестве предлога для нетестирования кода. Иногда можно услышать, как Haskell-программисты говорят: «Если код собирается, он правильный». Конечно, нет никакой гарантии, что программа, корректная с точки зрения типов, корректна в смысле правильного результата. В результате такого отношения в ряде исследований Haskell не стал сильно опережать остальные языки по качеству кода, как можно было бы ожидать. Кажется, что в коммерческих условиях необходимость чинить баги существует только до определенного уровня качества, что в основном связано с экономикой разработки программного обеспечения и толерантности конечного пользователя, и очень слабо связано с языком программирования или методологией разработки. Лучшим критерием было бы измерить, сколько проектов отстает от графика или поставляется с сильно сниженным функционалом.

Теперь, что касается утверждения, что модульное тестирование может заменить строгую типизацию. Рассмотрим общую практику рефакторин-

га в строго типизированных языках: изменение типа аргумента какой-либо функции. В сильно типизированных языках достаточно изменить декларацию этой функции, а затем исправить все ошибки сборки. В слабо типизированных языках, тот факт, что функция теперь ожидает другие данные, не может быть передан вызывающей стороне. Модульное тестирование может поймать некоторые из несоответствий, но тестирование практически всегда вероятностный, а не детерминированный процесс. Тестирование — плохая замена доказательству корректности.

2.3 Что такое типы?

Простейшее понимание типов: они представляют собой множества значений. Типу `Bool` (помните, конкретные типы начинаются с заглавной буквы на Haskell) соответствует множество из двух элементов: `True` и `False`. Тип `Char` — множество всех символов Unicode, например `'a'` или `'ą'`.

Множества могут быть конечными или бесконечными. Тип `String`, который, по сути, синонимом списка `Char`, — пример бесконечного множества.

Когда мы обьявляем `x`, как `Integer`:

```
x :: Integer
```

мы говорим, что это элемент множества целых чисел. `Integer` в Haskell — бесконечное множество, и может быть использовано для арифметики любой точности. Есть и конечное множество `Int`, которое соответствует машинному типу, как `int` в C++.

Есть некоторые тонкости, которые делают приравнивание типов к множествам сложным. Есть проблемы с полиморфными функциями, которые имеют цикличные определения, а также с тем, что вы не можете иметь множество всех множеств; но, как я и обещал, я не буду строгим математиком. Важно то, что имеется *категория множеств*, которая обозначается **Set**, и мы с ней будем работать. В **Set**, объекты — это множества, а морфизмы (стрелки) — функции.

Set — особая категория, потому что мы можем заглянуть внутрь ее объектов и это поможет интуитивно понять многое. Например, мы знаем, что пустое множество не имеет элементов. Мы знаем, что существуют специальные множества из одного элемента. Мы знаем, что функции отображают элементы одного множества в элементы другого. Они могут отображать два элемента в один, но не один элемент в два. Мы знаем, что тождественная функция отображает каждый элемент множества в себя, и так далее. Я планирую постепенно забывать всю эту информацию и вместо этого выразить все эти понятия в чисто категорийной форме, то есть в терминах объектов и стрелок.

В идеальном мире мы могли бы просто сказать, что типы в Haskell — множества, а функции в Haskell — математические функции между ними. Существует только одна маленькая проблема: математическая функция не выполняет какой-либо код — она знает только ответ. Функция в Haskell должна ответ вычислять. Это не проблема, если ответ может быть получен за конечное число шагов, каким бы большим оно ни было. Но есть некоторые вычисления, которые включают рекурсию, а те могут никогда не завершиться. Мы не можем просто запретить незавершающиеся функции в Haskell потому, что различить, завершается функция, или нет — знаменитая проблема остановки — неразрешима. Вот почему ученые-компьютерщики придумали гениальную идею, или грязный хак, в зависимости от вашей точки зрения, — расширить каждый тип специальным значением, названным *дном*, которое обозначается `_|_` или, в Unicode, как `⊥`. Это «значение» соответствует незавершающемуся вычислению. Так функция, объявленная как:

```
f :: Bool -> Bool
```

может вернуть `True`, `False`, или `_|_`; последнее значит, что функция никогда не завершается.

Интересно, что, как только вы включаете дно в систему типов, удобно рассматривать каждую ошибку времени исполнения как дно, и даже позволить функции возвращать дно явно. Последнее, как правило, осуществляется с помощью выражения `undefined`:

```
f  :: Bool -> Bool
f x = undefined
```

Это определение проходит проверку типов потому, что `undefined` вычисляется как дно, которое включено во все типы, в том числе и в `Bool`. Можно даже написать:

```
f :: Bool -> Bool
f  = undefined
```

(без `x`) потому, что дно еще и член типа `Bool -> Bool`.

Функции, которые могут возвращать дно, называются *частичными*, в отличие от обычных функций, которые возвращают правильные результаты для всех возможных аргументов.

Из-за дна, категория типов и функций Haskell обозначается **Hask**, а не **Set**. С теоретической точки зрения, это источник нескончаемых осложнений, поэтому на данном этапе я использую свой нож мясника и завершу эти рассуждения. С прагматической точки зрения можно игнорировать незавершающиеся функции (и дно) и работать с **Hask** как с полноценной категорией **Set**[1].

2.4 Зачем нам математическая модель?

Как программист, вы хорошо знакомы с синтаксисом и грамматикой языка программирования. Эти аспекты языка, как правило, формально описываются в самом начале спецификации языка. Но смысл и семантику языка гораздо труднее описать; это описание занимает намного больше страниц, редко достаточно формально, и почти никогда не полно. Отсюда никогда не заканчивающиеся дискуссии среди языковых юристов, и вся кустарная промышленность книг, посвященных толкованию тонкостей языковых стандартов.

Есть формальные средства для описания семантики языка, но из-за их сложности они в основном используются для упрощенных, академических языков, а не реальных гигантов промышленного программирова-

[1] Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons. *Fast and Loose Reasoning is Morally Correct*. В этой статье приводится обоснование игнорирования дна в большинстве контекстов.

ния. Один из таких инструментов называется *операционной семантикой* и описывает механику исполнения программы. Она определяет формализованный, идеализированный интерпретатор. Семантика промышленных языков, наподобие C++, как правило, описывается с помощью неформального рассуждения, часто в терминах «абстрактной машины».

Проблема в том, что о программах, использующих операционную семантику, очень трудно что-то доказать. Чтобы вывести некое свойство программы вы, по сути, должны «пропустить ее» через идеализированный интерпретатор.

Не важно, что программисты никогда формально не доказывают корректность. Мы всегда «думаем», что мы пишем правильные программы. Никто не сидит за клавиатурой, говоря: «О, я просто напишу несколько строк кода и посмотрю, что происходит». Мы считаем, что код, который мы пишем, будет выполнять определенные действия, которые произведут желаемые результаты. Мы, как правило, очень удивлены, если это не так. Это означает, что мы действительно думаем о программах, которые мы пишем, и мы, как правило, делаем это, запуская интерпретатор в наших головах. Просто, очень трудно уследить за всеми переменными. Компьютеры хороши для исполнения программ, люди — нет! Если бы мы были способны на это, нам бы не понадобились компьютеры.

Но есть и альтернатива. Она называется *денотационной семантикой* и основана на математике. В денотационной семантике для каждой языковой конструкции описывается математическая интерпретация. Таким образом, если вы хотите доказать свойство программы, вы просто доказываете математическую теорему. Вы думаете, что доказывание теорем трудно, но на самом деле мы, люди, строили математические методы тысячи лет, так что есть множество накопленных знаний, которые можно использовать. Кроме того, по сравнению с теоремами, которые доказывают профессиональные математики, задачи, с которыми мы сталкиваемся в программировании, как правило, довольно просты, если не тривиальны.

Рассмотрим определение функции факториала на Haskell, языке, легко поддающемуся денотационной семантике:

```
fact n = product [1..n]
```

Выражение [1..n] — это список целых чисел от 1 до n. Функция product перемножает все элементы списка. Точно так, как этого требует определение факториала, взятое из учебника. Сравните это с C:

```
int fact(int n)
{
   int i;
   int result = 1;
   for (i = 2; i <= n; ++i)
      result *= i;
   return result;
}
```

Нужно ли продолжать?

Хорошо, я сразу признаю, что это был дешевый прием! Факториал имеет очевидное математическое определение. Проницательный читатель может спросить: какова математическая модель для чтения символа с клавиатуры, или отправки пакета по сети? Долгое время это был бы неловкий вопрос, ведущий к довольно запутанным объяснениям. Казалось, денотационная семантика не подходит для значительного числа важных задач, которые необходимы для написания полезных программ, и которые могут быть легко решаемы операционной семантикой. Прорыву способствовала теория категорий. Еугенио Моджи обнаружил, что вычислительные эффекты могут быть преобразованы в монады. Это оказалось важным наблюдением, которое не только дало денотационной семантике новую жизнь и сделало чисто функциональные программы более удобными, но и дало новую информацию о традиционном программировании. Я буду говорить о монадах позже, когда мы освоим больше категорных инструментов.

Одним из важных преимуществ наличия математической модели для программирования является возможность выполнить формальное доказательство корректности программного обеспечения. Это может показаться не столь важным, когда вы пишете потребительский софт, но есть области программирования, где цена сбоя может быть огромной, или там, где человеческая жизнь находится под угрозой. Но даже при написании веб-приложений для системы здравоохранения, вы можете оценить то, что функции и алгоритмы из стандартной библиотеки языка Haskell идут в комплекте с доказательствами корректности.

2.5 Чистые и грязные функции

То, что мы называем функциями в C++ или любом другом императивном языке, не то же самое, что математики называют функциями. Математическая функция — просто отображение значений в значения.

Мы можем реализовать математическую функцию на языке программирования: такая функция, имея входное значение, будет рассчитать выходное значение. Функция для получения квадрата числа, вероятно, умножит входное значение само на себя. Она будет делать это при каждом вызове, и гарантированно произведет одинаковый результат каждый раз, когда она вызывается с одним и тем же аргументом. Квадрат числа не меняется при смене фаз Луны.

Кроме того, вычисление квадрата числа не должно иметь побочного эффекта, вроде выдачи вкусного ништячка вашей собаке. «Функция», которая это делает, не может быть легко смоделирована математической функцией.

В языках программирования функции, которые всегда дают одинаковый результат на одинаковых аргументах и не имеют побочных эффектов, называются *чистыми*. В чистом функциональном языке, наподобие Haskell, все функции чисты. Благодаря этому проще определить денотационную семантику этих языков и моделировать их с помощью теории категорий. Что касается других языков, то всегда можно ограничить себя чистым подмножеством, или размышлять о побочных эффектах отдельно. Позже мы увидим, как монады позволяют моделировать все виды эффектов, используя только чистые функции. В итоге мы ничего не теряем, ограничиваясь математическими функциями.

2.6 Примеры типов

Как только вы решите, что типы — это множества, вы можете придумать некоторые весьма экзотические примеры. Например, какой тип соответствует пустому множеству? Нет, это не `void` в C++, хотя этот тип обозначается в Haskell как `Void`. Это тип, который не наполнен ни одним значением (не населен). Вы можете определить функцию, которая принимает `Void`, но вы никогда не сможете ее вызвать. Чтобы ее

вызвать, вам придется обеспечить значение типа `Void`, а его там просто нет. Что касается того, что эта функция может вернуть — не существует никаких ограничений. Она может возвращать любой тип (хотя этого никогда не случится, потому что она не может быть вызвана). Другими словами, это функция, которая полиморфна по возвращаемому типу. Хаскеллеры назвали ее:

```
absurd :: Void -> a
```

(Помните, что `a` — это переменная типа, которая может быть любым типом.) Это название возникло случайно. Существует более глубокая интерпретация типов и функций с точки зрения логики под названием изоморфизм Карри-Говарда. Тип `Void` представляет неправдивость, а функция `absurd` — утверждение, что из лжи следует что угодно, как в латинской фразе «ex falso sequitur quodlibet».

Далее идет тип, соответствующий одноэлементному множеству. Это тип, который имеет только одно возможное значение. Это значение просто «имеется». Вы могли сразу его не признать, но это `void` в C++. Думайте о функциях *от* и *в* этот тип. Функция от `void` всегда может быть вызвана. Если это чистая функция, она всегда будет возвращать один и тот же результат. Вот пример такой функции:

```
int f44() { return 44; }
```

Можно подумать, что эта функция принимает «ничего», но, как мы только что видели, функция, которая принимает «ничего» не может быть вызвана, потому что нет никакого значения, представляющего тип «ничего». Итак, что же эта функция принимает? Концептуально, она принимает фиктивное значение, у которого есть только единственный экземпляр, так что мы можем явно его не указывать в коде. В Haskell, однако, есть символ этого значения: пустая пара скобок `()`. Таким образом, из-за забавного совпадения (или не совпадения?), вызов функции от `void` выглядит одинаково и на C++ и на Haskell. Кроме того, из-за поддержки Haskell лаконичности, тот же символ `()` используется и для типа, и для конструктора и для единственного значения, соответствующего одноэлементному множеству. Вот эта функция на Haskell:

```
f44    :: () -> Integer
f44 () = 44
```

Первая строка обьявляет, что `f44` преобразует тип `()`, названный «единица», в тип `Integer`. Вторая строка определяет, что `f44` с помощью сопоставления с образцом преобразует единственный конструктор для единицы, а именно `()` в число `44`. Вы вызываете эту функцию, предоставляя значение `()`:

```
f44 ()
```

Обратите внимание, что каждая функция от единицы эквивалентна выбору одного элемента из целевого типа (здесь, выбирается `Integer` `44`). На самом деле, вы можете думать о `f44`, как ином представлении числа `44`. Это пример того, как мы можем заменить прямое упоминание элементов множества на функцию (стрелку). Функции от единицы к некоторому типу A находятся во взаимно-однозначном соответствии с элементами множества A.

А как насчет функций, возвращающих `void`, или, на Haskell, возвращающих единицу? В C++ такие функции используются для побочных эффектов, но мы знаем, что такие функции — не настоящие, в математическом смысле этого слова. Чистая функция, которая возвращает единицу, ничего не делает: она отбрасывает свой аргумент.

Математически, функция от множества A к одноэлементному множеству отображает каждый элемент в единственный элемент этого множества. Для каждого A имеется ровно одна такая функция. Вот она для `Integer`:

```
fInt  :: Integer -> ()
fInt x = ()
```

Вы даете ей любое целое число, и она возвращает единицу. Следуя духу лаконичности, Haskell позволяет использовать символ подчеркивания в качестве аргумента, который отбрасывается. Таким образом, не нужно придумывать для него название. Код, приведенный выше, можно переписать в виде:

```
fInt  :: Integer -> ()
fInt _ = ()
```

Обратите внимание, что выполнение этой функции не только не зависит от значения, ей переданного, но и от типа аргумента.

Функции, которые могут быть определены одной и той же формулой для любого типа называются параметрически полиморфными. Вы можете реализовать целое семейство таких функций одним уравнением, используя параметр вместо конкретного типа. Как назвать полиморфную функцию от любого типа к единице? Конечно, мы назовем ее `unit`:

```
unit  :: a -> ()
unit _ = ()
```

В C++ вы бы реализовали ее так:

```
template<class T>
   void unit(T) {}
```

Далее в «типологии типов» можно рассмотреть набор из двух элементов. В C++ он обозначается `bool`, а в Haskell, что не удивительно, — `Bool`. Разница в том, что в C++ `bool` является встроенным типом, в то время как на Haskell он может быть определен следующим образом:

```
data Bool = True | False
```

(Читать это определение стоит так: `Bool` может быть или `True` или `False`.) В принципе, можно было бы описать этот тип и в C++:

```
enum bool { true, false };
```

Но в C++ `enum` на самом деле целое число. Можно использовать C++11 «`class enum`», но тогда пришлось бы уточнять значение именем класса: `bool::true` или `bool::false`, не говоря уже о необходимости включать соответствующий заголовок в каждом файле, который его использует.

Чистые функции из `Bool` просто выбирают два значения из целевого типа, одно, соответствующее `True`, а другое — `False`.

Функции, возвращающие `Bool`, называются *предикатами*. Например, библиотека `Data.Char` в Haskell содержит много предикатов, например `IsAlpha` или `isDigit`. В C++ есть похожая библиотека `<cctype>`, которая обьявляет, помимо прочего, функции `isalpha` и `isdigit`, но они возвращают `int`, а не булевое значение. Настоящие предикаты определены в `<locale>` и называются `ctype::is(alpha,c)` и `ctype::is(digit,c)`.

Упражнения

1. Определите функцию высшего порядка (или функциональный объект) `memoize` в вашем основном языке. Эта запоминающая функция принимает чистую функцию `f` в качестве аргумента и возвращает функцию, которая ведет себя почти так же, как `f`, за исключением того, что она вызывает исходную функцию только один раз для каждого значения аргумента, сохраняет результат у себя внутри, и возвращает этот сохраненный результат каждый раз, когда она вызывается с тем же самым значением аргумента. Вы можете описать запоминающую функцию, полученную из исходной, наблюдая за ее работой. Например, попробуйте создать запоминающую функцию из исходной, которая долго вычисляет свой результат. Вам придется некоторое время ждать результата при первом ее вызове, но при последующих вызовах, с теми же аргументами, вы должны немедленно получить тот же результат.

2. Попробуйте создать запоминающую функцию для функции из вашей стандартной библиотеки, которая обычно используется для получения случайных чисел. Она работает?

3. Большинство генераторов случайных чисел можно инициализировать начальным числом. Реализовать функцию, которая принимает начальное число, вызывает генератор случайных чисел с этим числом, и возвращает результат. Создайте из этой функции запоминающую функцию. Будет ли это работать?

4. Какие из следующих C++ функций являются чистыми? Попробуйте превратить их в запоминающие функции и понаблюдать, что происходит, когда вы вызываете их несколько раз: как функции с запоминанием и без.

   (a) Функция вычисления факториала из примера в тексте.

   (b) `std::getchar()`

   (c)
   ```
   bool f()
   {
   std::cout « "Hello!" « std::endl;
   return true;
   }
   ```

   (d)
   ```
   int f(int x)
   {
   static int y = 0;
   y += x;
   return y;
   }
   ```

5. Сколько имеется различных функций от `Bool` к `Bool`? Можете ли вы все их реализовать?

6. Изобразите схематично категорию, в которой объектами являются типы `Void`, `()` (единица) и `Bool`, со стрелками, соответствующими всем возможным функциям между этими типами. Пометьте стрелки именами функций.

Библиография

1. Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons, *Fast and Loose Reasoning is Morally Correct*[2] (в данной статье приводится обоснование игнорирования ⊥ в большинстве контекстов).

[2] https://www.cs.ox.ac.uk/jeremy.gibbons/publications/fast+loose.pdf

Глава 3

Категории, большие и малые

Понять пользу категорий можно, изучая различные примеры. Категории проявляются во всевозможных формах и размерах и часто возникают в самых неожиданных местах. Мы начнем с самых простых.

3.1 Без объектов

Самая простая категория — без объектов и, как следствие, без морфизмов. Это очень одинокая категория, сама по себе, но она важна в контексте других категорий, например, в категории всех категорий (да, такая есть). Если вы считаете, что пустое множество осмысленно, то почему бы не быть пустой категории?

3.2 Простые графы

Вы можете строить категории, просто соединяя объекты стрелками. Начните с любого ориентированного графа и превратите его в категорию, просто добавив больше стрелок. Во-первых, добавьте единичную стрелку в каждом узле. Потом, для любых двух стрелок, таких, что конец одной совпадает с началом другой (другими словами, любые две *компонуемые стрелки*), нужно добавить новую стрелку, которая будет их композицией. Каждый раз, когда вы добавляете новую стрелку, вам

нужно рассмотреть её композицию со всеми другими стрелками (кроме единичных). Таким путем может получиться бесконечное число стрелок, но это нормально.

На этот процесс можно смотреть как на создание категории, в которой есть объекты, один объект для каждой вершины графа, и все возможные цепочки компонуемых ребер в качестве морфизмов (вы можете воспринимать единичные морфизмы, как цепочки длины ноль.)

Такая категория называется *свободной категорией*, порожденной данным графом. Это пример свободной конструкции, процесса комплектования структуры путем расширения ее минимальным количеством элементов, с целью достижения соответствия законам этой конструкции (в данном случае, законам категории). Далее мы увидим больше примеров подобной конструкции.

3.3 Порядки

А теперь кое-что совсем другое! Категория, в которой морфизмы — это отношения между объектами: отношение меньше или равно. Давайте проверим, действительно ли она является категорией. У нас есть единичные морфизмы? Каждый объект меньше или равен самому себе! У нас есть композиция? Если $a \leqslant b$ и $b \leqslant c$, то $a \leqslant c$! Композиция ассоциативна? Да! Множество с таким отношением называется *предпорядком*, и мы показали, что предпорядок — это категория.

Можно также взять более сильную зависимость, которая удовлетворяет дополнительному условию, что если $a \leqslant b$ и $b \leqslant a$, то a должен быть такой же, как и b. Это называется *частичным порядком*.

Наконец, можно наложить условие, что любые два объекта связаны друг с другом, тогда получится *линейный* или *полный порядок*.

Охарактеризуем эти упорядоченные множества как категории. Предпорядок — это категория, где есть не более одного морфизма от любого объекта a к любому объекту b. Другое название для такой категории — *тонкая категория*. Предпорядок — это тонкая категория.

Множество морфизмов от объекта a к объекту b в категории $\mathbf{C}$ называется hom-*множеством* и записывается как $\mathbf{C}(a, b)$ (или, иногда,

$\mathbf{Hom_C}(a, b)$). Таким образом, каждое hom-множество в предпорядке либо пустое, либо одноэлементное, в том числе и hom-множество $\mathbf{C}(a, a)$, множество морфизмов от a к самому себе, которое должно быть одноэлементным и содержать только единичный морфизм. В предпорядке можно иметь циклы, в частичном же порядке они запрещены.

Очень важно уметь распознавать предпорядки, частичные и полные порядки при сортировке. Алгоритмы сортировки, например, быстрая сортировка, пузырьковая сортировка, сортировка слиянием, и т.д., могут работать корректно только на полных порядках. Частичные порядки могут быть отсортированы с помощью топологической сортировки.

3.4 Моноид как множество

Моноид— удивительно простая, но удивительно мощная концепция. Эта концепция лежит в основе арифметики: и сложение и умножение образуют моноид. Моноиды широко распространены в программировании. Они появляются в виде строк, списков, сворачиваемых структур данных, фьючерсов в параллельном программировании, событий в функциональном реактивном программирования и т.д.

Традиционно, моноид определяется как множество с бинарной операцией. Все, что требуется от этой операции — её ассоциативность и наличие единственного специального элемента, который ведет себя как единица по отношению к этой операции.

Например, натуральные числа со сложением и нулем образуют моноид. Ассоциативность означает, что:

$$(a + b) + c = a + (b + c)$$

(Другими словами, мы можем опустить скобки при сложении чисел.)

Нейтральный элемент — ноль потому, что:

$$0 + a = a$$

и

$$a + 0 = a$$

Вторая формула являетя излишней потому, что сложение коммутативно ($a + b = b + a$), но коммутативность не является частью определения моноида. Например, конкатенация не является коммутативной, но она образует моноид. Нейтральным элементом для конкатенации строк, кстати, будет пустая строка, которая может быть присоединена с любой стороны произвольной строки без ее изменения.

На Haskell мы можем определить класс типов для моноидов — тип, для которого существует нейтральный элемент, называемый `mempty`, и бинарная операция с названием `mappend`:

```
class Monoid m where
   mempty  :: m
   mappend :: m -> m -> m
```

Сигнатура типа для функции двух аргументов, `m -> m -> m`, на первый взгляд, может выглядеть странной, но она станет понятной после того, как мы поговорим о каррировании. Вы можете интерпретировать сигнатуру с несколькими стрелками двумя основными способами: как функцию с несколькими аргументами и с крайним правым типом в качестве возвращаемого, или как функцию одного аргумента (самого левого), возвращающую функцию. Последнюю интерпретацию можно подчеркнуть при помощи скобок (которые являются избыточными, так как стрелка правоассоциативна), например: `m -> (m -> m)`. Мы вернемся к этой интерпретации чуть позже.

Обратите внимание, что в Haskell нет никакого способа выразить моноидальные свойства `mempty` и `mappend` (то есть то, что `mempty` нейтральна и что `mappend` ассоциативна). Ответственность за это лежит на программисте.

Классы Haskell не так навязчивы, как классы C++. Когда вы определяете новый тип, вы не должны указывать его класс заранее. Вы можете полениться и объявить то, что данный тип будет экземпляром некоторого класса, гораздо позже. В качестве примера, давайте объявим, что тип `String` — моноид, предоставив реализацию `mempty` и `mappend` (это, по сути, сделали за вас в стандартном Prelude):

```
instance Monoid String where
```

```
    mempty  = ""
    mappend = (++)
```

Здесь мы снова использовали оператор конкатенации списков (++) потому, что строка, на самом деле, — список символов.

Немного о синтаксисе Haskell: любой инфиксный оператор может быть превращен в двухаргументную функцию с помощью заключения его в скобки. Имея две строки, вы можете объединить их, вставив между ними ++:

```
"Hello " ++ "world!"
```

или передав их в качестве двух аргументов в функцию (++):

```
(++) "Hello " "world!"
```

Заметьте, что аргументы функции не разделены запятыми и не окружены скобками (это, наверное, самое сложное, к чему нужно привыкнуть во время изучения Haskell).

Стоит заметить, что Haskell дает возможность выразить равенство функций напрямую:

```
mappend = (++)
```

Концептуально, это не то же самое, что и равенство значений, возвращаемых функцией:

```
mappend s1 s2 = (++) s1 s2
```

Первое представляет равенство морфизмов в категории **Hask** (или **Set**, если игнорировать дно). Такие формулы не только более емкие, но и часто обобщаемы и на другие категории. Второе называется *экстенсиональным равенством*, и заявляет тот факт, что для любых двух входных строк, результаты mappend и (++) равны. Так как значения аргументов иногда называют *точками* (например: значение f в точке x), то запись

в форме экстенсионального равенства называется *точечной нотацией*. Функциональное равенство без указания аргументов называется *бесточечной нотацией* (кстати, формулы в бесточечной нотации часто включают композицию функций, которую обозначают точкой между именами функций, так что новичкам такое название может казаться странным).

Ближе всего к объявлению моноида в C++ было бы использование синтаксиса понятий (концептов) стандарта C++20.

```
template<class T>
struct mempty;

template<class T>
T mappend(T, T) = delete;

template<class M>
concept Monoid = requires (M m)
{
   { mempty<M>::value() } -> std::same_as<M>;
   { mappend(m, m) }      -> std::same_as<M>;
};
```

Первое определение представляет собой структуру, предназначенную для хранения нейтрального элемента для каждой специализации.

Ключевое слово `delete` означает, что значение по умолчанию не определено: они должны быть указаны индивидуально для каждого случая. Так же, нет реализации по умолчанию для `mappend`.

Концепт `Monoid` проверяет, существуют ли подходящие определения `mempty` и `mappend` для данного типа `M`.

Реализация концепта `Monoid` может быть выполнена путем предоставления соответствующих специализаций и перегрузок:

```
template<>
struct mempty<std::string>
{
   static std::string
```

```
                    value() { return ""; }
}
template<>
std::string mappend(std::string s1,
                    std::string s2)
{
   return s1 + s2;
}
```

3.5 Моноид как категория

Это было «знакомое» определение моноида в терминах элементов множества. Но, как вы уже знаете, в теории категорий мы пытаемся уйти от множеств и их элементов, а вместо этого говорить об объектах и морфизмах. Так давайте немного изменим наш ход мысли и подумаем о применении бинарного оператора, как о «перемешивании» или «сдвиге» элементов внутри множества.

Например, существует операция добавления 5 к любому натуральному числу. Она отображает 0 в 5, 1 в 6, 2 в 7, и так далее. Это функция, определенная на множестве натуральных чисел. Это хорошо: у нас есть функция и множество. В общем, для любого числа n есть функция добавления n — «сумматор» n.

Как эти сумматоры компоновать? Композиция функции, которая добавляет 5, с функцией, которая добавляет 7, является функцией, которая добавляет 12. Таким образом, композиция сумматоров удовлетворяет правилам, эквивалентным правилам сложения. Это тоже хорошо: мы можем заменить сложение композицией функций.

Но это еще не все: существует также сумматор для нейтрального элемента, нулевой. Добавление нуля не изменяет элементы, так что это единичная функция в множестве натуральных чисел.

Вместо того, чтобы предоставлять традиционные правила сложения, я мог бы дать вам правила композиции сумматоров без потери информации. Обратите внимание, что композиция сумматоров ассоциативна,

поскольку композиция функций ассоциативна, и у нас есть нулевой сумматор, соответствующий единичной функции.

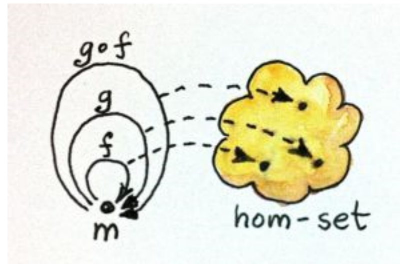
Проницательный читатель мог заметить, что отображение целых чисел в сумматоры следует из второго толкования сигнатуры типа `mappend` как `m -> (m -> m)`. Тип говорит нам о том, что `mappend` отображает элемент множества моноида в функцию, действующую на этом множестве.

Теперь я хочу, чтобы вы забыли, что имеете дело с множеством натуральных чисел и просто думать о нем, как о целостном объекте с совокупностью морфизмов — сумматоров. Моноид — это категория одного объекта. На самом деле, название моноид происходит от греческого моно — значит, одинокий. Каждый моноид может быть описан, как категория с одним объектом и набором морфизмов, которые следуют соответствующим правилам композиции.

Конкатенация строк интересна тем, что у нас есть выбор правого или левого добавления. Таблицы композиций этих моделей зеркально обратны друг другу. Можно легко убедиться, что добавлению справа «bar» к «foo» соответствует добавление слева «rab» к «oof».

Вы можете задать вопрос: каждый ли категориальный моноид — категория одного объекта — определяет единственное множество с бинарным оператором. Оказывается, мы всегда можем извлечь множество из категории одного объекта. Это будет множество морфизмов — сумматоров в нашем примере. Другими словами, мы имеем hom-множество $\mathbf{M}(m, m)$

одного объекта m в категории $\mathbf{M}$. Мы можем легко определить бинарный оператор в этом множестве: моноидальное произведение двух элементов этого множества — это элемент, соответствующий композиции соответствующих им морфизмов. Если вы предоставите два элемента $\mathbf{M}(m, m)$, соответствующие морфизмам f и g, их произведение будет соответствовать композиции $g \circ f$. Композиция всегда существует потому, что источник и результат этих морфизмов — один и тот же объект. И она ассоциативна по правилам категории. Тождественный морфизм является нейтральным элементом этого произведения. Таким образом, мы всегда можем восстановить моноид-множество из категориального моноида. Они, практически, одно и то же.



Моноидное $\mathbf{hom}$-множество рассматривается,
и как морфизмы, и как точки в множестве.

Существует только одна маленькая проблема для математиков: морфизмы не обязаны формировать множество. В мире категорий есть совокупности, большие, чем множества. Категория, в которой морфизмы между любыми двумя объектами образуют множество, называется *локально малой*. Как и было обещано, я буду игнорировать такие тонкости, но я решил, что нужно этот факт упомянуть.

Много интересных явлений в теории категорий происходит из за того, что элементы $\mathbf{hom}$-множества можно рассматривать и как морфизмы, которые следуют правилам композиции, и как элементы множества. Здесь композиция морфизмов в $\mathbf{M}$ соответствует моноидальному произведению в множестве $\mathbf{M}(m, m)$.

Упражнения

1. Сформируйте свободную категорию из:
   (a) графа с одной вершиной без ребер;
   (b) графа с одной вершиной и одним (направленным) ребром (подсказка: это ребро может компоноваться само с собой);
   (c) графа с двумя вершинами и одной стрелкой между ними;
   (d) графа с одной вершиной и 26 стрелками, помеченными буквами алфавита: a, b, c ... z;
2. Какого рода порядок в следующих случаях?
   (a) множество множеств с отношением включения: A содержится в B, если каждый элемент множества A является также и элементом множества B;
   (b) типы C++ со следующими отношением субтипирования: `T1` является подтипом `T2`, если указатель на `T1` может быть передан в функцию, которая ожидает указатель на `T2`, не вызывая ошибку компиляции.
3. Учитывая, что `Bool` представляет собой множество из двух значений `True` и `False`, покажите, что оно образует два (теоретико-множественных) моноида по отношению к, соответственно, оператору `&&` (И) и оператору `||` (ИЛИ).
4. Постройте представление моноида `Bool` с оператором `&&` как категорию: составьте список морфизмов и правила их композиции.
5. Постройте представление сложения по модулю 3 как моноидальную категорию.

Глава 4

Категории Клейсли

4.1 Композиция журналов

Вы видели, как моделировать типы и чистые функции в качестве категории. Также упоминалось, что существует способ моделирования побочных эффектов или нечистых функций, в рамках теории категорий. Давайте рассмотрим один такой пример: функции, которые регистрируют или отслеживают ход своего выполнения. Это то, что в императивном языке, скорее всего, будет реализовано путем изменения некоторого глобального состояния, например:

```
string logger;
bool negate(bool b)
{
   logger += "Not so! ";
   return !b;
}
```

Вы понимаете, что это не чистая функция, потому что ее запоминающая версия (с использованием функции `memoize` из упр.1 главы 2) не сможет создать журнал. Эта функция имеет *побочные эффекты*.

В современном программировании, мы стараемся держаться подальше от глобального изменяемого состояния, насколько это возможно — как минимум, из-за сложностей с параллелизмом. И вы никогда не размещаете подобный код в библиотеке.

К счастью для нас, можно сделать эту функцию чистой. Нужно просто передать журнал в явном виде в функцию и из нее. Давайте сделаем это путем добавления строкового аргумента, и возвращением пары из основного результата и строки, содержащей обновленный журнал:

```
pair<bool, string> negate(bool b, string logger)
{
   return make_pair(!b, logger + "Not so! ");
}
```

Эта функция чистая, она не имеет никаких побочных эффектов, она возвращает одну и ту же пару каждый раз, когда ее вызывают с одними и теми же аргументами, и ее результаты можно закешировать, если нужно. Однако, учитывая накопительный характер журнала, вам придется закешировать все возможные истории, которые могут привести к данному вызову. Таким образом, будут закешированы, например, записи:

```
negate(true, "It was the best of times. ");
```

и

```
negate(true, "It was the worst of times. ");
```

и так далее.

К тому же, это не очень хороший интерфейс для библиотечной функции. Пользователи могут свободно игнорировать строку в возвращаемом значении, так что в этом случае не возникает сложностей; но они вынуждены передавать строку в качестве входных данных, что может быть обременительно.

Есть ли способ сделать то же самое менее навязчиво? Есть ли способ разрулить ситуацию? В этом простом примере, основная цель функции `negate` — превратить одно логическое значение в другое. Ведение журнала вторично. Конечно, сообщение, которое регистрируется, является специфичным для функции, но агрегирование сообщений в один непрерывный журнал — отдельная задача. Мы все еще хотим, чтобы функция создавала строку, но мы хотели бы освободить ее от создания журнала. Вот компромиссное решение:

```
pair<bool, string> negate(bool b)
{
   return make_pair(!b, "Not so! ");
}
```

Идея в том, что журнал будет агрегироваться между вызовами функции.

Чтобы увидеть, как это можно сделать, давайте переключимся на чуть более реалистичный пример. У нас имеется одна функция преобразования строки, которая переводит строчные буквы в верхний регистр:

```
string toUpper(string s)
{
   string result;
   int (*toupperp)(int) = &toupper;
                          // toupper перегру-
жен
   transform(begin(s), end(s),
             back_inserter(result), toupperp);
   return result;
}
```

и другая, которая разбивает строку на вектор строк, разделяя ее по пробелам:

```
vector<string> toWords(string s)
                          { return words(s); }
```

Основная работа выполняется во вспомогательной функции `words`:

```
vector<string> words(string s)
{
   vector<string> result{""};
   for (auto i = begin(s); i != end(s); ++i)
      if (isspace(*i))
         result.push_back("");
      else
         result.back() += *i;
   return result;
}
```

Мы хотим изменить функции toUpper и toWords так, чтобы они добавляли строку сообщения поверх их обычных возвращаемых значений.

Мы «обогатим» возвращаемые значения этих функций. Давайте сделаем это в общем виде путем определения шаблона Writer, который инкапсулирует пару, первый компонент которой — значение произвольного типа A, а второй — строка:

```
template<class A>
   using Writer = pair<A, string>;
```

Вот обогащенные функции:

```
Writer<string> toUpper(string s)
{
   string result;
   int (*toupperp)(int) = &toupper;
   transform(begin(s), end(s),
             back_inserter(result), toupperp);
   return make_pair(result, "toUpper ");
}
Writer<vector<string>> toWords(string s)
{
   return make_pair(words(s), "toWords ");
}
```

Мы хотим скомпоновать эти функции в другую обогащенную функцию, которая переводит символы строки в верхний регистр и разделяет ее на слова, фиксируя при этом эти действия в журнале. Вот как мы можем это сделать:

```
Writer<vector<string>> process(string s)
{
   auto p1 = toUpper(s);
   auto p2 = toWords(p1.first);
   return make_pair(p2.first, p1.second +
                              p2.second);
}
```

Мы добились своей цели: агрегирование журнала больше не является заботой отдельных функций. Они производят свои собственные сообщения, которые затем, внешне, объединяются в больший журнал.

А теперь представьте себе всю программу, написанную в этом стиле. Это кошмар повторяющегося и подверженного ошибкам кода. Но мы программисты. Мы знаем, как бороться с повторяющимся кодом: мы его абстрагируем! Это, однако, не ваши привычные механические абстракции: мы должны абстрагировать саму композицию функций. Но композиция — суть теории категорий, поэтому перед тем, как писать код дальше, давайте проанализируем эту проблему с категорной точки зрения.

4.2 Категория Writer

Идея обогатить типы возвращаемых значений функций для того, чтобы привнести некоторую дополнительную функциональность, оказывается очень плодотворной. Мы увидим еще много примеров такого подхода. Отправной точкой является наша обычная категория типов и функций. Мы оставим типы в качестве объектов, но переопределим морфизмы, превратив их в обогащенные функции.

Например, предположим, что мы хотим обогатить функцию `isEven`, которая преобразует `int` в `bool`. Мы превратим ее в морфизм, который представлен обогащенной функцией. Важно то, что этот морфизм до сих пор считается стрелкой между объектами `int` и `bool`, хотя обогащенная функция возвращает пару:

```
pair<bool, string> isEven(int n)
{ return make_pair(n % 2 == 0, "isEven "); }
```

По законам категории, мы должны быть в состоянии скомпоновать этот морфизм с другим морфизмом, который идет от `bool` к чему угодно. В частности, мы должны быть в состоянии объединять его с нашей предыдущей функцией `negate`:

```
pair<bool, string> negate(bool b)
{ return make_pair(!b, "Not so! "); }
```

Очевидно, что мы не можем составить эти два морфизма так же, как мы составляем обычные функции, из-за несоответствия входа/выхода. Их композиция должна выглядеть примерно так:

```
pair<bool, string> isOdd(int n)
{
   pair<bool, string> p1 = isEven(n);
   pair<bool, string> p2 = negate(p1.first);
   return make_pair(p2.first, p1.second +
                              p2.second);
}
```

Итак, вот рецепт для композиции двух морфизмов в этой новой категории, которую мы конструируем:

1. Выполните обогащенную функцию, соответствующую первому морфизму.
2. Извлеките первый компонент пары-результата и передайте его в обогащенную функцию, соответствующую второму морфизму.
3. Соедините второй компонент (строку) первого результата и второй компонент (тоже строку) второго результата.
4. Верните новую пару, объединяющую первый компонент конечного результата с конкатенированной строкой.

Если мы хотим абстрагировать эту композицию в виде функции высшего порядка в C++, мы должны использовать шаблон, параметризованный тремя типами, соответствующими трем объектам в нашей категории. Следует взять две обогащенные функции, которые компонуемы в соответствии с нашими правилами, и вернуть третью обогащенную функцию:

```
template<class A, class B, class C>
   function<Writer<C>(A)>
               compose(function<Writer<B>(A)> m1,
                       function<Writer<C>(B)> m2)
{
   return [m1, m2](A x) { auto p1 = m1(x);
   auto p2 = m2(p1.first);
   return make_pair(p2.first, p1.second +
                              p2.second); };
}
```

Теперь мы можем вернуться к нашему примеру и реализовать композицию `toUpper` и `toWords` с помощью этого шаблона:

```
Writer<vector<string>> process(string s)
{
   return compose<string, string, vector<string>>
                            (toUpper, toWords)(s);
}
```

Тут все еще много избыточности с передачей типов в шаблон `compose`. Этого можно избежать, если у вас есть C++14-совместимый компилятор, который поддерживает обобщенные лямбда-функции с выводом возвращаемого типа (спасибо Эрику Ниблеру за код):

```
auto const compose = [](auto m1, auto m2)
{
   return [m1, m2](auto x)
   {
      auto p1 = m1(x);
      auto p2 = m2(p1.first);
      return make_pair(p2.first, p1.second +
                                 p2.second);
   };
};
```

В этом новом определении, реализация `process` упрощается:

```
Writer<vector<string>> process(string s)
{ return compose(toUpper, toWords)(s); }
```

Но мы еще не закончили. Мы определили композицию в нашей новой категории, но каковы единичные морфизмы? Это не наши регулярные тождественные функции! Они должны быть морфизмами от типа `A` обратно к типу `A`, значит, они — обогащенные функции вида:

```
Writer<A> identity(A);
```

Они должны вести себя как единицы по отношению к композиции. Если вы посмотрите на наше определение композиции, то увидите, что тождественный морфизм должен передать свой аргумент без изменения и только добавить пустую строку в журнал:

```
template<class A> Writer<A> identity(A x)
{ return make_pair(x, ""); }
```

Легко убедиться, что категория, которую мы только что определили, действительно законная категория. В частности, наша композиция тривиально ассоциативна. Если вы проследите, что происходит с первым компонентом каждой пары, вы увидите, что это просто обычная композиция функций, которая ассоциативна. Вторые компоненты соединяются как строки, а конкатенация также ассоциативна.

Проницательный читатель может заметить, что можно легко обобщить эту конструкцию на любой моноид, а не только строки. Мы хотели бы использовать `mappend` внутри `compose` и `mempty` внутри `identity` (вместо `+` и `""`). Действительно, нет никаких оснований ограничивать себя фиксированием в журнале только строк. Хороший создатель библиотек должен быть в состоянии определить необходимый минимум ограничений, которые необходимы библиотеке для работы — тут единственное требование к библиотеке работы с журналом заключается в том, что у журнала обладает моноидальными свойствами.

4.3 Writer на Haskell

Аналогичная конструкция на Haskell будет намного более лаконична и компилятор нам поможет намного больше. Давайте начнем с определения типа `Writer`:

```
type Writer a = (a, String)
```

Здесь я просто определяю псевдоним типа, эквивалент `typedef` (или `using`) в C++. Тип `Writer` параметризирован переменной типа и эквивалентен паре `a` и `String`. Синтаксис для пар минимален: всего два имени в скобках, через запятую.

Наши морфизмы — функции от произвольного типа к определенному типу `Writer`:

```
a -> Writer b
```

Мы объявим композицию как забавный инфиксный оператор `>=>`, который иногда называют «рыба»:

```
(>=>) :: (a -> Writer b) -> (b -> Writer c)
                         -> (a -> Writer c)
```

Это функция от двух аргументов, каждый из которых сам по себе функция, и она возвращает функцию. Первый аргумент имеет тип `(a -> Writer b)`, второй — `(b -> Writer c)`, а результат — `(a -> Writer c)`.

Вот определение этого инфиксного оператора — два аргумента `m1` и `m2`, появляются по обе стороны от рыбного символа:

```
m1 >=> m2 = \x ->
   let (y, s1) = m1 x
       (z, s2) = m2 y
   in  (z, s1 ++ s2)
```

В результате получается лямбда функция одного аргумента `x`. Лямбда записывается в виде обратной косой черты — можно думать о ней, как о греческой букве λ с ампутированной ногой.

Слово `let` позволяет объявить вспомогательные переменные. Здесь результат вызова `m1` представлен шаблоном, который соответствует паре возвращаемых переменных `(y,s1)`, а результат вызова `m2` с аргументом `y` из этого шаблона соответствует шаблону `(z,s2)`.

В Haskell подобные шаблоны — обычная альтернатива использованию аксессоров в C++. Помимо этого имеется довольно простое соответствие между этими двумя реализациями.

Значение `let` выражения содержится после `in`: здесь это пара, чей первый компонент `z`, а второй компонент — объединение двух строк, `s1 ++ s2`.

Я также определю тождественный морфизм для нашей категории, но по причинам, которые станут ясны значительно позже, я обозначу его `return`:

```
return  :: a -> Writer a
return x = (x, "")
```

Для полноты, запишем Haskell-версии обогащенных функций `upCase` (примечание: имеется в виду `toUpper` из C++ примера, но функция с таким именем уже есть в стандартном модуле Prelude) и `toWords`:

```
upCase   :: String -> Writer String
upCase s  = (map toUpper s, "upCase ")

toWords  :: String -> Writer [String]
toWords s = (words s, "toWords ")
```

Функция `map` соответствует функции `transform` в C++. Она применяет функцию на символах `toUpper` к строке `s`. Вспомогательная функция `words` определена в стандартной библиотеке Prelude.

Наконец, композиция этих двух функций строится с помощью оператора рыбы:

```
process :: String -> Writer [String]
process  = upCase >=> toWords
```

4.4 Категории Клейсли

Вы, наверное, догадались, что я не придумал эту категорию на лету. Это пример так называемой категории Клейсли — категории на основе монады. Мы пока не готовы обсуждать монады, но я хотел показать, что они могут делать. Для наших ограниченных целей, категория Клейсли имеет типы, как объекты. Морфизмы от типа A к типу B — это функции, которые идут от A к типу, полученному из B с помощью особого обогащения. Каждая категория Клейсли определяет свой собственный способ композиции таких морфизмов, а также тождественные морфизмы по отношению к этой композиции (позже мы увидим, что неточный термин «обогащение» соответствует понятию эндофунктора в категории).

Конкретная монада, которую я использовал здесь в качестве основы категории, обозначается **Writer**, и она используется для записи в журнал или отслеживания выполнения функций. Она также является примером более общего механизма для встраивания эффектов в чистые вычисления. Вы видели ранее, что мы могли бы моделировать типы языка программирования и функции в категории множеств (без учета дна, как обычно). Здесь мы расширили эту модель до несколько иной категории, категории, где морфизмы представлены обогащенными функциями, и их композиция делает больше, чем просто передает результат одной функции на вход другой. У нас на одну степень свободы больше: можно изменять саму композицию. Оказывается, что именно эта степень свободы позволяет дать простую денотационную семантику программам, которые в императивных языках традиционно реализованы с использованием побочных эффектов.

Упражнения

Функция, которая не определена для всех возможных значений аргумента называется частичной функцией. На самом деле это не функция в математическом смысле, так что она не соответствует стандартной категорной форме. Однако, она может быть представлена в виде функции, которая возвращает обогащенный тип `optional`:

```
template<class A> class optional
```

```
{
   bool _isValid;
   A _value;
   public:
      optional() : _isValid(false) {}
      optional(A v) : _isValid(true),
                      _value(v) {}
      bool isValid() const { return _isValid; }
      A value() const { return _value; }
};
```

В качестве примера, реализация обогащенной функции `safe_root`:

```
optional<double> safe_root(double x)
{
   if (x >= 0)
      return optional<double>{sqrt(x)};
   else
      return optional<double>{};
}
```

А теперь собственно упражнения:

1. Постройте категорию Клейсли для частичных функций (определите композицию и единицу).
2. Реализуйте обогащенную функцию `safe_reciprocal`, которая возвращает допустимое обращение своего аргумента, если он отличен от нуля.
3. Скомпонуйте функции `safe_root` и `safe_reciprocal` для реализации функции `safe_root_reciprocal`, которая вычисляет `SQRT(1/x)` всегда, когда это возможно.

Глава 5

Произведения и копроизведения

Древнегреческий драматург Еврипид писал «Всякий человек подобен своему окружению». Это верно и для теории категорий. Выделить определенный объект категории можно только путем описания характера его взаимоотношений с другими объектами (и самим собой), где отношения — это морфизмы.

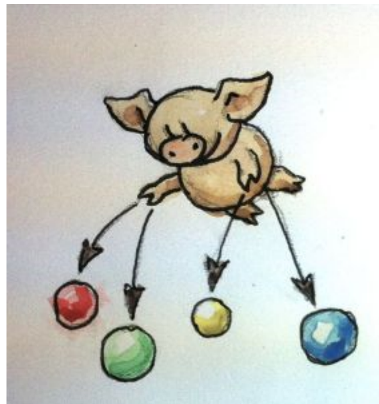
Для определения объектов в терминах их взаимоотношений теория категорий прибегает к т.н. *универсальным конструкциям*. Для этого можно выбрать некоторый шаблон, диаграмму из объектов и морфизмов определенной формы, и рассмотреть все подходящие под него конструкции рассматриваемой категории. Если шаблон достаточно распространен и категория достаточно велика, то, вероятно, найденных конструкций будет очень и очень много. Идея универсальной конструкции состоит в том, чтобы упорядочить конструкции по какому-то закону и выбрать из них наиболее подходящую.

Этот процесс можно сравнить с поиском в сети. Запрос пользователя — это наш шаблон. Если запрос не очень специфичен, то в ответ поисковая система выдаст множество подходящих документов, только часть из которых релевантны. Чтобы исключить нерелевантные ответы, пользователь уточняет запрос, что увеличивает точность поиска. В конце концов поисковая система проранжирует совпадения и, если повезет, искомый результат будет в самом начале списка.

5.1 Инициальный объект

Простейший шаблон состоит из одного-единственного объекта. Очевидно, под него подходит каждый объект категории, а их очень и очень много. Чтобы выбрать самый подходящий, нужно ввести на них иерархию. Единственное, что есть в нашем распоряжении, — это морфизмы. Если представлять себе морфизмы как стрелки, то может оказаться, что от одного конца категории до другого простирается всеобъемлющая цепочка стрелок. Это верно в упорядоченных категориях, например, в частичных порядках. Будем говорить, что объект a предшествует объекту b, если существует стрелка (морфизм) от a к b. Было бы естественно называть инициальным объект, который предшествует всем прочим (т.е. от него есть стрелка к любому другому объекту категории). Ясно, что, вообще говоря, инициального объекта может и не быть. Однако намного хуже то, что объектов, удовлетворяющих предыдущему определению, может быть больше одного. Как уточнить наше определение? Подсказку дают упорядоченные категории: в них между двумя объектами существует не более одной стрелки (есть всего один способ быть больше или меньше, чем другой объект). Это приводит нас к следующему определению:

> **Инициальный объект** — это объект, от которого к любому объекту категории исходит ровно один морфизм.



На самом деле, даже такое определение не гарантирует единственность инициального объекта (если он существует). Однако оно гарантирует единственность с точностью *до изоморфизма*. Изоморфизм — очень важное понятие для теории категорий, мы вскоре к нему вернемся.

Рассмотрим несколько примеров. Инициальный объект в частично упорядоченном множестве — это его наименьший элемент. Некоторые частично упорядоченные множества, такие как все целые числа (и положительные, и отрицательные), не имеют инициального объекта.

В категории множеств инициальный объект — это пустое множество. Напомним, пустое множество соответствует типу `Void` в Haskell (в C++ соответствующего типа нет), а единственная полиморфная функция от `Void` к любому другому типу имеет обозначение `absurd`:

```
absurd :: Void -> a
```

Существование такой функции и делает `Void` инициальным объектом категории множеств.

5.2 Терминальный объект

Продолжим эксперименты с шаблоном от единственного объекта, но изменим метод ранжирования. Будем говорить, что объект a следует за объектом b, если существует морфизм от b к a (обратите внимание на смену направления). Нас интересует «самый последний» объект категории. Из соображений единственности определим его так:

> **Терминальный объект** — это объект, к которому от любого объекта категории приходит ровно один морфизм.

Опять же терминальный объект единственен с точностью до изоморфизма, что будет вскоре доказано. Однако, сначала рассмотрим несколько примеров. В частично упорядоченном множестве терминальный объект (если он существует) является наибольшим объектом. В категории множеств терминальный объект — это синглетон. Напомним, синглетон соответствует типу `void` в C++ и типу `()` в Haskell и является типом, населенным ровно одним значением, неявным в C++ и явным в Haskell, обозначаемым тем же литералом `()`. Ранее было установлено, что существует единственная чистая функция из любого типа в `()`:

```
unit  :: a -> ()
unit _ = ()
```

так что все требования к терминальному объекту выполнены.

Заметим, что в данном случае условие единственности критически важно, поскольку существуют и другие множества (на самом деле, все множества кроме пустого), которые имеют входящие морфизмы от любого другого типа. Например, существует функция со значениями типа `Bool` (предикат), определенная для любого типа аргумента:

```
yes  :: a -> Bool
yes _ = True
```

Однако `Bool` не является терминальным объектом, потому что существует по крайней мере еще одна функция от любого типа к `Bool` (кроме `Void`, для которой обе функции есть `absurd`) .

```
no  :: a -> Bool
no _ = False
```

Требование единственности морфизмов дает нам необходимую точность, сужая набор подходящих терминальных объектов до одного.

5.3 Двойственность

Нельзя не заметить симметрию между определениями инициального и терминального объектов. Вся разница заключается в направлении морфизмов. Для любой категории $\mathbf{C}$ можно определить *двойственную категорию* $\mathbf{C}^{op}$, просто обратив все стрелки вспять. Двойственная категория автоматически удовлетворяет определению категории, если мы одновременно с обращением стрелок переопределим композицию. Если композицией исходных морфизмов $f :: a \to b$ и $g :: b \to c$ был морфизм $h :: a \to c$ с $h = g \circ f$, то композицией обращенных морфизмов $f^{op} :: b \to a$ и $g^{op} :: c \to b$ будет морфизм $h^{op} :: c \to a$ с $h^{op} = f^{op} \circ g^{op}$. Отметим, что обращение тождественной стрелки совпадает с ней самой.

Двойственность — это очень важное свойство категорий, удваивающее производительность труда категорного теоретика. Для любой конструкции имеет место двойственная и, доказав одну теорему, вы получаете вторую в виде бонуса. Конструкции двойственной категории часто имеют префикс «ко»: произведения и копроизведения, монады и комонады, конусы и коконусы, пределы и копределы и т.п. Однако не имеет смысла рассматривать ко-ко-конструкции, ведь дважды обращенная стрелка совпадает с исходной, так что, например, кокомонады совпадают с монадами.

Итак, терминальный объект — это инициальный объект в двойственной категории.

5.4 Изоморфизмы

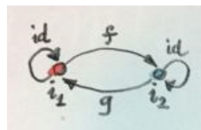
Программисты хорошо знают, что определить равенство не так уж и просто. Что означает, что два объекта равны? Должны ли они занимать одну и ту же область памяти (равенство указателей)? Или достаточно того, что значения всех их компонент совпадают? Считаются ли равными два комплексных числа, если одно из них задано действительной и мнимой компонентами, а другое — аргументом и модулем? Можно было бы понадеяться, что математики обладают сакральным знанием об определении равенства, но это не так. В математике также существует множество видов равенства, а также более слабое понятие изоморфизма и еще более слабое — эквивалентности.

Интуитивно объекты изоморфны, если они имеют одинаковый вид, внешне неразличимы: любая часть одного объекта взаимно однозначно соответствует некоторой части другого объекта; по показаниям доступных нам измерительных приборов объекты являются точными копиями друг друга. Математически это означает, что существуют взаимнообратные отображения объекта a в объект b и объекта b в объект a. Теория категорий обобщает отображения до морфизмов. Изоморфизм — это обратимый морфизм или, другими словами, пара морфизмов, обратных друг к другу.

Понятие обратимости выражается в терминах композиции и единичного морфизма: морфизм g является обратным к f, если их композиция является тождественным морфизмом. На самом деле это не одно, а два условия, поскольку есть два способа композиции пары морфизмов:

$$f \circ g = \text{id}$$
$$g \circ f = \text{id}$$

Когда мы говорим, что инициальный (терминальный) объект единственен с точностью до изоморфизма, то имеется в виду, что любые два инициальные (терминальные) объекты изоморфны. Это легко продемонстрировать. Предположим, что i_1 и i_2 — инициальные объекты в одной и той же категории. Поскольку i_1 инициальный, то существует единственный морфизм f от i_1 к i_2. Аналогично, поскольку i_2 инициальный, то существует единственный морфизм g от i_2 к i_1. Что можно сказать о композиции этих морфизмов?



Все морфизмы здесь единственные

Композиция $g \circ f$ должна быть морфизмом от i_1 к i_1. Но i_1 является инициальным объектом, так что существует ровно один морфизм от i_1 к i_1 и, поскольку начало и конец стрелки совпадают, эта вакансия уже занята тождественным морфизмом. Следовательно, они должны совпадать: морфизм $g \circ f$ является тождественным. Аналогично, морфизм $f \circ g$ также

совпадает с тождественным, поскольку может быть всего один морфизм от i_2 к i_2. Таким образом, f и g взаимнообратны, а два инициальных объекта изоморфны.

Заметим, что наше доказательство единственности инициального объекта с точностью до изоморфизма существенно использовало единственность морфизма от инициального объекта к самому себе. Однако важно ли то, что морфизмы f и g также единственны? Дело в том, что на самом деле мы доказали более строгое утверждение: инициальный объект единственен с точностью до единственного изоморфизма. Вообще говоря, между двумя объектами может быть более одного изоморфизма, но не в рассмотренном случае. «Единственность с точностью до единственного изоморфизма» — важное свойство всех универсальных конструкций.

5.5 Произведения

Следующая универсальная конструкция — это произведение. Мы уже знакомы с декартовым произведением двух множеств, состоящим из множества всех возможных пар. Какой же шаблон связывает множество-произведение с множествами-множителями? Если мы его выявим, то сможем обобщить понятие произведения на прочие категории.

С декартовым произведением связаны две функции (проекции), действующие от множества-произведения к соответствующему множеству-множителю. В Haskell эти функции называются `fst` и `snd`, которые выбирают первый и второй элементы пары соответственно:

```
fst       :: (a,b) -> a
fst (x,y) = x

snd       :: (a,b) -> b
snd (x,y) = y
```

Здесь функции определены при помощи сопоставления аргумента с образцом: образец соответствует любой паре `(x,y)` и извлекает ее компоненты в переменные `x` и `y`.

Эти определения можно упростить при помощи прочерков:

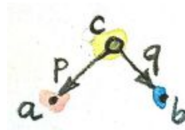
```
fst (x,_) = x
snd (_,y) = y
```

В C++ воспользуемся шаблонным функциями, например:

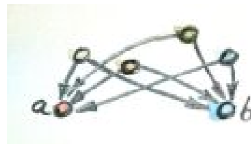
```
template<class A, class B>
   A fst(pair<A, B> const & p)
   {
      return p.first;
   }
```

С помощью этой (кажущейся скудной) информации о произведении попробуем сконструировать соответствующий ему шаблон из объектов и морфизмов в категории множеств. Этот шаблон будет состоять из объектов-множителей `a` и `b`, объекта `c` и проецирующих морфизмов `p` и `q`:

```
p :: c -> a
q :: c -> b
```



Все объекты `c`, которые удовлетворяют такому шаблону, будут рассматриваться как кандидаты в произведение. Таких объектов может быть очень много.



Для примера возьмем в качестве множителей два типа, а именно `Int` и `Bool`, и рассмотрим выбор кандидатов в их произведение.

Вот первый из них: `Int`. Может ли `Int` рассматриваться как кандидат в произведение `Int` и `Bool`? Да, может, вот соответствующие проекции:

```
p  :: Int -> Int
p x = x

q  :: Int -> Bool
q _ = True
```

Выглядит подозрительно, но вполне соответствует шаблону.

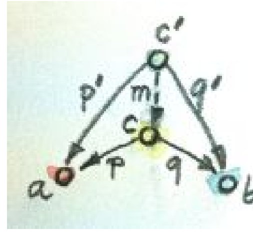
Вот еще кандидат: `(Int,Int,Bool)`. Это кортеж из трех элементов. Вот соответствующая пара проецирующих морфизмов (мы снова используем сопоставление с образцом):

```
p           :: (Int, Int, Bool) -> Int
p (x, _, _) = x

q           :: (Int, Int, Bool) -> Bool
q (_, _, b) = b
```

Внимательный читатель мог заметить, что первый кандидат слишком мал — он покрывает только `Int`-компоненту произведения, а второй слишком велик, потому что включает в себя явно фиктивную `Int`-компоненту.

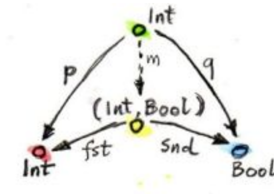
Мы пока что рассмотрели только первую составляющую универсальной конструкции — шаблон, но ничего не сказали о второй — о ранжировании. Нам нужен способ, позволяющий сравнить двух кандидатов, соответствующих шаблону, а именно объект `c` с проекциями `p` и `q` и объект `c'` с проекциями `p'` и `q'`. Хотелось бы положить, что `c` лучше `c'`, если существует морфизм `m` от `c'` к `c`, но это слишком слабое условие. Нужно еще и потребовать, чтобы проекции `p` и `q` были лучше (универсальнее), чем `p'` и `q'`. Это означает, что `p'` и `q'` могут быть восстановлены по `p` и `q` при помощи `m`:

```
p' = p . m
q' = q . m
```



Специалист по теории чисел сказал бы, что `m` является делителем (фактором) `p'` и `q'`. Представьте себе натуральные числа вместо морфизмов и умножение вместо композиции: тогда m окажется общим делителем `p'` и `q'`. Далее в подобных ситуациях мы будем говорить, что `m` *факторизует* `p'` и `q'`.

Для тренировки интуиции покажем, что пара `(Int,Bool)` с двумя каноническими проекциями `fst` и `snd` определенно лучше, чем рассмотренные выше кандидаты.



Отображение `m` для первого кандидата имеет вид:

```
m  :: Int -> (Int, Bool)
m x = (x, True)
```

Действительно, обе проекции `p` и `q` могут быть восстановлены как:

```
p x = fst (m x) = x
q x = snd (m x) = True
```

Морфизм `m` для второго примера также определен единственным образом:

```
m (x, _, b) = (x, b)
```

Мы показали, что (Int,Bool) лучше прежних обоих кандидатов. Покажем, что обратное неверно. Можно ли найти некоторый морфизм m′, который позволит восстановить fst и snd по p и q?

```
fst = p . m′
snd = q . m′
```

В первом примере q всегда возвращает True, однако в то же время существуют пары, в которых вторым компонентом выступает False. Так что восстановить snd по q нельзя.

Во втором примере дела обстоят иначе: у нас достаточно информации после p и q, чтобы восстановить fst и snd, однако факторизующий морфизм m′ определен неоднозначно. Действительно, поскольку и p, и q игнорируют второй элемент кортежа, морфизм m′ может поместить туда что угодно:

```
m′ (x, b) = (x, x, b)
```

или

```
m′ (x, b) = (x, 42, b)
```

и т. д.

Суммируя вышесказанное, для данного типа c с проекциями p и q существует единственный морфизм m от c к декартовому произведению (a,b), который факторизует p и q. На самом деле m просто комбинирует их в пару:

```
m  :: c -> (a, b)
m x = (p x, q x)
```

Это делает декартово произведение (a,b) наилучшим кандидатом и завершает рассмотрение этой универсальной конструкции для категории множеств.

Теперь забудем про множества и определим произведение двух объектов в произвольной категории при помощи той же универсальной конструкции. Такое произведение (если оно существует) единственно с точностью до единственного изоморфизма.

> **Произведение** объектов a и b — это такой оснащенный двумя проекциями объект c, что для любого другого оснащенного проекциями объекта c' существует единственный морфизм m от c' к c, факторизующий эти проекции.

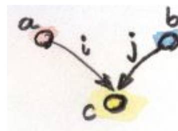
Функция (высшего порядка), которая строит факторизующий морфизм по двум проекциям, иногда называется *факторизатором*. В нашем случае она имеет вид:

```
factorizer    :: (c -> a) -> (c -> b)
                          -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

5.6 Копроизведения

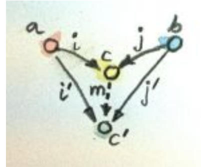
Как и всякая конструкция теории категорий, произведение имеет двойника, называемого копроизведением. Если мы обратим стрелки в шаблоне произведения, то получим объект `c`, оснащенный двумя *вложениями* `i` и `j`, морфизмами от `a` и `b` к `c`:

```
i :: a -> c
j :: b -> c
```



Нам также следует обратить порядок ранжирования: теперь объект `c` будет считаться лучше объекта `c'`, оснащенного вложениями `i'` и `j'`, если существует морфизм `m` от `c` к `c'`, факторизующий вложения:

```
i' = m . i
j' = m . j
```



Наилучший из таких объектов, обладающий единственным морфизмом ко всем прочим отвечающим шаблону объектам, называется копроизведением и, если он существует, единственен с точностью до единственного изоморфизма.

> **Копроизведение** объектов a и b — это такой оснащенный двумя вложениями объект c, что для любого другого оснащенного вложениями объекта c' существует единственный морфизм m от c к c', факторизующий эти вложения.

В категории множеств копроизведение — это *дизъюнктное объединение*. Элемент дизъюнктного объединения a и b — это либо элемент a, либо элемент b. Если два множества пересекаются, то дизъюнктное объединение содержит обе копии общей части. Можно считать, что элементы дизъюнктного объединения помечены метками множеств-источников.

Для программиста копроизведение — это помеченное объединение двух типов. C++ поддерживает непомеченные объединения; задача отслеживания, какой из членов объединения валиден, лежит на плечах программиста. Чтобы создать помеченное объединение, нужно определить метку — перечисление — и скомбинировать ее с объединением. Например, помеченное объединение `int` и `char const*` может выглядеть так:

```
struct Contact
{
   enum { isPhone, isEmail } tag;
   union
   {
      int phoneNum;
      char const * emailAddr;
```

```
    };
};
```

Два вложения могут быть реализованы, либо как конструкторы, либо как функции. Например, вот первое вложение в виде функции `PhoneNum`:

```
Contact PhoneNum(int n)
{
   Contact c;
   c.tag      = isPhone;
   c.phoneNum = n;
   return c;
}
```

Эта функция вкладывает `int` в `Contact`.

Помеченное объединение также называется *вариантом*, обобщенная реализация которого есть в библиотеке `boost` (`boost::variant`).

На Haskell можно скомпоновать помеченное объединение из любых типов данных, разделяя конструкторы вертикальной чертой. Рассмотренный выше `Contact` записывается так:

```
data Contact = PhoneNum Int | EmailAddr String
```

Здесь `PhoneNum` и `EmailAddr` выступают и в качестве конструкторов (вложений), и как метки для сопоставления с образцом (см. ниже). Например, вот как можно построить `Contact` по телефонному номеру:

```
helpdesk :: Contact
helpdesk  = PhoneNum 2222222
```

В отличие от канонической реализации произведения, встроенной в синтаксис Haskell как примитивная пара, каноническая реализация копроизведения является не специальной языковой конструкцией, а рядовым типом данных `Either` из стандартной библиотеки:

```
data Either a b = Left a | Right b
```

Этот тип данных параметризован двумя типами a и b и имеет два конструктора: Left, принимающим тип a, и Right, принимающим тип b.

По аналогии с факторизатором для произведения можно определить и факторизатор для копроизведения. Для данного кандидата в копроизведения в виде типа c с двумя вложениями i и j построим факторизующий морфизм:

```
factorizer      :: (a -> c) -> (b -> c)
                            -> Either a b -> c
factorizer i j (Left a)  = i a
factorizer i j (Right b) = j b
```

5.7 Асимметрия

Выше мы рассмотрели два множества двойственных определений: во-первых, определение терминального объекта может быть получено из определения инициального объекта обращением стрелок, во-вторых, таким же путем получается определение копроизведения из определения произведения. Однако, в категории множеств инициальный объект принципиально отличается от терминального, а копроизведение — от произведения. Ниже будет показано, что произведение ведет себя как умножение с терминальным объектом, играющим роль единицы, а копроизведение похоже на сложение, где вместо нуля — инициальный объект. В частности, для конечных множеств количество элементов произведения есть произведение количеств элементов в исходных множествах, а количество элементов копроизведения равно сумме исходных количеств.

Это показывает, что категория множеств не является симметричной относительно обращения стрелок.

Заметим, что несмотря на то, что от пустого множества исходит единственная стрелка к любому другому множеству (функция absurd), нет ни одного морфизма, направленного к нему. Синглетон (одноэлементное множество) имеет не только единственную стрелку, направленную к нему от любого множества, но и исходящие стрелки к каждому непустому множеству. Как мы видели ранее, эти исходящие от терминального

объекта стрелки играют важную роль в выборе элементов других множеств (в пустом множестве нет элементов, так что нечего и выбирать).

Взаимоотношение с синглетоном — это то, чем отличается произведение от копроизведения. Рассмотрим синглетон, состоящий из одного элемента `()`, как объект, отвечающий шаблону произведения. Оснастим его двумя проекциями: пусть `p` и `q` — функции от синглетона к каждой из компонент произведения. Обе они выбирают фиксированные элементы в соответствующих множествах. Поскольку произведение является универсальным, то существует (единственный) морфизм `m` от синглетона к произведению. Этот морфизм выбирает элемент из множества произведения, т.е. фиксированную пару, и факторизует проекторы:

```
p = fst . m
q = snd . m
```

Если подставить в эти формулы единственный элемент синглетона `()`, то получим:

```
p () = fst (m ())
q () = snd (m ())
```

Поскольку `m ()` — это элемент произведения, выбираемый морфизмом `m`, эти формулы означают, что элемент `p ()`, выбираемый проекцией `p` из первого множества, является первой компонентой пары, выбираемой `m`. Аналогично, `q ()` равен второй компоненте. Это полностью согласуется с трактовкой элементов множества-произведения как пар элементов из множеств-множителей.

Однако для копроизведения такой простой интерпретации не существует. Можно попробовать рассмотреть синглетон как кандидата в копроизведения, пытаясь вычленить из него элементы, но в таком случае мы получим два входящих вложения, а не две исходящие проекции. Вложения ничего не говорят о своих источниках (на самом деле, как мы видели ранее, в случае синглетона они с необходимостью игнорируют свой аргумент). То же касается и единственного морфизма от копроизведения к синглетону. Вид категории множеств со стороны инициального объекта (по стрелкам) совершенно отличен от вида со стороны терминального объекта (против стрелок).

Указанное различие не является специфическим свойством собственно множеств; это свойство функций, который выступают в качестве морфизмов категории множеств **Set**. Функции по своей природе асимметричны. Рассмотрим этот момент подробнее.

Функция должна быть определена для каждого элемента своей области определения (в программировании такая функция называется *тотальной*), но не обязана покрывать область значений целиком. Например, функции от синглетона попадают всего к одному элементу из области значений. Вырожденным случаем являются функции от пустого множества — они вообще не принимают никакого значения. В противоположной ситуации, когда значения функции покрывают все доступные значения, функция называется *сюръективной*.

Еще одним источником асимметрии является то, что функция может склеить много элементов из области определения в один элемент из области значений. Например, функции к синглетону склеивают все элементы исходного множества в один `()`. Выше рассматривалась полиморфная функция `unit`. Если под действием функции образы всех элементов различны, то функция называется *инъективной*.

Конечно же, существуют функции, которые хороши в обоих отношениях, т.е. являются и сюръективными, и инъективными. Они называются *биекциями* и, поскольку обратимы, образуют симметричную связь между областью определения и областью значений. В категории множеств изоморфизм — это то же самое, что биекция.

Упражнения

1. Покажите, что терминальный объект единственен с точностью до единственного изоморфизма.

2. Что представляет собой произведение в частично упорядоченном множестве? Подсказка: воспользуйтесь универсальной конструкцией.

3. Что представляет собой копроизведение в частично упорядоченном множестве?

4. Реализуйте аналог `Either` из Haskell на вашем основном языке программирования (но не на Haskell).

5. Покажите, что `Either` является более подходящим копроизведением, чем `int`, оснащенный двумя вложениями:

   ```
   int i(int n)  { return n; }
   int j(bool b) { return b ? 0 : 1; }
   ```

   Подсказка: определите функцию

   ```
   int m(Either const & e);
   ```

   которая факторизует `i` и `j`.

6. (Продолжение) Аргументируйте, почему `int` с двумя вложениями `i` и `j` не может оказаться предпочтительнее `Either`?

7. (Продолжение) А как насчет таких вложений?

   ```
   int i(int n)
   {
      if (n < 0) return n;
      return n + 2;
   }
   int j(bool b) { return b ? 0 : 1; }
   ```

8. Предложите неудачного кандидата в копроизведения для `int` и `bool`, который будет хуже `Either` тем, что допускает несколько морфизмов к `Either`.

Глава 6

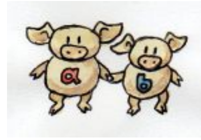
Простые алгебраические типы данных

В предыдущей статье были рассмотрены базовые операции над типами: произведение и копроизведение. Теперь покажем, что комбинирование этих механизмов позволяет построить многие из структур данных, используемых в повседневном программировании. Такое построение имеет существенное прикладное значение. Например, если мы умеем проверять на равенство базовые типы данных, а также знаем, как свести равенство произведения и копроизведения к равенству компонент, то операторы равенства для составных типов можно вывести автоматически. В Haskell для обширного подмножества составных типов автоматически выводятся операторы равенства и сравнения, конвертация в строку и обратно, и многие другие операции.

Рассмотрим подробнее место произведения и копроизведения типов в программировании.

6.1 Тип-произведение

Каноническая реализация типа-произведения в языках программирования — это пара. В Haskell пара является примитивным конструктором типов, а в C++ это относительно сложный шаблон из стандартной библиотеки.



Строго говоря, тип-произведение не коммутативен: нельзя подставить пару типа `(Int,Bool)` вместо `(Bool,Int)`, хотя они и содержат одни и те же данные. Однако произведение коммутативно с точностью до изоморфизма, задаваемого функцией `swap`, которая обратна самой себе:

```
swap        :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

Можно рассматривать такие пары как различные форматы хранения одной и той же информации, как big endian (большой обратный порядок байт) и little endian (малый обратный порядок байт).

Вкладывая одни пары в другие, можно скомбинировать в произведение сколько угодно типов. То же самое можно получить проще, если заметить, что вложенные пары эквивалентны кортежам. Это следствие того, что различные порядки вложения пар изоморфны между собой. Есть два возможных порядка комбинирования в тип-произведение трех типов `a`, `b` и `c` (в заданной последовательности). А именно,

```
((a, b), c)
```

или

```
(a, (b, c))
```

Эти типы различны в том смысле, что функции, ожидающей аргумент первого типа, нельзя передать аргумент второго, однако значения типов находятся во взаимно-однозначном соответствии. Вот функция, которая задает это отображение в одну сторону:

```
alpha                    :: ((a, b), c) ->
                               (a, (b, c))
alpha ((x, y), z)         = (x, (y, z))
```

А вот обратная к ней:

```
alpha_inv             :: (a, (b, c)) ->
                         ((a, b), c)
alpha_inv (x, (y, z)) = ((x, y), z)
```

Итак, имеет место изоморфизм типов; это просто разные способы переупаковки одних и тех же данных.

Рассматривая тип-произведение как бинарную операцию, мы видим, что указанный изоморфизм очень похож на ассоциативный закон в моноидах:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Единственная разница в том, что в моноиде оба произведения абсолютно совпадают, а для типов — равны с точностью до изоморфизма.

Если счесть это различие несущественным, то можно продлить аналогию с моноидами далее и показать, что синглетон `()` является нейтральным элементом относительно умножения типов, так же как 1 нейтральна относительно умножения чисел. Действительно, присоединение `()` к элементу типа `a` не добавляет никакой информации. Тип

```
(a, ())
```

изоморфен `a`, где изоморфизм задается функциями

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

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

Наши наблюдения можно формализовать в виде утверждения, что категория множеств **Set** является моноидальной, т.е. категорией, которая является еще и моноидом относительно умножения объектов (в данном случае — относительно декартова произведения). Ниже будет дано строгое определение.

В Haskell доступен более общий способ определения типа-произведения, особенно удобный, как мы вскоре убедимся, когда они сочетаются с типом-суммой. Воспользуемся именованными конструкторами с несколькими аргументами. Например, альтернативное определение пары выглядит так:

```
data Pair a b = P a b
```

Здесь `Pair a b` — это имя конструктора типов, параметризованное двумя другими типами `a` и `b`, а `P` — имя конструктора данных. Мы определяем конкретный тип, передавая в конструктор типов `Pair` два типа, и создаем пару этого типа, передавая соответствующие типизированные значения в конструктор `P`. Например, определим переменную `stmt` как пару из `String` и `Bool`:

```
stmt :: Pair String Bool
stmt  = P "This statement is" False
```

Первая строка — это декларация типа. Она состоит из конструктора типов `Pair` с `String` и `Bool` вместо `a` и `b`. Вторая строка определяет значение переменной, полученное применением конструктора данных `P` к конкретным строке и логическому значению. Еще раз: конструкторы типов используются для конструирования типов, конструкторы данных — для конструирования значений.

Поскольку пространства имен типов и данных в Haskell не пересекаются, то часто одно и то же имя используется для обоих конструкторов. Например,

```
data Pair a b = Pair a b
```

Если посмотреть на встроенный тип пары более пристально, то можно даже понять, что она на самом деле является вариацией на тему последней декларации, только конструктор `Pair` заменен на бинарный оператор `(,)`. Можно использовать `(,)` так же, как и любой другой именованный конструктор, в префиксной нотации:

```
stmt = (,) "This statement is" False
```

Аналогично, `(,,)` конструирует тройки и т. д.

Вместо использования обобщенных пар или кортежей, можно ввести отдельное имя для конкретных типов-произведений. Например,

```
data Stmt = Stmt String Bool
```

представляет собой произведение `String` и `Bool`, но обладает собственным именем и конструктором. Преимущество такого определения в том, что можно завести много типов с одним и тем же содержимым, но различной семантикой и функциональностью, которые система типов не позволит смешивать между собой.

Программирование на кортежах и многоаргументных конструкторах часто ведет к неразберихе и ошибкам, потому что надо все время отслеживать, какой компонент за что отвечает.

Было бы лучше иметь возможность давать компонентам имена собственные. Тип-произведение с именованными полями называется записью в Haskell и структурой (`struct`) в C.

6.2 Записи

Рассмотрим простой пример. Будем описывать химические элементы единой структурой, состоящей из двух строк (латинского названия и символа) и целого числа, соответствующего атомной массе. Для этого можно воспользоваться кортежем `(String, String, Int)` и все время держать в голове, какой компонент за что отвечает. Для извлечения компонент из кортежа будем применять сопоставление с образцом. Следующая функция проверяет, является ли символ химического элемента префиксом его латинского названия (например, He — префикс Helium):

```
startsWithSymbol :: (String, String, Int) ->
                                          Bool
startsWithSymbol (name, symbol, _)
                 = isPrefixOf symbol name
```

В таком коде легко ошибиться, его трудно читать и поддерживать. Гораздо лучше определить вместо кортежа запись:

```
data Element = Element { name         :: String
                       , symbol       :: String
                       , atomicNumber :: Int }
```

Эти представления изоморфны, в чем можно убедиться при помощи следующих взаимообратных преобразований:

```
tupleToElem     :: (String, String, Int) ->
                                      Element
tupleToElem (n, s, a)
                 = Element { name         = n
                           , symbol       = s
                           , atomicNumber = a }

elemToTuple  :: Element ->
                       (String, String, Int)
elemToTuple e = (name e, symbol e,
                              atomicNumber e)
```

Заметим, что имена полей записи одновременно являются и функциями доступа к этим полям. Например, `atomicNumber e` возвращает поле `atomicNumber` записи `e`. Таким образом функция `atomicNumber` имеет тип:

```
atomicNumber :: Element -> Int
```

С использованием записей типа `Element` функция `startsWithSymbol` становится более читаемой:

```
startsWithSymbol  :: Element -> Bool
startsWithSymbol e = isPrefixOf (symbol e)
                                  (name e)
```

На Haskell можно провернуть трюк, превращающий функцию `isPrefixOf` в инфиксный оператор, обрамив ее обратными апострофами. Это делает код более читаемым:

```
startsWithSymbol e = symbol e `isPrefixOf` name e
```

Мы смогли опустить скобки за счет того, что приоритет инфиксного оператора ниже приоритета вызова функции.

6.3 Тип-сумма

Аналогично тому, как произведение в категории множеств индуцирует тип-произведение, копроизведение порождает тип-сумму. Каноническая реализация тип-суммы на Haskell такова:

```
data Either a b = Left a | Right b
```

Как и пары, `Either`-ы коммутативны (с точностью до изоморфизма), могут быть вложенными, причем порядок вложения не важен (с точностью до изоморфизма). Поэтому мы можем, например, определить сумму, эквивалентную тройке:

```
data OneOfThree a b c = Sinistral a
                      | Medial b
                      | Dextral c
```

Оказывается, что **Set** образует (симметричную) моноидальную категорию относительно операции копроизведения. Место бинарной операции занимает дизъюнктная сумма, а нейтральным элементом является инициальный объект. В терминах типов `Either` — моноидальная операция, а ненаселенный тип `Void` — ее нейтральный элемент. Считайте, что `Either` — это сложение, а `Void` — это нуль. Действительно, добавление `Void` к тип-сумме не изменяет множество значений типа. Например, `Either a Void` изоморфно `a`. Это потому, что нет способа создать

версию `Right` этого типа — нет значения типа `Void`. Единственные обитатели `Either a Void` строятся с помощью конструкторов `Left`, они просто инкапсулируют значение типа `a`. Символически это может быть записано как

$$a + 0 = a$$

Тип-суммы очень часто встречаются в Haskell. В C++ их аналоги (объединения или варианты) используются существенно реже по ряду причин.

Во-первых, простейшие тип-суммы — это перечисления, которые реализованы в C++ при помощи `enum`. Эквивалентом

```
data Color = Red | Green | Blue
```

в C++ будет

```
enum { Red, Green, Blue };
```

Еще более простая тип-сумма

```
data Bool = True | False
```

в C++ является примитивным типом `bool`.

Далее, тип-суммы, кодирующие наличие или отсутствие значения, реализуются в C++ при помощи различных трюков с «невозможными» значениями, такими как пустые строки, отрицательные числа, нулевые указатели и т.д. На Haskell явная, намеренная опциональность значения записывается при помощи `Maybe`:

```
data Maybe a = Nothing | Just a
```

Тип `Maybe` является двойной тип-суммой. Мысленно превратим его конструкторы в отдельные типы. Первый примет вид:

```
data NothingType = Nothing
```

Это перечисление с единственным значением `Nothing`. Другими словами, это синглетон, эквивалентный типу `()`. Вторая часть

```
data JustType a = Just a
```

представляет собой обертку над типом `a`. Мы могли бы записать `Maybe` как

```
data Maybe a = Either () a
```

Более сложные тип-суммы имитируются в C++ при помощи указателей. Указатель может быть либо нулевым, либо указывать на значение определенного типа. Например, на Haskell список определен как (рекурсивная) тип-сумма:

```
data List a = Nil | Cons a (List a)
```

В C++ этот же тип записывается так:

```
template<class A>
   class List
   {
      Node<A> * _head;
      public:
         List() : _head(nullptr) {}   // Nil
         List(A a, List<A> l)         // Cons
           : _head(new Node<A>(a, l))
         {}
   };
```

Нулевой указатель здесь кодирует пустой список.

Заметим, что конструкторы `Nil` and `Cons` из Haskell превратились в два перегруженных конструктора класса `List` с аналогичными аргументами: без аргументов — в случае `Nil`, значение и список — для `Cons`. Класс `List` не нуждается в метке, которая бы отличала компоненты

тип-суммы; вместо этого он использует специальное значение nullptr для _head, чтобы представить Nil.

Важное различие между типами в Haskell и в C++ состоит в том, что в Haskell структуры данных неизменяемы. Если объект был создан при помощи определенного конструктора, то запоминается, какой конструктор и с какими аргументами использовался. Так, экземпляр класса Maybe, созданный как Just со значением "energy", никогда не превратится в Nothing. Аналогично пустой список останется пустым, а трехэлементный список всегда будет хранить одни и те же три элемента.

Неизменяемость делает конструкторы обратимыми: объект всегда можно разобрать на составные части, использованные при его создании. Такая деконструкция выполняется путем сопоставления с образцом, в качестве которого выступает тот или иной конструктор. Аргументы конструктора замещаются именами переменных (или другими образцами).

У типа List два конструктора, так что деконструкция произвольного List состоит из двух соответствующих сопоставлений с образцом. Первый образец совпадает с пустым списком Nil, второй — со списком, созданным при помощи Cons. Для примера определим простую функцию:

```
maybeTail            :: List a -> Maybe
                                   (List a)
maybeTail Nil         = Nothing
maybeTail (Cons _ t) = Just t
```

Первая часть определения maybeTail использует конструктор Nil как образец для сопоставления и возвращает Nothing. Вторая часть использует в качестве образца конструктор Cons. Первый аргумент образца представлен прочерком, поскольку нас не интересует содержащееся в нем значение (оно не будет использовано в правой части). Второй аргумент Cons связывается с переменной t (здесь и далее мы будем говорить о переменных, хотя, строго говоря, они неизменны: единожды связанная со значением переменная никогда не меняется). Значение функции на этом образце равно Just t. Итак, в зависимости от способа создания значения типа List, оно совпадет с одним из образцов. Если оно было создано при помощи Cons, то функция получит оба, использованных при этом, аргумента (первый из которых будет проигнорирован).

Более сложные суммы типов реализуются в C++ иерархией полиморфных классов. Семейство классов с общим прародителем можно трактовать как сумму типов, в которой неявной меткой компоненты служит таблица виртуальных функций. То, что в Haskell служит сопоставлением с образцом, реализуется в C++ вызовом функции диспетчеризации.

Вы не часто встретите в C++ использование `union` в качестве типа-суммы из-за его чрезмерной ограниченности. В `union` нельзя поместить даже `std::string`, потому что у этого класса есть конструктор копирования.

6.4 Алгебра типов

По отдельности тип-произведение и тип-сумма позволяют определить множество полезных структур данных, но настоящая мощь проистекает из их комбинации.

Подведем итоги вышеизложенного. Мы рассмотрели две коммутативные моноидальные структуры, лежащие в основе системы типов. Это тип-сумма с нейтральным элементом `Void` и тип-произведение с нейтральным элементом `()`. Их удобно представлять себе как сложение и умножение. В таком случае `Void` соответствует нулю, а `()` — единице.

Посмотрим, как далеко простирается эта аналогия. Например, верно ли, что умножение на ноль дает ноль? Другими словами, изоморфно ли любое произведение на `Void` типу `Void`?

Существуют ли пары, состоящие из, скажем, `Int` и `Void`? Для создания пары нужны оба значения. Значение типа `Int` — это не проблема, а вот с `Void` есть загвоздка: этот тип не населен (не существует ни одного значения этого типа). Таким образом, для любого типа `a` тип `(a, Void)` также не населен и, следовательно, эквивалентен `Void`. Другими словами, $a \cdot 0 = 0$.

Сложение и умножение чисел связаны дистрибутивным законом:

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

Выполняется ли он для тип-суммы и тип-произведения? Да, с точностью до изоморфизма. Левая часть тождества соответствует типу

```
(a, Either b c)
```

а правая —

```
Either (a, b) (a, c)
```

Предъявим функции, которые преобразуют типы туда и обратно:

```
prodToSum        :: (a, Either b c) ->
                    Either (a, b) (a, c)
prodToSum (x, e) = case e of
                     Left y  -> Left  (x, y)
                     Right z -> Right (x, z)

sumToProd  :: Either (a, b) (a, c) ->
                   (a, Either b c)
sumToProd e = case e of
                Left  (x, y) -> (x, Left  y)
                Right (x, z) -> (x, Right z)
```

Конструкция `case of` используется для сопоставления с образцом внутри функции. Стрелка разделяет образец и соответствующее ему выражение. Например, при вызове `prodToSum` с аргументом

```
prod1 :: (Int, Either String Float)
prod1  = (2, Left "Hi!")
```

переменная `e` в `case e of` будет равна `Left "Hi!"`. Она совпадет с образцом `Left y`, при подстановке `"Hi!"` вместо `y`. Поскольку переменная `x` ранее уже была связана с `2`, результатом конструкции `case of` (и всей функции) будет, как и ожидалось, `Left (2, "Hi!")`.

Доказательство того, что введенные функции взаимнообратны, оставим читателю в качестве упражнения. Они лишь перепаковывают одни и те же данные из одного формата в другой.

Два моноида, связанные дистрибутивным законом, в математике называются *полукольцом*. Это не полное кольцо, поскольку мы не в силах определить вычитание типов. Множество утверждений, верных для образующих полукольцо натуральных чисел, можно перенести на типы. Вот несколько примеров:

Числа	Типы
0	`Void`
1	`()`
$a+b$	`Either a b = Left a | Right b`
$a \cdot b$	`(a, b)` или `Pair a b = Pair a b`
$2=1+1$	`data Bool = True | False`
$1+a$	`data Maybe = Nothing | Just a`

Тип-список представляет особый интерес, поскольку он определен как решение уравнения. Определяемый тип встречается в обеих сторонах равенства:

```
List a = Nil | Cons a (List a)
```

Произведя обычные подстановки и заменив `List a` на `x`, получим

```
x = 1 + a * x
```

Это уравнение нельзя решить традиционными алгебраическими методами, поскольку типы нельзя вычитать или делить. Попробуем рекурсивно подставлять вместо `x` справа выражение `(1 + a*x)` и раскрывать скобки по дистрибутивности. Получим

```
x = 1 + a*x
x = 1 + a*(1 + a*x) = 1 + a + a*a*x
x = 1 + a + a*a*(1 + a*x) = 1 + a + a*a
                                + a*a*a*x ...
x = 1 + a + a*a + a*a*a + a*a*a*a ...
```

В конце концов мы придем к бесконечной сумме произведений (кортежей), которую можно трактовать так: список либо пуст — `1`; либо состоит из одного элемента — `a`; либо состоит из пары — `a*a`; либо из тройки — `a*a*a`, и т.д. Полученное формально определение полностью отвечает интуитивному представлению о списке как о строке, где вместо букв — значения типа `a`.

Мы вернемся к спискам и другим рекурсивным структурам после изучения функторов и неподвижных точек.

Решение уравнений с символьными переменными — это алгебра! Поэтому такие типы данных и называются алгебраическими типами данных (АТД).

Подводя итоги, дадим одну очень важную интерпретацию алгебры типов. Заметим, что тип-произведение для `a` и `b` должно содержать и значение типа `a`, и значение типа `b`, что влечет населенность обоих типов. С другой стороны, тип-сумма содержит либо значение типа `a`, либо значение типа `b`, так что достаточно, чтобы хотя бы один из них был населен. Логические операции конъюнкции и дизъюнкции образуют полукольцо, находящееся в следующем соответствии с алгеброй типов:

Логика	Типы
$false$	`Void`
$true$	`()`
$a\|\|b$	`Either a b = Left a | Right b`
$a\&\&b$	`(a, b)`

Эта аналогия может быть углублена и является основой изоморфизма Карри-Говарда между логикой и теорией типов. Мы вернемся к этому вопросу при рассмотрении функциональных типов.

Упражнения

1. Покажите, что `Maybe a` и `Either () a` изоморфны.

2. Рассмотрим следующую тип-сумму на Haskell:

   ```
   data Shape = Circle Float | Rect Float Float
   ```

   Чтобы определить на типе `Shape` функцию `area`, воспользуемся сопоставлением с образцом:

   ```
   area           :: Shape -> Float
   area (Circle r) = pi * r * r
   area (Rect d h) = d * h
   ```

Реализуйте `Shape` на C++ или Java как интерфейс и создайте два класса: `Circle` и `Rect`. Затем запишите `area` как виртуальную функцию.

3. (Продолжение) Несложно добавить новую функцию `circ`, которая вычисляет периметр `Shape`. В Haskell определение типа `Shape` останется неизменным; следующий код можно добавить в любое место программы:

```
circ             :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)
```

Добавьте `circ` в вашу программу на C++ или Java. Какие части исходной программы пришлось модифицировать?

4. (Продолжение) Добавьте в тип `Shape` новую фигуру `Square` и внесите соответствующие обновления в остальной код. Что пришлось поменять на Haskell? А что в C++ или Java? (Даже если вы не знаете Haskell, изменения должны быть довольно очевидны.)

5. Покажите, что формальное тождество $a + a = 2 \cdot a$ выполняется для типов (с точностью до изоморфизма). Напоминаем, 2 на языке типов соответствует `Bool` (см. таблицу выше).

Глава 7

Функторы

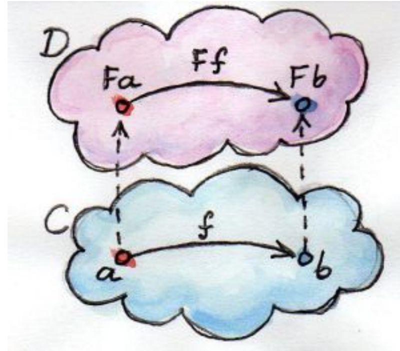
За понятием функтора стоит очень простая, но мощная идея (как бы банально это ни звучало). Просто сама теория категорий полна простых и мощных идей. Функтор есть отображение между категориями. Пусть даны две категории $\mathbf{C}$ и $\mathbf{D}$, а функтор F отображает объекты из $\mathbf{C}$ к объектам из $\mathbf{D}$ — это функция над объектами. Если a — это объект из $\mathbf{C}$, то будем обозначать его образ из $\mathbf{D}$ как $F\,a$ (без скобок). Но ведь категория — это не только объекты, но еще и соединяющие их морфизмы. Функтор также отображает и морфизмы — это функция над морфизмами. Но морфизмы отображаются не как попало, а так, чтобы сохранять связи. А именно, если морфизм f из $\mathbf{C}$ связывает объект a с объектом b,

$$f :: a \to b$$

то образ f в $\mathbf{D}$, $F\,f$, связывает образ a с образом b:

$$F\,f :: F\,a \to F\,b$$

(Надеемся, что такая смесь математических обозначений и синтаксиса Haskell понятна читателю. Мы не будем использовать скобки при применении функторов к объектам или морфизмам.)

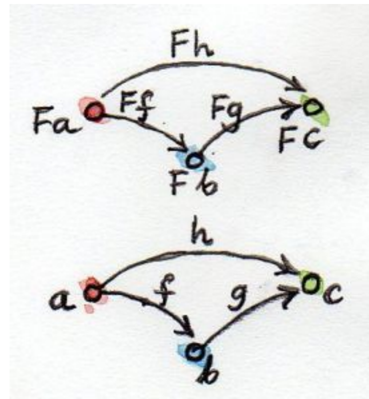


Как видите, функтор сохраняет структуру: что было связано в исходной категории, будет связано и в целевой. Но этим структура не исчерпывается: необходимо также поддержать композицию морфизмов. Если h — композиция f и g:

$$h = g \,.\, f$$

то потребуем, чтобы образ морфизма h под действием F был композицией образов f и g:

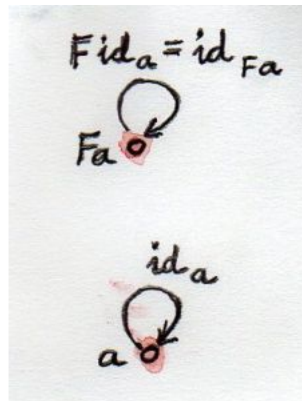
$$F\,h = F\,g \,.\, F\,f$$



Наконец, потребуем, чтобы все единичные (тождественные) морфизмы из **C** отображались к единичным морфизмам из **D**:

$$F\,\mathrm{id}_a = \mathrm{id}_{F\,a}$$

Здесь id_a — это единичный морфизм объекта a, а $\mathrm{id}_{F\,a}$ — единичный морфизм объекта $F\,a$.



Заметим, что все эти условия существенно ограничивают круг функций, подходящих в качестве морфизмов. Функторы должны сохранять структуру категории. Если представить себе категорию как набор объектов, переплетенных морфизмами, то функтор не имеет права оборвать ни одной нити этого кружева. Он может объединить несколько объектов, он может склеить морфизмы в один, но никогда не разрывает связей. Это ограничение аналогично понятию непрерывности из математического анализа. В этом смысле функторы «непрерывны» (хотя существует еще более ограничивающее определение непрерывности функторов). Как и функции, функторы могут выполнять и свертывание, и вложение. Аспект вложения наиболее ярко проявляется, когда исходная категория намного меньше целевой. В предельном случае исходная категория представляет собой категорию $\mathbf{1}$, состоящую из одного объекта и одного (единичного) морфизма. Функтор от категории $\mathbf{1}$ к любой другой категории просто выбирает определенный объект из этой категории. Имеет место полная аналогия с морфизмами из синглетона, выбирающими элементы из целевых множеств. Максимально свертывающий функтор называется постоянным функтором Δ_c. Он отображает каждый объект исходной категории в определенный объект c целевой категории, а всякий морфизм — в единичный морфизм id_c. Это как черная дыра, засасывающая все в точку сингулярности. Мы вернемся к рассмотрению этого функтора при обсуждении пределов и копределов.

7.1 Функторы в программировании

Вернёмся на землю, в мир программирования. Итак, у нас есть категория типов и функций. Рассмотрим функторы, отображающие эту категорию в себя — так называемые эндофункторы. Что представляет собой эндофунктор в категории типов? В первую очередь, он сопоставляет одним типам другие. Подобные отображения на самом деле нам знакомы, это типы, параметризованные другими типами. Рассмотрим несколько примеров.

Функтор Maybe

Определение `Maybe` есть отображение типа `a` в тип `Maybe a`:

```
data Maybe a = Nothing | Just a
```

Важная деталь: `Maybe` сам по себе — это не тип, а конструктор типа. Он принимает в качестве аргумента тип, такой как `Int` или `Bool`, и возвращает другой тип. `Maybe` без аргументов представляет собой функцию на типах. Но является ли `Maybe` функтором? (Здесь и далее, говоря о функторах в контексте программирования, почти всегда подразумеваются эндофункторы.) Ведь функтор — это не только отображение объектов (здесь — типов), но и отображение морфизмов (здесь — функций). Для любой функции от `a` к `b`

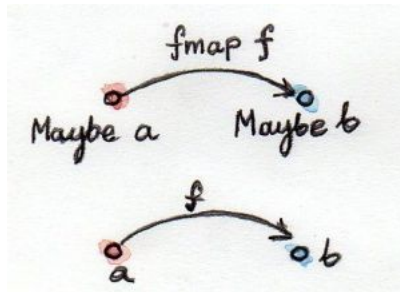
```
f :: a -> b
```

хотелось бы получить функцию от `Maybe a` к `Maybe b`. Чтобы определить такую функцию, нужно рассмотреть два случая, соответствующие двум конструкторам `Maybe`. В случае `Nothing` мы просто возвращаем `Nothing` обратно. Если же аргументом является `Just`, то применим функцию `f` к его содержимому. Итак, образ `f` под действием `Maybe` — это функция

```
f'          :: Maybe a -> Maybe b
f' Nothing  = Nothing
f' (Just x) = Just (f x)
```

(Кстати, в Haskell разрешено использовать апострофы в именах переменных, что очень удобно в подобных случаях.) В Haskell часть функтора, отвечающая за отображение морфизмов, реализуется функцией высшего порядка `fmap`. Для `Maybe` она имеет следующую сигнатуру:

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
```



Часто говорят, что `fmap` поднимает (осуществляет лифтинг) функцию. Поднятая функция работает на `Maybe`-значениях. Как обычно, из-за каррирования эта сигнатура может трактоваться двояко: либо как функция одной переменной, которая сама является функцией `(a -> b)`, возвращающая функцию `(Maybe a -> Maybe b)`, либо как функция двух переменных, возвращающая `Maybe b`:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

Следуя сказанному выше, приведем определение `fmap` для `Maybe`:

```
fmap _ Nothing  = Nothing
fmap f (Just x) = Just (f x)
```

Чтобы показать, что конструктор типов `Maybe` вместе с функцией `fmap` составляют функтор, осталось доказать, что `fmap` сохраняет единичные морфизмы и композицию. Эти утверждения называются *функториальными законами*, хотя они просто удостоверяют сохранение структуры категории.

Метод эквивалентных преобразований

Докажем функториальные законы методом *эквивалентных преобразований*, который является типичной техникой доказательств в Haskell. Метод опирается на то, что функции в Haskell определены набором равенств: левая часть равна правой, обе они могут быть подставлены друг вместо друга (возможно, при подстановке потребуется переименовать переменные во избежание конфликтов). Можно осуществить как подстановку тела функции в место ее вызова, так и вызов функции в этой точке. Рассмотрим в качестве примера тождественную функцию:

```
id x = x
```

Если мы встретим строку, скажем, `id y`, в некотором выражении, то ее всегда можно заменить на `y`. И вообще любое вхождение `id`, примененного к выражению, например, `id (y + 2)`, можно заменить на само это выражение `(y + 2)`. В обратную сторону: любое выражение `e` можно заменить на `id e`. В случае функций, определенных при помощи сопоставления с образцом, можно использовать каждое определение независимо. Например, согласно данному выше определению `fmap`, можно заменить `fmap f Nothing` на `Nothing` или наоборот. Рассмотрим этот подход на практике. Начнём с сохранения тождественности:

```
fmap id = id
```

Следует рассмотреть два случая: `Nothing` и `Just`. Начнём с `Nothing` (я описываю трансформацию левой части равенства в правую, используя псевдо-Haskell):

```
fmap id Nothing
= { по определению fmap }
Nothing
= { по определению id }
id Nothing
```

Заметьте, что на втором шаге мы подставили `id Nothing` вместо `Nothing`, используя определение `id` в обратную сторону. На практике подобные

доказательства делаются методом «поджигания фитиля с двух концов», вплоть до встречи в середине на одном и том же выражении, `Nothing` в данном случае. Второй случай также несложен:

```
fmap id (Just x)
= { по определению fmap }
Just (id x)
= { по определению id }
Just x
= { по определению id }
id (Just x)
```

Теперь покажем, что `fmap` сохраняет композицию:

```
fmap (g . f) = fmap g . fmap f
```

Начнём со случая `Nothing`:

```
fmap (g . f) Nothing
= { по определению fmap }
Nothing
= { по определению fmap }
fmap g Nothing
= { по определению fmap }
fmap g (fmap f Nothing)
```

Теперь пришло время `Just`:

```
fmap (g . f) (Just x)
= { по определению fmap }
Just ((g . f) x)
= { по определению композиции }
Just (g (f x))
= { по определению fmap }
fmap g (Just (f x))
= { по определению fmap }
fmap g (fmap f (Just x))
= { по определению композиции }
(fmap g . fmap f) (Just x)
```

Стоит подчеркнуть, что для функций с побочными эффектами в стиле C++, метод эквивалентных преобразований не работает. Рассмотрим пример:

```
int square(int x)
{
   return x * x;
}

int counter()
{
   static int c = 0;
   return c++;
}

double y = square(counter())
```

Метод эквивалентных преобразований позволил бы развернуть `square` и получить

```
double y = counter() * counter();
```

Определенно, такая трансформация некорректна и меняет результат выражения. Несмотря на это, если определить `square` как макрос, препроцессор C++ воспользуется методом эквивалентных преобразований с катастрофическим результатом.

Шаблонный тип `optional`

Функторы легко выражаются на Haskell, но описывать их можно на любом языке, поддерживающем обобщённое программирование и функции высшего порядка. Рассмотрим C++-ный аналог `Maybe`, шаблонный тип `optional`. Вот набросок реализации (полная реализация много сложнее, поскольку явно описывает разные способы передачи аргументов, семантику копирования и прочие вопросы характерного для C++ управления ресурсами):

```
template<class T>
   class optional
{
   bool _isValid; // the tag
   T _v;
   public:
      optional()    : _isValid(false) {}
                                   // Nothing
      optional(T x) : _isValid(true) , _v(x) {}
                                      // Just
      bool isValid() const { return _isValid; }
      T val() const { return _v; }
};
```

Приведённый шаблон обеспечивает половину описания функтора: отображение типов. Он сопоставляет новый тип `optional<T>` каждому типу `T`. Теперь опишем его действие над функциями:

```
template<class A, class B>
   std::function<optional<B>(optional<A>)>
   fmap(std::function<B(A)> f)
{
   return [f](optional<A> opt)
   {
      if (!opt.isValid())
         return optional<B>{};
      else
         return optional<B>{ f(opt.val()) };
   };
}
```

Это функция высшего порядка, принимающая функцию как аргумент и возвращающая функцию. А вот другой вариант, без каррирования:

```
template<class A, class B>
   optional<B> fmap(std::function<B(A)> f,
                        optional<A> opt)
```

```
{
   if (!opt.isValid())
      return optional<B>{};
   else
      return optional<B>{ f(opt.val()) };
}
```

С другой стороны, можно реализовать `fmap` как шаблонный метод `optional`. Такая неоднозначность в выборе делает абстракцию шаблона «функтор» в C++ проблемой. Должен ли функтор быть интерфейсом, дабы от него наследоваться (к сожалению, шаблонные функции не могут быть виртуальными)? А может быть, свободной шаблонной функцией, каррированной или нет? Сможет ли компилятор C++ корректно вывести недостающие типы, или их надо задавать явно? Представьте, что входная функция `f` преобразует `int` в `bool`. Как компилятор определит тип `g`:

```
auto g = fmap(f);
```

особенно, если в будущем появятся множество функторов со своими версиями `fmap`? (С другими видами функторов мы скоро познакомимся.)

Классы типов

Как же абстракция функтора реализована в Haskell? Используется механизм классов типов. Класс типов задаёт семейство типов, поддерживающих единый интерфейс. К примеру, класс объектов, сравнимых на равенство определяется так:

```
class Eq a where
   (==) :: a -> a -> Bool
```

Здесь утверждается, что тип `a` принадлежит классу `Eq`, если он поддерживает оператор `(==)`, который принимает два аргумента типа `a` и возвращает `Bool`. Чтобы убедить Haskell, что определённый тип относится к `Eq`, вам понадобится объявить его экземпляром (реализацией) класса, и предоставить реализацию `(==)`. Например, имея такое определение точки на плоскости (тип-произведение двух `Float`):

```
data Point = Pt Float Float
```

можно определить равенство точек:

```
instance Eq Point where
   (Pt x y) == (Pt x' y') = x == x' && y == y'
```

Здесь определяемый оператор `(==)` расположен инфиксно, между двумя образцами `(Pt x y)` и `(Pt x' y')`. Тело функции помещается справа от знака `=`. После того, как `Point` объявлен экземпляром `Eq`, можно непосредственно сравнивать точки на равенство. Обратите внимание, что в отличие от C++ или Java, вы не обязаны предоставлять класс или даже интерфейс `Eq` в момент определения `Point`, — это можно сделать позже. Замечу, что классы типов, — единственный механизм для перегрузки функций (и операторов) в Haskell. Нам он понадобится, чтобы перегружать `fmap` для разных функторов. Есть одна тонкость: функтор определяется не как тип, а как функция над типами, конструктор типов. Наш класс типов должен описывать семейство конструкторов типов, а не просто типов, как было в случае с `Eq`. К счастью, механизм классов типов Haskell работает с конструкторами типов так же, как и с простыми типами (хороший пример следования функциональной парадигме — даже в мире типов функции ничем не хуже объектов). Вот определение класса `Functor`:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Оно утверждает, что `f` есть `Functor` в том случае, если имеется функция `fmap` с данной сигнатурой. Здесь `f` — типовая переменная, того же рода, что и типовые переменные `a` и `b`. Но компилятор способен понять, что `f` представляет собой конструктор типов, отслеживая её использование: применение к другим типам, здесь `f a` и `f b`. Соответственно, именно конструктор типов мы объявляем экземпляром функтора, как в случае `Maybe`:

```
instance Functor Maybe where
   fmap _ Nothing  = Nothing
   fmap f (Just x) = Just (f x)
```

Замечу, что класс Functor, а также объявления многих общеупотребительных типов, включая Maybe с его экземплярами, входят в стандартную библиотеку Prelude.

Функторы в C++

Можем ли мы применить тот же подход в C++? Конструктору типов соответствует шаблонный класс, такой как optional, поэтому нам надо параметризовать fmap дважды шаблонным параметром F. Записывается это так:

```
template<template<class> F, class A, class B>
   F<B> fmap(std::function<B(A)>, F<A>);
```

Но как нам специализировать этот шаблон для разных функторов? К сожалению, частичная специализация шаблонных функций C++ запрещена. Следующий код вызовет ошибку компиляции:

```
template<class A, class B>
   optional<B> fmap<optional>
                    (std::function<B(A)> f,
                           optional<A> opt)
```

Вместо этого приходится вернуться к перегрузке функций, в результате мы возвращаемся к первоначальному определению fmap (без каррирования):

```
template<class A, class B>
   optional<B> fmap(std::function<B(A)> f,
                           optional<A> opt)
{
   if (!opt.isValid())
      return optional<B>{};
   else
      return optional<B>{ f(opt.val()) };
}
```

Это определение работает, но для правильной перегрузки требуется второй аргумент. Более общее определение fmap просто игнорируется.

Функтор List

Для лучшего понимания значимости функторов в программировании стоит рассмотреть больше примеров. Любой тип, параметризуемый другим типом — кандидат на роль функтора. К примеру, обобщённые контейнеры параметризуются типом своих элементов; рассмотрим список, очень простой контейнер:

```
data List a = Nil | Cons a (List a)
```

Здесь у нас конструктор типов `List`, представляющий собой отображение любого типа `a` в тип `List a`. Чтобы показать, что `List` есть функтор, нам необходимо определить поднятие функций вдоль функтора. Для заданной функции `a -> b` определим функцию `List a -> List b`:

```
fmap :: (a -> b) -> (List a -> List b)
```

Функция, действующая на `List` должна рассмотреть два случая, соответствующие двум конструкторам списка. Случай `Nil` тривиален, — `Nil` же и возвращаем, из пустого списка много не выжмешь. Случай `Cons` хитрее, поскольку затрагивает рекурсию. Подумаем: итак, у нас есть список `a`, функция `f`, превращающая `a` в `b`, и мы хотим получить список `b`. Очевидно, нужно использовать `f`, для преобразования каждого элемента списка из `a` в `b`. Что именно нужно сделать, если наш (непустой) список определён как `Cons` головы и хвоста? Применить `f` к голове и применить поднятое (`fmap`-нутое) `f` к хвосту. Это определение рекурсивное, мы определяем поднятое `f` через поднятое `f`:

```
fmap f (Cons x t) = Cons (f x) (fmap f t)
```

Важно, что в правой части `fmap f` применяется к списку более короткому, чем определяемый нами — а именно к хвосту последнего. Мы применяем рекурсию ко всё более коротким спискам, поэтому неизбежно приходим к пустому списку, или `Nil`. Но как мы уже определили, `fmap f` от `Nil` даёт `Nil`, тем самым завершая рекурсию. Окончательный результат получаем комбинированием новой головы `(f x)` с новым хвостом `(fmap f t)`, используя конструктор `Cons`. В итоге, вот наше объявление списка экземпляром функтора полностью:

```
instance Functor List where
   fmap _ Nil         = Nil
   fmap f (Cons x t) = Cons (f x) (fmap f t)
```

Если для вас привычнее C++, имеет смысл рассмотреть, по сути, базовый контейнер C++, `std::vector`.

Реализация `fmap` для `std::vector` — это просто обёртка вокруг `std::transform`:

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f,
                          std::vector<A> v)
{
   std::vector<B> w;
   std::transform( std::begin(v)
                 , std::end(v)
                 , std::back_inserter(w)
                 , f);
   return w;
}
```

С её помощью мы можем, к примеру, возвести в квадрат ряд чисел:

```
std::vector<int> v{ 1, 2, 3, 4 };
auto w = fmap([](int i) { return i*i; }, v);
std::copy( std::begin(w) , std::end(w) ,
           std::ostream_iterator(std::cout,
                                     ", "));
```

Большинство контейнеров C++ по сути функторы. Это обеспечено наличием итераторов, которые можно передать `std::transform`, — примитивному родственнику `fmap`. К сожалению, простота функтора теряется под нагромождением итераторов и временных переменных (как в реализации `fmap` выше). Из хороших новостей — планируемая к включению в C++ библиотека интервалов (ranges) значительно выявляет их, интервалов, функциональную природу.

Функтор Reader

Теперь, когда мы развили какую-то интуицию, в частности о том, что функторы это своего рода контейнеры, вот вам пример, на первый взгляд выглядящий совершенно иначе. Рассмотрим отображение типа `a` на тип функций, возвращающих `a`. Мы ещё не добрались до обсуждения функциональных типов на должном теоретико-категорном уровне, но у нас, как программистов, есть их определённое понимание. В Haskell функциональные типы строятся с помощью конструктора типов — стрелки `(->)`, который принимает два типа: тип аргумента и тип результата. Вы его уже встречали в инфиксной записи, `a -> b`, но с помощью скобок можно ничуть не хуже использовать и префиксную запись:

```
(->) a b
```

Как и обычные функции, ти́повые функции нескольких аргументов можно применять частично. И когда мы подаём стрелке один аргумент, она всё ещё ждёт другой. То есть,

```
(->) a
```

представляет собой конструктор типов; нужен ещё один тип `b`, чтобы получился полноценный тип `a -> b`. Таким образом, стрелка определяет целое семейство конструкторов типов, параметризованных `a`. Давайте выясним, является ли оно также семейством функторов. Чтобы не путать два параметра, переименуем их, подчеркнув роли. В соответствии с нашими предыдущими определениями функторов, тип аргумента назовём `r`, а тип результата `a`. Итак, наш конструктор берёт любой тип `a`, и отображает его на тип `r -> a`. Чтобы обосновать функторность, нам надо поднять функцию `a -> b` до функции, принимающей `r -> a` и возвращающей `r -> b`. Это типы, создаваемые действием конструктора `(->) r` на `a` и `b`, соответственно. Вот какая получается сигнатура `fmap`:

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Получаем своего рода головоломку: имея функции `f :: a -> b` и функции `g :: r -> a`, создать `r -> b`. Есть только один способ их композиции, и результат как раз такой, как нам надо. Получается такая реализация `fmap`:

```
instance Functor ((->) r) where
   fmap f g = f . g
```

Сработало! Минималист может ещё сократить запись, воспользовавшись префиксной формой:

```
fmap f g = (.) f g
```

теперь аргументы можно опустить, получая отождествление двух функций:

```
fmap = (.)
```

Комбинацию конструктора типов `(->) r` и такой реализации `fmap` называют функтором «reader».

7.2 Функторы как контейнеры

Мы познакомились с примерами использования функторов в программировании для определения контейнеров общего назначения, или хотя бы объектов, содержащих какие-то значения того типа, которым функтор параметризован. Функтор reader кажется исключением, поскольку мы не думаем о функциях как о данных. Но как мы видели, к функциям применима мемоизация, когда вычисление сводится к поиску в таблице. Таблица это уже данные. С другой стороны, поскольку Haskell ленив, традиционный контейнер вроде списка может на деле реализовываться как функция. Посмотрите, например, на бесконечный список натуральных чисел, который можно описать так:

```
nats :: [Integer]
nats  = [1..]
```

Пара квадратных скобок в первой строке, — встроенный типовый конструктор Haskell для списков. Во второй строке с помощью скобок формируется списковый литерал. Очевидно, подобный бесконечный список в памяти разместить невозможно. Компилятор вместо этого создаёт функцию, которая генерирует `Integer`-ы по запросу. Haskell фактически размывает границу между кодом и данными: список можно считать функцией, а функцию можно считать таблицей, сопоставляющей аргументам результаты. Это иногда даже практично, при условии, что область определения конечна, и не слишком велика. Но вот реализовать `strlen` как табличную функцию было бы не практично, поскольку различных строк бесконечно много. Программисты обычно не любят бесконечности, но теория категорий научит вас щёлкать их как орешки. Множество ли это всех возможных строк, или всех состояний вселенной, из прошлого, настоящего или будущего, мы можем работать с этим! Так что я думаю о функторных объектах (объектах типа, сконструированного с помощью эндофунктора) как содержащих значения первоначального типа (аргумента функтора), даже если такое значение не представлено физически. Пример функтора в C++, — `std::future`, который возможно когда-нибудь будет содержать значение, если повезёт; и если вам нужно это значение, возможно, придётся остановиться и подождать, пока другой поток завершит вычисления. Другой пример, это компонента системы ввода/вывода в Haskell или ее будущая версия из нашей вселенной, которая может содержать введённую пользователем фразу `"Hello World!"`, высвеченную на мониторе. При такой интерпретации, функтор есть нечто, способное содержать значение или значения того типа, которым оно параметризовано. Или содержать рецепт создания таких значений. Нас не заботит, можем ли мы посмотреть на это значение, — это не относится к обязательным свойствам функтора. Всё что нас интересует, — возможность манипулировать такими значениями с помощью функций. Если значение было доступно, — мы сможем также увидеть и результат применения функции, если нет, нам достаточно того, чтобы манипуляции подчинялись композиции, а манипуляция с тождественной функцией ничего не меняла. Просто чтобы показать, насколько нам безразличен доступ к значениям внутри функторного объекта, вот вам конструктор, который полностью игнорирует свой аргумент `a`:

```
data Const c a = Const c
```

— типовый конструктор `Const` принимает два типа, `c` и `a`. Чтобы создать функтор, нам надо его частично применить, так же, как мы это делали с конструктором стрелки `(->)`.

```
fmap :: (a -> b) -> Const c a -> Const c b
```

Поскольку наш функтор игнорирует свой типовый аргумент, будет естественным в реализации `fmap` игнорировать функциональный аргумент, — функцию не к чему применять:

```
instance Functor (Const c) where
    fmap _ (Const v) = Const v
```

В C++ это выглядит немного нагляднее (вот уж не думал, что скажу такое!), за счёт более чёткого различия между типовыми аргументами, которые работают на этапе компиляции, и значениями, работающими во время выполнения.

```
template<class C, class A>
    struct Const
{
    Const(C v) : _v(v) {}
    C _v;
};
```

C++-реализация `fmap` также игнорирует функциональный аргумент, и по сути преобразует тип `Const`, не трогая значение.

```
template<class C, class A, class B>
    Const<C, B> fmap(std::function<B(A)> f,
                                Const<C, A> c)
{
    return Const<C, B>{c._v};
}
```

Несмотря на странный вид, функтор `Const` играет важную роль во многих построениях. В теории категорий это частный случай функтора Δ_c, упомянутого ранее — эндофункторной разновидности чёрной дыры. В будущем мы познакомимся с ним поближе.

7.3 Композиция функторов

Несложно убедиться, что для функторов между категориями композиция работает точно также, как для функций между множествами. Композиция двух функторов, при действии на объекты, есть просто композиция соответствующих отображений объектов, аналогичная ситуация с действием на морфизмы. После прыжка вдоль двух функторов, единичные морфизмы остаются единичными, также как и композиция морфизмов остаётся таковой для образов. Ничего особенного. В частности, легко сочетаются эндофункторы. Помните функцию `maybeTail` (из предыдущей главы)? Давайте её перепишем, применительно к стандартным спискам Haskell:

```
maybeTail        :: [a] -> Maybe [a]
maybeTail []      = Nothing
maybeTail (x:xs) = Just xs
```

(Конструктор пустого списка, который мы обозначили `Nil`, заменён пустой парой квадратных скобок `[]`. Конструктор `Cons` заменён инфиксным оператором `:` (двоеточие).) Результат `maybeTail` имеет тип композиции двух функторов, `Maybe` и `[]`, действующих на `a`. Каждый из этих функторов снабжён собственной версией `fmap`, но что если мы хотим применить некоторую функцию `f` к содержимому композиции: `Maybe list`? Необходимо пролезть сквозь два слоя функторов. Используя `fmap` можно пройти сквозь внешний, `Maybe`. Но мы не можем отправить `f` внутрь `Maybe`, поскольку `f` не работает со списками. Нам нужно послать `(fmap f)`, для действия на внутренний список. Например, давайте возведём в квадрат элементы `Maybe list` целых чисел:

```
square x = x * x

mis :: Maybe [Int]
mis  = Just [1, 2, 3]

mis2 = fmap (fmap square) mis
```

Компилятор, проанализировав типы выяснит, что для внешнего `fmap` следует использовать реализацию из экземпляра `Maybe`, а для внутреннего, — реализацию от функтора `list`. Возможно, не совсем очевидно, что код выше можно переписать так:

```
mis2 = (fmap . fmap) square mis
```

Но вспомните, что о `fmap` можно думать как о функции единственного аргумента:

```
fmap :: (a -> b) -> (f a -> f b)
```

В нашем случае, второй `fmap` в `(fmap . fmap)` принимает на вход

```
square :: Int -> Int
```

и возвращает функцию типа

```
[Int] -> [Int]
```

Затем первый `fmap` берёт получившуюся функцию и в свою очередь возвращает

```
Maybe [Int] -> Maybe [Int]
```

И наконец, эта функция применяется к `mis`. Таким образом, композиция двух функторов это функтор, для которого `fmap` есть композиция соответствующих `fmap`. Возвращаясь к теории категорий: достаточно очевидно, что композиция функторов ассоциативна (и отображение объектов ассоциативно, и отображение функторов). Кроме того, в каждой категории имеется единичный функтор, который отображает каждый объект в него же, и аналогично действует с морфизмами. То есть, функторы имеют все необходимые свойства морфизмов некоторой категории. Но что это будет за категория? Это должна быть категория, в которой объекты — это категории, а морфизмы — это функторы. Это категория категорий. Но категория всех категорий должна включать себя также

и себя, что ведёт к появлению парадоксов того же типа, что делают множество всех множеств невозможным. Однако, существует категория всех малых категорий **Cat** (которая велика, и, соответственно, своим членом не является). *Малая* — это такая категория, объекты которой составляют множество, в противоположность чему-то, большему, чем множество. Заметьте, что в теории категорий даже бесконечное несчётное множество считается малым. Наверное, я рассказываю вам всё это, поскольку считаю поразительным, как одни и те же структуры повторяются на разных уровнях абстракции. Позже мы увидим, что функторы тоже составляют категорию.

Упражнения

1. Можно ли превратить типовый конструктор `Maybe` в функтор, определив

   ```
   fmap _ _ = Nothing
   ```

   который игнорирует оба своих аргумента? (Подсказка: проверьте функторные законы)

2. Докажите функторные законы для функтора reader (Подсказка: там всё просто).

3. Реализуйте функтор reader на вашем втором любимом языке (первый, естественно, Haskell).

4. Докажите законы функторов для списка. Начните с допущения, что законы выполняются для хвостовой части списка (то есть используйте индукцию).

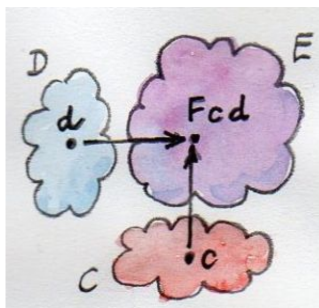
Глава 8

Функториальность

Теперь рассмотрим, как можно строить новые функторы из уже известных. В частности, было бы интересно посмотреть, какие конструкторы типов (соответствующие отображениям между объектами в категории) могут быть расширены до функторов (которые включают в себя отображения между морфизмами).

8.1 Бифункторы

Так как функторы являются морфизмами в **Cat** (категория категорий), интуитивное понимание морфизмов — и функций, в частности, — также переносится и на функторы. Например, так же, как и в случае функции двух аргументов, можно говорить о функторе двух аргументов, или *бифункторе*. На объектах, бифунктор отображает каждую пару объектов, один из категории **C**, а другой из категории **D**, к объекту в категории **E**.



Заметим, что это говорит только о том, что имеется отображение от декартова произведения $\mathbf{C} \times \mathbf{D}$ категорий $\mathbf{C}$ и $\mathbf{D}$ к категории $\mathbf{E}$.

Это довольно просто. Но функториальность означает, что бифунктор должен также отображать и морфизмы. На этот раз, однако, он должен отобразить пару морфизмов, один из $\mathbf{C}$, а другой из $\mathbf{D}$, к одному морфизму из $\mathbf{E}$.

Опять же, пара морфизмов это только единственный морфизм в категорном произведении $\mathbf{C} \times \mathbf{D}$. Определим морфизм в декартовом произведении категорий как пару морфизмов, которые связывают одну пару объектов с другой парой объектов. Эти пары морфизмов могут быть скомпонованы очевидным образом:

$$(f, g) \circ (f', g') = (f \circ f', g \circ g')$$

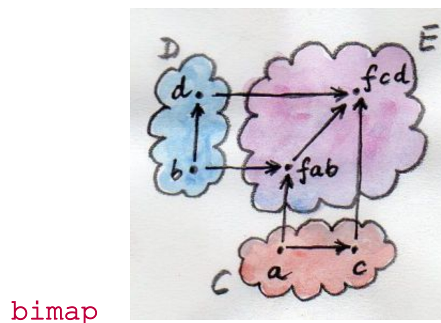
Композиция ассоциативна и имеет единицу — пару единичных морфизмов $(\mathrm{id}, \mathrm{id})$. Так что декартово произведение категорий действительно является категорией.

Более простой способ понимания бифункторов базируется на том, что они являются функторами отдельно по каждому аргументу. Так что вместо того, чтобы транслировать функториальные законы — ассоциативность и сохранение единицы — от функторов к бифункторам, достаточно проверить их по отдельности для каждого аргумента. Однако, вообще говоря, раздельной функториальности недостаточно для доказательства совместной функториальности. Категории, в которых совместная функториальность нарушается, называются *предмоноидальными*. Под *функториальностью* понимается то, что оно действует на морфизмах как истинный функтор.

Давайте определим бифунктор на Haskell. В этом случае все три категории совпадают с категорией типов Haskell. Бифунктор является конструктором типа, который принимает два аргумента типа. Вот определение класса типов `Bifunctor`, взятое непосредственно из библиотеки `Control.Bifunctor`:

```
class Bifunctor f where
   bimap    :: (a -> c) -> (b -> d) ->
                            f a b -> f c d
   bimap g h = first g . second h
   first    :: (a -> c) -> f a b -> f c b
   first g   = bimap g id
   second   :: (b -> d) -> f a b -> f a d
   second    = bimap id
```

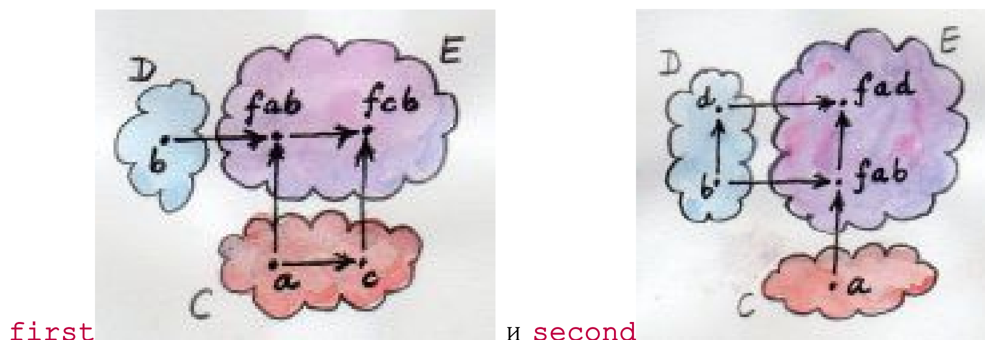
Переменная типа `f` представляет бифунктор. Во всех сигнатурах типа он всегда применяется к двум аргументам типа. Первый тип сигнатуры определяет `bimap`: отображение двух функций одновременно. Результатом является поднятая функция, `(f a b -> f c d)`, работающая на типах, порожденных конструктором бифункторного типа. Существует реализация по умолчанию `bimap` в терминах функций `first` и `second` (но это не всегда работает, потому что, например, `first g . second h` может не совпадать с `second h . first g`).



`bimap`

Две другие сигнатуры типа, функции `first` и `second`, являются двумя `fmap`, свидетельствующими о функториальности `f` по первому и второму аргументу, соответственно.

`first`



и `second`

Определение класса типов обеспечивает реализацию по умолчанию для обоих из них с точки зрения `bimap`.

При объявлении экземпляра `Bifunctor` имеется выбор: либо реализовать `bimap`, принимая значения по умолчанию для `first` и `second`, или реализовать одновременно `first` и `second` и принять значение по умолчанию для `bimap` (конечно, можно реализовать все варианты, чтобы убедиться, что они приводят к одному и тому же результату).

8.2 Произведение и копроизведение как бифункторы

Важным примером бифунктора является категорное произведение — произведение двух объектов, определяемое универсальной конструкцией. Если произведение существует для любой пары объектов, отображение этих объектов к их прозведению бифункториально. Это верно в общем и, в частности, в Haskell. Вот экземпляр от `Bifunctor` для конструктора пары — простейший тип-произведение:

```
instance Bifunctor (,) where
  bimap f g (x, y) = (f x, g y)
```

Собственно, выбора здесь нет: `bimap` просто применяет первую функцию к первой компоненте, а вторую функцию — ко второй компоненте пары. Код в значительной степени описывает сам себя, с учетом типа:

```
bimap :: (a -> c) -> (b -> d) -> (a, b) ->
                                          (c, d)
```

Действие бифунктора заключается здесь в создании пары типов, например:

```
(,) a b = (a, b)
```

В силу двойственности, копроизведение, если оно определено для каждой пары объектов в категории, также является бифунктором. На Haskell, это можно проиллюстрировать на примере конструктора типа `Either` как экземпляра `Bifunctor`:

```
instance Bifunctor Either where
   bimap f _ (Left x)  = Left (f x)
   bimap _ g (Right y) = Right (g y)
```

Этот код также описывает сам себя.

А теперь вспомните, что мы говорили о моноидальных категориях? Моноидальная категория определяет бинарный оператор, действующий на объектах, вместе с единичным объектом. Я упоминал, что **Set** является моноидальной категорией по отношению к декартовому произведению, с одноэлементным множеством в качестве единицы. И оно также является моноидальной категорией относительно несвязного объединения, с пустым множеством в качестве единицы. Что я не упомянул, так это то, что одним из требований, предъявляемых к моноидальной категории является то, что бинарный оператор должен быть бифунктором. Это очень важное требование — мы хотим, чтобы моноидальное произведение было совместимым со структурой категории, которая определяется морфизмами. Сейчас мы находимся еще на шаг ближе к полному определению моноидальной категории (нам еще необходимо узнать больше о естественности, прежде чем мы сможем говорить об этом).

8.3 Функториальные алгебраические типы данных

Мы уже видели несколько примеров параметризованных типов данных, которые оказались функторами — мы сумели определить `fmap` для них. Сложные типы данных построены из более простых типов данных. В частности, алгебраические типы данных (АТД) создаются с использованием сумм и произведений. Мы узнали, что суммы и произведения функториальны. Мы также знаем, что к функторам можно применять операцию композиции. Так что, если мы сможем показать, что основные строительные блоки АТД функториальны, мы будем знать, что параметризованные АТД также функториальны.

Так каковы же строительные блоки параметризованных алгебраических типов данных? Во-первых, есть элементы, которые не зависят от параметра типа функтора, как `Nothing` в `Maybe` или `Nil` в `List`. Они эквивалентны функтору `Const`. Напомню, что функтор `Const` игнорирует параметр типа (на самом деле, только второй параметр типа представляет интерес для нас, первый параметр остается постоянным).

Тогда есть элементы, которые просто инкапсулируют сам параметр типа, как `Just` в `Maybe`. Они эквивалентны тождественному функтору. Я упоминал тождественный функтор ранее, в качестве тождественного морфизма в **Cat**, но не дал его определение на Haskell. Вот оно:

```
data Identity a = Identity a

instance Functor Identity where
   fmap f (Identity x) = Identity (f x)
```

Вы можете воспринимать `Identity` как простейший контейнер, который всегда хранит только одно (неизменное) значение типа `a`.

Все остальное в алгебраических структурах данных строится из этих двух примитивов с использованием произведений и сумм.

С этим пониманием давайте по новому посмотрим на конструктор типа `Maybe`:

```
data Maybe a = Nothing | Just a
```

Это — тип-сумма двух типов, и, как мы знаем, она функториальна. Первая часть, `Nothing` может быть представлена в виде `Const()`, действующего на `a` (первый параметр типа `Const` установлен в единицу — позже мы увидим более интересные использования `Const`). Вторая часть — это просто другое название для функтора идентичности. Мы могли бы определить `Maybe`, с точностью до изоморфизма, как:

```
type Maybe a = Either (Const () a) (Identity a)
```

Таким образом, `Maybe` представляет собой композицию бифунктора `Either` с двумя функторами, `Const()` и `Identity` (`Const` действительно бифунктор, но здесь мы используем его частичное применение).

Мы уже видели, что композиция функторов является функтором — мы можем легко убедиться, что то же самое относится и к бифункторам. Все что нам нужно, это понять, как композиция бифунктора с двумя функторами работает на морфизмах. При заданных двух морфизмах, мы просто поднимаем первый из них с одним функтором, а другой — с другим функтором. Затем поднимаем полученную пару поднятых морфизмов с бифунктором.

Мы можем выразить эту композицию на Haskell. Давайте определим тип данных, параметризированный бифунктором `bf` (это тип переменной, которая является конструктором типа, принимающим два типа в качестве аргументов), двумя функторами `fu` и `gu` (конструкторы типа, которые принимают по одному типу переменной) и двумя обычными типами `a` и `b`. Мы применяем `fu` к `a` и `gu` к `b`, а затем применяем `bf` к полученным в результате двум типам:

```
newtype BiComp bf fu gu a b = BiComp (bf (fu a)
                                         (gu b))
```

Это композиция на объектах или типах. Обратите внимание на то, как на Haskell мы применяем конструкторы типов к типам, — так же, как мы применяем функции к аргументам. Синтаксис такой же.

Если появляется чувство неуверенности, попробуйте применить `BiComp` к `Either`, `Const()`, `Identity`, `a` и `b`, именно в таком порядке. Вы получите нашу версию из `Maybe b` (`a` игнорируется).

Новый тип данных `BiComp` является бифунктором на `a` и `b`, но только если `bf` сам по себе есть `Bifunctor`, а `fu` и `gu` появляются как `Functors`. Компилятор должен знать, что появится определение `bimap`, доступное для `bf`, а также определения `fmap` для `fu` и `gu`. На Haskell это выражается в качестве предварительного условия в объявлении экземпляра: набора ограничений класса с последующей двойной стрелкой:

```
instance (Bifunctor bf, Functor fu, Functor gu) =>
   Bifunctor (BiComp bf fu gu) where
      bimap f1 f2 (BiComp x) = BiComp
                  ((bimap (fmap f1) (fmap f2)) x)
```

Реализация `bimap` для `BiComp` дается в терминах `bimap` для `bf` и двух `fmap` для `fu` и `gu`. Компилятор автоматически выводит все типы и выбирает правильные перегруженные функции всякий раз, когда `BiComp` используется.

`x` в определении `bimap` имеет тип:

```
bf (fu a) (gu b)
```

что довольно громоздко. Внешний `bimap` «прорывает» внешний слой `bf`, а два `fmap` «зарываются» под `fu` и `gu`, соответственно. Если типы `f1` и `f2` есть:

```
f1 :: a -> a'
f2 :: b -> b'
```

то конечный результат должен иметь тип `bf (fu a') (gu b')`:

```
bimap :: (fu a -> fu a')  ->
             (gu b -> gu b') ->
              bf (fu a) (gu b) ->
                  bf (fu a') (gu b')
```

Если вы любите головоломки, подобные виды манипуляций над типами помогут скоротать время.

Так получается, что мы не предоставили доказательства того, что `Maybe` является функтором — этот факт следует из того, как он был построен в виде суммы двух функториальных примитивов.

Проницательный читатель может задать вопрос: если вывод экземпляра `Functor` для алгебраических типов данных настолько механический, не может ли он быть автоматизирован и выполнен компилятором? В самом деле, может быть, и да. Для этого необходимо обеспечить определенное расширение Haskell и включить такую строку в начало исходного файла:

```
{-# LANGUAGE DeriveFunctor #-}
```

а затем добавить `deriving Functor` в структуру данных

```
data Maybe a = Nothing | Just a
   deriving Functor
```

и соответствующий `fmap` будет реализован.

Регулярность алгебраических структур данных позволяет вывести экземпляры не только из `Functor`, но и из нескольких других классов типов, в том числе класса типа `Eq`, упомянутого ранее. Существует также возможность обучения компилятора, чтобы получить экземпляры ваших собственных классов типов, но это немного более продвинутая идея: вы указываете поведение для основных строительных блоков, сумм и произведений, и пусть компилятор сделает все остальное.

8.4 Функторы в C++

Если вы программист на C++, то, очевидно, основываясь на своих убеждениях, можете решить, насколько необходима реализация функторов. Тем не менее, вы должны признать существование некоторых типов алгебраических структур данных в C++. Если из такой структуры данных

создается обобщенный шаблон, вы должны быть в состоянии быстро реализовать fmap для него.

Давайте рассмотрим структуру данных, дерево, которую мы определяем на Haskell как рекурсивную тип-сумму:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
   deriving Functor
```

Как я уже упоминал ранее, один из способов реализации тип-суммы в C++ осуществляется через иерархию классов. Было бы естественно в объектно-ориентированном языке программирования реализовать fmap в качестве виртуальной функции базового класса Functor, а затем переопределить ее во всех подклассах. К сожалению, это невозможно, потому что fmap представляет собой шаблон, параметризованный не только по типу действующего объекта (указатель this), но и типом возвращаемого значения функции, которая была применена к нему. Виртуальные функции не могут быть шаблонными в C++. Мы будем реализовывать fmap как родовую свободную функцию, заменяя соответствие шаблону с помощью dynamic_cast.

Базовый класс должен задать, по крайней мере, одну виртуальную функцию для того, чтобы поддерживать динамическую специализацию, поэтому мы сделаем деструктор виртуальным (что является хорошей идеей в любом случае):

```
template<class T>
struct Tree
{ virtual ~Tree() {}; };
```

Node есть тип-произведение:

```
template<class T>
struct Node : public Tree<T>
{
   Tree<T> * _left;
   Tree<T> * _right;
   Node(Tree<T> * l, Tree<T> * r) : _left(l),
                                    _right(r) {}
};
```

При реализации `fmap` мы воспользуемся динамической диспетчеризацией на типе `Tree`. В случае `Leaf` применяется версия `Identity` из `fmap`, а в случае `Node` он рассматривается как бифунктор, компонуемый с двумя копиями функтора `Tree`. Как программист на C++, вы, вероятно, не анализируете код с такой точки зрения, но это хорошее упражнение для развития категорного мышления.

```
template<class A, class B>
Tree<B> * fmap(std::function<B(A)> f,
                         Tree<A> * t)
{
   Leaf<A> * pl = dynamic_cast <Leaf<A>*>(t);
   if (pl)
      return new Leaf<B>(f (pl->_label));
   Node<A> * pn = dynamic_cast<Node<A>*>(t);
   if (pn)
      return new Node<B>
                ( fmap<A>(f, pn -> _left),
                  fmap<A>(f, pn -> _right));
   return nullptr;
}
```

Для упрощения я решил игнорировать вопросы памяти и управления ресурсами, но в работоспособном коде вы, вероятно, будете использовать интеллектуальные указатели (уникальные или общие, в зависимости от вашего стиля программирования).

Сравните это с реализацией `fmap` на Haskell:

```
instance Functor Tree where
   fmap f (Leaf a)    = Leaf (f a)
   fmap f (Node t t') = Node (fmap f t)
                             (fmap f t')
```

Также, эта реализация может быть получена компилятором автоматически.

8.5 Функтор Writer

Я обещал, что вернутся к категории Клейсли, описанной ранее. Морфизмы в этой категории были представлены как *обогащенные* функции, возвращающие структуру данных `Writer`.

```
type Writer a = (a, String)
```

Я говорил, что обогащение так или иначе связано с эндофункторами. И, действительно, конструктор типа `Writer` функториален по `a`. Мы даже не должны реализовать `fmap` для него, потому что это только простой тип-произведение.

Но что представляет собой соотношение между категорией Клейсли и функтором, в общем? Категория Клейсли, будучи категорией, определяет композицию и тождественный морфизм. Позвольте напомнить вам, что композиция задается оператором рыбы (`>=>`):

```
(>=>)    :: (a -> Writer b) -> (b -> Writer c)
                            -> (a -> Writer c)
m1 >=> m2 = \x -> let (y, s1) = m1 x
                      (z, s2) = m2 y
                  in (z, s1 ++ s2)
```

а тождественный морфизм — функцией `return`:

```
return  :: a -> Writer a
return x = (x, "")
```

Оказывается, что, если смотреть на типы этих двух функций достаточно долго (я имею в виду, достаточно *долго*), можно найти способ объединить их для создания функции с правой сигнатурой типа в качестве `fmap`. Как вот это:

```
fmap f = id >=> (\x -> return (f x))
```

Здесь оператор рыбы сочетает в себе две функции: одной из них является `id`, а другой — лямбда-функция, которая применяет `return` к результату, действуя с `f` на свой аргумент. Самый сложный момент здесь, вероятно, использование `id`. Разве это не аргумент оператора рыбы, как предполагается, функция, которая принимает обычный тип и возвращает обогащенный тип? Ну, не совсем. Никто не говорит, что `a` в `a -> Writer b` должен быть обычным типом. Это тип переменной, так что это может быть что угодно, в частности, это может быть обогащенный тип, такой как `Writer b`.

Так что, `id` примет `Writer a` и превратит его в `Writer a`. Оператор рыбы будет вылавливать значение `a` и передаст его в качестве `x` в лямбда-функцию. Там, `f` превратит его в `b`, а `return` обогатит его, превратив в `Writer b`. Собирая все вместе, мы в конечном итоге получаем функцию, которая принимает `Writer a` и возвращает `Writer b` — это именно то, что должна делать `fmap`.

Обратите внимание на то, что этот аргумент является очень общим: можно заменить `Writer` любым конструктором типа. До тех пор, пока поддерживаются оператор рыбы и `return`, вы можете также определить и `fmap`. Так что, обогащение в категории Клейсли — всегда функтор (не любой функтор, однако, приводит к категории Клейсли).

Вы можете спросить, — определенная нами `fmap`, — это та же `fmap`, которая сформирована компилятором с помощью `deriving Functor`? Это связано с тем, как Haskell реализует полиморфные функции, что называется *параметрическим полиморфизмом*. Это — источник так называемых *бесплатных теорем*. Одна из этих теорем гласит, что, если имеется реализация `fmap` для данного конструктора типа, который сохраняет тождественность, то она должны быть уникальной, т.е. единственной.

8.6 Ковариантные и контравариантные функторы

Теперь, когда мы рассмотрели функтор записи, давайте вернемся к функтору чтения. Он основан на частичном применении конструктора типа функция-стрелка:

```
(->) r
```

Можно переписать его как синоним типа:

```
type Reader r a = r -> a
```

для которого экземпляр от `Functor`, как мы уже видели раньше, выглядит так:

```
instance Functor (Reader r) where
   fmap f g = f . g
```

Но так же, как и конструктор типа пары, или конструктор типа `Either`, конструктор функционального типа принимает два аргумента — типы. Пара и `Either` функториальны по обоим аргументам — они являются бифункторами. Является ли конструктор функционального типа также бифунктором?

Давайте попробуем сделать его функториальным по первому аргументу. Сначала создадим синоним типа — такого же, как `Reader`, но с переставленными местами аргументами:

```
type Op r a = a -> r
```

На этот раз мы фиксируем тип возвращаемого значения `r` и изменяем тип аргумента `a`. Давайте посмотрим, сможем ли мы как-нибудь обеспечить соответствие типам в целях реализации `fmap`, которая будет иметь следующую сигнатуру типа:

```
fmap :: (a -> b) -> (a -> r) -> (b -> r)
```

С помощью всего лишь двух функций, принимающих `a`, и возвращающих, соответственно, `b` и `r`, нет просто никакого способа построить функцию принятия `b` и возврата `r`! Было бы иначе, если бы мы могли как-то перевернуть первую функцию, так чтобы она принимала `b` и

возвращала `a` вместо этого. Мы не можем инвертировать произвольную функцию, но мы можем перейти к двойственной категории.

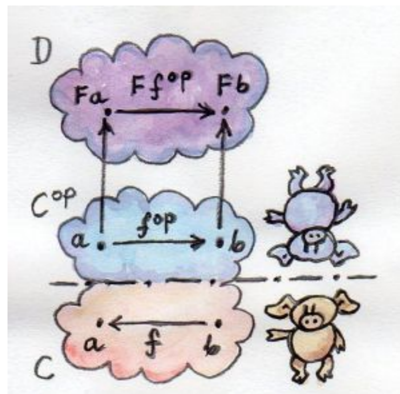
Краткое напоминание: для любой категории $\mathbf{C}$ существует двойственная категория $\mathbf{C}^{op}$. Эта категория с теми же объектами, что и в $\mathbf{C}$, но направления всех стрелок которой изменены на противоположные.

Рассмотрим функтор , который идет от $\mathbf{C}^{op}$ к какой-либо другой категории $\mathbf{D}$:

$$F :: \mathbf{C}^{op} \to \mathbf{D}$$

Такой функтор отображает морфизм $f^{op} :: a \to b$ из $\mathbf{C}^{op}$ к морфизму $F\, f^{op} :: F\, a \to F\, b$ из $\mathbf{D}$. Но морфизм f^{op} соответствует некоторому морфизму $f :: b \to a$ в исходной категории $\mathbf{C}$. Обратите внимание на инверсию.

Теперь, F является регулярным функтором, но существует другое отображение, которое можно определить на основе F, и которое не является функтором — назовем его G: это отображение от $\mathbf{C}$ к $\mathbf{D}$. Оно отображает объекты точно так же, как и F, но когда дело доходит до отображения морфизмов, оно меняет их направление: принимает морфизм $f :: b \to a$ из $\mathbf{C}$, отображает его сначала к противоположному морфизму $f^{op} :: a \to b$, а затем использует функтор F на нем, чтобы получить $F\, f^{op} :: F\, a \to F\, b$.



Принимая во внимание, что $F\, a$ совпадает с $G\, a$ и $F\, b$ является таким же, как $G\, b$, в итоге можно записать $G\, f :: (b \to a) \to (G\, a \to G\, b)$.

Это — «функтор с завихрением». Отображение категорий, которое обращает направление морфизмов таким образом, называется *контравариантным функтором*. Обратите внимание на то, что контравариантный

функтор — это просто обычный функтор от противоположной категории. Кстати, регулярные функторы того вида, которые мы изучали до сих пор, называются *ковариантными функторами*.

Вот определяющий класс типов контравариантных функторов (на самом деле, контравариантных эндофункторов) на Haskell:

```
class Contravariant f where
   contramap :: (b -> a) -> (f a -> f b)
```

Наш конструктор типа `Op` является его экземпляром:

```
instance Contravariant (Op r) where
   -- (b -> a) -> Op r a -> Op r b
   contramap f g = g . f
```

Обратите внимание, что функция `f` вставляет себя перед (т.е. справа) содержимым `Op` — функцией `g`.

Определение `contramap` для `Op` может быть сделано более компактным, если воспользоваться специальной функцией перестановки аргументов:

```
flip      :: (a -> b -> c) -> (b -> a -> c)
flip f y x = f x y
```

С ее помощью мы получаем:

```
contramap = flip (.)
```

Важные примеры функторов предоставляют hom-множества. Пусть $\mathbf{C}$ — категория с малыми hom-множествами, так что каждое множество $\mathrm{hom}(a,b)=\{f \mid f : a \to b \text{ в } \mathbf{C}\}$ является малым и потому входит как объект в категорию $\mathbf{Set}$ всех малых множеств. Тогда для каждого объекта $a \in \mathbf{C}$ имеется *ковариантный* hom-*функтор*

$$\mathbf{C}(a,-) = \mathrm{hom}(a,-) : \mathbf{C} \to \mathbf{Set}\,;$$

его функция объектов отображает каждый объект b в множество $\hom(a,b)$, а функция стрелок отображает каждую стрелку $k : b \to b'$ в функцию

$$\hom(a,k) : \hom(a,b) \to \hom(a,b')\,,$$

которая строится по правилу $f \mapsto k \circ f$ для каждой стрелки $f : a \to b$.

Контравариантный hom-*функтор* для произвольного объекта $b \in \mathbf{C}$ можно записать в ковариантном виде как

$$\mathbf{C}(-,b) = \hom(-,b) : \mathbf{C}^{op} \to \mathbf{Set}\,;$$

отображающий каждый объект a в множество $\hom(a,b)$, а каждую стрелку $g : a \to a'$ категории $\mathbf{C}$ в функцию

$$\hom(g,b) : \hom(a',b) \to \hom(a,b)\,,$$

которая строится по правилу $f \mapsto f \circ g$. Таким образом, для каждой стрелки $f : a' \to b$ имеем

$$\hom(-,k)f = k \circ f, \qquad \hom(g,-)f = f \circ g\,.$$

8.7 Профункторы

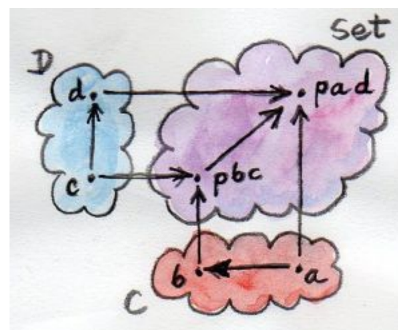
Мы уже видели, что оператор функция-стрелка контравариантен по первому аргументу и ковариантен по второму. Есть ли название для такого случая? Оказывается, что, если целевой категорией является **Set**, то такой оператор называется *профунктором*. Поскольку контравариантный функтор эквивалентен ковариантному функтору от двойственной категории, профунктор определяется следующим образом:

$$\mathbf{C}^{op} \times \mathbf{D} \to \mathbf{Set}$$

Так как, в первом приближении, типы Haskell являются множествами, мы применяем имя `Profunctor` к конструктору типа `p` двух аргументов, который не функториален по первому аргументу, но является функториальным по второму. Вот соответствующий типовый класс из библиотеки `Data.Profunctor`:

```
class Profunctor p where
   dimap    :: (a -> b) -> (c -> d) -> p b c
                                    -> p a d
   dimap f g = lmap f . rmap g
   lmap     :: (a -> b) -> p b c -> p a c
   lmap f    = dimap f id
   rmap     :: (b -> c) -> p a b -> p a c
   rmap      = dimap id
```

Все три функции поставляются с реализацией, по умолчанию. Так же, как в ситуации с `Bifunctor`, при объявлении экземпляра `Profunctor` у вас есть выбор: либо реализации `dimap`, принимая значения по умолчанию для `lmap` и `rmap`, или реализации как `lmap` так и `rmap`, принимя значение по умолчанию для `dimap`:



dimap

Теперь мы можем утверждать, что оператор функция-стрелка является экземпляром `Profunctor`:

```
instance Profunctor (->) where
   dimap ab cd bc = cd . bc . ab
   lmap           = flip (.)
   rmap           = (.)
```

Профункторы имеют свое применение в библиотеке линз Haskell. Мы увидим их снова, когда будем говорить о концах и ко-концах.

8.8 hom-функтор

Приведенные выше примеры являются отражением более общего утверждения о том, что отображение, которое использует пару объектов a и b и сопоставляет ей множество морфизмов между ними, $\hom$-множество $\mathbf{C}(a, b)$, является функтором. Это есть функтор от категории произведения $\mathbf{C}^{\mathsf{op}} \times \mathbf{C}$ к категории множеств $\mathbf{Set}$.

Определим его действие на морфизмах. Морфизм в $\mathbf{C}^{\mathsf{op}} \times \mathbf{C}$ — это пара морфизмов из $\mathbf{C}$:

$$f :: a' \to a$$
$$g :: b \to b'$$

Поднятие этой пары должно быть морфизмом (функцией) множества $\mathbf{C}(a, b)$ в множество $\mathbf{C}(a', b')$. Просто надо выбрать любой элемент h из $\mathbf{C}(a, b)$ (это морфизм от a к b) и назначить ему:

$$g \circ h \circ f$$

что является элементом в $\mathbf{C}(a', b')$.

Ясно, что $\hom$-функтор — это частный случай профунктора.

Упражнения

1. Покажите, что тип данных:

   ```
   data Pair a b = Pair a b
   ```

   является бифунктором. Для укрепления уверенности в понимании материала реализуйте все методы `Bifunctor` и, используя эквациональные рассуждения, покажите, что эти определения совместимы с реализациями по умолчанию всякий раз, когда они могут быть применены.

2. Покажите существование изоморфизма между стандартным определением `Maybe` и этим, «без сахара»:

```
type Maybe' a = Either (Const () a)
                       (Identity a)
```

Подсказка: определите два отображения между этими двумя реализациями. В дополнение, с помощью эквациональных рассуждений, покажите, что они являются обратными по отношению друг к другу.

3. Попробуем другую структуру данных. Я называю ее `PreList`, потому она является предшественницей `List`. Она заменяет рекурсию параметром типа `b`.

   ```
   data PreList a b = Nil | Cons a b
   ```

   Восстановите предыдущее определение `List`, рекурсивно применяя `PreList` к себе (мы узнаем, как это делается, когда будем говорить о неподвижных точках).

   Покажите, что `PreList` является экземпляром `Bifunctor`.

4. Покажите, что следующие типы данных определяют бифункторы на `a` и `b`:

   ```
   data K2 c a b = K2 c
   data Fst a b  = Fst a
   data Snd a b  = Snd b
   ```

   В дополнение, сверьте свои решения с приведенными в статье Конора Макбрайда[1]

   (C. McBride. Clowns to the Left of me, Jokers to the Right).

5. Определите бифунктор в языке, отличном от Haskell. Реализуйте `bimap` для типичной пары на этом языке.

6. Можно ли `std:map` считать бифунктором или профунктором в двух шаблонных аргументах `Key` и `T`? Как бы вы перепроектировали этот тип данных, чтобы это было так?

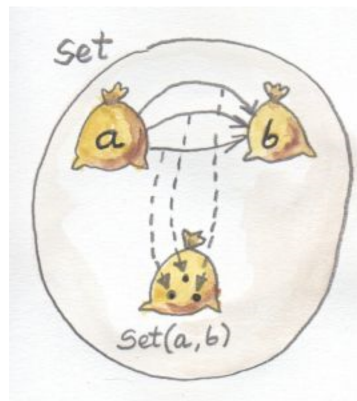
[1] http://strictlypositive.org/CJ.pdf

Глава 9

Функциональные типы

До сих пор я старался не говорить о смысле типов функций, поскольку тип функции отличается от других типов.

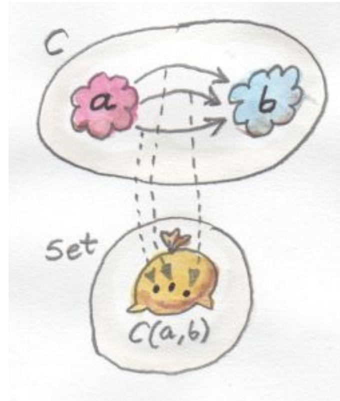
Возьмем, например, `Integer`: это просто множество целых чисел. `Bool` есть множество, состоящее из двух элементов. Но тип функции $a \to b$ говорит о большем: это множество морфизмов между объектами a и b. Совокупность морфизмов между двумя объектами в любой категории называется hom-*множеством*. Просто так получилось, что в категории **Set** каждое hom-множество представляет собой объект в той же категории — потому что оно, в конце концов, множество.



hom-множество в **Set** это просто множество

Это не относится к другим категориям, где hom-множества являются

внешними по отношению к определенной категории. Они даже называются *внешними* hom-множествами.



hom-множество в категории $\mathbf{C}$ является внешним множеством

Этот самореферентный характер категории $\mathbf{Set}$ делает типы функций особенными. Но есть способ, по крайней мере, в некоторых категориях, строить объекты, представляющие hom-множества. Такие объекты называются *внутренними* hom-множествами.

9.1 Универсальная конструкция

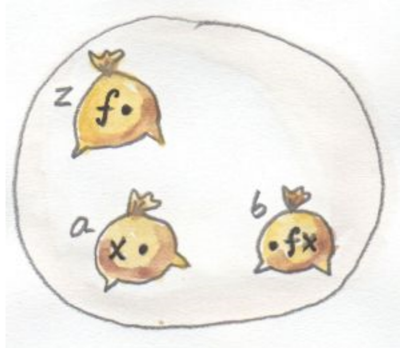
Забудем на мгновение, что функциональные типы являются множествами и попытаемся построить тип функции, или более обще, внутреннее hom-множество, с нуля. Как обычно, мы будем искать аналогии в категории $\mathbf{Set}$, но тщательно избегать использования каких-либо свойств множеств, так что такое конструирование будет автоматически работать и для других категорий.

Функциональный тип может рассматриваться в качестве составного типа из-за его отношения к типу аргументов и типу результата. Мы уже сталкивались с конструкциями составных типов — тех, которые участвуют в отношениях между объектами. Мы использовали универсальные конструкции, чтобы определить произведение и копроизведение типов. Мы можем использовать тот же прием, чтобы определить тип функции.

Нам понадобится шаблон, который включает в себя три объекта: тип функции, который мы строим, типы аргументов и тип результата.

Очевидный шаблон, который соединяет эти три типа, называется *применением функции* или *оценкой*. Имея кандидата для типа функции, обозначим его z (обратите внимание, что, если мы не в категории **Set**, это просто объект, как и любой другой объект), и тип аргумент a (некоторый объект), применение отображает эту пару к типу результата b (некоторый объект). Таким образом, имеются три объекта, два из них — фиксированные (представляющие тип аргументов и тип результата).

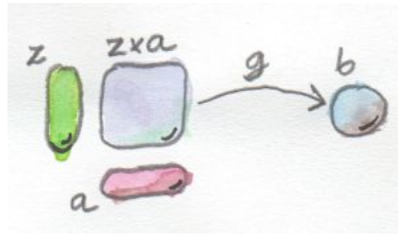
У нас также есть применение, которое является отображением. Как мы включим это отображение в наш шаблон? Если бы нам разрешили заглянуть внутрь объектов, мы могли бы образовать пару из функции f (типа z) с аргументом x (типа a) и отобразить ее к $f\,x$ (применение f к x, которое имеет тип b).



В **Set** мы можем выбрать функцию f из множества функций z, аргумент x из множества (типа) a. Мы получим элемент $f\,x$ в множестве (типа) b.

Но вместо того, чтобы иметь дело с отдельными парами (f, x), мы можем говорить обо всем произведении функций типа z и аргументов типа a. Произведение $z \times a$ является объектом, и мы можем выбрать, в качесте морфизма применения, стрелку g от этого объекта к b. В **Set** g будет функцией, которая отображает каждую пару (f, x) к $f\,x$.

Так что, вот этот шаблон: произведение двух объектов z и a, присоединенных к другому объекту b морфизмом g.

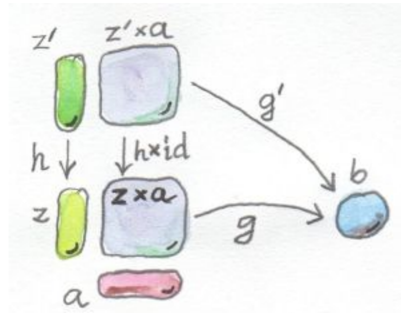


Шаблон объектов и морфизмов, являющийся отправной точкой универсальной конструкции

Является ли эта модель достаточно конкретной, чтобы выделить тип функции, используя универсальную конструкцию? Не в каждой категории, но в категориях, представляющих для нас интерес. И еще вопрос: существует ли возможность определить функциональный объект без предварительного определения произведения? Есть категории, в которых нет ни одного произведения, или нет произведения для всех пар объектов. Ответ на этот вопрос отрицательный: не существует функционального типа, если нет тип-произведения. Мы вернемся к этому позже, когда будем говорить об экспонентах.

Давайте рассмотрим универсальную конструкцию. Начнем с шаблона объектов и морфизмов. Это неоднозначная конструкция, и она, как правило, порождает множество частных случаев. В частности, в **Set**, в значительной степени все связано со всем. Мы можем взять любой объект z, сформировать его произведение с a, и там же найдется функция от этого произведения к b (кроме случаев, когда b является пустым множеством).

Вот когда мы применяем наше секретное оружие: ранжирование. Это обычно делается в соответствии с требованием иметь единственное отображение между объектами кандидатами — это отображение, которое каким-то образом факторизует нашу конструкцию. В этом случае, мы полагаем, что z вместе с морфизмом g от $z \times \mathsf{a}$ к b лучше, чем какой-либо другой z' со своим собственным применением g', тогда и только тогда, когда существует единственное отображение h от z' к z, так что имеем применение факторов g' посредством применения g (подсказка: прочитайте это предложение, глядя на рисунок ниже).



Создание ранжирования между кандидатами на функциональный объект

Теперь сложная часть, и это главная причина, по которой я отложил эту конкретную универсальную конструкцию до сих пор. Основываясь на морфизме $h :: z' \to z$, мы хотим завершить диаграмму, которая имеет как z' так и z, скрещенные с a. Что нам действительно нужно, учитывая отображение h от z' к z, это отображение от $z' \times a$ к $z \times a$. Теперь, после обсуждения функториальности произведения, мы знаем, как это сделать. Поскольку произведение является функтором (точнее эндобифунктором), можно поднять пары морфизмов. Другими словами, мы можем определить не только произведение объектов, но и произведение морфизмов.

Так как мы не касаемся второго компонента произведения $z' \times a$, мы будем поднимать пару морфизмов (h, id), где id является тождественным морфизмом на a.

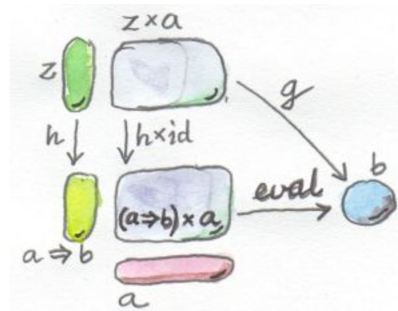
Итак, вот как мы можем факторизовать одно применение функции, g, из другого применения, g':

$$g' = g \circ (h \times \text{id})$$

Ключевым моментом здесь является действие произведения на морфизмы.

Третья часть универсального построения заключается в выборе объекта, который является универсально лучшим. Обозначим этот объект как $a \Rightarrow b$ (следует думать об этом обозначении как о символическом имени для объекта, не следует путать его с ограничением класса типов в Haskell — я буду обсуждать различные способы его именования позже). Этот объект обладает своим собственным применением — морфизмом от $(a \Rightarrow b) \times a$ к b — который мы обозначим $eval$. Объект $a \Rightarrow b$ является наилучшим, если какой-либо другой кандидат для функционального

объекта может быть однозначно сопоставлен с ним таким образом, что его морфизм применения g дофакторизуется через $eval$. Этот объект действительно лучше, чем любой другой объект, в соответствии с нашим ранжированием.



Определение универсального функционального объекта. Это та же диаграмма, что и выше, но теперь объект $a \Rightarrow b$ является универсальным.

Формально:

> *Функциональный объект* от a к b — это объект $a \Rightarrow b$ вместе с морфизмом
>
> $$eval :: ((a \Rightarrow b) \times a) \to b$$
>
> такие, что для любого другого объекта z с морфизмом
>
> $$g :: z \times a \to b$$
>
> существует единственный морфизм
>
> $$h :: z \to (a \Rightarrow b)$$
>
> который факторизует g через $eval$:
>
> $$g = eval \circ (h \times \text{id})$$

Естественно, нет никакой гарантии, что объект $a \Rightarrow b$ существует для любой пары объектов a и b в данной категории. Но это всегда так в **Set**. Кроме того, в **Set**, этот объект изоморфен hom-множеству $\mathbf{Set}(a, b)$.

Именно поэтому, в Haskell, мы интерпретируем функциональный тип `a -> b` как категорный функциональный объект $a \Rightarrow b$.

9.2 Карринг

Давайте повторно рассмотрим всех кандидатов на функциональный объект. На этот раз, однако, будем думать о морфизме g как о функции двух переменных, z и a.

$$g :: z \times a \to b$$

Будучи морфизмом от произведения, g как нельзя лучше подходит на роль функции от двух переменных. В частности, в **Set**, g является функцией от пар значений, одно из множества z, а другое из множества a.

С другой стороны, универсальное свойство говорит нам, что для каждого такого g существует единственный морфизм h, который отображает z к функциональному объекту $a \Rightarrow b$.

$$h :: z \to (a \Rightarrow b)$$

В **Set** это просто означает, что h является функцией, которая принимает одну переменную типа z и возвращает функцию от a к b. Это характеризует h как функцию высшего порядка. Поэтому универсальная конструкция устанавливает взаимно однозначное соответствие между функциями двух переменных и функциями от одной переменной, возвращающих функции. Это соответствие называется *каррингом*, а h называется *каррированной версией* g.

Это соответствие взаимно однозначное, так как для любой заданной функции g существует единственная функция h, а для данной h всегда можно восстановить двухаргументную функцию g, используя формулу:

$$g = eval \circ (h \times \mathrm{id})$$

Функция g называется *некаррированной версией* h.

Карринг по существу встроен в синтаксис Haskell. Функция возвращающая функцию:

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

часто рассматривается как функция от двух переменных. Вот как мы воспринимаем не разделенную скобками сигнатуру:

```
a -> b -> c
```

Эта интерпретация проявляется в том, как мы определим функции нескольких аргументов. Например:

```
catstr      :: String -> String -> String
catstr s s' = s ++ s'
```

Та же функция может быть записана как функция одного аргумента, возвращающая лямбда-функцию:

```
catstr' s = \s' -> s ++ s'
```

Эти два определения эквивалентны, и любой из них может быть частично применен лишь к одному аргументу, производя функцию одного аргумента:

```
greet :: String -> String
greet  = catstr "Hello "
```

Строго говоря, функцией двух переменных является та, которая принимает пару (тип-произведение):

```
(a, b) -> c
```

Преобразование между этими двумя представлениями тривиально, и две функции (высшего порядка), которые делают это, названы, что не удивительно, `curry` и `uncurry`:

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

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

Обратите внимание на то, что `curry` является факторизатором для универсальной конструкции функционального объекта. Это становится очевидным, если записать эту функцию в таком виде:

```
factorizer  :: ((a, b)->c) -> (a->(b->c))
factorizer g = \a -> (\b -> g (a, b))
```

(В качестве напоминания: факторизатор производит факторизующую функцию из кандидата.)

В не функциональных языках, наподобие C++, карринг возможен, но реавлизация его нетривиальна. Вы можете думать о функциях от нескольких аргументов в C++, как о соответствующих функциях Haskell, принимающих кортежи (хотя, чтобы запутать этот вопрос еще больше, в C++ вы можете определить функции, которые принимают явно `std::tuple`, а также функции, принимающие переменное число переменных, или функции, принимающие списки инициализаторов).

Вы можете частично применить функцию C++ с использованием шаблона `std::bind`. Например, с помощью функции от двух строковых аргументов:

```
std::string catstr(std::string s1, std::string s2)
{
   return s1 + s2;
}
```

вы можете определить функцию от одной строковой переменной:

```
using namespace std::placeholders;

auto greet = std::bind(catstr, "Hello ", _1);
std::cout « greet("Haskell Curry");
```

Scala, который является более функциональным языком, чем C++ или Java, находится несколько ближе к Haskell. Если вы предполагаете, что функция, которую вы определяете, будет частично применяться, вы можете определить ее с несколькими списками аргументов:

```
def catstr(s1: String)(s2: String) = s1 + s2
```

Конечно, это требует некоторой предусмотрительности или предвидения от автора соответствующей библиотеки.

9.3 Экспоненциалы

В математической литературе, функциональный объект, или внутренний hom-объект между двумя объектами a и b, часто называют *экспоненциалом* и обозначают b^a. Обратите внимание на то, что тип аргумента находится в показателе. Эта запись, на первый взгляд, может показаться странной, но имеет смысл, если вы думаете о взаимосвязи между функциями и произведениями. Мы уже видели, что необходимо использовать произведение в универсальной конструкции внутреннего hom-объекта, но связь гораздо глубже, чем видится.

Это становится более наглядным, когда вы посчитаете количество функций между конечными типами — типами, которые имеют конечное число значений, такие как `Bool`, `Char` или даже `Int` или `Double`. Такие функции, по крайней мере, в принципе, могут быть полностью превращены в структуры данных. И в этом суть эквивалентности между функциями, которые являются морфизмами и функциональными типами, которые являются объектами.

Например (чистая) функция от `Bool` полностью определяется парой значений: одно соответствует `False`, а другое соответствует `True`. Множество всех возможных функций от `Bool` к, скажем `Int`, есть множество всевозможных пар значений, каждое типа `Int`. Это то же самое, что и произведение $Int \times Int$ или, применяя другую форму записи, Int^2.

В качестве другого примера, давайте рассмотрим тип `char` из C++, который содержит 256 значений (тип `Char` из Haskell содержит большее их количество, потому что Haskell использует Unicode). Имеют-

ся несколько функций в стандартной библиотеки C++, которые обычно реализуются с помощью поиска. Такие функции, как `isupper` или `isspace` реализуются с помощью таблиц, которые эквивалентны кортежу из 256 логических значений. Кортеж имеет тип-произведение, таким образом, мы имеем дело с произведением 256 логических значений: `bool` $\times$ `bool` $\times$ `bool` $\times$ ... $\times$ `bool`, т.е. если «умножать» `bool` сам на себя 256 (или `char`) раз, то получим `bool` в степени `char` или $bool^{char}$.

Сколько значений в типе, определенном как кортеж из 256 значений `bool`? Ровно 2^{256}. Это также число различных функций от `char` к `bool`, каждая функция соответствует единственному кортежу длины 256. Таким же образом можно подсчитать, что число функций от `bool` к `char` равно 256^2, и так далее. Экспоненциальное обозначение для типов функций имеет смысл и в этих случаях.

Мы, вероятно, не хотели бы постоянно пересчитывать количество функций от `int` или `double`. Но эквивалентность между функциями и типами данных существует. Есть также бесконечные типы, например, для списков, строк или деревьев. Подсчет функций от этих типов требует памяти бесконечной емкости. Но Haskell является ленивым языком, поэтому граница между лениво оцениваемыми (бесконечными) структурами данных и функциями размыта. Это объясняет идентификацию функционального типа в Haskell категорным экспоненциальным объектом, который лучше соответствует нашему представлению о данных.

9.4 Декартово замкнутые категории

Хотя я буду продолжать использовать категорию множеств в качестве модели для типов и функций, конечно стоит отметить, что существует большое семейство категорий, которые могут быть использованы для этой цели. Эти категории называются *декартово замкнутыми*, и **Set** является лишь одним из примеров такой категории.

Декартово замкнутая категория должна содержать:

1. терминальный объект,
2. произведение для любой пары объектов и

3. экспоненциал для любой пары объектов.

Если вы рассматриваете экспоненциал в качестве произведения (возможно, бесконечного), то вы можете представлять декартово замкнутую категорию, как поддерживающую произведение произвольной арности. В частности, терминальный объект можно рассматривать как произведение нулевого количества объектов — как нулевую степень объекта.

Что интересно, декартово замкнутые категории с точки зрения компьютерной науки обеспечивают модели для простого типизированного лямбда-исчисления, которая образует основу всех типизированных языков программирования.

Терминальный объект и произведение имеют двойственные аналоги: инициальный объект и копроизведение, соответственно. Декартово замкнутая категория, которая содержит эти объекты и в ней произведение может быть дистрибутивно по копроизведению

$$a \cdot (b + c) = a \cdot b + a \cdot c$$
$$(b + c) \cdot a = b \cdot a + c \cdot a$$

называется *бидекартово замкнутой* категорией. Мы увидим в следующем параграфе, что бидекартово замкнутые категории, среди которых **Set** является ярким примером, имеют некоторые интересные свойства.

9.5 Экспоненциалы и АТД

Интерпретация типов функций, как экспонент, очень хорошо вписывается в схему алгебраических типов данных. Оказывается, что все основные тождества из алгебры, связывающие числовые нуль и единицу, суммы, произведения и степени, неизменны, по существу, в любой бидекартово замкнутой категории для, соответственно, инициальных и терминальных объектов, копроизведений, произведений и экспонент. У нас нет связанных с ними утверждений, их надо формулировать и доказывать (например, касающихся сопряжений, или лемму Йонеды), но я буду упоминать их здесь, тем не менее, в качестве источника ценных интуиций.

Нулевая степень

$a^0 = 1$

В категорной интерпретации, мы заменяем 0 инициальным объектом, 1 — терминальным объектом, а эквивалентность — изоморфизмом. Экспоненциал является внутренним hom-объектом. Данный экспоненциал представляет собой совокупность морфизмов, идущих от инициального объекта к произвольному объекту a. По определению инициального объекта, существует ровно один такой морфизм, поэтому hom-множество $\mathbf{C}(0, \mathrm{a})$ является одноэлементным множеством (синглетоном). Синглетон является терминальным объектом в **Set**, так что эта эквивалентность тривиально работает в **Set**. То, о чем мы говорим, означает, что это работает в любой бидекартово замкнутой категории.

В Haskell, мы заменяем 0 на `Void`, 1 на единичный тип `()`, а экспоненциал на функциональный тип. Утверждается, что множество функций от `Void` к любому типу `a` эквивалентно единичному типу, который является синглетоном. Другими словами, существует только одна функция `Void -> a`. Мы уже сталкивались с этой функцией: она обозначается `absurd`.

Здесь ситуация немного сложнее, по двум причинам. Одной из них является то, что в Haskell не существует «необитаемых» типов — каждый тип содержит «результат бесконечного вычисления», или дно. Вторая причина заключается в том, что все реализации `absurd` эквивалентны, потому что, независимо от того, что они делают, никто никогда не сможет их выполнить. Нет никакого значения, которое может быть передано в `absurd` (а если и удастся передать в него бесконечное вычисление, результат никогда не возвратится!).

Степени первого порядка

$1^a = 1$

Это тождество, при интерпретации в **Set**, переопределяет терминальный объект: существует единственный морфизм от любого объекта к терминальному объекту. В общем, внутренний hom-объект от a к терминальному объекту изоморфен самому терминальному объекту.

В Haskell есть только одна функция от любого типа `a` к единице. Мы уже сталкивались с этой функцией — она именуется `unit`. Вы также можете думать о ней как функции `const`, частично применяемой к `()`.

Первая степень

$a^1 = a$

Это переформулировка наблюдения, что морфизмы от терминального объекта могут быть использованы для того, чтобы выбрать «элементы» объекта a. Множество таких морфизмов изоморфно самому объекту. В **Set** и в Haskell существует изоморфизм между элементами множества `a` и функциями, которые выбирают эти элементы, `() -> a`.

Экспоненциалы сумм

$a^{b+c} = a^b \times a^c$

Категорным языком, это говорит о том, что экспоненциал от копроизведения двух объектов изоморфен произведению двух экспоненциалов. В Haskell это алгебраическое тождество имеет весьма практическую интерпретацию. Оно говорит нам о том, что функция от тип-суммы двух типов эквивалентна паре функций от отдельных типов. Это только анализ конкретного случая, который мы используем при определении функций на суммах. Вместо того, чтобы записывать одно определение функции с выражением `case`, мы обычно делим его на два (или более) определений, касающихся каждого типа конструктора отдельно. Например, возьмем функцию от тип-суммы `(Either Int Double)`:

```
f :: Either Int Double -> String
```

Она может быть определена как пара функций от, соответственно, `Int` и `Double`:

```
f (Left n)  = if n < 0   then "Negative int"
                         else "Positive int"
f (Right x) = if x < 0.0 then "Negative double"
                         else "Positive double"
```

Здесь, `n` есть `Int`, а `x` — `Double`.

Экспоненциалы экспоненциалов

$(a^b)^c = a^{b \times c}$

Это как раз способ выражения каррирования исключительно в терминах экспоненциальных объектов. Функция, возвращающая функцию, эквивалентна функции от произведения (функции с двумя аргументами).

Экспоненциалы над произведениями

$(a \times b)^c = a^c \times b^c$

В Haskell: функция, возвращающая пару, эквивалентна паре функций, каждая из которых производит один из элементов этой пары.

Довольно удивительно, что эти простые алгебраические тождества можно поднять до теории категорий, что имеет практическое применение и в функциональном программировании.

9.6 Изоморфизм Карри-Говарда

Я уже упоминал о соответствии между логикой и алгебраическими типами данных. Тип `Void` и тип единицы `()` соответствуют логическим константам: ложь и истина. Тип-произведение и тип-сумма соответствуют логической конъюнкции $\wedge$ (и) и логической дизъюнкции $\vee$ (или), соответственно. В этой схеме функциональный тип, определенный выше, соответствует логической импликации $\Rightarrow$. Другими словами, тип `a -> b` может быть прочитана как «если `a` то `b`».

В соответствии с изоморфизмом Карри-Говарда, каждый тип может быть интерпретирован как суждение — заявление или решение, которое может быть истинным или ложным. Такое суждение считается истинным, если тип населен, и ложь, если это не так. В частности, логическая импликация является истинной, если тип соответствующей

функции населен, а это значит, что существует функция этого типа. Реализация функции, следовательно, является доказательством теоремы. И написание программ эквивалентно доказательству теоремы. Давайте посмотрим несколько примеров.

Возьмем функцию `eval` которую мы ввели в определении функционального объекта. Ее сигнатура:

```
eval :: ((a -> b), a) -> b
```

Она принимает пару, состоящую из функции и ее аргумента, и выдает результат соответствующего типа. Это есть реализация на Haskell следующего морфизма:

$$eval : (a \Rightarrow b) \times a \to b$$

который определяет тип функции $a \Rightarrow b$ (или экспоненциальный объект b^a). Приведем эту сигнатуру к логическому предикату, используя изоморфизмом Карри-Говарда:

$$((a \Rightarrow b) \wedge a) \Rightarrow b$$

Вот как вы можете прочитать это суждение: если истинно то, что b следует из a, и a — истина, то и b должно быть истиной. Оно несет идеальный интуитивный смысл и известно с древности как правило *модус поненс*. Мы можем доказать эту теорему, реализуя функцию:

```
eval        :: ((a -> b), a) -> b
eval (f, x) = f x
```

Если иметь пару, состоящую из функции f, принимающую a и возвращающую b, и значение x типа a, можно получить конкретное значение типа b, просто применив функцию f к x. При реализации этой функции было показано, что тип $((a \to b), a) \to b$ необитаем. Поэтому правило *модус поненс* верно в нашей логике.

А как насчет предиката, который является явно ложными? Например: «если a или b истинно, то a должно быть истиной»:

$$a \vee b \Rightarrow a$$

Это, очевидно, не верно, потому что, если выбрать a, который является ложным, и b, который истинен, то результат должен быть истиным, хотя по допущению он ложен (это контрпример).

Отображая этот предикат в сигнатуру функции, используя изоморфизм Карри-Говарда, получаем:

```
Either a b -> a
```

Сколько ни пытаться, невозможно реализовать эту функцию — вы не можете произвести значение типа `a`, если функция вызывается со значением `Right` (напоминаю, что речь идет о чистых функциях).

И, наконец, мы приходим к смыслу функции `absurd`:

```
absurd :: Void -> a
```

Учитывая, что `Void` преобразуется в ложь, получаем:

```
false => a
```

Все следует из лжи (*ex falso quodlibet*). Вот одно из возможных доказательств (реализации) этого утверждения (функции) на Haskell:

```
absurd (Void a) = absurd a
```

где `Void` определяется как:

```
newtype Void = Void Void
```

Вообще, тип `Void` сложный. Это определение делает невозможным сконструировать значение, потому что для того, чтобы создать его, вы должны его обеспечить. Поэтому функция `absurd` никогда не может быть вызвана.

Это все интересные примеры, но есть ли практическая сторона изоморфизма Карри-Говарда? Наверное, в повседневном программировании,

нет. Но есть языки программирования, такие как Agda или Coq, которые используют преимущества изоморфизма Карри-Говарда доказывать теоремы.

Компьютеры не только помогают математикам делать свою работу — они являются инструментом, помогающим революционировать самые основы математики. Одной из последних горячих тем исследования в этой области называют гомотопическую теорию типов, которая является результатом исследований на стыке топологии, теории категорий, (общей) теории типов. Она полна для булевских и целых чисел, произведений и копроизведений, функциональных типов и т.д. И, как будто, чтобы рассеять любые сомнения, теория формулируется в Coq и Agda. Компьютеры участвуют в преобразовании мира более чем одним способом.

Библиография

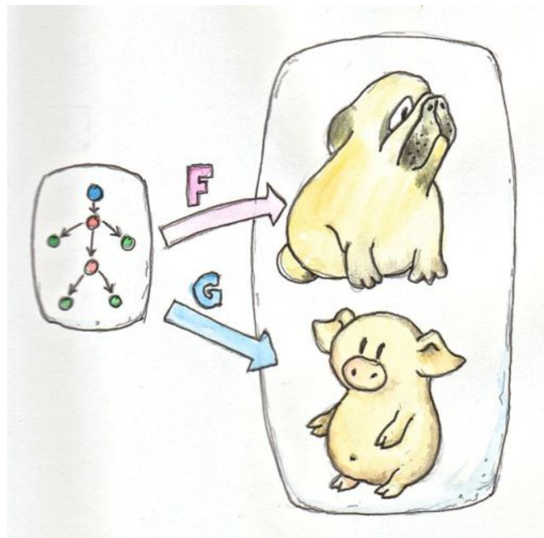
1. Ralph Hinze, Daniel W. H. James, *Reason Isomorphically*! WGP '10 September 26, 2010, Baltimore, Maryland, USA[1] (статья содержит доказательства в теории категорий всех алгебраических тождеств, которые упоминались выше).

[1] http://www.cs.ox.ac.uk/ralf.hinze/publications/WGP10.pdf

Глава 10

Естественные преобразования

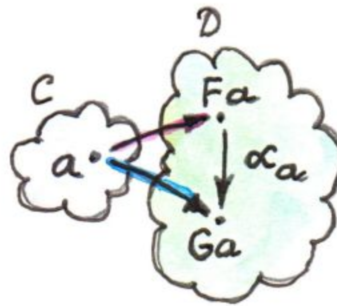
Мы говорили о функторах как об отображениях между категориями, которые сохраняют их структуру. Функтор «встраивает» одну категорию в другую. Он может объединить несколько сущностей в одно целое, но никогда не разрывает связи. Одним из способов представить это является то, что с помощью функтора мы моделируем одну категорию внутри другой. Исходная категория служит моделью, планом, для некоторой структуры, которая является частью целевой категории.



Существует много способов вложения одной категории в другую. Ино-

гда они эквивалентны, иногда очень различаются. Один способ позволяет свернуть всю исходную категорию в один объект, другой может отобразить каждый объект на другой объект и каждый морфизм свести к другому морфизму. Один и тот же способ может быть реализован различными методами. Естественные преобразования помогают нам сравнить эти реализации. Они являются отображениями функторов — специальными отображениями, сохраняющими их функториальную природу.

Рассмотрим два функтора F и G между категориями $\mathbf{C}$ и $\mathbf{D}$. Если сосредоточиться только на одном объекте a из $\mathbf{C}$, он отображается на два объекта: $F\,a$ и $G\,a$. Поэтому отображению функторов следует сопоставить отображение от $F\,a$ к $G\,a$.



Обратите внимание на то, что $F\,a$ и $G\,a$ являются объектами в одной и той же категории $\mathbf{D}$. Отображения между объектами в этой категории не должны идти вразрез с ней. Мы не хотим создавать дополнительные связи между объектами. Так что естественно использовать существующие связи, а именно морфизмы. Естественное преобразование является выбором морфизмов: для каждого объекта a, оно выбирает один морфизм от $F\,a$ к $G\,a$. Если мы обозначим естественное преобразование α, этот морфизм называется компонентом α на a и обозначается α_a.

$$\alpha_a :: F\,a \to G\,a$$

Имейте в виду, что a является объектом $\mathbf{C}$, в то время как α_a — морфизм в $\mathbf{D}$.

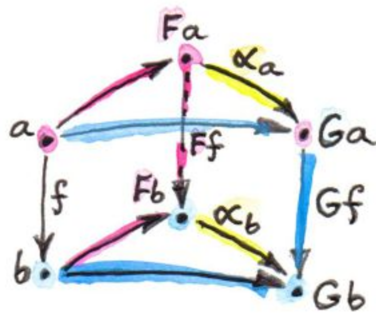
Если для какого-нибудь a не существует морфизма между $F\,a$ и $G\,a$ в $\mathbf{D}$, то не может быть никакого естественного преобразования между F и G.

Конечно, это только половина дела, потому что функторы отображают не только объекты, но и морфизмы. Итак, что же естественное преобразование делает с этими отображениями? Оказывается, что отображение морфизмов фиксировано — при любом естественном преобразовании между F и G, $F\,f$ должно быть преобразовано в $G\,f$. Более того, отображение морфизмов двумя функторами резко ограничивает выбор, совместимый с определением естественного преобразования. Рассмотрим морфизм f между двумя объектами a и b в $\mathbf{C}$. Он отображается на два морфизма, $F\,f$ и $G\,f$ в $\mathbf{D}$:

$$F\,f :: F\,a \to F\,b$$
$$G\,f :: G\,a \to G\,b$$

Естественное преобразование α предоставляет два дополнительных морфизмы, которые завершают диаграмму в $\mathbf{D}$:

$$\alpha_a :: F\,a \to G\,a$$
$$\alpha_b :: F\,b \to G\,b$$



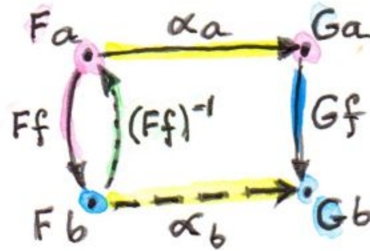
Теперь у нас есть два пути следования от $F\,a$ к $G\,b$. Чтобы убедиться в том, что они равноправны, мы должны наложить *условие естественности*, заключающееся в том, что для любого f имеет место:

$$G\,f \circ \alpha_a = \alpha_b \circ F\,f$$

Это условие естественности является довольно жестким требованием. Например, если морфизм $F\,f$ обратим, естественность определяет α_b в

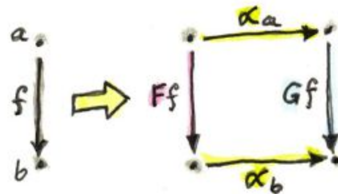
терминах α_a. Она транспортирует α_a вдоль f:

$$\alpha_b = (G\, f) \circ \alpha_a \circ (F\, f)^{-1}$$



Если имеется более одного обратимого морфизма между двумя объектами, все они должны быть согласованы. В общем, однако, морфизмы не являются обратимыми, и вы можете видеть, что существование естественных преобразований между двумя функторами далеко не гарантировано. Таким образом, дефицит или избыток функторов, которые связаны естественными преобразованиями, могут многое рассказать о структуре категорий, между которыми они действуют. Мы увидим некоторые примеры этого, когда будем говорить о пределах и лемме Йонеды.

Если посмотреть на естественное преобразование покомпонентно, можно сказать, что оно отображает объекты в морфизмы. В связи с естественностью, можно также сказать, что оно отображает морфизмы на коммутативные квадраты — существует один коммутативный естественный квадрат в $\mathbf{D}$ для любого морфизма из $\mathbf{C}$.



Это свойство естественных преобразований оказывается очень удобным во многих категориальных конструкциях, которые часто включают коммутативные диаграммы. При разумном выборе функторов многие из этих

условий коммутативности могут быть преобразованы в условия естественности. Мы увидим примеры этого, когда доберемся до пределов, копределов и замыканий.

И, наконец, естественные преобразования могут быть использованы для определения изоморфизма функторов. Сказать, что два функтора естественно изоморфны, это почти то же, если говорить, что они одинаковы. Естественный изоморфизм определяется как естественное преобразование, компоненты которого являются изоморфизмами (обратимыми морфизмами).

10.1 Полиморфные функции

Мы говорили о роли функторов (или, точнее, эндофункторов) в программировании. Они соответствуют конструкторам типов, которые отображают типы на типы. Они также отображают функции на функции, и это отображение осуществляется с помощью функции более высокого порядка `fmap` (или `transform`, `then` и тому подобное в C++).

Чтобы построить естественное преобразование, мы начинаем с некоторогог объекта, здесь типа `a`. Один функтор, скажем `F`, отображает его к типу `F a`. Другой функтор, `G`, отображает его к `G a`. Компонент естественного преобразования `alpha` в `a` является функцией от `F a` к `G a`. В псевдо-Haskell:

$alpha_a$ `:: F a -> G a`

Естественное преобразование является полиморфной функцией, которая определена для всех типов `a`:

```
alpha :: forall a. F a -> G a
```

`forall a` является необязательным в Haskell (а на самом деле, требует включения языкового расширения `ExplicitForAll`). Обычно, подобное выражение следует записывать так:

```
alpha :: F a -> G a
```

Имейте в виду, что на самом деле, это семейство функций, параметризованных `a`. Это еще один пример краткости синтаксиса Haskell. Подобная конструкция в C ++ будет немного более многословной:

```
template<class A> G<A> alpha(F<A>);
```

Существует более глубокое различие между полиморфными функциями в Haskell и обобщенными функциями C ++, и это отражается в том, как эти функции реализованы и типово проверяемы. На Haskell полиморфная функция должна быть определена единообразно для всех типов. Одна формула должна работать для всех типов. Это называется *параметрическим полиморфизмом*.

C ++, с другой стороны, поддерживает по умолчанию *специализированный полиморфизм*, который означает, что шаблон не обязан быть четко определен для всех типов. Работа шаблона для данного типа определяется во время создания экземпляра, когда конкретный тип заменяется параметром типа. Проверка типов откладывается, что, к сожалению, часто приводит к непонятным сообщениям об ошибках.

В C ++ имеется также механизм перегрузки функций и специализации шаблона, что позволяет создавать различные определения одной той же функции для разных типов. В Haskell эта функциональность обеспечивается классами типов и семействами типов.

Параметрический полиморфизм в Haskell имеет неожиданное последствие — любая полиморфная функции типа:

```
alpha :: F a -> G a
```

где `F` и `G` являются функторами, автоматически удовлетворяет условию естественности. Здесь, в категорных обозначениях (`f` является функцией $f :: a \to b$):

$$G\, f \circ \alpha_a = \alpha_b \circ F\, f$$

На Haskell действие функтора `G` на морфизма `f` реализуется с использованием `fmap`. Сначала запишем это в псевдо-Haskell, с аннотациями явных типов:

```
fmapG f . alphaa = alphab . fmapF f
```

Из-за выведения типов, эти аннотации не нужны, и имеет место равенство:

```
fmap f . alpha = alpha . fmap f
```

Это все же не настоящий Haskell — функция равенства не выражается в коде, но это тождество, которое может быть использовано программистом в эквациональном рассуждении, или компилятором для выполнения оптимизаций.

Причина, по которой условие естественности выполняется автоматически в Haskell, имеет отношение к, так называемым, «бесплатным теоремам». Параметрический полиморфизм, который используется для определения естественных преобразований в Haskell, накладывает очень сильные ограничения на реализацию одной формулы для всех типов. Эти ограничения переводятся в эквациональные теоремы о таких функциях. В случае функций, которые преобразовываются функторами, бесплатные теоремы являются условиями естественности (можно прочитать больше о бесплатных теоремах в блоге автора https://bartoszmilewski.com/2014/09/22/parametricity-money-for-nothing-and-theorems-for-free/)

Один из вариантов понимания функторов Haskell, который я упоминал ранее, — считать их обобщенными контейнерами. Мы можем продолжить эту аналогию и рассматривать естественные преобразования как рецепты для переупаковки содержимого одного контейнера в другой контейнер. Мы не касаемся самих предметов: не изменяем их и не создаем новые. Мы просто копируем (некоторые из) их, иногда несколько раз, в новый контейнер.

Условие естественности становится утверждением, что не имеет значения, изменяем ли мы сначала содержимое исходного контейнера с помощью `fmap`, а переупаковываем позже, или переупаковываем в первую очередь, а затем изменяем содержимое в новом контейнере, имеющим свою собственную реализацию `fmap`. Эти два действия, переупаковка и `fmap`инг, ортогональны.

Давайте рассмотрим несколько примеров естественных преобразований на Haskell. Первый пример касается функтора списка и функтора `Maybe`.

Естественное преобразование возвращает голову списка, но только если список не пуст:

```
safeHead        :: [a] -> Maybe a
safeHead []      = Nothing
safeHead (x:xs) = Just x
```

Эта функция полиморфная по `a`. Он работает для любых типов `a`, без каких-либо ограничений, так что это пример параметрического полиморфизма. И поэтому — это действительно естественное преобразование между двумя функторами. Но, просто чтобы убедиться, давайте проверим условие естественности.

```
fmap f . safeHead = safeHead . fmap f
```

Имеем два случая для рассмотрения, пустой список:

```
fmap f (safeHead []) = fmap f Nothing
                     = Nothing
safeHead (fmap f []) = safeHead []
                     = Nothing
```

и не пустой список:

```
fmap f (safeHead (x:xs)) = fmap f (Just x)
                         = Just (f x)
safeHead (fmap f (x:xs)) = safeHead
                          (f x : fmap f xs)
                         = Just (f x)
```

Используем реализацию `fmap` для списков:

```
fmap f []     = []
fmap f (x:xs) = f x : fmap f xs
```

и для `Maybe`:

```
fmap f Nothing  = Nothing
fmap f (Just x) = Just (f x)
```

Интересен случай, когда одним из функторов является тривиальный функтор `Const`. Естественное преобразование от или к функтору `Const` выглядит точно так же, как и функция, которая является полиморфной, либо на типе возвращаемого ею значения, либо на типе ее аргумента.

Например, `length` можно рассматривать как естественное преобразование от функтора списка к функтору `Const Int`:

```
length        :: [a] -> Const Int a
length []      = Const 0
length (x:xs) = Const (1 + unConst (length xs))
```

Здесь `unConst` используется для отделения конструктора `Const`:

```
unConst           :: Const c a -> c
unConst (Const x) = x
```

Конечно, реально `length` определяется так:

```
length :: [a] -> Int
```

что эффективно скрывает тот факт, что это естественное преобразование.

Нахождение параметрически полиморфной функции от функтора `Const` немного сложнее, так как требует создания значения из ничего. Лучшее, что мы можем сделать, это:

```
scam          :: Const Int a -> Maybe a
scam (Const x) = Nothing
```

Другим распространенным функтором, с которым мы уже встречались, и который будет играть важную роль в лемме Йонеды, является функтор `Reader`. Перепишем его определение, используя `newtype`:

```
newtype Reader e a = Reader (e -> a)
```

Он параметризуется двумя типами, но (ковариантно) функториален он только по второму:

```
instance Functor (Reader e) where
   fmap f (Reader g) = Reader (\x -> f (g x))
```

Для каждого типа `e` вы можете определить семейство естественных преобразований от `Reader e` к любому другому функтору `f`. Позже мы увидим, что члены этого семейства всегда находятся во взаимно однозначном соответствии с элементами `f e` (лемма Йонеды).

Например, рассмотрим несколько тривиальный тип единицы `()` с одним элементом `()`. Функтор `Reader ()` принимает любой тип `a` и отображает его в функцию типа `() -> a`. Это все функции, которые выбирают один элемент из множества `a`. Многие из них являются элементами из `a`. Теперь давайте рассмотрим естественные преобразования от этого функтора к функтору `Maybe`:

```
alpha :: Reader () a -> Maybe a
```

Есть только два из них, `dumb` и `obvious`:

```
dumb (Reader _)     = Nothing
obvious (Reader g) = Just (g ())
```

(Единственное, что можно сделать с `g` — применить его к единичному значению `()`)

И действительно, как это и следует из леммы Йонеды, они соответствуют двум элементам типа `Maybe ()`, которыми являются `Nothing` и `Just ()`. Мы вернемся к лемме Йонеды позже, а пока можете считать это небольшой рекламой данной леммы.

10.2 За пределами естественности

Параметрически полиморфная функция между двумя функторами (в том числе и между граничными значениями функтора `Const`) это всегда естественное преобразование. Так как все стандартные алгебраические типы данных являются функторами, любая полиморфная функция между такими типами является естественным преобразованием.

В нашем распоряжении имеются также функциональные функции, а они функториальны в типе возвращаемого значения. Мы можем использовать их для создания функторов (как, например, функтора `Reader`) и определения естественных преобразований, которые являются функциями высшего порядка.

Тем не менее, функциональные типы не ковариантны по типу аргумента. Они контравариантны. Конечно, контравариантные функторы эквивалентны ковариантным функторам из двойственной категории. Полиморфные функции между двумя контравариантными функторами к тому же являются естественными преобразованими в категорном смысле, за исключением того, что они работают над функторами от противоположной категории к типам Haskell.

Возможно, вы помните пример контравариантного функтора, рассмотренный раньше:

```
newtype Op r a = Op (a -> r)
```

Этот функтор контравариантен по `a`:

```
instance Contravariant (Op r) where
   contramap f (Op g) = Op (g . f)
```

Можно написать полиморфную функцию от, скажем, `Op Bool` к `Op String`:

```
predToStr (Op f) = Op (\x -> if f x then "T"
                                    else "F")
```

Но так как эти два функтора не ковариантны, то приведенное не является естественным преобразованием в Hask. Тем не менее, поскольку они оба контравариантны, то удовлетворяют «противоположному» условию естественности:

```
contramap f . predToStr = predToStr . contramap f
```

Обратите внимание на то, что функция `f` должна работать в направлении, противоположном тому, в каком используется `fmap`, в связи с сигнатурой `contramap`:

```
contramap :: (b -> a) -> (Op Bool a
                      -> Op Bool b)
```

Имеются ли конструкторы типов, которые не являются функторами, не важно, ковариантными или контравариантными? Вот один из примеров:

```
a -> a
```

Это не функтор, так как один и тот же тип `a` используется как в отрицательной (контравариантный), так и в положительной роли (ковариантный). Невозможно реализовать `fmap` или `contramap` для этого типа. Поэтому функция сигнатуры:

```
(a -> a) -> f a
```

где `f` — произвольный функтор, не может быть естественным преобразованием. Полезно отметить, что существует обобщение естественных преобразований, называемое диестественными преобразованиями, которые имеют дело с подобными случаями. Мы рассмотрим их, когда будем обсуждать концы.

10.3 Категория функторов

Теперь, когда мы знаем об отображениях между функторами — об естественных преобразованиях — вполне естественно задать вопрос, образуют ли функторы категорию? Действительно, это так! Для каждой пары категорий, $\mathbf{C}$ и $\mathbf{D}$, существует конкретная категория функторов. Объектами в этой категории являются функторы от $\mathbf{C}$ к $\mathbf{D}$, а морфизмами — естественные преобразования между этими функторами.

Мы должны определить композицию двух естественных преобразований, но это довольно легко. Компоненты естественных преобразований являются морфизмами, а мы знаем, как соединять морфизмы.

Действительно, рассмотрим естественное преобразование α от функтора F к функтору G. Его составляющая для объекта a есть некоторый морфизм:

$$\alpha_a :: F\,a \to G\,a$$

Необходимо скомпоновать α с β, которое является естественным преобразованием от функтора G к функтору H. Компонента β для a есть морфизм:

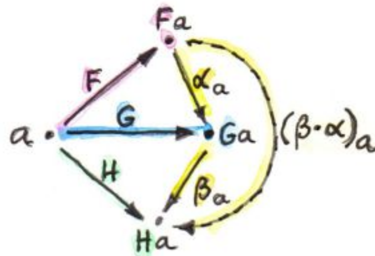
$$\beta_a :: G\,a \to H\,a$$

Эти морфизмы компонуемы и их композицией является еще один морфизм:

$$\alpha_a \circ \beta_a :: F\,a \to H\,a$$

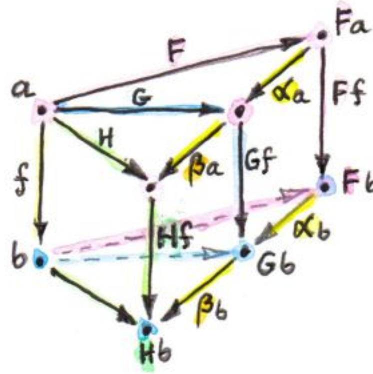
Мы будем использовать этот морфизм как компоненту естественного преобразования $\beta \cdot \alpha$ — композицию двух естественных преобразований α и β (β после α):

$$(\beta \cdot \alpha)_a = \beta_a \circ \alpha_a$$



Один (но пристальный) взгляд на диаграмму убеждает нас в том, что результат этой композиции действительно является естественным преобразованием от F к H:

$$H\ f \circ (\beta \cdot \alpha)_a = (\beta \cdot \alpha)_a \circ F\ f$$



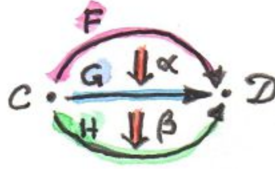
Композиция естественных преобразований ассоциативна, так как их компоненты, которые являются регулярными морфизмами, являются ассоциативными.

Наконец, для каждого функтора F существует тождественное естественное преобразование 1_F, компонентами которого являются тождественные морфизмы:

$$\text{id}_{F\,a} :: F\,a \to F\,a$$

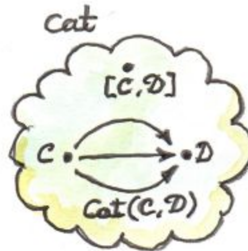
Итак, в самом деле, функторы образуют категорию.

Немного об обозначениях. Следуя Маклейну я использую точку для указания копозиции естественных преобразований, которая только что была описана. Проблема заключается в том, что существует два способа композиции естественных преобразований. Один из них называется вертикальной композицией, так как функторы, как правило, располагают вертикально на описывающих их диаграммах. Вертикальная композиция имеет важное значение при определении категории функторов (горизонтальную композицию я объясню вскоре).



Категория функторов между категориями **C** и **D** записывается в виде **Fun**(**C**,**D**) или [**C**,**D**], а иногда и как $\mathbf{D}^{\mathbf{C}}$. Последнее обозначение предполагает, что категория функторов можно рассматриваться как функциональный объект (экспоненциал) в какой-либо другой категории. Действительно ли это так?

Давайте посмотрим на иерархию абстракций, которые мы использовали. Мы начали с категории, которая представляет собой совокупность объектов и морфизмов. Сами категории (или, строго говоря, малые категории, чьи объекты образуют множества) являются объектами в категории **Cat** более высокого уровня. Морфизмы в этой категории — функторы. hom-множество в **Cat** есть множество функторов. Например **Cat**(**C**,**D**) представляет собой множество функторов между двумя категориями **C** и **D**.



Категория функторов [**C**,**D**] также является множеством функторов между двумя категориями (плюс естественные преобразования в качестве морфизмов). Ее объекты — те же, что и в **Cat**(**C**,**D**). Кроме того, категория функторов, будучи категорией, должна сама быть объектом **Cat** (так получилось, что категория функторов между двумя малыми категориями сама является малой). У нас есть отношения между hom-множеством в категории и некоторым объектом в той же категории. Ситуация в точности такая же, как и в случае с экспоненциальным объектом, с которым

мы встречались в предыдущем разделе. Давайте посмотрим, как мы можем построить последний в **Cat**.

Как вы помните, для того, чтобы построить экспоненциал, необходимо сначала определить произведение. В **Cat** это оказывается относительно легко, потому что малые категории представляют собой множества объектов, и мы знаем, как определять декартовы произведения множеств. Таким образом, объект в категорном произведении $\mathbf{C}\times\mathbf{D}$ это просто пара объектов, (c, d), один из **C**, а другой из **D**. Аналогичным образом, морфизм между двумя такими парами, (c, d) и (c', d'), представляет собой пару морфизмов, (f, g), где $f :: c \to c'$ и $g :: d \to d'$. Такие пары морфизмов составляются покомпонентно и всегда имеется тождественная пара — это просто пара единичных морфизмов. Кратко говоря, **Cat** является полномасштабной декартово замкнутой категорией, в которой существует экспоненциал $\mathbf{D}^{\mathbf{C}}$ для любой пары категорий. Под «объектом» в **Cat** я подразумеваю категорию, так что $\mathbf{D}^{\mathbf{C}}$ является категорией, которую мы можем сопоставить с категорией функторов между **C** и **D**.

10.4 2-категории

Исходя из изложенного, давайте более внимательнее рассмотрим **Cat**. По определению, любое hom-множество в **Cat** есть множество функторов. Но, как мы уже видели, функторы между двумя объектами, имеют более богатую структуру, чем множество. Они образуют категорию с естественными преобразованиями, выступающими в качестве морфизмов. Так как функторы рассматриваются как морфизмы в **Cat**, естественные преобразования являются морфизмами между морфизмами.

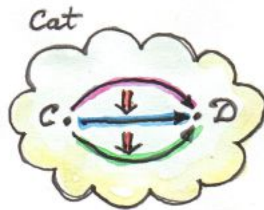
Эта более богатая структура является примером 2-категории, являющейся обобщением категории, где, помимо объектов и морфизмов (которые можно было бы назвать 1-морфизмами в данном контексте), имеются также 2-морфизмы, которые играют роль морфизмов между морфизмами.

В случае **Cat**, рассматриваемой как 2-категория, имеем:

- объекты — (малые) категории;
- 1-морфизмы — функторы между категориями;

- 2-морфизмы — естественные преобразования между функторами.

Вместо hom-множества между двумя категориями $\mathbf{C}$ и $\mathbf{D}$, мы имеем hom-категорию — категорию функторов $\mathbf{D}^{\mathbf{C}}$. У нас есть композиция регулярных функторов: функтор F из $\mathbf{D}^{\mathbf{C}}$ соединяется с функтором G из $\mathbf{E}^{\mathbf{D}}$, давая $G \circ F$ из $\mathbf{E}^{\mathbf{C}}$. Но также имеется композиция и в каждой hom-категории — вертикальная композиция естественных преобразований или 2-морфизмов между функторами.



С двумя видами композиции в 2-категориях, возникает вопрос: каким образом они взаимодействуют друг с другом?

Рассмотрим два функтора или 1-морфизма в $\mathbf{Cat}$:

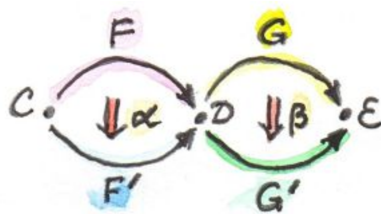
$$F :: \mathbf{C} \to \mathbf{D}$$
$$G :: \mathbf{D} \to \mathbf{E}$$

и их композицию:

$$G \circ F :: \mathbf{C} \to \mathbf{E}$$

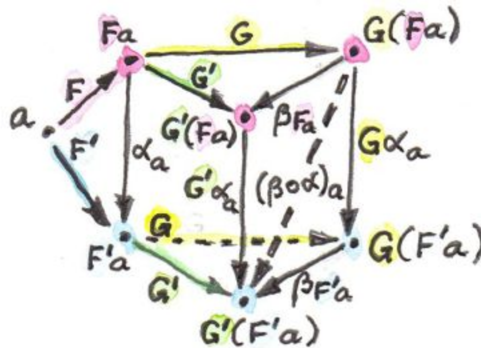
Предположим, имеются два естественных преобразования, α и β, действующие, соответственно, на функторы F и G:

$$\alpha :: F \to F'$$
$$\beta :: G \to G'$$



Обратите внимание на то, что нельзя применить вертикальную композицию к этой паре, потому что результат α отличается от источника β. Фактически, они являются членами двух различных категорий функторов: $\mathbf{D}^{\mathbf{C}}$ и $\mathbf{E}^{\mathbf{D}}$. Можно, однако, применить композицию к функторам F' и G', потому что результат F' является источником G' — это категория $\mathbf{D}$. Каково соотношение между функторами $G' \circ F'$ и $G \circ F$?

Имея α и β, можно ли определить естественное преобразование от $G \circ F$ к $G' \circ F'$? Вот эскиз этой конструкции:



Как обычно, мы начинаем с объекта a из $\mathbf{C}$. Его образ разделяется на два объекта в $\mathbf{D}$: $F\,a$ и $F\,a'$. Существует также морфизм, компонент α, соединяющий эти два объекта:

$$\alpha_a :: F a \to F' a$$

При переходе от $\mathbf{D}$ к $\mathbf{E}$, эти два объекта расщепляются на четыре объекта: $G\,(F\,a)$, $G'\,(F\,a)$, $G\,(F'\,a)$, $G'\,(F'\,a)$.

У нас также есть четыре морфизмам, образующие квадрат. Два из них являются компонентами естественного преобразования β:

$$\beta_{F\,a} :: G\,(F\,a) \to G'\,(F\,a)$$
$$\beta_{F'\,a} :: G\,(F'\,a) \to G'\,(F'\,a)$$

Два других морфизма являются образами α_a под двумя функторами (морфизмы отображений функторов):

$$G\,\alpha_a :: G\,(F\,a) \to G\,(F'\,a)$$
$$G'\,\alpha_a :: G'\,(F\,a) \to G'\,(F'\,a)$$

Слишком уж много морфизмов. Наша цель состоит в том, чтобы найти морфизм, который направлен от $G\,(F\,a)$ к $G'\,(F'\,a)$, кандидата на компоненту естественного преобразования, соединяющего два функтора $G \circ F$ и $G' \circ F'$. На самом деле существует не один, а два способа сделать это: $G'\,\alpha_a \circ \beta_{F\,a}$ и $\beta_{F'\,a} \circ G\,\alpha_a$. Они, оказывается, равны, так как сформированный нами квадрат является естественным квадратом для β.

Мы только что определили компоненту естественного преобразования от $G \circ F$ к $G' \circ F'$. Доказательство естественности для этого преобразования довольно просто, при условии, что у вас есть достаточно терпения.

Мы называем это естественное преобразование *горизонтальной композицией* α и β:

$$\beta \circ \alpha :: G \circ F \to G' \circ F'$$

Опять же, следуя Маклейну, я использую маленький кружок для горизонтальной композиции, хотя вы можете также встретить символ $*$ для этого обозначения.

Вот категорное правило: каждый раз, когда вы встречаете композицию, вы должны искать категорию. Если имеется вертикальная композиция естественных преобразований, то это часть категории функторов. Но как насчет горизонтальной композиции? Какая категория связана с ней?

Способ прояснить это — взглянуть на **Cat** под другим углом. Естественные преобразования можно воспринимать не как стрелки между функторами, а как стрелки между категориями. Естественное преобразование находится между двумя категориями, теми, которые связаны функторами, и которые оно трансформирует. Мы можем думать о естественном преобразовании, как соединяющем эти две категории.

Давайте сосредоточимся на двух объектах из **Cat** — на категориях **C** и **D**. Имеется множество естественных преобразований, которые связывают функторы, соединяющие **C** с **D**. Эти естественные преобразования являются нашими новыми стрелками от **C** к **D**. К тому же, существуют

естественные преобразования, связывающие функторы от **D** к **E**, которые можно рассматривать как новые стрелки от **D** к **E**. Горизонтальная композиция представляет собой композицию этих стрелок.

У нас также есть тождественная стрелка от **C** к **C**. Это тождественное естественное преобразование, которое отображает тождественный функтор от **C** к себе. Обратите внимание на то, что единица для горизонтальной композиции является единицей и для вертикальной композиции, но не наоборот.

И, наконец, оба вида композиции удовлетворяют закону чередования:

$$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$$

Я процитирую Маклейна: «Читатель может наслаждаться выписыванием очевидных диаграмм, необходимых для доказательства этого факта».

Существуют еще обозначения, которые могут пригодиться далее. В приведенной новой интерпретации **Cat** имеются два способа установления связей между объектами: используя функтор или используя естественное преобразование. Мы можем, однако, повторно интерпретировать функторные стрелки, как особый вид естественного преобразования: тождественное естественное преобразование, действующее на этом функторе. Так что, вы часто будете встречать обозначение:

$$F \circ \alpha$$

где F — функтор от **D** к **E**, а α — естественное преобразование между двумя функторами, идущими от **C** к **D**. Так как нельзя составлять композицию функтора с естественным преобразованием, то приведенное выражение интерпретируется как горизонтальная композиция тождественного естественного преобразования 1_F с α.

Аналогично:

$$\alpha \circ F$$

— горизонтальная композиция α с 1_F.

10.5 Заключение

На этом завершается первая часть книги. Мы изучили базовую лексику теории категорий. Вы можете думать об объектах и категориях как о

существительных, а о морфизмах, функторах и естественных преобразованиях как о глаголах. Морфизмами соединяются объекты, функторы связывают категории, естественные преобразования управляют функторами.

Но мы также сталкивались с тем, что, представляемое как действие на одном уровне абстракции, становится объектом на следующем уровне. Множество морфизмов превращается в функциональный объект. В качестве объекта оно может быть источником или целью другого морфизма. Это территория функций высшего порядка.

Функтор отображает объекты на объекты, поэтому мы можем использовать его в качестве конструктора типа или параметрического типа. Функтор также отображает морфизмы, так что, к примеру, `fmap` — это функция высшего порядка. Есть несколько простых функторов, наподобие `Const`, произведения и копроизведения, которые могут быть использованы для создания большого разнообразия алгебраических типов данных. Функциональные типы к тому же функториальны, как ковариантно, так и контравариантно, и могут быть использованы для расширения алгебраических типов данных.

Функторы можно рассматривать как объекты в категории функторов. Таким образом, они становятся источниками и целями морфизмов — естественных преобразований. Естественное преобразование представляет собой особый тип полиморфных функций.

Упражнения

1. Определите естественное преобразование от функтора `Maybe` к функтору списка. Докажите условие естественности для него.

2. Определите, по крайней мере, два различных естественных преобразования между `Reader ()` и функтором списка. Сколько имеется различных списков `()`?

3. Продолжите предыдущее упражнение с `Reader Bool` и `Maybe`.

4. Покажите, что горизонтальная композиция естественного преобразования удовлетворяет условию естественность (подсказка: ис-

пользуйте компоненты). Это хорошее упражнение на понимание диаграмм.

5. Напишите небольшое эссе на тему того, как можно создать очевидные диаграммы, необходимые для доказательства закона чередования.

6. Создайте несколько тестовых примеров для противоположного условия естественности преобразований между различными контравариантными функторами. Вот один вариант:

```
op :: Op Bool Int
op  = Op (\x -> x > 0)
```

и

```
f  :: String -> Int
f x = read x
```

Часть II

Глава 11

Теория категорий и декларативное программирование

В первой части книги я утверждал, что и теория категорий и программирование имеют дело с композицией. В программировании, вы декомпозируете проблему, пока не достигнете того уровня детализации, когда вы сможете решить каждую подзадачу, а затем собираете решения снизу вверх. Есть, грубо говоря, два способа сделать это: говоря компьютеру, что надо сделать, или, предлагая, как это сделать. Первый из этих способов называется декларативным, а другой — императивным.

В этом можно убедиться даже на базовом уровне. Сама композиция может быть определена декларативно, как, например, `h` представляет собой выполнение `g` после выполнения `f`:

```
h = g . f
```

или императивно: сначала вызвать `f`, запомнить результат этого вызова, а затем вызвать `g` с этим результатом (в качестве аргумента):

```
h x = let y = f x
      in g y
```

Императивная версия программы обычно описывается как последовательность действий, упорядоченных во времени. В частности, вызов `g` не может произойти до завершения выполнения `f`. По крайней мере, это концептуальная картина; в ленивом же языке, с вызовом по необходимости при передаче аргументов, фактическое исполнение может действовать по-другому.

На самом деле, в зависимости от возможностей компилятора, может быть незначительным или отсутствовать различие между выполнением декларативного и императивного кода. Но эти две методологии различаются, иногда резко, в подходах к решению проблем, в сопровождении и тестируемости полученного кода.

Основной вопрос, когда сталкиваются с проблемой, всегда ли есть выбор между декларативным и императивным подходами к ее решению? И, если есть декларативное решение, всегда ли оно может быть переведено в машинный код? Ответ на этот вопрос далеко не очевиден, и, если бы мы могли найти его, то, вероятно, совершили бы переворот в понимании этой области.

Позвольте уточнить. Подобная двойственность существует в физике, которая либо указывает на какой-то глубокий основополагающий принцип, или говорит нам о том, как работает наш мозг. Ричард Фейнман упоминает эту двойственность как вдохновение, в своей работе по квантовой электродинамике.

Существуют две формы выражения большинства законов физики. Одна использует локальные, или бесконечно малые, аспекты. Мы смотрим на состояние системы в малой окрестности, и пытаемся предсказать, как она будет развиваться в следующие моменты времени. Это, как правило, выражается с помощью дифференциальных уравнений, которые должны быть интегрированы, или суммируемы, за определенный период времени.

Обратите внимание на то, как этот подход напоминает императивное мышление: мы достигаем окончательного решения, выполняя последовательность небольших шагов, каждый в зависимости от результата предыдущего. На самом деле, компьютерное моделирование физических систем обычно реализуется путем преобразования дифференциальных уравнений в разностные уравнения и их итерацию. Это похоже на ситуацию с космическими кораблямии, анимированными в игре “Астероиды”. На каждом временном шаге, положение космического корабля изменяет-

ся путем добавления небольшого приращения, которое рассчитывается путем умножения его скорости на интервал времени. Скорость, в свою очередь, изменяется на малую величину, пропорциональную ускорению, которое задается силой, деленной на массу.

Они являются непосредственными кодировками дифференциальных уравнений, соответствующих законам движения Ньютона:

$$F= m\frac{dv}{dt}$$
$$v= \frac{dx}{dt}$$

Подобные методы могут быть применены к более сложным проблемам, таких, как распространение электромагнитных полей с помощью уравнений Максвелла, или даже поведение кварков и глюонов внутри протона с использованием решетки QCD (квантовой хромодинамики).

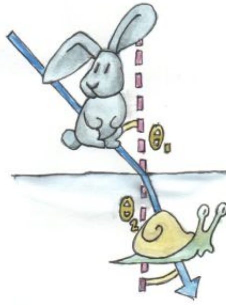
Это локальное мышление в сочетании с дискретностью пространства и времени, которое поощряется за счет использования цифровых вычислительных машин, нашли свое крайнее выражение в героической попытке Стивена Вольфрама свести сложность всей вселенной к системе клеточных автоматов.

Другой подход носит глобальный характер. Мы смотрим на начальное и конечное состояние системы и рассчитываем траекторию, которая соединяет их путем минимизации некоторого функционала. Простейшим примером является принцип наименьшего времени Ферма. Он утверждает, что световые лучи распространяются вдоль путей, которые сводят к минимуму время их перемещения. В частности, при отсутствии отражающих или преломляющих объектов, световой луч от точки A до точки B пройдет самый короткий путь, который представляет собой прямую

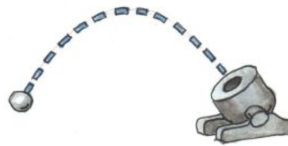
линию. Но свет распространяется медленнее в плотных (прозрачных) материалах, таких как вода или стекло. Так что если вы выбираете начальную точку в воздухе, а конечную точку под водой, то более «выгодным» для света является более длительное перемещение в воздухе, чем прохождение через воду. Путь, минимальный по времени, заставляет луч преломляться на границе воздуха и воды, что приводит к закону Снелла:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2}$$

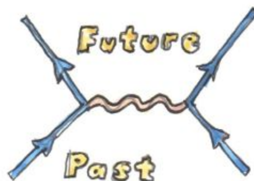
где v_1 — скорость света в воздухе, а v_2 — скорость света в воде.



Все из классической механики можно вывести из принципа наименьшего действия. Действие может быть расчитано для любой траектории путем интегрирования лагранжиана, который представляет собой разницу между кинетической и потенциальной энергией (заметьте: это разность, а не сумма — сумма дала бы полную энергию). Когда вы стреляете из орудия, чтобы поразить заданную цель, снаряд сначала летит вверх, повышая свою потенциальную энергию, и находится там некоторое время, внося отрицательный вклад в действие. Он будет также замедляться в верхней части параболы, минимизируя кинетическую энергию. Затем начнется ускорение для быстрого вхождения в область низкой потенциальной энергии.



Наибольший вклад Фейнмана заключается в понимании того, что принцип наименьшего действия можно перенести на квантовую механику. Там, опять же, задача формулируется в терминах исходного и конечного состояний. Интеграл пути Фейнмана между этими состояниями используется для вычисления вероятности перехода.



Дело в том, что есть любопытная необъяснимая двойственность в том, как мы можем описать законы физики. Мы можем использовать локальную картину, в которой все действия происходят последовательно и небольшими порциями. Или же мы можем использовать глобальную картину, где мы объявляем начальные и конечные условия и все, что из этого следует.

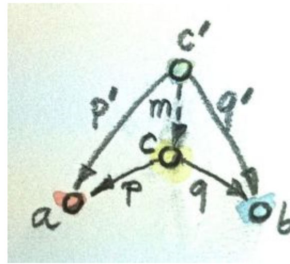
Глобальный подход может быть также использован в программировании, например, при реализации задачи трассировки лучей. Мы фиксируем положение глаз и положения источников света, а также пути для световых лучей, которые могут быть учтены в реализации. Мы явно не минимизируем время прохождения каждого луча, но используем закон Снелла и геометрию отражения для этого.

Самая большая разница между локальным и глобальным подходами заклюсается в их трактовке пространства и, что еще важнее, времени. Локальный подход предоставляет немедленное понимание здесь и сейчас, в то время как глобальный подход принимает долгосрочный статический вид, как если бы будущее было предопределено, и мы анализируем только свойства некоторой вечной вселенной.

Как нигде, лучше всего это показано в подходе функционального реактивного программирования (ФРП) к взаимодействию с пользователем. Вместо того, чтобы писать отдельные обработчики для всех возможных действий пользователя, имеющим доступ к некоторым общим изменяемым состоянием, ФРП рассматривает внешние события как бесконечный список, и применяет к нему ряд преобразований. Концептуально,

список всех будущих действий доступен в качестве входных данных в нашей программе. С точки зрения программы нет никакой разницы между списком цифр числа π, списком псевдослучайных чисел, или списком позиций мыши, подключенной к компьютеру и манипулируемой пользователем. В каждом случае, если вы хотите получить n-й элемент, вы должны сначала пройти через первые $n-1$ элементы. При применении временных событий, мы называем это свойство причинно-следственной связью.

Так при чем же здесь теория категории? Я буду утверждать, что теория категорий поощряет глобальный подход и, следовательно, поддерживает декларативное программирование. В первую очередь, в отличие от исчисления, она не содержит встроенного понятия расстояния, или окрестности, или времени. Все, что мы имеем, это абстрактные объекты и абстрактные связи между ними. Если вы можете добраться от A к B через ряд шагов, вы также можете заменить это одним прыжком. Кроме того, важнейшим инструментом теории категорий является универсальная конструкция, которая является воплощением глобального подхода. Мы видели ее в действии, например, в определении категорного произведения. Это было сделано путем выявления некоторого его свойства (весьма декларативный подход), а именно, этот объект оснащен двумя проекциями, и это лучший такой объект — он оптимизирует определенное свойство: свойство факторизации проекций других подобных объектов.



Сравните это с принципом минимального времени Ферма, или принципом наименьшего действия.

С другой стороны, это контрастирует с традиционным определением декартова произведения, которое гораздо более декларативное. Вы описываете, как создать элемент произведения, выбирая один элемент из од-

ного множества, а другой элемент — из другого множества. Это рецепт создания пары. И еще существует возможность разборки пары.

Почти в каждом языке программирования, включая и функциональные языки, вроде Haskell, тип-произведения, тип-копроизведения и функциональные типы встроены, а не определяется универсальными конструкциями; хотя были попытки создания категорных языков программирования (см., например, тезис Tatsuya Hagino[1]).

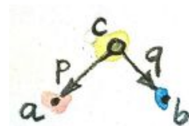
Используются ли непосредственно или нет, категорные определения оправдывают уже существующие программные конструкции, и приводят к новым. Самое главное, что теория категорий обеспечивает мета-язык для рассуждений о компьютерных программах на декларативном уровне. Она также предлагает рассуждения о проблеме спецификации, прежде чем решение будет реализовано в коде.

[1]http://web.sfc.keio.ac.jp/~hagino/thesis.pdf

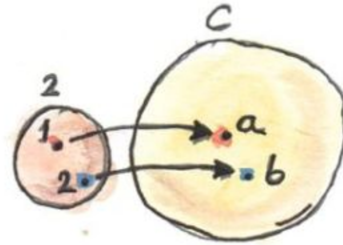
Глава 12

Пределы и копределы

Похоже, что в теории категорий все связано со всем, и все можно рассматривать с разных точек зрения. Возьмем, например, универсальную конструкцию произведения. Теперь, когда мы знаем больше о функторах и естественных преобразованиях, можем ли мы упрощать, а возможно, и обобщать их? Давайте попробуем.

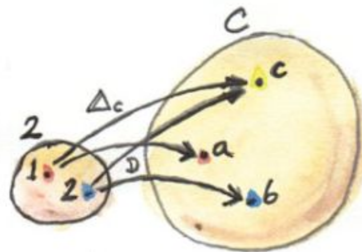


Конструкция произведения начинается с выбора двух объектов a и b, чье произведение мы хотим построить. Но что означает выбрать объекты? Можно ли перефразировать это действие в категорной форме? Два объекта формируют шаблон — очень простой шаблон. Мы можем абстрагировать этот шаблон в категорию — очень простую категорию, но, тем не менее, категорию. Эту категорию будем обозначать **2**. Она содержит только два объекта, 1 и 2, и ни одного морфизма, кроме двух обязательных, тождественных. Теперь можно перефразировать выбор двух объектов в категории **C** как акт, определяющий функтор D от категории **2** к **C**. Функтор отображает объекты на объекты, поэтому его образом являются всего два объекта (или это может быть один, если функтор коллапсирует объекты, это тоже хорошо). Он также отображает морфизмы — в этом случае он просто отображает тождественные морфизмы в тождественные морфизмы.



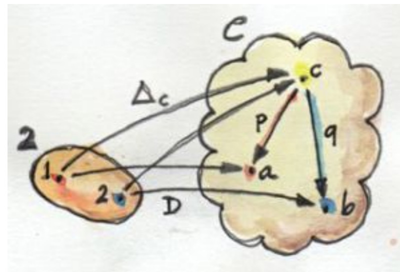
Привлекательность такого подхода заключается в том, что он основывается на категорных понятиях, не используя нестрогих описаний вроде «выбор объектов», взятых прямо из лексикона охотников-собирателей — наших предков. И, между прочим, он также легко обобщается, потому что ничто не может помешать нам при определении моделей использовать категории более сложные, чем **2**.

Но давайте продолжим. Следующим шагом в определении произведения является выбор кандидата на роль объекта c. Здесь снова, мы могли бы перефразировать выбор в терминах функтора от одноэлементной категории. И в самом деле, если бы мы использовали расширения Кана, это было бы правильным решением. Но так как мы не готовы для этого, все же есть еще один трюк: мы можем использовать постоянный функтор Δ от категории **2** к категории **C**. Выбор c в **C** может быть произведен с помощью Δ_c. Напомню, что Δ_c отображает все объекты в c, а все морфизмы — в $\mathtt{id}_c$.



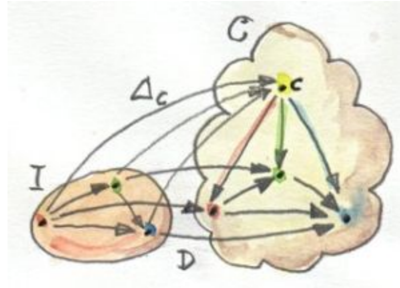
Теперь у нас есть два функтора, Δ_c и D, идущие между **2** и **C**, так что вполне естественно спросить о естественных преобразованиях между ними. Поскольку существует только два объекта в **2**, естественное преобразование будет включать два компонента. Объект 1 в **2** отображается

в c посредством Δ_c, а a — посредством D. Таким образом, компонент естественного преобразования между Δ_c и D на объекте 1 есть морфизм от c к a. Мы можем назвать его p. Аналогично, второй компонент представляет собой морфизм q от c к b — образ объекта 2 в $\mathbf{2}$ под D. Но это такая же картина, как и в случае двух проекций, которые мы использовали в нашем первоначальном определении произведения. Таким образом, вместо того, чтобы говорить о выборе объектов и проекций, мы можем только говорить об отборе функторов и естественных преобразований. Так получилось, что в этом простом случае условие естественности для нашего преобразования тривиально выполняется, потому что нет морфизмов (кроме тождественных) в $\mathbf{2}$.

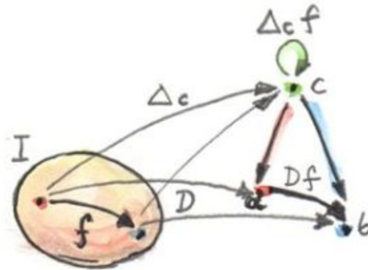


Обобщение этой конструкции на другие категории, отличные от $\mathbf{2}$, те, которые, например, содержат нетривиальные морфизмы, будет вводить условия естественности на преобразование между Δ_c и D. Мы называем такое преобразование *конусом*, потому что образ Δ является вершиной конуса / пирамиды, стороны которого образованы компонентами естественного преобразования. Образ D образует основание конуса.

В общем, чтобы построить конус, мы начинаем с категории $\mathbf{I}$, которая определяет шаблон. Это небольшая, часто конечная категория. Возьмем функтор D от $\mathbf{I}$ к $\mathbf{C}$ и назовем его (или его изображение) диаграммой. Выберем c в $\mathbf{C}$ в качестве вершины нашего конуса. Мы используем его для определения постоянного функтора Δ_c от $\mathbf{I}$ к $\mathbf{C}$. Естественное преобразование от Δ_c к D есть тогда наш конус. Для конечной категории $\mathbf{I}$ это просто совокупность морфизмов, соединяющих c с диаграммой: образ $\mathbf{I}$ под D.



Естественность требует, чтобы все треугольники (грани пирамиды) в этой диаграмме были коммутативны. Действительно, возьмем любой морфизм f в $\mathbf{I}$. Функтор D отображает его в морфизм $D\,f$ в $\mathbf{C}$, морфизм, который образует основание некоторого треугольника. Постоянный функтор Δ_c отображает f к тождественному морфизму на c. Δ схлопывает два конца морфизма в один объект, а естественностью квадрата влечет коммутируемость треугольника. Две стороны этого треугольника являются компонентами естественного преобразования.



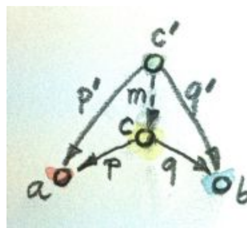
Так что это конус, который является *универсальным конусом*, аналогично мы выбирали универсальный объект для нашего определения произведения.

Есть много способов следовать этому. Например, мы можем определить категорию конусов, основанную на данном функторе D. Объекты в этой категории являются конусами. Не каждый объект c в $\mathbf{C}$ может быть вершиной конуса, хотя бы потому, что может не существовать ни одного естественного преобразования между Δ_c и D.

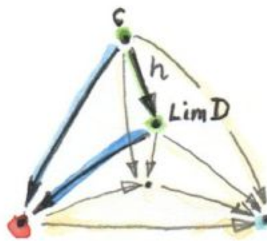
Для того, чтобы завершить определение этой категории, необходимо определить морфизмы между конусами. Они будут в полной мере опре-

деляться морфизмами между их вершинами. Но не любой морфизм годится для этого. Помните, что в конструкции произведения мы ввели условие, что морфизмы между кандидатами в объекты (вершины) должны быть общими факторами для проекций. Например:

```
p' = p . m
q' = q . m
```



Это условие приводит, в общем случае, к свойству: треугольники, у которых одна сторона является факторизующим морфизмом, являются коммутативными.



Коммутирующий треугольник соединяет два конуса факторизующим морфизмом h (в данном случае нижний конус является универсальным, с $\mathrm{Lim}D$ в качестве его вершины).

Мы примем эти факторизующие морфизмы в качестве морфизмов в категории конусов. Легко проверить, что эти морфизмы действительно компонуемы, и что тождественный морфизм также является факторизующим. Конусы, поэтому, образуют категорию.

Теперь мы можем определить универсальный конус в качестве терминального объекта в категории конусов. Определение терминального объекта утверждает, что существует единственный морфизм от любого другого объекта к этому объекту. В нашем случае это означает, что существует единственный факторизующий морфизм от вершины любого другого конуса к вершине универсального конуса. Мы называем этот универсальный конус пределом диаграммы D и обозначаем $\mathrm{Lim}D$ (в литературе, часто используется стрелка влево, к $\mathbf{I}$, под обозначением Lim). В качестве условного термина называют пределом (или предельным объектом) вершину этого конуса.

Интуитивно, предел воплощает в себе свойства всей диаграммы в одном объекте. Например, предел нашей двухобъектной диаграммы является произведением двух объектов. Произведение (вместе с двумя проекциями) содержит информацию об обоих объектах. И, будучи универсальным средством, что он не несет посторонней (лишней) информации.

12.1 Предел как естественный изоморфизм

Существует, однако, что-то неудовлетворительное в этом определении предела. Оно работоспособно, но я имею в виду условие коммутативности для треугольников, которые связываются любые два конуса. Оно воспринималось бы гораздо более элегантным, если бы удалось заменить его некоторым условием естественности. Но как?

Так что, мы уже имеем дело не с одним конусом, а с целой коллекцией (на самом деле, категорией) конусов. Если существует предел (для прояснения, нет никакой гарантии этого), то один из этих конусов является универсальным конусом. Для любого другого конуса мы имеем единственный факторизующий морфизм, который отображает его вершину, давайте обозначим ее c, к вершине универсального конуса, которую мы обозначили $\mathrm{Lim}D$ (на самом деле, я могу пропустить слово «другого», потому что тождественный морфизм отображает универсальный конус к себе и тривиально дофакторизовывается через себя). Позвольте мне повторить важную деталь: для любого конуса существует единственный морфизм специального вида, т.е. имеется отображение конусов в специальных морфизмы, и это отображение взаимно однозначно.

Этот специальный морфизм является членом $\hom$-множества $\mathbf{C}(c, \mathrm{Lim}D)$. Остальным членам этого $\hom$-множества менее повезло в том смысле, что они не факторизуют отображение двух конусов. Все, что мы хотим, это иметь возможность выбрать, для каждого c, один морфизм из множества $\mathbf{C}(c, \mathrm{Lim}D)$ — морфизм, который удовлетворяет определенному условию коммутативности. Не правда ли, что это звучит как определение естественного преобразования? И это, безусловно, так!

Но, что за функторы связаны с этой трансформацией?

Один функтор есть отображение c к множеству $\mathbf{C}(c, \mathrm{Lim}D)$. Это функтор от $\mathbf{C}$ к $\mathbf{Set}$ — он отображает объекты на множества. Фактически, это контравариантный функтор. Вот как мы определяем его действие на морфизмах: рассмотрим морфизм f от c' к c:

$$f :: c' \to c$$

Наш функтор отображает c' на множество $\mathbf{C}(c', \mathrm{Lim}D)$. Для того, чтобы определить действие этого функтора на f (другими словами, чтобы поднять f), мы должны определить соответствующее отображение между $\mathbf{C}(c, \mathrm{Lim}D)$ и $\mathbf{C}(c', \mathrm{Lim}D)$. Так что давайте выберем один элемент u из $\mathbf{C}(c, \mathrm{Lim}D)$ и посмотрим, сможем ли мы сопоставить его с какой-нибудь элементом из $\mathbf{C}(c', \mathrm{Lim}D)$. Элементом $\hom$-множества является морфизм, так что, имеем:

$$u :: c \to \mathrm{Lim}D$$

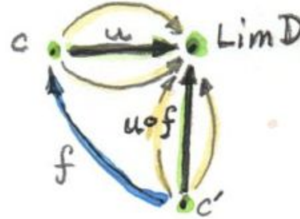
Мы можем предварительно составить композицию f с u, чтобы получить:

$$u\ .\ f :: c' \to \mathrm{Lim}D$$

И это элемент из $\mathbf{C}(c', \mathrm{Lim}D)$ — так что, на самом деле, мы нашли отображение морфизмов, на Haskell это будет выглядеть так:

```
contramap    :: (c' -> c) -> (c  -> LimD)
                          -> (c' -> LimD)
contramap f u = u . f
```

Обратите внимание, что инверсия в порядке следования `c` и `c'` характеризует контравариантный функтор.



Для того, чтобы определить естественное преобразование, нам нужен другой функтор, также отображение от **C** к **Set**. Но на этот раз мы рассмотрим множество конусов. Конусы являются именно естественными преобразованиями, поэтому мы рассмотрим множество естественных преобразований $Nat(\Delta_c, D)$. Отображение от c к этому конкретному множеству естественных преобразований является (контравариантным) функтором. Как мы можем показать это? Опять же, давайте определим его действие на морфизме:

$$f :: c' \to c$$

Подъем f должен быть отображением естественных преобразований между двумя функторами, которые идут от **I** к **C**:

$$Nat(\Delta_c, D) \to Nat(\Delta_{c'}, D)$$

Как мы отображаем естественные преобразования? Каждое естественное преобразование является выбором морфизмов — его компонентов — один морфизм на каждый элемент из **I**. Компонент некоторого α (члена $Nat(\Delta_c, D)$) при a (объект из **I**) есть морфизм:

$$\alpha_a :: \Delta_c\, a \to D\, a$$

или, используя определение постоянного функтора Δ,

$$\alpha_a :: c \to D\, a$$

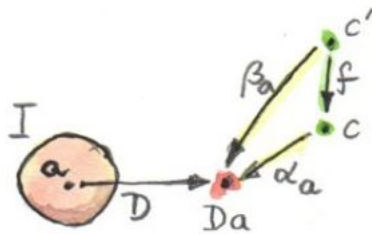
С учетом f и α, мы должны построить β — член $Nat(\Delta_{c'}, D)$. Его составляющей при a должен быть морфизм:

$$\beta_a :: c' \to D\, a$$

Мы можем легко получить этот член (β_a) из первого (α_a), строя предкомпозицию его с f:

$$\beta_a = \alpha_a \mathbin{.} f$$

Достаточно легко показать, что эти компоненты действительно складываются в естественное преобразование.



Учитывая морфизм f, мы, таким образом, построили отображение между двумя естественными преобразованиями, покомпонентно. Это отображение определяет `contramap` для функтора:

$$c \to Nat(\Delta_c, D)$$

Я проделал все это для того, чтобы показать вам, что у нас есть два (контравариантных) функтора от **C** к **Set**. Я не делал никаких предположений — эти функторы вообще существуют.

Кстати, первый из этих функторов играет важную роль в теории категорий, и мы увидим его снова, когда будем говорить о лемме Йонеды. Существует название для контравариантных функторов от произвольной категории **C** к **Set**: они называются *предпучками*. Первый функтор называется *представимым предпучком*. Второй функтор — также предпучок.

Теперь, когда у нас есть два функтора, мы можем говорить о естественных преобразованиях между ними. Так что, без дальнейших церемоний,

вот вывод: функтор D от $\mathbf{I}$ к $\mathbf{C}$ имеет предел $\mathrm{Lim}D$ тогда и только тогда, когда существует естественный изоморфизм между двумя функторами (которые я только что определил):

$$\mathbf{C}(c, \mathrm{Lim}D) \simeq Nat(\Delta_c, D)$$

Позвольте напомнить вам, что такое естественный изоморфизм. Это естественное преобразование, каждый компонент которого является изоморфизмом, то есть обратимым морфизмом.

Я не собираюсь следовать доказательству этого утверждения. Процедура эта довольно проста, если не утомительна. Когда имеют дело с естественными преобразованиями, то обычно сосредотачиваются на компонентах, которые являются морфизмами. В этом случае, так как целью обоих функторов является $\mathbf{Set}$, компоненты естественного изоморфизма будут функциями. Это функции высшего порядка, потому что они идут от hom-множества к множеству естественных преобразований. Опять же, вы можете проанализировать функцию, учитывая то, что она делает со своим аргументом: здесь аргумент будет морфизмом — членом $\mathbf{C}(c, \mathrm{Lim}D)$, а результат будет естественным преобразованием — членом $Nat(\Delta_c, D)$, или то, что мы называем конусом. Это естественное преобразование, в свою очередь, имеет свои собственные компоненты, которые являются морфизмами. Так что здесь везде морфизмы, сверху-до-низу, и если вы сможете проследить за ними, вы сможете доказать вышеуказанное утверждение.

Наиболее важным результатом здесь является то, что условие естественности для этого изоморфизма есть именно условие коммутативности для отображения конусов.

В качестве предварительного упоминания ближайших понятий, позвольте мне отметить, что множество $Nat(\Delta_c, D)$ можно рассматривать как hom-множество в категории функторов; так что наш естественный изоморфизм связывает два hom-множества, что указывает на еще более общий характер взаимосвязи, называемой сопряжением.

12.2 Примеры пределов

Мы уже видели, что категорное произведение является пределом диаграммы, порожденной простой категорией **2**.

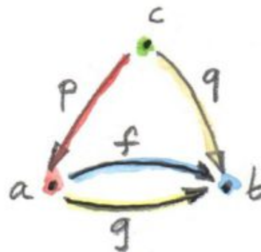
Существует еще более простой пример предела: терминальный объект. Первая приходящая на ум мысль — это одноэлементная категория, ведущая к терминальному объекту, но на самом деле все оказывается еще проще: терминальный объект является пределом, порожденным пустой категорией. Функтор от пустой категории не выбирает объектов, поэтому конус стягивается просто к вершине. Универсальный конус является одинокой вершиной, которая имеет единственный морфизм, приходящий к ней от любого другого вершины. Согласитесь, что это есть определение терминального объекта.

Следующий интересный предел называется *уравнителем*. Это предел, порожденный двухобъектной категорией с двумя параллельными морфизмами между этими объектами (и, как обычно, с тождественными морфизмами). Эта категория выбирает диаграмму в **C**, состоящую из объектов `a` и `b`, и двух морфизмов:

```
f :: a -> b
g :: a -> b
```

Для построения конуса над этой диаграммой, необходимо добавить вершину `c` с двумя проекциями:

```
p :: c -> a
q :: c -> b
```



Имеем два треугольника, которые должны быть коммутативными:

```
q = f . p
q = g . p
```

Это говорит нам о том, что q однозначно определяется одной из этих формул, скажем, q = f . p, и мы можем исключить ее из рассмотрения. Таким образом, мы останемся только с одним условием:

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

Понимание этого состоит в том, если мы ограничим наше внимание к **Set**, образ функции p выбирает подмножество a. При ограничении на это подмножество, функции f и g равны.

Например, возьмем в качестве a двумерную плоскость, параметризуемую координатами x и y, пусть b — действительная прямая, и положим:

```
f (x, y) = 2 * y + x
g (x, y) = y - x
```

Уравнителем этих функций является множество действительных чисел (вершина c) и функция:

```
p t = (t, (-2) * t)
```

Обратите внимание на то, что (p t) определяет прямую линию на двумерной плоскости. Вдоль этой линии обе функции равны.

Конечно, имеются и другие множества c′ и функции p′, которые могут привести к равенству:

```
f . p′ = g . p′
```

но все они однозначно факторизуются через p. Например, мы можем взять одноэлементное множество () в качестве c′ и функцию:

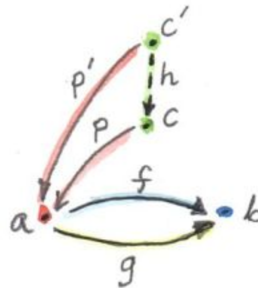
```
p′() = (0, 0)
```

Это хороший конус, потому что `f(0,0) = g(0,0)`. Но он не является универсальным, благодаря единственной факторизации через `h`:

```
p' = p . h
```

с

```
h () = 0
```



Уравнитель, таким образом, может быть использован для решения уравнений вида $fx = gx$. Но он имеет гораздо более широкие возможности, так как определен не алгебраически, а в терминах объектов и морфизмов.

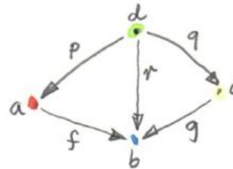
Еще более общая идея решения уравнения воплощается в другом пределе — обратном образе. Здесь опять имеются два морфизма, которые мы хотим уравнять, но на этот раз их домены различны. Будем исходить из трехобъектной категории формы $1 \to 2 \leftarrow 3$. Диаграмма, соответствующая этой категории, содержит три объекта, `a`, `b` и `c`, и два морфизма:

```
f :: a -> b
g :: c -> b
```

Такую диаграмму часто называют ко-пролетом (cospan).

Конус, построенный на вершине этой схемы, содержит саму вершину, `d`, и три морфизма:

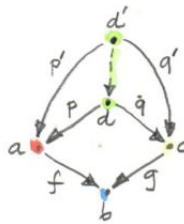
```
p :: d -> a
q :: d -> c
r :: d -> b
```



Условия коммутативности говорят нам, что `r` полностью определяется другими морфизмами, и может быть исключен из рассмотрения. Таким образом, мы останемся только со следующим условием:

```
g . q = f . p
```

Обратный образ есть универсальный конус этой диаграммы.



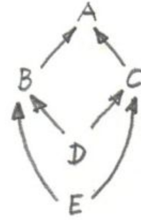
Опять же, если сместить фокус к множествам, можно мыслить объект `d` как состоящий из пар элементов `a` и `c`, для которых `f`, действуя на первом компоненте, равна `g`, действующей на второй компоненте. Если это все еще носит слишком общий характер, рассмотрим частный случай, в котором `g` является постоянной функцией, скажем, `g _ = 1,23` (при условии, что `b` является множеством действительных чисел). Тогда вы, на самом деле, решаете уравнение:

```
f x = 1.23
```

В этом случае выбор `c` не имеет никакого значения (до тех пор, пока это не пустое множество), так что мы можем считать его синглетоном. Множество `a`, например, может быть множеством трехмерных векторов, а `f` — длиной вектора. Тогда обратный образ есть множество пар `(v,())`, где `v` является вектором длины 1,23 (решение уравнения $\sqrt{(x^2 + y^2 + r^2)} = 1,23$), а `()` является пустым (фиктивным) элементом синглетона.

Но обратные образы имеют более общие приложения, в том числе в программировании. Например, рассмотрим классы C++ в качестве категории, в которой морфизмы являются стрелками, соединяющими подклассы с суперклассами. Рассмотрим наследование транзитивного свойства, так что, если C наследуется от B, а B наследуется от A, то будем говорить, что C наследуется от A (в конце концов, можно передать указатель к C, который, как ожидается, указывает на A). Кроме того, будем считать, что C наследуется от C, т.е. имеется тождественная стрелка для каждого класса. Таким образом, создание подклассов совмещено с созданием подтипов. C++ также поддерживает множественное наследование, так что можно построить ромбовидную диаграмму наследования с двумя классами B и C, унаследованными от A, и четвертым классом D с множественным наследованием от B и C. Так что, D получит две копии A, что нежелательно, но при этом можно использовать виртуальное наследование, чтобы иметь только одну копию A для D.

Что бы значило то, что D являлся бы обратным образом в этой схеме? Это означало бы, что любой класс E, который множественно наследуется от B и C, также является подклассом D. Это непосредственно не выразимо в C++, где образование подтипов является номинальным (компилятор C++ не будет выводить такого рода классовых отношений — это потребует «изощренной типизации»). Но мы могли бы выйти за пределы отношения образования подтипов, и вместо того, чтобы спрашивать, есть ли переход от E к D, спросить будет ли он безопасным или нет. Этот переход был бы безопасным, если D был бы комбинацией скелетонов B и C, без каких-либо дополнительных данных и переопределения методов. И, конечно же, не может быть никакого обратного образа, если существует конфликт имен между некоторыми методами B и C.



Существует также возможность более расширенного использования обратного образа при выводе типов. Так, часто возникает необходимость унификации типов двух выражений. Например, предположим, что компилятор хочет определить тип функции:

```
twice f x = f (f x)
```

Компилятор задаст предварительные типы для всех переменных и подвыражений. В частности, он назначит:

```
f       :: t0
x       :: t1
f x     :: t2
f (f x) :: t3
```

откуда будет выведено, что:

```
twice :: t0 -> t1 -> t3
```

Он также сформирует набор ограничений, вытекающих из правил применения функции:

```
t0 = t1 -> t2  --
потому, что f применяется к x
t0 = t2 -> t3  --
так как f применяется к (f x)
```

Эти ограничения должны быть унифицированы путем нахождения множества типов (или переменных типа), которые при подстановке на место неизвестных типов в обоих выражениях, производят один и тот же тип. Вот одна из таких замен:

```
t1 = t2 = t3 = Int
twice :: (Int -> Int) -> Int -> Int
```

но, очевидно, это не самый обобщенный подход. Наиболее общее замещение получается с помощью обратного образа. Я не буду вдаваться в детали, поскольку что они выходят за рамки этой книги, но вы можете убедиться, что результатом будет:

```
twice :: (t -> t) -> t -> t
```

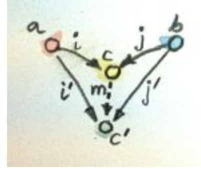
со свободной переменной типа `t`.

12.3 Копределы

Так же, как и все конструкции в теории категорий, пределы имеют противоположную конструкцию в двойственных категориях. Когда вы меняете направление всех стрелок в конусе, вы получаете ко-конус, и среди них один универсальный, который называется *копределом*. Обратите внимание на то, что инверсия влияет также и на факторизующий морфизм, который теперь направлен от универсального ко-конуса к другим ко-конусам.

Ко-конус с факторизующим морфизмом h, соединяющим две вершины

Типичным примером копредела является копроизведение, что соответствует диаграмме, порожденной категорией **2**, которую мы использовали в определении произведения.



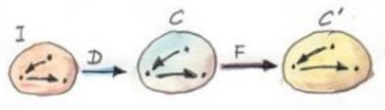
И произведение, и копроизведение, воплощают суть пары объектов, каждый по-своему.

Так же, как терминальный объект был пределом, так и инициальный объект является копределом соответствующей диаграммы, основанной на пустой категории.

Двойственным к обратному образу является понятие амальгамы (кодекартов квадрат). Она основан на диаграмме, называемой *пролетом*, порожденной категорией со схемой $1 \leftarrow 2 \rightarrow 3$.

12.4 Непрерывность

Я ранее говорил, что функторы близки идее непрерывных отображений категорий, в том смысле, что они никогда не разрушают существующие связи (морфизмы). Фактическое определение непрерывного функтора F от категории $\mathbf{C}$ к категории $\mathbf{C}'$ включает в себя требование о том, что функтор сохраняет пределы. Каждая диаграмма D в $\mathbf{C}$ может быть отображена на диаграмму $F \circ D$ в $\mathbf{C}'$ простой композицией двух функторов. Условие непрерывности F утверждает, что, если диаграмма D имеет предел $\text{Lim}D$, то диаграмма $F \circ D$ также имеет предел, и он равен $F\,(\text{Lim}D)$.



Обратите внимание на то, что, так как функторы отображают морфизмы на морфизмы и композиции на композиции, изображение конуса всегда остается конусом. Коммутативный треугольник всегда отображается на коммутативный треугольник (функторы сохраняют композицию). То же самое верно и для факторизующих морфизмов: образ факторизующего морфизм является также факторизующим морфизмом. Таким образом, каждый функтор почти непрерывен. Что может не выполняться, так это условие единственности. Факторизующий морфизм в $\mathbf{C}'$ может быть не единственным. Могут существовать и другие «более лучшие конусы» в $\mathbf{C}'$, которые отсутствовали в $\mathbf{C}$.

$\hom$-функтор есть пример непрерывного функтора. $\hom$-функтор $\mathbf{C}(a, b)$ является контравариантным по первой переменой и ковариантным — по второй. Другими словами, это функтор:

$$\mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$$

Когда его второй аргумент фиксирован, то функтор $\hom$-множества (который становится представляемым предпучком) отображает копределы в $\mathbf{C}$ на пределы в $\mathbf{Set}$, а когда первый аргумент является фиксированным, то пределы отображаются на пределы.

В Haskell, $\hom$-функтор является отображением любых двух типов на функциональный тип, так что это просто параметризованный функциональный тип. Когда мы фиксируем второй параметр, скажем, задавая `String`, мы получаем контравариантный функтор:

```
newtype ToString a = ToString (a -> String)
instance Contravariant ToString where
   contramap f (ToString g) = ToString (g . f)
```

Непрерывность означает, что, когда `ToString` применяется к копределу, например к копроизведению `Either b c`, то будет производиться предел; в этом случае это произведение двух функциональных типов:

```
ToString (Either b c) ~ (b -> String,
                         c -> String)
```

Действительно, любая функция `Either b c` реализуется в качестве конкретного выражения, содержащего поддержку двух ситуаций парой функций.

Аналогичным образом, когда фиксируется первый аргумент hom-множества, мы получаем уже известный функтор. Его непрерывность означает, что, например, любая функция, возвращающая произведение, эквивалентна произведению функций; в частности:

```
r -> (a, b) ~ (r -> a, r -> b)
```

Я знаю, о чем вы думаете: не нужно знать теорию категорий, чтобы понять эти вещи. И вы правы! Тем не менее, я считаю удивительным, что такие результаты могут быть получены из основных принципов, не прибегая к терминологии битов и байтов, процессорных архитектур, компьютерных технологий, или даже лямбда-исчисления.

Если вам интересно, откуда появились названия «предел» и «непрерывность», — они являются обобщением соответствующих понятий из математических исчислений. В соответствующем исчислении пределы и непрерывность определяются в терминах открытых окрестностей, а открытые множества, которые определяют топологию, образуют категорию (частично упорядоченных множеств).

Упражнения

1. Как бы вы охарактеризовали амальгаму в категории классов C++?
2. Покажите, что предел тождественного функтора $\text{Id} :: \mathbf{C} \to \mathbf{C}$ является инициальным объектом.
3. Подмножества заданного множества образуют категорию. Морфизм в этой категории определяется как стрелка, соединяющая два множества, первое из которых является подмножеством второго. Что представляет собой обратный образ таких двух множеств в этой категории? Что здесь является амальгамой? Каковы инициальный и терминальный объекты?

4. Можете ли вы догадаться, что является коуравнителем в категории из предыдущего упражнения?

5. Покажите, что в категории, содержащей терминальный объект, обратный образ вместе с терминальным объектом является произведением.

6. Аналогично, покажите, что амальгама из инициального объекта (если таковой существует) является копроизведением.

Глава 13

Свободные моноиды

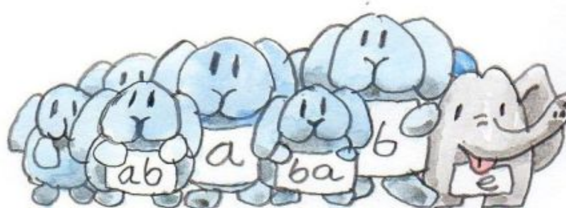
Моноиды являются важным понятием и в теории категорий и в программировании. Категории соответствуют строго типизированным языкам, моноиды — нетипизированным языкам. Это потому, что в моноиде можно компоновать любые две стрелки, так же, как в нетипизированном языке можно соединить любые две функции (конечно, при выполнении такая программа может завершиться аварийно).

Мы уже видели, что моноид может быть описан как категория с одним объектом, где вся логика кодируется в правилах композиции морфизмов. Эта категорная модель полностью эквивалентна более традиционному теоретико-множественному определению моноида, когда мы «перемножаем» два элемента множества, чтобы получить третий элемент. Этот процесс «умножения» может быть представлен двумя шагами, на первом — образуется пара элементов, а на втором эта пара идентифицируется с существующим элементом — их «произведением».

Что происходит, когда мы отказываться от второй части умножения — идентификации пар с существующими элементами? Мы можем, например, начать с произвольного множества, образуя всевозможные пары элементов, и назвать их новыми элементами. Затем мы будем образовывать пары из этих новых элементов со всеми возможными другими элементами этого множества, и так далее. Это цепная реакция — мы будем добавлять новые элементы постоянно. В результате, полученное бесконечное множество будет почти моноидом. Но моноиду необходимы также единичный элемент и закон ассоциативности. Что же, мы можем

добавить специальный единичный элемент и «узаконить» лишь некоторые из пар, которые удовлетворяют законам единицы и ассоциативности.

Давайте посмотрим, как это работает на простом примере. Начнем с множества из двух элементов, $\{a, b\}$. Будем называть a и b образующими свободного моноида. Сначала добавим в это множество специальный элемент e в качестве единицы. Далее добавим все пары элементов и назовем их *произведениями*. Произведением a и b будет пара (a, b). Произведением b и a будет пара (b, a), произведением a с a будет (a, a), b с b — (b, b). Также могут образовываться пары с e, такие как (a, e), (e, b) и т.д., но мы будем их отождествлять с a, b и т.д. Так что на этом этапе мы добавим только (a, a), (a, b), (b, a) и (b, b), и в конечном итоге получим множество $\{e, a, b, (a, a), (a, b), (b, a), (b, b)\}$.



На следующем этапе мы будем добавлять такие элементы, как: $(a, (a, b))$, $((a, b), a)$, и т.д. При этом мы должны быть убеждены в том, что выполняется закон ассоциативности, так что мы будем отождествлять $(a, (b, a))$ с $((a, b), a)$, и т.д. Другими словами, нет необходимости в расстановке внутренних скобок.

Можно догадаться, что конечным результатом этого процесса будет создание множества всевозможных списков, содержащих a и b. Таким образом, если мы представим e в качестве пустого списка, мы можем понять, что наше «умножение» есть ничто иное, как конкатенация (списков).

Такого рода конструкция, в которой можно генерировать всевозможные комбинации исходных элементов, а также выполнять минимальное количество идентификаций, достаточное для выполнения требуемых законов, называется *свободной конструкцией*. То, что мы только что сделали, это построение свободного моноида из множества образующих $\{a, b\}$.

13.1 Свободный моноид в Haskell

В Haskell, двухэлементное множество эквивалентно типу `Bool`, а свободный моноид, порожденный этим множеством, эквивалентен типу `[Bool]` — списку `Bool` (ясознательно не затрагиваю проблем, связанных с бесконечными списками).

Моноид на Haskell определяется ти́повым классом:

```
class Monoid m where
   mempty  :: m
   mappend :: m -> m -> m
```

Это как раз говорит о том, что каждый `Monoid` должен иметь нейтральный элемент, который назван `mempty`, и бинарную функцию (умножение), названную `mappend`. Законы единицы и ассоциативности не могут быть выражены на Haskell и должны проверяться программистом каждый раз, когда создается экземпляр моноида.

Тот факт, что список любого типа образует моноид, описывается таким определением его экземпляра:

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

Это означает, что пустой список `[]` является единицей, а а конкатенация списков `(++)` — бинарной операцией.

Как мы уже видели, список типа `a` соответствует свободному моноид, с множеством из одного элемента `a`, выступающего в качестве образующего. Множество натуральных чисел с умножением не является свободным моноидом, потому что мы идентифицируем много произведений. Сравните, например:

```
2 * 3 = 6
[2] ++ [3] = [2, 3] --
это не то же самое, что [6]
```

Это было просто, но вопрос в том, можем ли мы создать такую свободную конструкцию в теории категорий, где нам не позволено заглянуть внутрь объектов? Мы будем использовать нашу рабочую лошадку: универсальную конструкцию.

Второй интересный вопрос, может ли моноид быть получен из некоторого свободного моноида, идентификацией более чем минимального количества элементов, требуемых в соответствии с законами? Я покажу, что это непосредственно следует из универсальной конструкции.

13.2 Универсальная конструкция свободного моноида

Если вспомнить наши предыдущие эксперименты с универсальными конструкциями, можно заметить, что требуется немного для их построения — надо выбрать объект, который наилучшим образом соответствует заданному шаблону. Так что, если мы хотим использовать универсальную конструкцию, чтобы «построить» свободный моноид, мы должны рассмотреть совокупность моноидов, из которых выбрать один. Нам нужна целая категория моноидов для такого выбора. Но, образуют ли моноиды категорию?

Давайте сначала посмотрим на моноиды как на множества с дополнительной структурой, определяемой единицей и умножением. Выберем в качестве морфизмов те функции, которые сохраняют моноидальную структуру. Такие функции, сохраняющие структуру, называются *гомоморфизмами*. Моноидный гомоморфизм должен сопоставить произведению двух элементов произведение отображений этих элементов:

```
h (a * b) = h a * h b
```

и он должен отображать единицу в единицу.

Для примера, рассмотрим гомоморфизм от списка целых чисел к целым числам. Если мы отображаем `[2]` в `2`, а `[3]` в `3`, то мы должны иметь отображение `[2,3]` в `6`, потому что конкатенация

```
[2] ++ [3] = [2, 3]
```

приводит к умножению

```
2 * 3 = 6
```

Теперь давайте забудем о внутренней структуре отдельных моноидов, и будем рассматривать их как объекты с соответствующими морфизмами. Мы получим категорию **Mon** моноидов.

Хорошо, прежде чем мы, может быть, забудем о внутренней структуре, отметим одно важное свойство. Каждый объект **Mon** может быть тривиальным образом отображен на множество. Это будет просто множество элементов этой категории. Этот множество называется основным множеством. На самом деле, мы не только можем отображать объекты **Mon** на множества, но и морфизмы **Mon** (гомоморфизмы) — на функции. Это кажется тривиальным, но вскоре нам понадобится. Эти отображения объектов и морфизмов от **Mon** к **Set** фактически являются функтором. Так как этот функтор «забывает» моноидальную структуру (находясь внутри простого множества, мы не различаем единичный элемент и не заботимся об умножения), то он называется *забывающим функтором*. Забывающие функторы неизменно проявляют себя в теории категорий.

Теперь мы имеем две различные точки зрения на **Mon**. Мы можем относиться к ней так же, как и к любой другой категории с объектами и морфизмами. С этой точки зрения мы не рассматриваем внутреннюю структуру моноидов. Все, что мы можем сказать о конкретном объекте в **Mon**, это то, что он соединяется сам собой и с другими объектами посредством морфизмов. Таблица «умножения» морфизмов, правила композиции, выводится из другой точки зрения: рассмотрения моноидов как множеств. Переходя к теории категорий мы не теряем эту точку зрения полностью, мы все еще можем получить доступ к структуре моноида через забывающий функтор.

Для применения универсальной конструкции нам необходимо определить специальное свойство, которое позволил бы использовать категорию моноидов для выбора наилучшего кандидата на роль свободного моноида. Но свободный моноид определяется своими образующими. Выбор различных образующих порождает различные свободные моноиды (список элементов типа `Bool` не то же самое, что список элементов типа `Int`). Наша конструкция должна начинаться с множества образующих. Таким образом, мы вернулись к множествам!

Именно здесь забывающий функтор вступает в игру. Мы можем использовать его для рентгена наши моноидов и определить образующие в их рентгеновских снимках. Вот как это работает.

Будем исходить из множества образующих `x`, являющимся элементом **Set**. Шаблон, который мы будем использовать для установления соответствия состоит из моноида `m` — объекта **Mon** — и функции `p` в **Set**:

```
p :: x -> U m
```

где `U` — забывающий функтор от **Mon** к **Set**. Это странный гетерогенный шаблон — половина в **Mon**, а половина в **Set**.

Идея заключается в том, что функция `p` будет определять множество образующих внутри рентгеновского изображения `m`. Не имеет значения, что функции могут вести себя непредсказуемым образом при определении точек внутри множеств (они могут сворачивать их). Все будет отсортировано посредством универсальной конструкции, что поможет выбрать лучшего представителя этой модели.

Мы также должны определить ранжирование среди кандидатов. Предположим, у нас есть еще один кандидат: моноид `n` и функция, которая идентифицирует образующие в его рентгеновском изображении:

```
q :: x -> U n
```

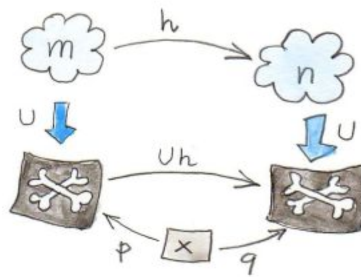
Мы будем говорить, что `m` лучше, чем `n`, если существует морфизм моноидов (это структуро-сохраняющий гомоморфизм):

```
h :: m -> n
```

образ которого при U (помните, что U функтор, поэтому он отображает морфизмы к функциям) пропускается через p:

```
q = U h . p
```

Если думать о p как выбирающем образующие в m, а о q — как выбирающем «те же» образующие в n, то можно думать о h, как об отображении этих образующих между двумя моноидами. Напомним, что h, по определению, сохраняет моноидальную структуру. Это означает, что произведение двух образующих в одном моноиде будет отображаться на произведение соответствующих образующих во втором моноиде, и так далее.



Это ранжирование может быть использовано для нахождения лучшего кандидата — свободного моноида:

Определение. Будем говорить, что m (вместе с функцией p) является *свободным моноидом* с образующими x тогда и только тогда, когда существует единственный морфизм h от m к любому другому моноиду n (вместе с функцией q), что удовлетворяет приведенному ранее свойсту факторизации.

Кстати, это ответ на наш второй вопрос. Функция U h является такой, что имеет возможность для сворачивания нескольких элементов U m в один элемент U n. Это сворачивание соответствует отождествлению некоторых элементов свободного моноида. Поэтому любой моноид с образующими x может быть получен из свободного моноида, основанного на x, отождествлением некоторых элементов. Свободным моноидом

является тот, который получен с помощью минимального числа отождествлений.

Мы вернемся к свободным моноидам, когда будем говорить о сопряжениях.

Упражнения

1. Вы можете подумать (как и я, изначально), что требование о том, что гомоморфизм моноидов сохраняет единицу, является излишним. В конце концов, мы знаем, что для всех `a`

   ```
   h a * h e = h (a * e) = h a
   ```

   Так что он действует как правый блок (и, по аналогии, как левый блок). Проблема заключается в том, что ha, для всех a может покрывать только подмоноид целевого моноида. «Истинная» единица может находиться за пределами образа h. Покажите, что изоморфизм между моноидами, сохраняющий умножение, должен автоматически сохранять единицу.

2. Рассмотрим моноидный гомоморфизм от списков целых чисел с операцией конкатенации к целым числам с операцией умножения. Что является образом пустого списка `[]`? Предположим, что все одноэлементные списки преобразуются в целые числа, которые они содержат, то есть `[3]` отображается в `3` и т.д. Что будет образом списка `[1,2,3,4]`? Сколько различных списков могут быть отображены к целому числу `12`? Существуют ли другой гомоморфизм между двумя моноидами?

3. Что представляет собой свободный моноид, порожденный одноэлементным множеством? Можете ли вы показать, что такое «порождение» изоморфно?

Глава 14

Представимые функторы

Пора бы немного поговорить о множествах. У математиков отношение к теории множеств варьируется от любви до ненависти. Это язык сборки математики — по крайней мере, так используется теория множеств. Теория категорий пытается, в некотором смысле, отойти от теории множеств. Например, известно, что множество всех множеств не существует, а категория всех множеств **Set** не только существует, но и является, пожалуй, одним из основных источников развития теории категорий. С другой стороны, мы предполагаем, что морфизмы между любыми двумя объектами в этой категории образуют множество. Его даже назвали hom-множеством. Справедливости ради, надо сказать, что имеется раздел теории категорий, в котором морфизмы не образуют множества. Вместо этого они являются объектами в другой категории. Эти категории, которые используют hom-объекты, а не hom-множества, называются *обогащенными* категориями. В дальнейшем, однако, мы будем придерживаться категорий с хорошими старомодными hom-множествами.

Множество похоже на безликий сгусток, находящийся за пределами категорных объектов. Множество содержит элементы, но вы не можете рассказать многого об этих элементах. В конечном множестве можно пересчитать его элементы. С помощью кардинальных чисел можно оценить количество элементов и бесконечного множества. Множество натуральных чисел, например, меньше множества действительных чисел, хотя оба бесконечны. Но может показаться удивительным, что множество рациональных чисел имеет тот же размер, что и множество натуральных чисел.

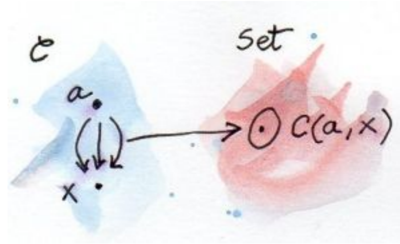
Кроме этого, вся информация о множествах может быть закодирована в функциях между ними, особенно обратимых, называемых изоморфизмами. Для всех намерений и целей изоморфные множествах идентичны. Перед тем, как я вызову гнев фундаментальных математиков, позвольте отметить, что различие между равенством и изоморфизмом имеет принципиальное значение. На самом деле это одна из главных проблем новейшей ветви математики, Гомотопической Теории Типов (HoTT). Я упоминаю HoTT, потому что это чисто математическая теория, которая развивается, отталкиваясь от изучения вычислений, и один из ее главных сторонников, Владимир Воеводский, проявил большое прозрение при изучении доказывателя теорем Coq. Взаимодействие между математикой и программированием идет в обоих направлениях.

Важный урок, касающийся множеств, заключается в том, что надо сравнивать множества, а не элементы. Например, мы можем сказать, что данное множество естественных преобразований изоморфно некоторому множеству морфизмов, так как множество это просто набор. Изоморфизм в этом случае просто означает, что для каждого естественного преобразования из одного множества существует единственный морфизм в другом множестве, и наоборот. В этом смысле они могут быть в паре друг с другом. Вы не можете сравнивать яблоки с апельсинами, если они являются объектами из разных категорий, но вы можете сравнить множества яблок с множествами апельсинов. Часто трансформация категорной задачи в задачу теоретико-множественную дает нам необходимое понимание или даже возможность доказывать полезные теоремы.

14.1 hom-функтор

Каждая категория оснащена каноническим семейством отображений к $\mathbf{Set}$. Эти отображения являются фактически функторами, поскольку они сохраняют структуру категории. Давайте построим одно такое отображение.

Зафиксируем объект a в $\mathbf{C}$ и выберем другой объект x, также в $\mathbf{C}$. hom-множество $\mathbf{C}(a, x)$ представляет собой множество — объект в $\mathbf{Set}$. Когда мы меняем x, сохраняя a фиксированным, $\mathbf{C}(a, x)$ также будет меняться в $\mathbf{Set}$. Таким образом, мы имеем отображение от x к $\mathbf{Set}$.



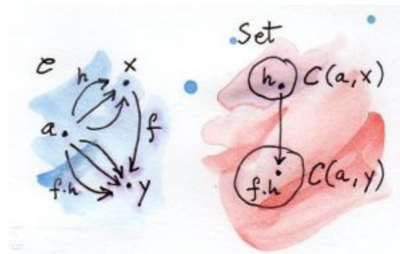
Если мы хотим подчеркнуть, что рассматриваем hom-множество как отображение по второму аргументу, будем использовать обозначение $\mathbf{C}(a, -)$ с подчеркиванием, выступающим в качестве заполнителя для аргумента.

Такое отображение объектов легко переносится на отображение морфизмов. Рассмотрим морфизм f в $\mathbf{C}$ между двумя произвольными объектами x и y. Объект x отображается на множество $\mathbf{C}(a, x)$, а объект y отображается на множество $\mathbf{C}(a, y)$, способом, который мы только что определили. Если это отображение будет функтором, f должен быть отображен в функцию между двумя множествами: $\mathbf{C}(a, x) \longrightarrow \mathbf{C}(a, y)$.

Определим эту функцию поточечно, то есть для каждого аргумента в отдельности. Для аргумента мы должны выбрать произвольный элемент $\mathbf{C}(a, x)$ — обозначим его h. Морфизмы компонуемы, если они согласованы, т.е. цель h должна соответствовать источнику f, поэтому их композиция:

$$f \circ h :: a \to y$$

есть морфизм от a к y. Поэтому он является членом $\mathbf{C}(a, y)$.



Мы только что вышли на функцию от $\mathbf{C}(a, x)$ к $\mathbf{C}(a, y)$, которая может служить образом f. Если нет опасности путаницы, мы будем записывать

эту поднятую функцию, как $\mathbf{C}(a, f)$, а ее действие на морфизм h, как:

$$\mathbf{C}(a, f)\, h = f \circ h$$

Так как эта конструкция работает в любой категории, она также должна работать и в категории типов Haskell. В Haskell hom-функтор более известен как функтор `Reader`:

```
type Reader a x = a -> x
instance Functor (Reader a) where
   fmap f h = f . h
```

Теперь давайте рассмотрим, что произойдет, если, вместо того, чтобы фиксировать источник hom-множества, мы зафиксируем цель. Другими словами, мы задаемся вопросом, является ли функтором отображение $\mathbf{C}(-, a)$?

Это так, но вместо того, чтобы быть ковариантным, он контравариантен. Это потому, что это тот же самый вид согласования морфизмов встык после композиции по f, а не до композиции, как это имело место с $\mathbf{C}(a, -)$.

Мы уже встречали этот контравариантный функтор на Haskell. Мы назвали его `Op`:

```
type Op a x = x -> a
instance Contravariant (Op a) where
   contramap f h = h . f
```

И, наконец, если мы позволим обоим объектам меняться, мы получаем профунктор $\mathbf{C}(-, -)$, который контравариантен по первому аргументу и ковариантен по второму. Мы сталкивались с этим профунктором ранее, когда говорили о функториальности:

```
instance Profunctor (->) where
   dimap ab cd bc = cd . bc . ab
   lmap           = flip (.)
   rmap           = (.)
```

Важный вывод заключается в том, что это наблюдение справедливо в любой категории: отображение объектов в hom-множества функториально. Так как контравариантность эквивалента отображению от противоположной категории, мы можем кратко констатировать этот факт в форме:

$$\mathbf{C}(-,=) :: \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$$

14.2 Представимые функторы

Мы видели, что при любом выборе объекта a в $\mathbf{C}$, мы получаем функтор от $\mathbf{C}$ к $\mathbf{Set}$. Такого рода структуры, сохраняющие отображения к $\mathbf{Set}$, часто называют представлениями. Мы представляем объекты и морфизмы из $\mathbf{C}$ как множества и функции в $\mathbf{Set}$.

Сам функтор $\mathbf{C}(a,-)$ иногда называют представимым. В более общем смысле, любой функтор F, который естественным образом изоморфен hom-функтору, при некотором выборе a, называется *представимым*. Такой функтор должен быть обязательно многозначным, так как об этом говорит запись $\mathbf{C}(a,-)$.

Я говорил, что мы часто думаем об изоморфных множествах как об идентичных. В более общем плане, мы думаем об изоморфных объектах в категории как об идентичных. Это потому, что объекты не имеют структуры, а только отношения к другим объектам (и к самим себе) посредством морфизмов.

Например, мы уже говорили о категории моноидов $\mathbf{Mon}$, которая изначально была смоделирована из множеств. Но мы были осторожны, выбрая в качестве морфизмов только те функции, которые сохраняли моноидальную структуру этих множеств. Так что, если два объекта в $\mathbf{Mon}$ изоморфны, а это означает, что существует обратимый морфизм между ними, то они имеют точно такую же структуру. Если мы взглянем на множества и функции, на которых они были основаны, мы увидим, что единичный элемент одного моноида сопоставляется с единичным элементом другого, и что произведение двух элементов отображается на произведению их отображений.

Те же рассуждения можно применить к функторам. Функторы между двумя категориями образуют категорию, в которой естественные преоб-

разования играют роль морфизмов. Таким образом, два функтора изоморфны, и могут рассматриваться как идентичные, если существует обратимое естественное преобразование между ними.

Давайте проанализируем определение представимых функторов с этой точки зрения. Для представимости F мы требуем, чтобы: существовал объект a в $\mathbf{C}$; одно естественное преобразование α от $\mathbf{C}(a, -)$ к F; еще одно естественное преобразование β в обратном направлении; и что их композиция должна быть тождественным естественным преобразованием.

Давайте рассмотрим компонент α на каком-нибудь объекте x. Это функция в $\mathbf{Set}$:

$$\alpha_x :: \mathbf{C}(a, x) \to F\, x$$

Условие естественности для этого преобразования говорит нам, что для любого морфизма f от x к y, следующая диаграмма коммутативна:

$$F\, f \circ \alpha_x = \alpha_y \circ \mathbf{C}(a, f)$$

В Haskell мы заменяем естественные преобразования полиморфными функциями:

```
alpha :: forall x. (a -> x) -> F x
```

с необязательным квантором `forall`. Условие естественности:

```
fmap f . alpha = alpha . fmap f
```

Условие естественности удовлетворяется автоматически в связи с параметричностью (это одна из «бесплатных» теорем, упомянутая мною ранее), с пониманием того, что `fmap` слева определяется функтором `F`, а справа — считывающим функтором. Так как `fmap` для считывателя это просто функция, действующая до композиции, можно быть еще более категоричным. Элемент из $\mathbf{C}(a, x)$, действуя на `h`, упрощает условие естественности до:

```
fmap f (alpha h) = alpha (f . h)
```

Другое преобразование, `beta`, действует в обратном направлении:

```
beta :: forall x. F x -> (a -> x)
```

Оно должно удовлетворять условиям естественности и быть обратным к `alpha`:

```
alpha . beta = id = beta . alpha
```

Позже мы увидим, что естественное преобразование от $\mathbf{C}(a, -)$ к любому **Set**-значному функтору всегда существует, пока Fa не пусто (лемма Йонеды), но не является обязательно обратимым.

Позвольте привести один пример на Haskell со списочным функтором и `Int` в качестве `a`. Вот работоспособное естественное преобразование:

```
alpha  :: forall x. (Int -> x) -> [x]
alpha h = map h [12]
```

Я произвольно выбрал число `12` и создал из него одноэлементный список. Затем я применяю `fmap h` к этому списку и получаю тип списка, возвращаемого `h` (на самом деле таких преобразований столько, сколько существует целых чисел).

Условие естественности эквивалентно компонуемости `map` (списочная версия `fmap`):

```
map f (map h [12]) = map (f . h) [12]
```

Но если нам нужно найти обратное преобразование, мы должны отталкиваться от списка произвольного типа `x` к функции, возвращающей `x`:

```
beta :: forall x. [x] -> (Int -> x)
```

Вы можете мыслить получение x из списка, например, с использованием head, но это не будет работать для пустого списка. Обратите внимание на то, что вообще нет какого-нибудь выбора для типа a (вместо Int), который будет работать в этом случае. Так что списочный функтор не является представимым.

Помните, мы говорили о (эндо-) функторах Haskell, похожими на контейнеры? В том же духе мы можем думать о представимых функторах как о контейнерах для хранения результатов вызовов функций (члены hom-множеств в Haskell — это просто функции). Представляющий объект, тип a в $\mathbf{C}(a, -)$, мыслится как тип ключа, с помощью которого мы можем получить доступ к табличным значениям функции. Преобразование α назовем tabulate, а обратное ему, β, — index. Вот (слегка упрощенное) определение класса Representable:

```
class Representable f where
   type Rep f :: *
   tabulate   :: (Rep f -> x) -> f x
   index      :: f x -> Rep f -> x
```

Обратите внимание на то, что представляющий тип a, который здесь обозначен Rep f, является частью определения Representable. Звездочка просто означает, что Rep f является типом (в отличие от конструктора типа, или других, более экзотических видов).

Бесконечные списки или потоки, которые не пусты, являются представимыми.

```
data Stream x = Cons x (Stream x)
```

Вы можете рассматривать их как содержащие запомненные значения функции, принимающей Integer в качестве аргумента (строго говоря, я должен был использовать для этого неотрицательные натуральные числа, но я не хотел усложнять код).

Для функции tabulate вы создаете бесконечный поток значений. Конечно, это возможно только потому, что Haskell ленив (значения вычисляются по требованию). Вы получаете доступ к запомненным значениям с помощью index:

```
instance Representable Stream where
   type Rep Stream      = Integer
   tabulate f           = Cons (f 0)
                          (tabulate (f . (+1)))
   index (Cons b bs) n = if n == 0
                          then b
                          else index bs (n - 1)
```

Очень примечательно, что вы можете реализовать единую схему мемоизации, чтобы покрыть целое семейство функций с произвольными типами возврата.

Представимость для контравариантных функторов определяется аналогично, за исключением того, что мы фиксируем второй аргумент $\mathbf{C}(-, a)$. Или, что то же самое, мы можем рассматривать функторы от $\mathbf{C}^{op}$ к $\mathbf{Set}$, так как $\mathbf{C}^{op}(a, -)$ — это $\mathbf{C}(-, a)$.

Существует интересный разворот к представимости. Помните, что в декартово замкнутых категориях $\hom$-множества внутренне можно рассматривать как экспонциальные объекты? $\hom$-множество $\mathbf{C}(a, x)$ эквивалентно x^a, и для представимого функтора F можно записать: $-^a = F$.

Прологарифмируем последнее выражение, просто ради удовольствия: $a = \log F$

Конечно, это чисто формальное преобразование, но если вам известны некоторые из свойств логарифмов, это оказывается весьма полезным. В частности, выясняется, что функторы, которые основаны на тип-произведении могут быть представлены тип-суммами, и что функторы тип-сумм, вообще говоря, не представимы (например: списочный функтор).

Наконец, обратите внимание, что представимый функтор дает нам две различные реализации одного и того же — одной функции, одной структуры данных. Они имеют одинаковое содержание — одни и те же значения извлекаются с использованием одних и тех же ключей. Про это ощущение «одинаковости» я уже говорил. Два естественно изоморфных функтора идентичны настолько, насколько это позволяет их содержание. С другой стороны, эти два представления часто реализуются по-разному и могут иметь различные рабочие характеристики. Мемоизации используется в качестве повышения производительности и может привести к существенному сокращению времени работы. Возможность генерировать

различные представления одного и того же базового вычисления является очень ценным на практике. Так что, как ни странно, несмотря на то, что это не касается производительности в общем, теория категорий предоставляет широкие возможности для изучения альтернативных реализаций, которые имеют практическую ценность.

Упражнения

1. Покажите, что hom-функторы отображают идентичные морфизмы из $\mathbf{C}$ в соответствующие идентичные функции из $\mathbf{Set}$.

2. Покажите, что `Maybe` не представим.

3. Является ли функтор `Reader` представимым?

4. Используя представление `Stream`, мемоизуйте функцию, которая вычисляет квадрат своего аргумента.

5. Покажите, что `tabulate` и `index` для `Stream`, на самом деле, обратны друг другу.
   (Подсказка: используйте индукцию)

6. Функтор:

   ```
   Pair a = Pair a a
   ```

   является представимым. Можете ли вы угадать тип, представляющий его? Используйте `tabulate` и `index`.

Глава 15

Лемма Йонеды

Большинство конструкций в теории категорий являются обобщениями результатов из других более конкретных областей математики. Такие понятия, как произведения, копроизведения, моноиды, экспоненты и т.д. были известны задолго до появления теории категорий. Возможно, они известны под другими наименованиями в разных областях математики. Так, декартово произведение в теории множеств и конъюнкция в логике являются конкретными примерами абстрактной идеи категорного произведения.

Лемма Йонеды выделяется в этом отношении как радикальное утверждение о категориях вообще, не имеющее прецедентов в других областях математики. Некоторые говорят, что ее ближайший аналог — теорема Кэли в теории групп (каждая группа изоморфна группе подстановок некоторого множества).

Область действия леммы Йонеды — это произвольная категория $\mathbf{C}$ вместе с функтором F от $\mathbf{C}$ к $\mathbf{Set}$. Мы видели в предыдущем разделе, что некоторые функторы со значениями в $\mathbf{Set}$ представимы, то есть изоморфны hom-функторам. Лемма Йонеды говорит, что все функторы, ведущие к $\mathbf{Set}$, могут быть получены из hom-функторов через естественные преобразования и явно перечисляет все такие преобразования.

Когда я говорил о естественных преобразованиях, я упомянул, что условие естественности может быть довольно ограничительным. Когда вы определяете компонент естественного преобразования в одном объекте,

естественность может быть достаточно сильной, чтобы «транспортировать» этот компонент к другому объекту, который связан с ним посредством морфизма. Чем больше стрелок между объектами в исходной и целевой категориях, тем больше ограничений для транспортировки компонентов естественных преобразований. А **Set** — категория, весьма насыщенная стрелками.

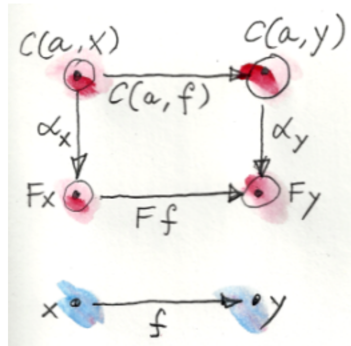
Лемма Йонеды сообщает, что естественное преобразование между hom-функтором и любым другим функтором F полностью определяется заданием значения его единственной компоненты только в одной точке! Остальная часть естественного преобразования просто следует из условий естественности.

Итак, рассмотрим условие естественности между двумя функторами, участвующими в лемме Йонеды. Первым функтором является hom-функтор. Он отображает любой объект x из **C** к множеству морфизмов $\mathbf{C}(a, x)$ — для фиксированного объекта a из **C**. Мы также видели, что он отображает любой морфизм f от $x \to y$ к $\mathbf{C}(a, f)$.

Второй функтор — произвольный функтор F с множеством значений в **Set**.

Обозначим через α естественное преобразование между этими двумя функторами. Поскольку мы работаем в **Set**, компоненты естественного преобразования, скажем, α_x или α_y, являются просто регулярными функциями между множествами:

$$\alpha_x :: \mathbf{C}(a, x) \to F\,x$$
$$\alpha_y :: \mathbf{C}(a, y) \to F\,y$$



А поскольку это просто функции, мы можем рассмотреть ее значения в определенных точках. Но что такое точка в множестве $\mathbf{C}(a, x)$? Вот ключевое замечание: каждая точка множества $\mathbf{C}(a, x)$ является также и морфизмом h от a к x.

Итак, квадрат естественности для α:

$$\alpha_y \circ \mathbf{C}(a, f) = F\,f \circ \alpha_x$$

становится поточечным, когда действует на h:

$$\alpha_y\,(\mathbf{C}(a, f)\,h) = (F\,f)\,(\alpha_x\,h)$$

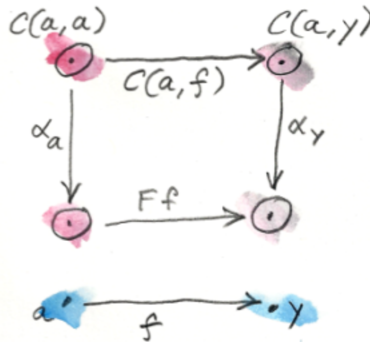
Вы можете вспомнить из предыдущего раздела, что действие hom-функтора $\mathbf{C}(a, -)$ на морфизме f было определено как предварительная композиция:

$$\mathbf{C}(a, f)\,h = f \circ h$$

что приводит к:

$$\alpha_y\,(f \circ h) = (F\,f)\,(\alpha_x\,h)$$

Насколько сильно это условие, можно увидеть, полагая $x = a$.



В этом случае h становится морфизмом от a к a. Мы знаем, что существует, по крайней мере, один такой морфизм, $h = \text{id}_a$. Давайте задействуем его:

$$\alpha_y\,f = (F\,f)\,(\alpha_a\,\text{id}_a)$$

Обратите внимание, что произошло: левая часть — это действие α_y на произвольный элемент f из $\mathbf{C}(a, y)$. И это полностью определяется единственным значением α_a при id_a. Мы можем выбрать любое такое значение, и оно будет генерировать естественное преобразование. Поскольку

значения α_a находятся в множестве $F\,a$, любая точка в $F\,a$ будет определять некоторое α.

И наоборот, при любом естественном преобразовании α от $\mathbf{C}(a,-)$ к F вы можете вычислить его при id_a, чтобы получить точку в $F\,a$.

Мы только что доказали лемму Йонеды:

Лемма. *Между естественными преобразованиями от* $\mathbf{C}(a,-)$ *к* F *и элементами из* $F\,a$ *существует взаимно однозначное соответствие.*

Другими словами,

$$\mathbf{Nat}(\mathbf{C}(a,-),F) \cong F\,a$$

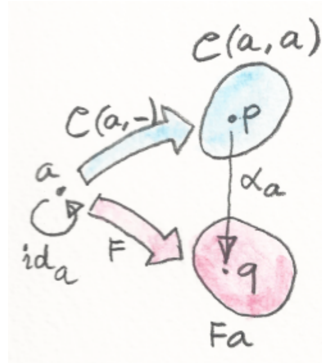
Или, если мы используем обозначение $[\mathbf{C},\mathbf{Set}]$ для категории функторов между $\mathbf{C}$ и $\mathbf{Set}$, то множество естественных преобразований является просто $\hom$-множеством в этой категории, и мы можем написать:

$$[\mathbf{C},\mathbf{Set}](\mathbf{C}(a,-),F) \cong F\,a$$

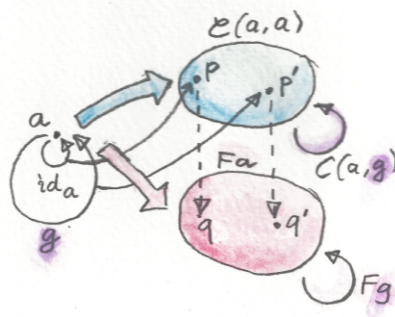
Позже я покажу, что это соответствие на самом деле является естественным изоморфизмом.

Теперь давайте попробуем интуитивно понять этот результат. Самое удивительное, что это естественное преобразование в целом кристаллизуется из одного места зародышеобразования: значения, которое мы присваиваем ему при id_a. Оно распространяется от этой точки, следуя условию естественности. Оно заполняет образ $\mathbf{C}$ в $\mathbf{Set}$. Итак, давайте сначала разберемся, что за образ $\mathbf{C}$ находится под $\mathbf{C}(a,-)$.

Начнем с самого образа. Под $\hom$-функтором $\mathbf{C}(a,-)$, a отображается на множество $\mathbf{C}(a,a)$. С другой стороны, под функтором F оно отображается на множество $F\,a$. Компонента естественного преобразования α_a является некоторой функцией от $\mathbf{C}(a,a)$ к $F\,a$. Давайте сосредоточимся только на одной точке множества $\mathbf{C}(a,a)$, точке, соответствующей морфизму id_a. Чтобы подчеркнуть тот факт, что это всего лишь точка в множестве, обозначим ее p. Компонента α_a должна отображать p к некоторой точке q из $F\,a$. Я покажу вам, что любой выбор q приводит к единственному естественному преобразованию.



Первое утверждение состоит в том, что выбор одной точки q однозначно определяет остальную часть функции α_a. Действительно, возьмем любую другую точку p' в $\mathbf{C}(a,a)$, соответствующую некоторому морфизму g от a к a. И вот в чем заключается магия леммы Йонеды: g можно рассматривать как точку p' в множестве $\mathbf{C}(a,a)$. В то же время, она выбирает две функции между множествами. Действительно, при гомоморфизме морфизм g отображается к функции из $\mathbf{C}(a,g)$, а под F он отображается к $F\,g$.



Теперь рассмотрим действие $\mathbf{C}(a,g)$ в нашей исходной точке p, которой, как вы помните, соответствует id_a. Оно определяется как предварительная композиция, $g \circ \mathrm{id}_a$, которая равна g, что соответствует точке p'. Таким образом, морфизм g отображается в функцию, которая при действии на p производит p', которая представляет собой g. Мы оказались в исходной точке!

Разберем теперь действие $F\,g$ на q. Это некоторая q' — точка в $F\,a$. Чтобы завершить построение квадрата естественности, p' должна быть

отображена в q' под α_a. Мы выбрали произвольную p' (произвольную g) и вывели ее отображение под α_a. Таким образом, функция α_a полностью определена.

Второе утверждение заключается в том, что α_a однозначно определяется для любого объекта x в $\mathbf{C}$, который связан с a. Здесь рассуждения аналогичны, за исключением того, что теперь у нас есть еще два множества: $\mathbf{C}(a, x)$ и $F\,x$, а морфизм g от a к x отображается под hom-функтором к:

$$\mathbf{C}(a, g) :: \mathbf{C}(a, a) \to \mathbf{C}(a, x)$$

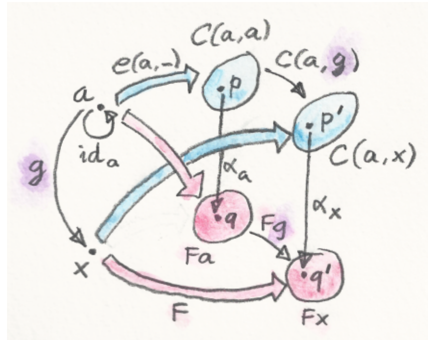
а под F к:

$$Fg :: Fa \to Fx$$

Опять же, $\mathbf{C}(a, g)$, действующее на p, задается предварительной композицией $g \circ \mathrm{id}_a$, которая соответствует точке p' из $\mathbf{C}(a, x)$. Естественность определяет значение α_x, действующее на p' для получения:

$$q' = (F\,g)\,q$$

Поскольку p' произвольно, вся функция α_x определяется таким образом:



Что, если в $\mathbf{C}$ есть объекты, не имеющие связи с a? Все они сопоставляются под $\mathbf{C}(a, -)$ с одним множеством — пустым множеством. Напомним, что пустое множество является инициальным объектом в категории множеств. Это означает, что от этого множества существует единственная функция к любому другому множеству. Мы назвали эту функцию `absurd`. Итак, здесь опять же нет выбора для компонента естественного преобразования: это может быть только `absurd`.

Один из способов понимания леммы Йонеды состоит в осознании того, что естественные преобразования между **Set**-значно определенными функторами являются просто семействами функций, а функции в общем случае являются забывающими. Функция может сворачивать информацию и охватывать только некоторые части своей кообласти. Единственными функциями, которые не являются забывающими, являются те, которые являются обратимыми — изоморфизмы. Отсюда следует, что наилучшие, сохраняющие структуру, **Set**-значные функторы являются представимыми. Они являются либо $\hom$-функторами, либо функторами, которые естественно изоморфны $\hom$-функторам. Любой другой функтор F получается из $\hom$-функтора посредством преобразования с забыванием. Такое преобразование может не только потерять информацию, но и охватить лишь малую часть образа функтора F в **Set**.

15.1 Йонеда в Haskell

Мы уже встречали $\hom$-функтор в Haskell в виде считывающего функтора:

```
type Reader a x = a -> x
```

Этот считыватель отображает морфизмы (здесь, функции) с помощью предкомпозиции:

```
instance Functor (Reader a) where
   fmap f h = f . h
```

Лемма Йонеды говорит, что считывающий функтор естественно сопоставить с любым другим функтором.

Естественное преобразование является полиморфной функцией. Итак, к данному функтору F имеется отображение от считывающего функтора:

```
alpha :: forall x . (a -> x) -> F x
```

Как обычно, `forall` является необязательным, но я специально записываю его явно, чтобы подчеркнуть параметрический полиморфизм естественных преобразований.

Лемма Йонеды сообщает нам, что эти естественные преобразования находятся во взаимно однозначном соответствии с элементами `F a`:

```
forall x . (a -> x) -> F x ≅ F a
```

Правая часть этого тождества — это то, что мы обычно рассматриваем как структуру данных. Помните интерпретацию функторов как обобщенных контейнеров? `F a` — это контейнер, содержащий `a`. Но левая часть является полиморфной функцией, которая принимает функцию в качестве аргумента. Лемма Йонеды утверждает, что эти два представления эквивалентны — они содержат одну и ту же информацию.

Еще один способ сказать это — дайте мне полиморфную функцию типа:

```
alpha :: forall x . (a -> x) -> F x
```

и я создам контейнер для `a`. Трюк здесь, который был использован при доказательстве леммы Йонеды, в том, что мы называем эту функцию `id`, чтобы получить элемент `F a`:

```
alpha id :: F a
```

Обратное также верно, т.е. при значении типа `F a`:

```
fa :: F a
```

можно определить полиморфную функцию:

```
alpha h = fmap h fa
```

корректного типа. Вы можете легко переключаться между этими двумя представлениями.

Преимущество наличия нескольких представлений состоит в том, что одно из них может быть проще для построения, чем другое, или что одно может быть более эффективным в некоторых приложениях, чем другое.

Простейшей иллюстрацией этого принципа является преобразование кода, которое часто используется при построении компилятора: стиль продолжения передачи или CPS. Это самое простое применение леммы Йонеды к тождественному функтору. Замена `F` тождественным морфизмом приводит к:

```
forall r . (a -> r) -> r ≅ a
```

Интерпретация этой формулы заключается в том, что любой тип `a` может быть заменен функцией, которая принимает «обработчик» для `a`. Обработчик — это функция, принимающая `a` и выполняющая остальную часть вычисления — продолжение (тип `r` обычно инкапсулирует некоторый код состояния.)

Этот стиль программирования очень распространен в пользовательских интерфейсах, асинхронных системах и параллельном программировании. Недостатком CPS является то, что он включает инверсию управления. Код разделяется между производителями и потребителями (обработчиками), и его сложно скомбинировать. Любой, кто занимался каким-либо нетривиальным веб-программированием, знаком с кошмаром кода спагетти от взаимодействующих обработчиков состояния. Как мы увидим позже, разумное использование функторов и монад может восстановить некоторые свойства композиции CPS.

15.2 Ко-Йонеда

Как обычно, мы получаем бонусную конструкцию, инвертируя направление стрелок. Лемма Йонеды может быть применена к обратной категории $\mathbf{C}^{op}$, чтобы предоставить отображение между контравариантными функторами.

Эквивалентно, мы можем получить лемму ко-Йонеды путем фиксации целевого объекта наших hom-функторов вместо источника. Мы получаем контравариантный hom-функтор от $\mathbf{C}$ к $\mathbf{Set}$: $\mathbf{C}(-, a)$. Контравариантный вариант леммы Йонеды устанавливает взаимно однозначное соответствие между естественными преобразованиями от этого функтора к любому другому контравариантному функтору F и элементами множества $F\,a$:

$$\mathbf{Nat}(\mathbf{C}(-, a), F) \cong F\,a$$

Вот Haskell версия леммы ко-Йонеды:

```
forall x . (x -> a) -> F x ≅ F a
```

Обратите внимание, что в некоторых литературных источниках именно эта контравариантная версия называется леммой Йонеды.

Упражнения

1. Покажите, что две функции `phi` и `psi`, которые образуют изоморфизм Йонеды, на Haskell, являются инверсиями друг друга.

   ```
   phi       :: (forall x . (a -> x) -> F x)
                                        -> F a
   phi alpha = alpha id
   psi       :: F a -> (forall x . (a -> x)
                                        -> F x)
   psi fa h  = fmap h fa
   ```

2. Дискретная категория — это категория, которая имеет объекты, но не морфизмы, отличные от тождественных. Как работает лемма Йонеды для функторов от такой категории?

3. Список единиц `[()]` не содержит никакой другой информации, кроме длины этого списка. Таким образом, в качестве типа данных его можно рассматривать как кодировку целых чисел. Пустой список кодирует ноль, синглетон `[()]` (значение, а не тип) кодирует единицу и так далее. Постройте другое представление этого типа данных, используя лемму Йонеды для функтора списка.

Глава 16

Вложение Йонеды

Ранее мы видели, что при фиксации объекта a в категории $\mathbf{C}$, отображение $\mathbf{C}(a, -)$ является (ковариантным) функтором от $\mathbf{C}$ к $\mathbf{Set}$.

$$x \to \mathbf{C}(a, x)$$

(Кообласть — $\mathbf{Set}$, потому что $\hom$-множество $\mathbf{C}(a, x)$ является множеством) Мы называем это отображение $\hom$-функтором, ранее мы определили его действие также и на морфизмах.

Теперь давайте изменять a в этом отображении. Мы получаем новое отображение, которое назначает $\hom$-функтор $\mathbf{C}(a, -)$ любому a.

$$a \to \mathbf{C}(a, -)$$

Это отображение объектов из категории $\mathbf{C}$ в функторы, которые являются объектами в категории функторов (см. подраздел о функторных категориях в разделе «Естественные преобразования»). Давайте используем обозначение $[\mathbf{C},\mathbf{Set}]$ для категории функторов от $\mathbf{C}$ к $\mathbf{Set}$. Напомню, что $\hom$-функторы являются прототипическими представимыми функторами.

Каждый раз, когда мы имеем сопоставление объектов между двумя категориями, естественно спросить, является ли такое отображение функтором. Другими словами, можем ли мы поднять морфизм из одной категории в морфизм другой категории. Морфизм в $\mathbf{C}$ является просто

элементом из $\mathbf{C}(a, b)$, но морфизм в категории функторов $[\mathbf{C},\mathbf{Set}]$ является естественным преобразованием. Поэтому мы ищем отображение морфизмов на естественные преобразования.

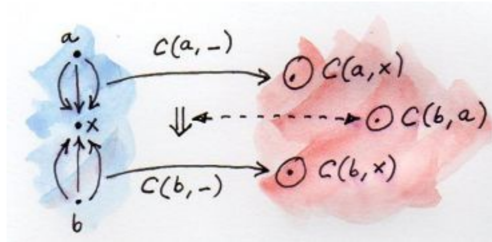
Посмотрим, удастся ли найти естественное преобразование, соответствующее морфизму $f :: a \to b$. Во-первых, давайте посмотрим, на что отображаются a и b. Они отображаются на два функтора: $\mathbf{C}(a, -)$ и $\mathbf{C}(b, -)$. Нам требуется естественное преобразование между этими двумя функторами.

И вот трюк — мы используем лемму Йонеды:

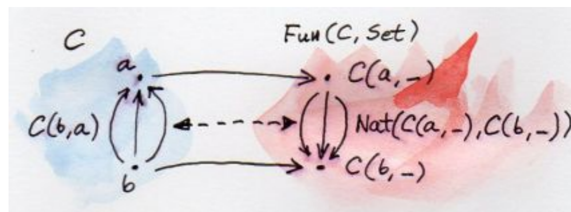
$$[\mathbf{C},\mathbf{Set}](\mathbf{C}(a, -), F) \cong F\, a$$

и заменяем функтор F на hom-функтор $\mathbf{C}(b, -)$:

$$[\mathbf{C},\mathbf{Set}](\mathbf{C}(a, -),\mathbf{C}(b, -)) \cong \mathbf{C}(b, a)$$



Это как раз то естественное преобразование между двумя hom-функторами, которые мы искали, но с небольшим завихрением: у нас есть отображение между естественным преобразованием и морфизмом — элементом из $\mathbf{C}(b, a)$, которое идет в «неправильном» направлении. Но это нормально, это означает только то, что рассматриваемый нами функтор контравариантен.



На самом деле, мы получили даже больше, чем рассчитывали. Отображение от $\mathbf{C}$ к $[\mathbf{C},\mathbf{Set}]$ является не только контравариантным функтором — это *вполне точный* (т.е. полный и точный) функтор. Полнота и точность — это те свойства функторов, которые описывают, как они отображают hom-множества.

Точный функтор *инъективен* на hom-множествах, т.е. он отображает разные морфизмы в разные морфизмы. Другими словами, он их не объединяет.

Полный функтор *сюръективен* на hom-множествах, т.е. он отображает одно hom-множество на другое hom-множество, полностью покрывающее последнее.

Вполне точный функтор F является *биекцией* на hom-множествах — взаимно однозначным соответствием элементов обоих множеств. Для каждой пары объектов a и b из исходной категории $\mathbf{C}$ существует биекция между $\mathbf{C}(a, b)$ и $\mathbf{D}(F\, a, F\, b)$, где $\mathbf{D}$ является целевой категорией из F (в нашем случае категорией функторов $[\mathbf{C},\mathbf{Set}]$). Заметьте, это не означает, что F является биекцией объектов. В $\mathbf{D}$ могут быть объекты, которые не находятся в образе F, и мы не можем ничего сказать о hom-множествах этих объектов.

16.1 Вложение

Описанный нами (контравариантный) функтор, который отображает объекты из $\mathbf{C}$ к функторам из $[\mathbf{C},\mathbf{Set}]$:

$$a \to \mathbf{C}(a, -)$$

определяет *вложение* Йонеды. Он *вкладывает* категорию $\mathbf{C}$ (строго говоря, категорию $\mathbf{C}^{op}$, из-за контравариантности) внутрь категории функторов $[\mathbf{C},\mathbf{Set}]$. Он не только отображает объекты из $\mathbf{C}$ к функторам, но и скрупулезно сохраняет все связи между ними.

Это очень полезный результат, потому что математики знают много о категории функторов, особенно функторов с кообластью $\mathbf{Set}$. Мы можем получить много информации о произвольной категории $\mathbf{C}$, встраивая ее в категорию функторов.

Конечно, существует двойственная версия вложения Йонеды, которую иногда называют вложением ко-Йонеды. Обратите внимание, что мы могли бы начать с фиксации целевого объекта (а не исходного объекта) каждого hom-множества $\mathbf{C}(-, a)$. Это дало бы нам контравариантный hom-функтор. Контравариантные функторы от $\mathbf{C}$ к $\mathbf{Set}$ являются знакомыми для нас предпучками (см., например, раздел «Пределы и копределы»). Вложение ко-Йонеды определяет вложение категории $\mathbf{C}$ в категорию предпучков. Его действие на морфизмы определяется так:

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(-, a), \mathbf{C}(-, b)) \cong \mathbf{C}(a, b)$$

Опять же, математикам известно многое о категории предпучков, поэтому возможность встравать произвольную категорию в нее — большое достижение.

16.2 Применимость в Haskell

На Haskell вложение Йонеды может быть представлено как изоморфизм естественных преобразований между считывающими функторами, с одной стороны, и функциями (идущими в противоположном направлении), с другой стороны:

```
forall x. (a -> x) -> (b -> x) ≅ b -> a
```

(напомним, что считыващий функтор эквивалентен `((->) a)`).

Левая часть этого тождества является полиморфной функцией, которая при заданной функции от `a` к `x` и значении типа `b` может выдавать значение типа `x` (я не каррирую, опуская круглые скобки, функцию `b -> x`). Единственный способ, которым это можно сделать для всех `x`, — это наделение этой функции возможностью преобразования `b` в `a`. Она должна иметь скрытый доступ к функции `b -> a`.

Допуская такой преобразователь, `btoa`, можно определить левую сторону, назвав ее `fromY`, как:

```
fromY    :: (a -> x) -> b -> x
fromY f b = f (btoa b)
```

И наоборот, имея функцию fromY, мы можем восстановить преобразователь, применив fromY к тождественной функции:

```
fromY id :: b -> a
```

Это устанавливает биекцию между функциями типов fromY и btoa.

Альтернативный способ рассмотрения этого изоморфизма состоит в том, что он является CPS-кодированием функции от b к a. Аргумент a -> x является продолжением (обработчиком). Результатом будет функция от b к x, которая при вызове со значением типа b будет выполнять продолжение, предварительно скомпонованное с кодируемой функцией.

Вложение Йонеды также объясняет некоторые из альтернативных представлений структур данных в Haskell. В частности, оно обеспечивает очень полезное представление линз (https://bartoszmilewski.com/2015/07/13/from-lenses-to-yoneda-embedding/) из библиотеки Control.Lens.

16.3 Пример с предпорядком

Этот пример был предложен Робертом Харпером и является приложением вложения Йонеды в категорию, определенную предпорядком. Предпорядок — это множество с отношением упорядочения между его элементами, которое традиционно записывается как $\leqslant$. «Пред» в этом термине означает, что мы только требуем, чтобы это отношение было транзитивным и рефлексивным, но не обязательно антисимметричным (так что возможно появление циклов).

Множество с отношением предпорядка порождает категорию. Объектами являются элементы этого множества. Морфизм от объекта a к объекту b либо не существует, если объекты не могут быть сопоставлены или если не верно, что $a \leqslant b$; либо морфизм существует, если $a \leqslant b$, тогда он указывает на связь от a к b. При этом может быть не более одного морфизма от одного объекта к другому. Поэтому любое $\hom$-множество в такой категории является либо пустым, либо одноэлементным множеством. Подобная категория называется *тонкой*.

Легко убедиться в том, что эта конструкция действительно относится к категории: стрелки могут быть компонуемы, потому что, если $a \leqslant b$ и

$b \leqslant c$, то $a \leqslant c$, а композиция является ассоциативной. Также имеются тождественные стрелки, поскольку каждый элемент равен (меньше или равен) самому себе (рефлексивность лежащего в основе отношения).

Теперь мы можем применить вложение ко-Йонеды к категории предпорядка. В частности, нам интересно ее действие на морфизмы:

$$[\mathbf{C},\mathbf{Set}](\mathbf{C}(-,a),\mathbf{C}(-,b)) \cong \mathbf{C}(a,b)$$

hom-множество в правой части непусто тогда и только тогда, когда $a \leqslant b$ — в данном случае это одноэлементное множество. Следовательно, если $a \leqslant b$, то существует единственное естественное преобразование слева. В противном случае естественное преобразование отсутствует.

Итак, что же такое естественное преобразование между hom-функторами в предпорядке? Это должно быть семейство функций между множествами $\mathbf{C}(-,a)$ и $\mathbf{C}(-,b)$. В предпорядке каждое из этих множеств может быть либо пустым, либо одноэлементным. Рассмотрим, какие функции есть в нашем распоряжении.

Существует функция от пустого множества к себе (тождественная функция, действующая на пустом множестве), также функция $absurd$ от пустого множества к одноэлементному множеству (она ничего не делает, так как ее нужно определить для элементов пустого множества, которых нет), и функция от одноэлементного множества к себе (тождественная функция, действующая на одноэлементном множестве). Единственная комбинация, которая запрещена, — это отображение от одноэлементного множества к пустому множеству (какое значение имела бы такая функция при действии на единственный элемент?).

Таким образом, наше естественное преобразование никогда не свяжет одноэлементное hom-множество с пустым hom-множеством. Другими словами, если $x \leqslant a$ (одноэлементное hom-множество $\mathbf{C}(x,a)$), то $\mathbf{C}(x,b)$ не может быть пустым. Непустота $\mathbf{C}(x,b)$ означает, что x меньше или равно b. Таким образом, существование рассматриваемого естественного преобразования требует, чтобы для каждого x, если $x \leqslant a$, то $x \leqslant b$.

$$\forall x,\ x \leqslant a \Rightarrow x \leqslant b$$

С другой стороны, ко-Йонеда говорит нам, что существование этого естественного преобразования эквивалентно тому, что $\mathbf{C}(a,b)$ непусто

или $a \leqslant b$. Собирая все вместе, получаем:

$$a \leqslant b, \text{ если и только если для всех } x,\ x \leqslant a \Rightarrow x \leqslant b$$

Мы могли бы прийти к этому результату непосредственно. Интуитивно, если $a \leqslant b$, то все элементы, которые предшествуют a, должны предшествовать и b. И наоборот, когда вы подставляете a для x справа, то отсюда следует, что $a \leqslant b$. Но надо признать, что получение этого результата через вложение Йонеды является намного более захватывающим.

16.4 Естественность

Лемма Йонеды устанавливает изоморфизм между множеством естественных преобразований и объектом в $\mathbf{Set}$. Естественные преобразования являются морфизмами в категории функторов $[\mathbf{C},\mathbf{Set}]$. Множество естественных преобразований между любыми двумя функторами является hom-множеством в этой категории. Лемма Йонеды — это изоморфизм:

$$[\mathbf{C},\mathbf{Set}](\mathbf{C}(a,-),F) \cong F\,a$$

Этот изоморфизм оказывается естественным как по F, так и по a. Другими словами, это естественность в (F,a), паре, взятой из категорного произведения $[\mathbf{C},\mathbf{Set}] \times \mathbf{C}$. Обратите внимание, что здесь мы рассматриваем F как объект в категории функторов.

Давайте подумаем, что это значит. Естественный изоморфизм — это обратимое естественное преобразование между двумя функторами. И действительно, правая часть нашего изоморфизма является функтором. Это функтор от $[\mathbf{C},\mathbf{Set}] \times \mathbf{C}$ к $\mathbf{Set}$. Его действие на пару (F,a) является множеством — результатом вычисления функтора F на объекте a. Он называется функтором вычисления.

Левая часть также является функтором, переводящим (F,a) в множество естественных преобразований $[\mathbf{C},\mathbf{Set}](\mathbf{C}(a,-),F)$.

Чтобы показать, что это действительно функторы, мы должны определить их действие на морфизмах. Но что такое морфизм между парами (F,a) и (G,b)? Это пара морфизмов (Φ,f), первый из которых является морфизмом между функторами — естественным преобразованием, а второй — регулярным морфизмом в $\mathbf{C}$.

Функтор вычисления берет эту пару (Φ, f) и отображает ее в функцию между двумя множествами $F\,a$ и $G\,b$. Мы можем легко построить такую функцию по компоненте Φ в точке a (которая отображает $F\,a$ в $G\,a$) и морфизму f, поднятому посредством G:

$$(G\,f) \circ \Phi_a$$

Обратите внимание, что из-за естественности Φ, это то же самое, что и:

$$\Phi_b \circ (F\,f)$$

Я не собираюсь доказывать естественность всего изоморфизма — после того, как вы уяснили, что такое функторы, доказательство является довольно механическим. Это следует из того, что наш изоморфизм строится из функторов и естественных преобразований. Просто тяжело сделать это не верным (но это уже будет относиться к известной области медицины).

Упражнения

1. Выразите вложение ко-Йонеды на Haskell.
2. Покажите, что биекция, которую мы установили между `fromY` и `btoa`, является изоморфизмом (эти два отображения являются обратными друг другу).
3. Разработайте вложение Йонеды для моноида. Какой функтор соответствует одному объекту моноида? Какие естественные преобразования соответствуют моноидным морфизмам?
4. Каково применение ковариантного вложения Йонеды в предпорядки? (Вопрос, предложенный Гершомом Базерманом)
5. Вложение Йонеды может быть использовано для встраивания произвольной категории функторов $[\mathbf{C},\mathbf{D}]$ в категорию функторов $[[\mathbf{C},\mathbf{D}],\mathbf{Set}]$. Выясните, как это работает над морфизмами (которые в данном случае являются естественными преобразованиями).

Часть III

Глава 17

Все о морфизмах

Если я еще не убедил вас, что теория категорий — это среда обитания морфизмов, то я не выполнил свою работу должным образом. Поскольку следующая тема посвящена сопряжениям, которые определяются в терминах изоморфизмов hom-множеств, имеет смысл пересмотреть наши интуитивные представления о строительных блоках hom-множеств. Кроме того, вы увидите, что сопряжения предоставляют более общий язык для описания множества конструкций, которые мы изучали ранее, поэтому он может помочь переосмыслить и их тоже.

17.1 Функторы

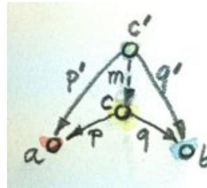
Во-первых, вы должны действительно думать о функторах как об отображениях морфизмов — точка зрения, которая подчеркивается в Haskell при определении класса типов `Functor`, который обволакивает `fmap`. Конечно, функторы отображают и объекты — концы морфизмов — в противном случае мы не могли бы говорить о сохранении структуры. Объекты указывают нам, какие пары морфизмов являются компонуемыми. Цель одного морфизма должна совпадать с источником другого, если они должны быть скомпонованы. Так что, если мы хотим, чтобы композиция морфизмов отображалась на композицию поднятых морфизмов, то в значительной степени это определяется отображением их граничных точек.

17.2 Коммутативные диаграммы

Многие свойства морфизмов выражаются через коммутативные диаграммы. Если конкретный морфизм можно определить как композицию других морфизмов более чем одним способом, то мы получаем коммутативную диаграмму.

В частности, коммутативные диаграммы составляют основу почти всех универсальных конструкций (с понятными исключениями инициального и терминального объектов). Мы видели это в определениях произведений, копроизведений, различных других (ко) пределов, экспоненциальных объектов, свободных моноидов и т.д.

Произведение представляет собой простой пример универсальной конструкции. Мы выбираем два объекта a и b и смотрим, существует ли объект c, вместе с парой морфизмов p и q, который обладает универсальным свойством, чтобы быть их произведением.

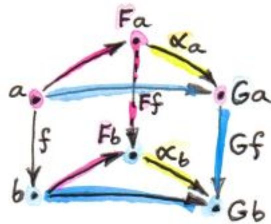


Произведение является частным случаем предела. Предел определяется в терминах конусов. Конус в общем случае строится, исходя из коммутативности диаграмм, которая может быть заменена подходящим условием естественности для отображаемых функторов. Таким образом, коммутативность сводится к роли языка сборки относительно языка более высокого уровня, языка естественных преобразований.

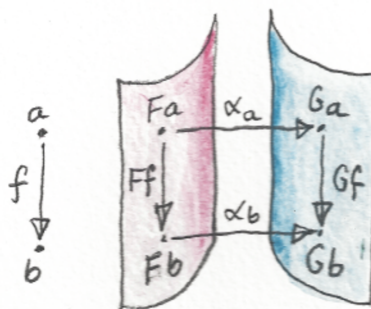
17.3 Естественные преобразования

Вообще, естественные преобразования очень полезны всякий раз, когда нам нужно перейти от морфизмов к коммутативным квадратам. Две противоположные стороны естественного квадрата являются отображениями некоторым морфизмом f двух функторов F и G. Другие стороны

являются компонентами естественного преобразования (которые также являются морфизмами).



Естественность означает, что когда вы переходите к «соседнему» компоненту (под соседним я имею в виду связанный с ним морфизм), вы следуете структуре и категории, и функторов. Не имеет значения, используете ли вы сначала компонент естественного преобразования для преодоления разрыва между объектами, а затем переходите к его соседнему компоненту с помощью функтора, или наоборот. Оба направления ортогональны. Естественное преобразование перемещает фокус зрения влево и вправо, а функторы перемещают его вверх и вниз или назад и вперед — как вам проще это представить. Вы можете визуализировать изображение функтора в виде листа в целевой категории. Естественное преобразование отображает один такой лист, соответствующий F, на другой лист, соответствующий G.



Мы видели примеры такой ортогональности на Haskell. Там действие функтора изменяет содержимое контейнера без изменения его формы, в то время как естественное преобразование переупаковывает нетрону-

тое содержимое в другой контейнер. Порядок этих операций не имеет значения.

Мы наблюдали, что конусы в определении предела заменены естественными преобразованиями. Естественность гарантирует, что стороны каждого конуса удовлетворяют соотношению коммутативности. Тем не менее, предел определяется в терминах отображений между конусами. Эти отображения также должны удовлетворять условиям коммутативности (например, треугольники в определении произведения должны быть коммутативными).

Эти условия также могут быть заменены согласно естественности. Вы можете вспомнить, что универсальный конус, или предел, определяется как естественное преобразование между (контравариантным) hom-функтором:

$$F :: c \to \mathbf{C}(c, \mathrm{Lim}D)$$

и (также контравариантным) функтором, который отображает объекты из $\mathbf{C}$ в конусы, которые сами являются естественными преобразованиями:

$$G :: c \to \mathbf{Nat}(\Delta_c, D)$$

Здесь Δ_c — постоянный функтор, а D — функтор, который определяет диаграмму в $\mathbf{C}$. Оба функтора F и G обладают корректно определенными действиями над морфизмами из $\mathbf{C}$. Так получилось, что это конкретное естественное преобразование между F и G является изоморфизмом.

17.4 Естественные изоморфизмы

Естественный изоморфизм — это изоморфизм, который является естественным преобразованием, каждая компонента которого обратима, — это способ формулировки в теории категорий того, что «две вещи одинаковы». Компонент такого преобразования должен быть изоморфизмом между объектами — морфизмом, имеющим обратный. Если визуализировать образы функторов в виде листов, то естественный изоморфизм — это взаимно однозначное обратимое отображение между этими листами.

17.5 hom-множества

Так что же такое морфизмы? Они более структурны, чем объекты: в отличие от объектов, морфизмы имеют два конца. Но если вы зафиксируете объекты, являющиеся источником и целью, морфизмы между ними образуют неинтересное множество (по крайней мере для локально малых категорий). Мы можем дать имена элементам этого множества, например, f или g, чтобы отличать одно от другого, но что же в действительности делает их различными?

Существенное различие между морфизмами в данном hom-множестве заключается в том, что их можно соединять с другими морфизмами (из примыкающих hom-множеств). Если существует морфизм h, композиция которого (слева либо справа) с f отличается от его композиции с g, например:

$$h \circ f \neq h \circ g$$

то мы можем непосредственно «наблюдать» разницу между f и g. Но если различие не бросается в глаза, мы можем использовать функторы для увеличения масштаба hom-множества. Функтор F может отображать два морфизма в разные морфизмы:

$$F\, f \neq F\, g$$

в более богатой категории, где примыкающие hom-иножества обеспечивают большее разрешение, например,

$$h' \circ F\, f \neq h' \circ F\, g$$

где h' не входит в образ F.

17.6 Изоморфизмы hom-множества

Многие категорные конструкции базируются на изоморфизмах между hom-множествами. Но поскольку hom-множества — это просто множества, то обыкновенный изоморфизм между ними говорит вам о немногом. Для конечных множеств изоморфизм между ними просто отражает то, что они имеют одинаковое количество элементов. Если изоморфные множества бесконечны, их мощность должна быть одинаковой. Но любой значимый изоморфизм hom-множеств должен учитывать

композицию. А композиция охватывает более одного hom-множества. Нам необходимо определить изоморфизмы, охватывающие целые наборы hom-множеств, при этом наложив некоторые условия совместимости, которые связаны с композицией. И естественный изоморфизм как раз подходит для этого.

Но что представляет собой естественный изоморфизм hom-множеств? Натуральность является свойством отображений между функторами, а не множеств. То есть в действительности мы говорим о естественном изоморфизме многозначных hom-функторов. Эти функторы являются большим, чем просто многозначными функторами. Их действие на морфизмах индуцируется соответствующими hom-функторами. Морфизмы канонически сопоставляется с hom-функторами использованием слева или справа от знака композиции (в зависимости от ковариации функтора).

Вложение Йонеды является одним из примеров такого изоморфизма. Оно отображает hom-множества из $\mathbf{C}$ к hom-множествам из категории функторов; и это естественно. Один функтор в вложении Йонеды является hom-функтором из $\mathbf{C}$, а другой отображает объекты к множествам естественных преобразований между hom-множествами.

Определение предела также является естественным изоморфизмом между hom-множествами (второй, опять же, в категории функторов):

$$\mathbf{C}(c, \mathrm{Lim}D) \simeq \mathbf{Nat}(\Delta_c, D)$$

Оказывается, что наше построение экспоненциального объекта или свободного моноида можно также переписать в виде естественного изоморфизма между hom-множествами.

Это не совпадение — мы увидим далее, что это всего лишь разные примеры сопряжений, которые определяются как естественные изоморфизмы hom-множеств.

17.7 Асимметрия hom-множеств

Есть одно наблюдение, которое поможет нам понять сопряжения. hom-множества, вообще говоря, несимметричны. hom-множество $\mathbf{C}(a, b)$ ча-

сто сильно отличается от hom-множества $\mathbf{C}(b, a)$. Предельным проявлением этой асимметрии является частичнй порядок, рассматриваемый как категория. В частичном порядке морфизм от a к b существует тогда и только тогда, когда a меньше или равно b. Если a и b различны, то не может быть никакого морфизма, идущего другим путем от b к a. Так что, если hom-множество $\mathbf{C}(a, b)$ не пусто, что в данном случае означает, что это одноэлементное множество, то $\mathbf{C}(b, a)$ должно быть пустым, если $a = b$. Стрелки в этой категории ориентированы в одном направлении.

Предпорядок, основанный на отношении, которое не обязательно является антисимметричным, также является «обычно» направленным, за исключением редких циклов. Удобно думать о произвольной категории как об обобщении предпорядка.

Предпорядок — это тонкая категория — все hom-множества либо одноточечные, либо пустые. Мы можем представить общую категорию как «толстый» предпорядок.

Упражнения

1. Рассмотрите некоторые вырожденные случаи условия естественности и изобразите соответствующие диаграммы. Например, что произойдет, если любой из функторов F или G отобразит оба объекта a и b (концы морфизма $f :: a \to b$) к одному и тому же объекту, например, $F\,a = F\,b$ или $G\,a = G\,b$? (Обратите внимание, что таким образом вы получаете конус или ко-конус.) Затем рассмотрите случаи, когда, либо $F\,a = G\,a$, либо $F\,b = G\,b$. Наконец, что получится, если вы начнете с морфизма, который замыкается на себя — $f :: a \to a$?

Глава 18

Сопряжения

В математике имеются различные способы выразить то, что одна вещь похожа на другую. Самый строгий из них — равенство. Две сущности равны, если нет возможности отличить одну от другой. Тогда одну из них можно заменить другой в любом мыслимом контексте. Например, вы заметили, что мы использовали равенство морфизмов каждый раз, когда мы говорили о коммутативных диаграммах? Это потому, что морфизмы образуют множество (hom-множество), а элементы множества можно проверять на равенство.

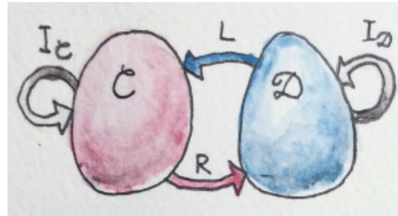
Но равенство часто чересчур строго. Существует много примеров того, что вещи, одинаковые для многих целей и задач, не являются фактически равными. Например, тип пары `(Bool,Char)` не является строго равным типу `(Char,Bool)`, но мы понимаем, что эти пары содержат одну и ту же информацию. Это понятие лучше всего передается изоморфизмом между двумя типами — морфизмом, который обратим. Так как это морфизм, он сохраняет структуру, а «изо» означает, что это часть круиза, который возвращает вас в исходную точку, независимо от того, с какой стороны вы стартуете. В случае пар этот изоморфизм назван `swap`:

```
swap      :: (a,b) -> (b,a)
swap (a,b) = (b,a)
```

`swap` оказывается обратным себе.

18.1 Сопряжение и пара «единица/коединица»

Когда мы говорим об изоморфности категорий, мы выражаем это в терминах отображений между категориями — функторами. Мы хотели бы иметь возможность сказать, что две категории $\mathbf{C}$ и $\mathbf{D}$ изоморфны, если существует функтор R («правый») от $\mathbf{C}$ к $\mathbf{D}$, который обратим. Другими словами, существует еще один функтор L («левый») от $\mathbf{D}$ обратно к $\mathbf{C}$, который, когда он соединен с R, равен тождественному функтору I. Имеются две возможные композиции $R \circ L$ и $L \circ R$, и два возможных тождественных функтора — один, находящийся в $\mathbf{C}$ и другой — в $\mathbf{D}$.



Но вот сложный момент: что значит для двух функторов быть равными? Что мы подразумеваем под таким равенством:

$$R \circ L = I_{\mathbf{D}}$$

или под этим:

$$L \circ R = I_{\mathbf{C}}$$

Было бы разумно определить равенство функторов в терминах равенства объектов. Два функтора, действуя на равные объекты, должны приводить к равным объектам. Но мы, вообще говоря, не имеем понятия равенства объектов для произвольной категории. Оно просто не входит в определение (погружаясь в размышление «что такое равенство на самом деле», мы попали бы во владения гомотопической теории типов).

Вы могли бы утверждать, что функторы являются морфизмами в категории категорий, поэтому они должны быть сопоставимы по принципу равенства. И действительно, до тех пор, пока мы говорим о малых категориях, где объекты образуют множество, мы действительно можем использовать равенство элементов множества для сравнения объектов.

Но, напомню, что **Cat** на самом деле является **2**-категорией. hom-множества в **2**-категории имеют дополнительную структуру — существуют 2-морфизмы, действующие между 1-морфизмами. В **Cat** 1-морфизмы являются функторами, а 2-морфизмы — естественными преобразованиями. Поэтому более естественным (не могу избежать этого каламбура!) будет считать естественные изоморфизмы заменой для равенства при разговоре о функторах.

Таким образом, вместо изоморфизма категорий имеет смысл рассмотреть более общее понятие эквивалентности. Две категории **C** и **D** эквивалентны, если мы можем найти два встречных функтора между ними, композиция которых (в любом порядке) естественно изоморфна тождественному функтору. Другими словами, если существует двухстороннее естественное преобразование между композицией $R \circ L$ и тождественным функтором $I_{\mathbf{D}}$, и другое — между композицией $L \circ R$ и тождественным функтором $I_{\mathbf{C}}$.

Сопряжение даже слабее эквивалентности, поскольку оно не требует, чтобы композиция двух функторов была изоморфна тождественному функтору. Вместо этого оно предусматривает существование одностороннего естественного перехода от $I_{\mathbf{D}}$ к $R \circ L$, а также от $L \circ R$ к $I_{\mathbf{C}}$. Вот сигнатуры этих двух естественных преобразований:

$$\eta :: I_{\mathbf{D}} \to R \circ L$$
$$\varepsilon :: L \circ R \to I_{\mathbf{C}}$$

η называется единицей, а ε — коединицей сопряжения.

Обратите внимание на асимметрию между этими двумя определениями. В общем случае у нас нет двух остальных отображений:

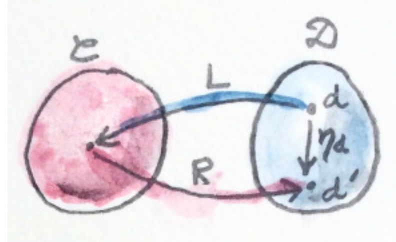
$R \circ L \to I_{\mathbf{D}}$	не обязательно
$I_{\mathbf{C}} \to L \circ R$	не обязательно

Из-за этой асимметрии функтор L называется *левым сопряженным* к функтору R, а функтор R является *правым сопряженным* к L (разумеется, левое и правое имеют смысл, только если вы изображаете свои диаграммы определенным образом).

Компактное обозначение для сопряжения:

$$L \dashv R$$

Для лучшего понимания сопряжения, проанализируем единицу и коединицу более подробно.

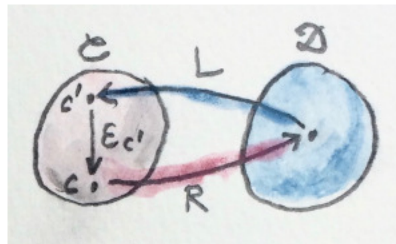


Начнем с единицы. Это естественное преобразование, так что это семейство морфизмов. Для объекта d из $\mathbf{D}$, компонента η является морфизмом между $I\,d$, равным d, и $(R \circ L)\,d$, который на рисунке обозначен как d':

$$\eta_d :: d \to (R \circ L)\,d$$

Заметим, что композиция $R \circ L$ является эндофунктором в $\mathbf{D}$.

Это формула выражает то, что мы можем выбрать любой объект d из $\mathbf{D}$ в качестве отправной точки и использовать круизный функтор $R \circ L$ для выбора целевого объекта d'. Затем направляем стрелку — морфизм η_d — к цели.



Аналогично, компонент коединицы ε можно описать так:

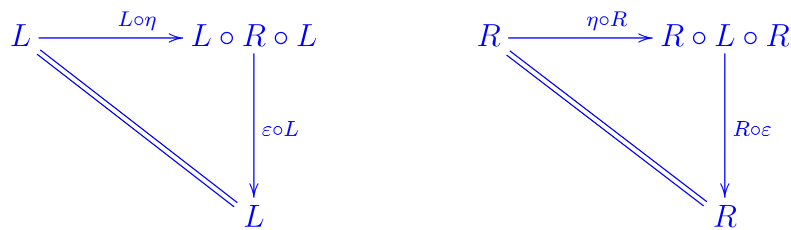
$$\varepsilon_c :: (L \circ R)\,c \to c$$

Это говорит о том, что мы можем выбрать любой объект c из $\mathbf{C}$ в качестве цели и использовать круизный функтор $L \circ R$, чтобы выбрать источник c'. Затем мы направляем стрелку — морфизм $\varepsilon_{c'}$ — от источника к цели.

Еще один вариант рассмотрения единицы и коединицы состоит в том, что единица позволяет *вводить* композицию $R \circ L$ там, куда бы мы могли вставить тождественный функтор на $\mathbf{D}$, а коединица позволяет нам исключить композицию $L \circ R$, заменив ее тождественным функтором на $\mathbf{C}$. Это приводит к некоторым «очевидным» условиям согласованности, которые гарантируют, что введение композиции, сопровождаемое исключением ее «зеркального» изображения, ничего не меняет:

$$L = L \circ I_{\mathbf{D}} \to L \circ R \circ L \to I_{\mathbf{C}} \circ L = L$$
$$R = I_{\mathbf{D}} \circ R \to R \circ L \circ R \to R \circ I_{\mathbf{C}} = R$$

Они называются треугольными тождествами, потому что обеспечивают коммутативность следующих диаграмм:



Это диаграммы из категории функторов: стрелки являются естественными преобразованиями, а их композиция является горизонтальной композицией естественных преобразований. В обозначениях компонент, это тождества:

$$\varepsilon_{L\,d} \circ L\,\eta_d = \mathrm{id}_{L\,d}$$
$$R\,\varepsilon_c \circ \eta_{R\,c} = \mathrm{id}_{R\,c}$$

Мы часто встречаем единицу и коединицу в Haskell под разными именами. Единица известна как `return` (или `pure`, в определении `Applicative`):

```
return :: d -> m d
```

а коединица — как `extract`:

```
extract :: w c -> c
```

Здесь m — (эндо) функтор, соответствующий $R \circ L$, а w — (эндо) функтор, соответствующий $L \circ R$. Как мы увидим позже, они являются частью определения монады и комонады, соответственно.

Если вы рассматриваете эндофунктор как контейнер, единица (или return) является полиморфной функцией, которая создает окружение по умолчанию вокруг значения произвольного типа. Коединица (или extract) делает обратное: извлекает или выдает одно значение из контейнера.

Мы увидим позже, что каждая пара сопряженных функторов определяет монаду и комонаду. И наоборот, каждая монада или комонада может быть разложена на пару сопряженных функторов — эта факторизация, однако, не является единственной.

В Haskell мы часто используем монады, но редко факторизуем их на пары сопряженных функторов, прежде всего потому, что эти функторы обычно выводят нас за рамки **Hask**.

Тем не менее, мы можем определить сопряжеения эндофункторов на Haskell. Вот часть определения из Data.Functor.Adjunction:

```
class (Functor f, Representable u) =>
   Adjunction f u | f -> u, u -> f where
      unit   :: a -> u (f a)
      counit :: f (u a) -> a
```

Это определение требует некоторого пояснения. Прежде всего, оно описывает класс многопараметрического типа — двумя параметрами являются f и u. Оно устанавливает отношение, называемое Adjunction, между этими двумя конструкторами типа.

Дополнительные условия, после вертикальной черты, указывают функциональные зависимости. Например, f -> u означает, что u определяется через f (связь между f и u является функцией, здесь, на конструкторах типов). Напротив, u -> f означает, что, если нам известно u, то f определяется однозначно.

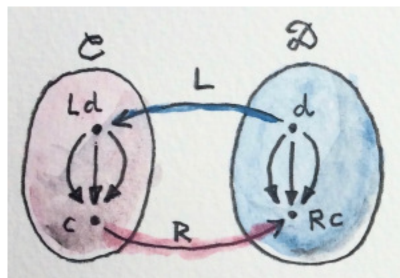
Я объясню далее, в какой момент в Haskell мы можем наложить условие, чтобы правый сопряженный u был *представимым* функтором.

18.2 Сопряжения и hom-множества

Существует эквивалентное определение сопряжения в терминах естественных изоморфизмов hom-множеств. Это определение хорошо сочетается с универсальными конструкциями, которые мы изучали до сих пор. Каждый раз, когда вы встречаете утверждение, что существует единственный морфизм, который факторизует какую-то конструкцию, вы должны понимать это как отображение некоторого множества в hom-множество. В этом и заключается смысл «подбора единственного морфизма».

Кроме того, факторизация часто описывается в терминах естественных преобразований. Факторизация включает коммутативные диаграммы — некоторый морфизм равен композиции двух других морфизмов (факторов). Естественное преобразование отображает морфизмы на коммутативные диаграммы. Таким образом, в универсальной конструкции мы переходим от морфизма к коммутативной диаграмме, а затем к уникальному (единственному) морфизму. Мы приходим к отображению морфизма к морфизму или от одного hom-множества к другому (обычно, в разных категориях). Если это отображение обратимо, и если его можно естественным образом распространить по всем hom-множествам, то получаем сопряжение.

Основное различие между универсальными конструкциями и сопряжениями в том, что последние определены глобально — для всех hom-множеств. Например, используя универсальную конструкцию, вы можете определить произведение двух выбранных объектов, даже если оно не существует ни для какой другой пары объектов в этой категории. Как мы скоро увидим, если произведение какой-либо пары объектов существует в категории, его также можно определить и через сопряжение.



Вот альтернативное определение сопряжения с использованием hom-множеств. Как и раньше, имеем два функтора $L :: \mathbf{D} \to \mathbf{C}$ и $R :: \mathbf{C} \to \mathbf{D}$. Далее, выбираем два произвольных объекта: исходный объект d из $\mathbf{D}$ и целевой объект c из $\mathbf{C}$. Мы можем отобразить исходный объект d на $\mathbf{C}$, используя L. Два объекта в $\mathbf{C}$, $L\,d$ и c, определяют hom-множество:

$$\mathbf{C}(L\,d, c)$$

Аналогично, можно отобразить целевой объект c, используя R. Теперь имеются два объекта в $\mathbf{D}$, d и $R\,c$, которые также определяют hom-множество:

$$\mathbf{D}(d, R\,c)$$

Мы говорим, что L является левым сопряженным к R тогда и только тогда, когда существует изоморфизм hom-множеств:

$$\mathbf{C}(L\,d, c) \cong \mathbf{D}(d, R\,c)$$

который естественен и по d, и по c.

Естественность означает, что источник d можно выбирать, непрерывно перебирая объекты из $\mathbf{D}$, также как и целевой объект c в $\mathbf{C}$. Более точно, мы имеем естественное преобразование φ между следующими двумя (ковариантными) функторами от $\mathbf{C}$ к $\mathbf{Set}$, которые так действуют на объекты:

$$c \to \mathbf{C}(L\,d, c)$$
$$c \to \mathbf{D}(d, R\,c)$$

Другое естественное преобразование, ψ, действует между следующими (контравариантными) функторами:

$$d \to \mathbf{C}(L\,d, c)$$
$$d \to \mathbf{D}(d, R\,c)$$

Оба эти естественные преобразования должны быть обратимыми.

Нетрудно показать, что оба определения сопряжения эквивалентны. Например, выведем единичное преобразование, исходя из изоморфизма hom-множеств:

$$\mathbf{C}(L\,d, c) \cong \mathbf{D}(d, R\,c)$$

Так как этот изоморфизм работает с любым объектом c, он также должен работать с $c = L\,d$:

$$\mathbf{C}(L\,d, L\,d) \cong \mathbf{D}(d, (R \circ L)\,d)$$

Мы знаем, что левая часть должна содержать хотя бы один морфизм, тождественный. Естественное преобразование отобразит этот морфизм к элементу из $\mathbf{D}(d, (R \circ L)\,d)$ или, если вставить тождественный функтор I, отобразит морфизм к:

$$\mathbf{D}(I\,d, (R \circ L)\,d)$$

Мы получаем семейство морфизмов, параметризованных по d. Они образуют естественное преобразование между функтором I и функтором $R \circ L$ (условие естественности легко проверить). Это искомая единица, η.

Наоборот, исходя из существования единицы и коединицы, мы можем определить преобразования между $\hom$-множествами. Например, возьмем произвольный морфизм f в $\hom$-множестве $\mathbf{C}(L\,d, c)$. Мы хотим определить φ, действующее на f, порождающее морфизм в $\mathbf{D}(d, R\,c)$.

Выбор невелик. Единственное, что мы можем попробовать, это поднять f, используя R. Это приведет к получению морфизма $R\,f$ от $R\,(L\,d)$ к $R\,c$ — морфизма, являющегося элементом $\mathbf{D}((R \circ L)\,d, R\,c)$.

Нам нужно, чтобы компонента φ была морфизмом от d к $R\,c$. Это не проблема, так как мы можем использовать компонент η_d для перехода от d к $(R \circ L)\,d$. Мы получаем:

$$\varphi_f = R\,f \circ \eta_d$$

Другое направление аналогично, и, следовательно, это вывод ψ.

Возвращаясь к определению `Adjunction` из Haskell, естественные преобразования φ и ψ заменяются полиморфными (на `a` и `b`) функциями `leftAdjunct` и `rightAdjunct`, соответственно. Функторы L и R обозначаются через `f` и `u`:

```
class (Functor f,
```

```
 Representable u) => Adjunction
     f u | f -> u, u -> f where
leftAdjunct  :: (f a -> b)
                -> (a -> u b)
rightAdjunct :: (a -> u b)
                -> (f a -> b)
```

Эквивалентность между формулировкой `unit` / `counit` и формулировкой `leftAdjunct` / `rightAdjunct` фиксируется такими отображениями:

```
unit           = leftAdjunct id
counit         = rightAdjunct id
leftAdjunct f  = fmap f . unit
rightAdjunct f = counit . fmap f
```

Очень полезно проследить за переводом категорного описания сопряжения в код Haskell. Я очень рекомендую это упражнение.

Теперь мы готовы объяснить, почему в Haskell правый сопряженный является автоматически представимым функтором. Это следует из того, что в первом приближении категорию типов Haskell можно рассматривать как категорию множеств.

Когда правая категория $\mathbf{D}$ есть $\mathbf{Set}$, правый сопряженный R является функтором от $\mathbf{C}$ к $\mathbf{Set}$. Такой функтор представим, если мы можем найти объект rep в $\mathbf{C}$ такой, что hom-функтор $\mathbf{C}(rep, -)$ естественно изоморфен R. Оказывается, что если R — правое сопряженное к некоторому функтору L от $\mathbf{Set}$ к C, такой объект всегда существует — это образ синглетона $()$ под L:

$$rep = L\,()$$

В самом деле, сопряжение говорит нам, что следующие два hom-множества естественно изоморфны:

$$\mathbf{C}(L\,(), c) \cong \mathbf{Set}((), R\,c)$$

Для заданного c правая часть представляет собой набор функций от одноэлементного множества $()$ к $R\,c$. Мы видели ранее, что каждая такая

функция выбирает один элемент из множества $R\,c$. Множество таких функций изоморфно множеству $R\,c$. Итак, мы имеем:

$$\mathbf{C}(L\,(), -) \cong R$$

что показывает — R действительно представим.

18.3 Произведение из сопряжения

Ранее мы ввели ряд понятий, используя универсальные конструкции. Многие из этих концепций, когда они определены глобально, легче выразить с помощью сопряжений. Простейший нетривиальный пример — произведение. Суть универсальной конструкции произведения — это способность факторизации любого кандидата на роль произведения, через универсальное произведение.

Точнее, произведение двух объектов a и b — это объект $(a \times b)$ (или `(a,b)` в нотации Haskell), снабженный двумя морфизмами fst и snd такими, что для любого другого претендента c, с морфизмами $p :: c \to a$ и $q :: c \to b$, существует единственный морфизм $m :: c \to (a, b)$, который факторизует p и q через fst и snd.

Как мы уже видели, на Haskell можно реализовать функцию `factorizer`, которая генерирует этот морфизм с помощью двух проекций:

```
factorizer :: (c -> a) -> (c -> b)
                       -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

Легко проверить, что выполнены условия факторизации:

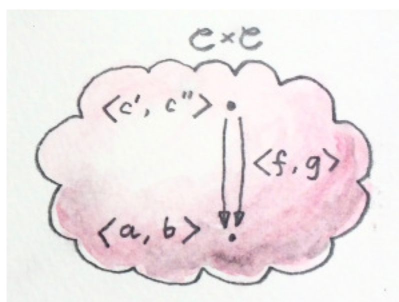
```
fst . factorizer p q = p
snd . factorizer p q = q
```

Мы имеем отображение, которое использует пару морфизмов `p` и `q` и порождает другой морфизм `m = factorizer p q`.

Как мы можем перевести это в отображение между двумя $\hom$-множествами, которое требуется для определения сопряжения? Хитрость заключается в том, чтобы выйти за пределы **Hask** и рассматривать пару морфизмов как единый морфизм в категории произведений.

Позвольте напомнить вам, что представляет собой категория произведений. Возьмем две произвольные категории $\mathbf{C}$ и $\mathbf{D}$. Объектами в категории произведений $\mathbf{C} \times \mathbf{D}$ являются пары объектов, один из $\mathbf{C}$, а другой из $\mathbf{D}$. Морфизмами являются пары морфизмов, один из $\mathbf{C}$, другой из $\mathbf{D}$.

Чтобы определить произведение в некоторой категории $\mathbf{C}$, необходимо начать с категории произведений $\mathbf{C} \times \mathbf{C}$. Пары морфизмов из $\mathbf{C}$ являются одиночными морфизмами в категории произведений $\mathbf{C} \times \mathbf{C}$.



Вначале может показаться немного странным, что мы используем категорию произведений для определения произведения. Это, однако, очень разные произведения. Но нам не требуется универсальная конструкция для определения категории произведений. Все, что нам нужно — это понятие пары объектов и пары морфизмов.

Тем не менее, пара объектов из $\mathbf{C}$ не является объектом в $\mathbf{C}$. Это объект в другой категории, $\mathbf{C} \times \mathbf{C}$. Мы можем записать пару формально как $\langle a, b \rangle$, где a и b являются объектами $\mathbf{C}$. Универсальная конструкция, с другой стороны, необходима, чтобы определить объект $(a \times b)$ (или `(a,b)` в Haskell), который является объектом в той же категории $\mathbf{C}$. Этот объект должен представлять пару $\langle a, b \rangle$ способом, определенным универсальной конструкцией. Он не всегда существует и, даже если он существует для некоторых объектов, то может не существовать для других пар объектов из $\mathbf{C}$.

Давайте теперь рассмотрим `factorizer` как отображение $\hom$-множеств. Первое $\hom$-множество находится в категории произведений $\mathbf{C} \times \mathbf{C}$, а

второе — в $\mathbf{C}$. Общий морфизм в $\mathbf{C} \times \mathbf{C}$ был бы парой морфизмов $\langle f, g \rangle$:

$$f :: c' \to a$$
$$g :: c'' \to b$$

с потенциально различными c' и c''. Но для определения произведения нас интересует специальный морфизм в $\mathbf{C} \times \mathbf{C}$, пара p и q, которые совместно используют один и тот же исходный объект c. Ничего страшного: в определении сопряжения источник левого hom-множества не является произвольным объектом — это результат действия левого функтора L, действующего на некоторый объект из правой категории. Функтор, отвечающий требованиям, легко угадать — это диагональный функтор Δ от $\mathbf{C}$ к $\mathbf{C} \times \mathbf{C}$, действующий на объекты следующим образом:

$$\Delta\, c = \langle c, c \rangle$$

Левая сторона hom-множества в нашем сопряжении должна иметь вид:

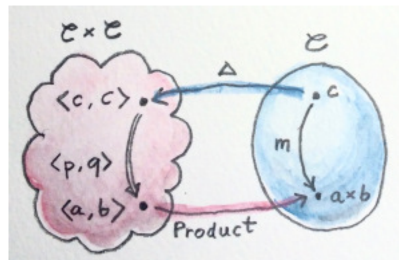
$$(\mathbf{C} \times \mathbf{C})(\Delta\, c, \langle a, b \rangle)$$

Это hom-множество в категории произведений. Его элементами являются пары морфизмов, которые мы считаем аргументами для `factorizer`:

$$(c \to a) \to (c \to b) \cdots$$

Правая сторона hom-множества обитает в $\mathbf{C}$, и располагается между исходным объектом c и результатом некоторого функтора R, действующего на целевой объект в $\mathbf{C} \times \mathbf{C}$. Это функтор, который отображает $\langle a, b \rangle$ к объекту произведения, $a \times b$. Мы осознаем этот элемент hom-множества как результат действия `factorizer`:

$$\cdots \to (c \to (a, b))$$



У нас еще нет полного сопряжения. Для этого необходимо, чтобы функция `factorizer` была обратима — мы строим изоморфизм между hom-множествами. Обратная к `factorizer` функция должна начинаться от морфизма m — морфизма от некоторого объекта c к объекту произведения $a \times b$. Другими словами, m должен быть элементом из:

$$\mathbf{C}(c, a \times b)$$

Обратный факторизатор должен отображать m к морфизму $\langle p, q \rangle$ из $\mathbf{C} \times \mathbf{C}$, который идет от $\langle c, c \rangle$ к $\langle a, b \rangle$; другими словами, это морфизм, который является элементом из:

$$(\mathbf{C} \times \mathbf{C})(\Delta\, c, \langle a, b \rangle)$$

Если это отображение существует, мы заключаем, что существует правый сопряженный к диагональному функтору. Этот функтор определяет произведение.

На Haskell мы всегда можем построить обратный к `factorizer`, сомпозируя `m` с, соответственно, `fst` и `snd`.

```
p = fst . m
q = snd . m
```

Чтобы завершить доказательство эквивалентности двух способов определения произведения, нам нужно еще показать, что отображение между hom-множествами естественно по a, b и c. Я оставляю это упражнение для подготовленного читателя.

Подведем итоги того, что мы сделали: категорное произведение может быть определено глобально как правое сопряженное к диагональному функтору:

$$(\mathbf{C} \times \mathbf{C})(\Delta\, c, \langle a, b \rangle) \cong \mathbf{C}(c, a \times b)$$

Здесь $a \times b$ — результат действия правого сопряженного функтора $\mathit{Product}$ на пару $\langle a, b \rangle$. Заметим, что любой функтор из $\mathbf{C} \times \mathbf{C}$ является бифунктором, поэтому $\mathit{Product}$ является бифунктором. На Haskell, бифунктор $\mathit{Product}$ записывается просто как `(,)`. Вы можете применить его к двум типам и получить тип их произведения, например:

```
(,) Int Bool ~ (Int, Bool)
```

18.4 Экспоненциал из сопряжения

Экспоненциал b^a, или функциональный объект $a \Rightarrow b$, можно определить, используя универсальную конструкцию. Эта конструкция, если она существует для всех пар объектов, может рассматриваться как сопряжение. Опять же, суть в том, чтобы сконцентрироваться на утверждении:

> *Для любого другого объекта* z *с морфизмом* $g :: z \times a \to b$
> *существует единственный морфизм* $h :: z \to (a \Rightarrow b)$

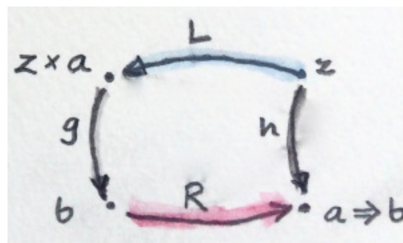
Это утверждение устанавливает отображение между hom-множествами.

В этом случае мы имеем дело с объектами в одной категории, поэтому два сопряженных функтора являются эндофункторами. Левый (эндо-) функтор L, действуя на объект z, порождает $z \times a$. Это функтор, соответствующий взятию произведения с некоторым фиксированным a.

Правый (эндо-) функтор R, действуя на b, порождает функциональный объект $a \Rightarrow b$ (или b^a). Опять же, a фиксируется. Сопряжение между этими двумя функторами часто записывают как:

$$_ \times a \dashv (-)^a$$

Отображение hom-множеств, лежащих в основе этого сопряжения, лучше всего рассмотреть, перерисовав диаграмму, которую мы использовали в универсальной конструкции функционального объекта в главе 9.



Обратите внимание, что морфизм $eval$[1] — это не что иное, как коединица этого сопряжения:

$$(a \Rightarrow b) \times a \to b$$

[1] См. главу 9 об универсальной конструкции.

где

$$(a \Rightarrow b) \times a = (L \circ R)\, b$$

Я уже упоминал, что универсальная конструкция определяет единственный объект, с точностью до изоморфизма. Вот почему у нас есть «конкретное» произведение и «конкретный» экспоненциал. Это свойство также сводится к сопряжению: если функтор имеет сопряженный, то этот сопряженный единственен с точностью до изоморфизма.

Упражнения

1. Выведите квадрат естественности для ψ, преобразования между двумя (контравариантными) функторами:

   $$a \to \mathbf{C}(L\, a, b)$$
   $$a \to \mathbf{D}(a, R\, b)$$

2. Выведите коединицу ε, начинающуюся с изоморфизма hom-множеств, во втором определении сопряжения.

3. Завершите доказательство эквивалентности двух определений сопряжения.

4. Покажите, что копроизведение может быть определено сопряжением. Начните с определения факторизатора для копроизведения.

5. Показать, что копроизведение является левым сопряженным к диагональному функтору.

6. Определить сопряжение между произведением и функциональным объектом на Haskell.

Глава 19

Свободные/забывающие сопряжения

19.1 Свободный моноид из сопряжения

Свободные конструкции — это мощная область приложений для сопряжений. Свободный функтор определяется как левый сопряженный к забывающему функтору. Забывающий функтор — это, как правило, довольно простой функтор, который забывает какую-то структуру. Например, ряд интересных категорий построен на множествах. Но категорные объекты, которые абстрагируют эти множества, не имеют внутренней структуры — у них нет элементов. Тем не менее, эти объекты часто содержат информацию о множествах, в том смысле, что имеется отображение — функтор — от заданной категории $\mathbf{C}$ к категории $\mathbf{Set}$. Множество, соответствующее некоторому объекту в $\mathbf{C}$, называется его основным множеством.

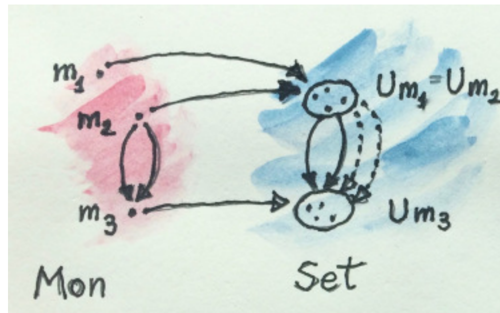
Моноиды — это такие объекты, у которых есть базовые множества — множества элементов. Существует забывающий функтор U от категории моноидов $\mathbf{Mon}$ к категории множеств, который отображает моноиды в их базовые множества. Он также отображает моноидные морфизмы (гомоморфизмы) к функциям между множествами.

Мне нравится думать, что $\mathbf{Mon}$ имеет раздвоение личности. С одной стороны, это набор множеств с умножением и единичными элементами. С другой стороны, это категория с невыразительными объектами,

структура которых закодирована в связывающих их морфизмах. Каждая множество-функция, сохраняющая умножение и единицу, порождает морфизм в **Mon**.

Что нужно иметь в виду:

- моноидов, сопоставляемых одному множеству, может быть много, и
- моноидных морфизмовможет быть меньше (или больше), чем функций между их базовыми множествами.



Моноиды m_1 и m_2 имеют одно и тоже базовое множество. Между базовыми множествами m_2 и m_3 существует больше функций, чем морфизмов.

Функтор F, который является левым сопряженным к забывающему функтору U, является свободным функтором, который строит свободные моноиды из своих образующих множеств. Сопряжение следует из свободной моноидной универсальной конструкции, которую мы обсуждали ранее[1].

В терминах hom-множеств, мы можем записать это сопряжение как:

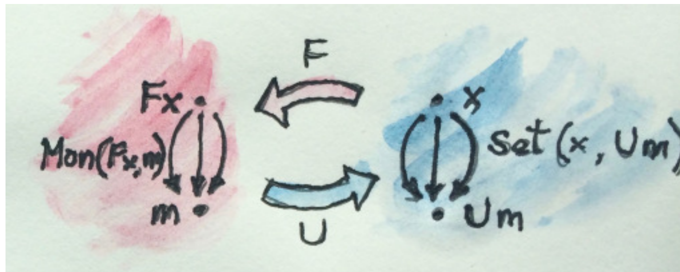
$$\mathbf{Mon}(F\,x, m) \cong \mathbf{Set}(x, U\,m)$$

Этот (естественный по x и m) изоморфизм означает, что:

- Для каждого моноидного гомоморфизма между свободным моноидом $F\,x$, порожденным x, и произвольным моноидом m, существует единственная функция, которая вставляет множество образующих x в лежащее в основе множество m. Это функция из $\mathbf{Set}(x, U\,m)$.

[1]См. главу 13 о свободных моноидах.

- Для каждой функции, которая вставляет x в лежащее в основе множество некоторого m, существует моноидный мономорфизм между свободным моноидом, порожденным x, и моноидом m (это морфизм, который мы обозначили через h в нашей универсальной конструкции).



Интуитивно, $F\,x$ является «максимальным» моноидом, который можно построить на основе x. Если бы мы могли заглянуть внутрь моноидов, мы увидели бы, что любой морфизм, принадлежащий $\mathbf{Mon}(F\,x, m)$, вставляет этот свободный моноид в какой-нибудь другой моноид m. Он делает это, возможно идентифицируя некоторые элементы. В частности, он встраивает образующие $F\,x$ (то есть элементы x) в m. Сопряжение демонстрирует, что вложение x, которое задается функцией из $\mathbf{Set}(x, U\,m)$ справа, однозначно определяет вложение моноидов слева и наоборот.

В Haskell списковая структура данных является свободным моноидом (с некоторыми оговорками: см. в блоге Дэн Доэль[2]). Тип списка `[a]` является свободным моноидом с типом `a`, представляющим множество образующих. Например, тип `[Char]` содержит единичный элемент — пустой список `[]` — и синглетоны, наподобие `['a']`, `['b']` — образующие свободного моноида. Остальное генерируется путем применения «произведения». Здесь произведение двух списков просто присоединяет один к другому. Присоединение ассоциативно и унитально (т.е. существует нейтральный элемент — здесь, пустой список). Свободный моноид, сгенерированный посредством `Char`, является не чем иным, как множеством всех строк символов из `Char`. В Haskell он обозначается `String`:

[2] http://comonad.com/reader/2015/free-monoids-in-haskell/

```
type String = [Char]
```

(`type` определяет синоним типа — другое имя для существующего типа).

Другим интересным примером является свободный моноид, построенный только из одного образующего. Это тип списка единиц, `[()]`. Его элементы — `[]`, `[()]`, `[(),()]` и т.д. Каждый такой список может быть описан одним натуральным числом — его длиной. Больше информации, закодированной в списке единиц, нет. Умножение двух таких списков приводит к созданию нового списка, длина которого представляет собой сумму длин его составляющих. Легко видеть, что тип `[()]` изоморфен аддитивному моноиду натуральных чисел (с нулем). Вот две функции, которые являются обратными друг к другу, свидетельствуя об этом изоморфизме:

```
toNat  :: [()] -> Int
toNat   = length
toLst  :: Int -> [()]
toLst n = replicate n ()
```

Для простоты я использовал тип `Int`, а не `Natural`, но идея одна и та же. Функция `replicate` создает список длины `n`, предварительно заполненный заданным значением — здесь, единицей `()`.

19.2 Немного интуиции

Ниже приведены некоторые неформальные аргументы, далеко не строгие, но они помогают в формировании интуиции.

Чтобы получить некоторое представление о свободных / забывающих сопряжениях, полезно задуматься о том, что функторы и функции по своей природе являются преобразователями, теряющими некоторую информацию исходных источников. Функторы могут сворачивать несколько объектов и морфизмов, функции могут группировать несколько элементов множества. Кроме того, их образ может охватывать только часть их кообласти.

«Типичное» $\hom$-множество из **Set** будет содержать целый спектр функций, начиная с тех, которые являются наименее «забывающими» (например, инъекциями или, возможно, изоморфизмами), и заканчивая постоянными функциями, которые сворачивают всю область в один элемент (если таковой имеется).

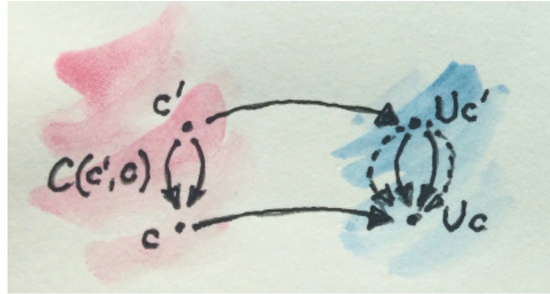
Я склонен думать так же и о морфизмах в произвольной категории. Это всего лишь ментальная модель, но она полезна, особенно при рассмотрении тех сопряжений, в которых одна из категорий есть **Set**.

Формально мы можем говорить только о морфизмах, которые являются обратимыми (изоморфизмами) или необратимыми. Последний вид морфизмов может быть и забывающим. Существует также понятие моно- и эпиморфизмов, которые обобщают идею инъективных (не коллапсирующих) и сюръективных (покрывающих всю совокупность) функций, но возможно существование морфизма, который одновременно является моно и эпи, но все еще необратим.

В «свободном $\dashv$ забывающем» сопряжении у нас находится более ограниченная категория $\mathbf{C}$ слева и менее ограниченная категория $\mathbf{D}$ справа. Морфизмов в $\mathbf{C}$ «меньше», они должны сохранить некоторую дополнительную структуру. В случае **Mon** они должны сохранять умножение и единицу. Морфизмы в $\mathbf{D}$ не должны сохранять столько структуры, поэтому их больше.

Когда мы применяем забывающий функтор U к объекту c из $\mathbf{C}$, мы думаем об этом как об выявлении «внутренней структуры» c. На самом деле, если $\mathbf{D}$ есть **Set**, мы рассматриваем функтор U как определяющий внутреннюю структуру c — лежащего в основе множества (в произвольной категории мы не можем говорить о строении объекта, кроме как через его связи с другими объектами, но здесь мы просто «размахиваем руками»).

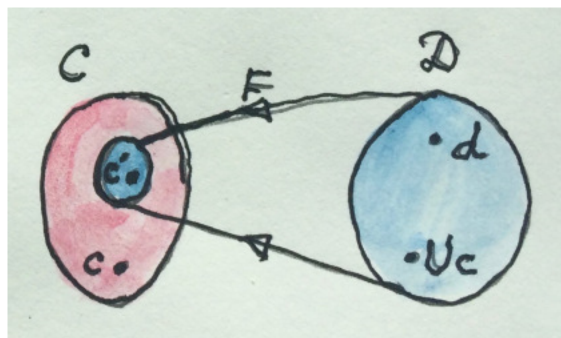
Если мы отображаем два объекта c' и c, используя U, мы ожидаем, что в общем случае отображение $\hom$-множества $\mathbf{C}(c', c)$ будет охватывать только подмножество $\mathbf{D}(U\,c', U\,c)$. Это потому, что морфизмы в $\mathbf{C}(c', c)$ должны сохранять дополнительную структуру, а в $\mathbf{D}(U\,c', U\,c)$ — нет.



Но так как сопряжение определяется как *изоморфизм* конкретных hom-множеств, мы должны быть очень разборчивыми с выбором c'. В сопряжении, c' выбирается не произвольно из $\mathbf{C}$, а из (предположительно меньшего) образа свободного функтора F:

$$\mathbf{C}(F\,d, c) \cong \mathbf{D}(d, U\,c)$$

Поэтому образ F должен состоять из объектов, которые имеют несколько морфизмов, идущих к произвольному c. В действительности, должно быть столько же сохраняющих структуру морфизмов от $F\,d$ к c, поскольку существуют не сохраняющие структуру морфизмы от d к $U\,c$. Это означает, что образ F должен состоять из существенно бесструктурных объектов (так что нет никакой структуры, которую можно было бы сохранить морфизмами). Такие «бесструктурные» объекты называются свободными объектами.



В примере моноида, свободный моноид не имеет никакой структуры, кроме того, что он генерируется законами единицы и ассоциативности. Кроме того, все умножения производят совершенно новые элементы.

В свободном моноиде $2 * 3$ это не 6, а новый элемент $[2, 3]$. Поскольку нет идентификации $[2, 3]$ с 6, морфизм от этого свободного моноида к любому другому моноиду m позволяет отображать их по-отдельности. Но также было бы хорошо, чтобы он отображал оба, $[2, 3]$ и 6 (их произведение), на один и тот же элемент m. Или, можно определить подобным образом $[2, 3]$ и 5 (их сумму) в аддитивном моноиде и т.д. Различные идентификации порождают разные моноиды.

Это приводит к еще одной интересной интуиции: свободные моноиды, вместо выполнения моноидальной операции, накапливают аргументы, которые были переданы ей. Вместо того, чтобы умножать 2 и 3, они запоминают 2 и 3 в списке. Преимущество этой схемы состоит в том, что нам не нужно указывать, какую моноидальную операцию мы будем использовать. Мы можем продолжать накапливать аргументы, и только в конце применять оператор к результату. И именно тогда мы можем выбрать, какой оператор применять. Мы можем добавить числа, или умножить их, или выполнить сложение по модулю 2, и так далее. Свободный моноид отделяет создание выражения от его обработки. Мы еще рассмотрим эту идею, когда будем говорить об алгебрах.

Эта интуиция обобщается на другие, более сложные свободные конструкции. Например, мы можем накапливать целые деревья выражений перед их преобразованием. Преимущество такого подхода состоит в том, что мы можем трансформировать такие деревья, чтобы сделать их обработку быстрее или менее потребляющей память. Это, например, делается при реализации матричного исчисления, в противовес немедленным вычислениям, которые приводят к большому количеству распределений временных массивов для хранения промежуточных результатов.

Упражнения

1. Рассмотрим свободный моноид, построенный из одноэлементного множества в качестве его образующего. Покажите, что существует взаимно однозначное соответствие между морфизмами от этого свободного моноида к любому моноиду m и функциями от одноэлементного множества к множеству, лежащему в основе m.

Глава 20

Монады: определение программиста

Программисты разработали целую мифологию вокруг монад, возможно одного из самых абстрактных и сложных понятий в программировании. Есть люди, которые «добираются до этого», и те, кто этого не делают. Для многих момент, когда они понимают концепцию монады, подобен мистическому переживанию. Монада абстрагирует сущность столь многих разнообразных конструкций, что мы просто не имеем хорошей аналогии для нее в повседневной жизни. Мы ходим на ощупь в темноте, как те слепые, которые трогают разные части слона, торжествующе восклицая: «Это веревка», «Это ствол дерева» или «Это репейник»!

Позвольте мне прямо сказать: весь мистицизм вокруг монады является результатом недоразумения. Монада — это очень простая концепция. Путаницу же вызывает разнообразие приложений монады.

В рамках исследования по этой теме я искал области применимости клейкой ленты. Вот небольшой перечень того, для чего вы можете ее использовать:

- для уплотнения труб,
- для закрепления промывателей CO_2 на борту Apollo 13,
- для обработки наростов,

- для исправления проблемы с iPhone 4 от Apple,
- при изготовлении платья к выпускному вечеру,
- при строительстве подвесного моста.

Теперь представьте, что вы не знаете, что такое клейкая лента, и пытаетесь это понять на основе приведенного списка. Удачи!

Поэтому я хотел бы добавить еще один элемент в коллекцию клише «монада подобна ...»: монада похожа на клейкую ленту. Ее приложения очень разнообразны, но принцип применения очень прост — она склеивает вещи. Точнее, она осуществляет композицию вещей.

Этим частично объясняются трудности, с которыми многие программисты, особенно приходящие из императивного фона, сталкиваются при попытке понять монаду. Проблема в том, что мы не привыкли думать о программировании в терминах композиции функций. Это понятно. Мы часто даем имена промежуточным значениям, а не передаем их непосредственно из функции в функцию. Мы также встраиваем короткие сегменты кода склеивания, а не абстрагируем их во вспомогательные функции. Вот реалистичная реализация функции подсчета длины вектора на языке C:

```
double vlen(double * v)
{
   double d = 0.0;
   int n;
   for (n = 0; n < 3; ++n)
      d += v[n] * v[n];
   return sqrt(d);
}
```

Сравните ее со стилизованной версией на Haskell, которая делает функциональную композицию явной:

```
vlen = sqrt . sum . fmap (flip (^) 2)
```

(Здесь, чтобы сделать содержимое более загадочным, я частично применил оператор возведения в степень (^), установив его второй аргумент равным 2.)

Я не утверждаю, что бесточечный стиль Haskell всегда лучше, просто функциональная композиция является основой всего, что мы делаем в программировании. И хотя мы эффективно составляем функции, Haskell делает все возможное, чтобы обеспечить синтаксис императивного стиля, названный do нотацией для монадической композиции. Мы увидим ее использование позже. Но сначала позвольте мне объяснить, почему нам нужна монадическая композиция в первую очередь.

20.1 Категория Клейсли

Ранее мы дошли до монады Writer, обогащая регулярные функции. Особое обогащение было выполнено путем объединения возвращаемых значений со строками или, более общо, с элементами моноида. Теперь мы можем признать, что такое обогащение является функтором:

```
newtype Writer w a = Writer (a, w)

instance Functor (Writer w) where
   fmap f (Writer (a, w)) = Writer (f a, w)
```

Впоследствии мы нашли способ составления обогащенных функций или стрелок Клейсли, которые являются функциями вида:

```
a -> Writer w b
```

Внутри композиции мы реализовали накопление журнала.

Теперь мы готовы к более общему определению категории Клейсли. Начнем с категории $\mathbf{C}$ и эндофунктора m. Соответствующая категория Клейсли $\mathbf{K}$ имеет те же объекты, что и $\mathbf{C}$, но ее морфизмы другие. Морфизм между двумя объектами a и b в $\mathbf{K}$ реализуется как морфизм:

$$a \to m\, b$$

в исходной категории **C**. Важно иметь в виду, что мы рассматриваем стрелку Клейсли в **K** как морфизм между a и b, а не между a и $m\,b$.

В нашем примере m был специализирован до `Writer w`, для некоторого фиксированного моноида `w`.

Стрелки Клейсли образуют категорию только в том случае, если мы можем определить для них правильную композицию. Если есть композиция, которая ассоциативна и имеет тождественную стрелку для каждого объекта, то функтор m называется *монадой*, а результирующая категория называется категорией Клейсли.

В Haskell, композиция Клейсли определяется с помощью оператора рыбы `>=>`, а тождественная стрелка является полиморфной функцией, называемой `return`. Вот определение монады с использованием композиции Клейсли:

```
class Monad m where
   (>=>)  :: (a -> m b) -> (b -> m c)
                        -> (a -> m c)
   return :: a -> m a
```

Имейте в виду, что существует много эквивалентных способов определения монады, и что приведенное не является первичным в экосистеме Haskell. Мне нравится ее концептуальная простота и интуиция, которые она предоставляет, но есть и другие определения, которые более удобны при программировании. Мы поговорим о них незамедлительно.

В данной формулировке законы монады очень легко выразить. Они не могут быть применены на Haskell, но их можно использовать для эквациональных рассуждений. Это просто стандартные законы композиции для категории Клейсли:

```
                      -- асоциативность
(f >=> g) >=> h       = f >=> (g >=> h)
                      -- левая единица
return    >=> f       = f
                      -- правая единица
f         >=> return = f
```

Такое определение также выражает то, чем на самом деле является монада: это способ составления обогащенных функций. Речь идет не о побочных эффектах или состоянии, а о композиции. Как мы увидим позже, обогащенные функции могут использоваться для выражения разнообразных эффектов или состояний, но это не то, для чего предназначена монада. Монада — это липкая клейкая лента, которая связывает один конец обогащенной функции с другим концом обогащенной функции.

Возвращаясь к нашему примеру с `Writer`: функции ведения журнала (стрелки Клейсли для функтора `Writer`) образуют категорию, потому что `Writer` является монадой:

```
instance Monoid w => Monad (Writer w) where
   f >=> g = \a ->
      let Writer (b, s)  = f a
          Writer (c, s') = g b
      in  Writer (c, s `mappend` s')
   return a = Writer (a, mempty)
```

Законы монады для `Writer w` удовлетворяются до тех пор, пока выполняются законы моноида для `w` (они также не могут быть применены в Haskell).

Есть полезная стрелка Клейсли, определенная для монады `Writer`, называемая `tell`. Ее единственная цель — добавить в журнал свой аргумент:

```
tell  :: w -> Writer w ()
tell s = Writer ((), s)
```

Мы будем использовать ее позже в качестве строительного блока для других монадических функций.

20.2 Анатомия >=>

При реализации оператора `>=>` для разных монад вы вскоре понимаете, что много кода повторяется и его легко можно сократить. Начнем с

того, что композиция Клейсли из двух функций должна возвращать функцию, поэтому ее реализация также может начинаться с лямбда-функции, принимающей аргумент типа a:

```
(>=>)  :: (a -> m b) -> (b -> m c)
                     -> (a -> m c)
f >=> g = \a -> ...
```

Единственное, что мы можем сделать с этим аргументом, — передать его в f:

```
f >=> g = \a -> let mb = f a
                in ...
```

В этот момент мы должны получить результат типа m c, имея в распоряжении объект типа m b и функцию g :: b -> m c. Давайте определим функцию, которая сделает это. Эта функция называется связыванием и обычно записывается в виде инфиксного оператора:

```
(>>=) :: m a -> (a -> m b) -> m b
```

Для каждой монады вместо определения оператора >=> мы можем определить связывание. На самом деле, в стандартном определении монады Haskell используется связывание:

```
class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   return :: a -> m a
```

Вот определение связывания для монады Writer:

```
(Writer (a, w)) >>=
   f = let Writer (b, w') = f a
       in  Writer (b, w `mappend` w')
```

Это действительно короче определения оператора `>=>`.

Возможно дальнейшее анатомирование связывания, пользуясь тем, что `m` является функтором. Мы можем использовать `fmap` для применения функции `a -> m b` к содержимому `m a`. Это превратит `a` в `m b`. Поэтому результат применения имеет тип `m (m b)`. Это не совсем то, что мы хотим — нам нужен результат типа `m b`, но мы близки к этому. Все, что нам нужно, это функция, которая сворачивает или сглаживает двойное применение `m`. Такая функция именуется `join`:

```
join :: m (m a) -> m a
```

Используя `join`, мы можем переписать связывание как:

```
ma >>= f = join (fmap f ma)
```

Это приводит нас к третьему варианту определения монады:

```
class Functor m => Monad m where
   join   :: m (m a) -> m a
   return :: a -> m a
```

Здесь мы явно потребовали, чтобы `m` был функтором. Мы не должны были этого делать в двух предыдущих определениях монады потому, что любой конструктор типа `m`, поддерживающий оператор `>=>` или связывание, автоматически является функтором. Например, можно определить `fmap` в терминах связывания и `return`:

```
fmap f ma = ma >>= \a -> return (f a)
```

Для полноты, вот `join` для монады `Writer`:

```
join   :: Monoid w => Writer w (Writer w a)
                              -> Writer w a
join (Writer ((Writer (a, w')), w))
        = Writer (a, w `mappend` w')
```

20.3 do-нотация

Один из способов написания кода с использованием монад состоит в том, чтобы работать со стрелками Клейсли — компонуя их с помощью оператора `>=>`. Этот способ программирования является обобщением бесточечного стиля. Бесточечный код является компактным и часто довольно элегантным. Вообще, хотя это может быть трудно понять, гранича при этом с таинственностью. Именно поэтому большинство программистов предпочитают давать имена функциональным аргументам и промежуточным значениям.

Когда речь идет о монадах, это означает преимущество оператора связывания над оператором `>=>`. Связывание использует монадическое значение и возвращает монадическое значение. Программист может предпочесть давать имена этим значениям. Но вряд ли это улучшение. На самом деле мы хотим притвориться, что имеем дело с обычными значениями, а не с монадическими контейнерами, инкапсулирующими их. Так работает императивный код — побочные эффекты, такие как обновление глобального журнала, в основном скрыты от просмотра. И это то, что нотация `do` эмулирует в Haskell.

Вы могли бы задаться вопросом, зачем вообще использовать монады? Если мы хотим сделать невидимыми побочные эффекты, почему бы не придерживаться императивного стиля? Ответ заключается в том, что монада дает нам гораздо лучший контроль над побочными эффектами. Например, журнал в монаде `Writer` передается от функции к функции и никогда не подвергается глобальному воздействию. Нет возможности исказить журнал или создать расу данных. Кроме того, монадический код четко разграничен и отделен от остальной части программы.

`do`-нотация — это просто синтаксический сахар для монадической композиции. Она очень похожа на императивный код, но напрямую преобразуется в последовательность связываний и лямбда-выражений.

Например, рассмотрим функцию `process`, которая использовалась ранее при иллюстрации композицию стрелок Клейсли в монаде `Writer`. Используя текущие определения, ее можно переписать так:

```
process :: String -> Writer String [String]
process  = upCase >=> toWords
```

Эта функция переводит все символы входной строки в верхний регистр и разбивает их на слова, все время фиксируя эти действия в журнале.

В do-нотации это будет выглядеть так:

```
process s = do
            upStr <- upCase s
            toWords upStr
```

Здесь upStr — это просто String, хотя upCase вызывает Writer:

```
upCase  :: String -> Writer String String
upCase s = Writer (map toUpper s, "upCase ")
```

Это происходит из-за того, что блок do удаляет синтаксический сахар (десахаризирует), компилируя содержимое к виду:

```
process s =
   upCase s >>= \ upStr -> toWords upStr
```

Монадический результат upCase привязан к лямбда-выражению, которое принимает String. Это имя этой строки, которое отображается в блоке do. При чтении строки:

```
upStr <- upCase s
```

мы говорим, что upStr *получает* результат upCase s.

Псевдо-императивный стиль еще более выражен, когда мы встраиваем строку toWords в журнал. Мы заменяем его вызовом tell, который записывает в журнал строку "toWords ", после чего следует вызов return с результатом разделения строки upStr, используя words. Обратите внимание, что words — это регулярная функция, работающая над строками.

```
process s = do
            upStr <- upCase s
            tell "toWords "
            return (words upStr)
```

Здесь каждая строка в блоке do вводит новое вложенное связывание в коде десахаризации:

```
process s =
   upCase s >>= \upStr ->
      tell "toWords " >>= \() ->
         return (words upStr)
```

Как правило, do-блоки состоят из строк (или подблоков), которые либо используют стрелку влево, чтобы ввести новые имена, которые затем доступны в остальной части кода, либо выполняются исключительно для побочных эффектов. Операторы связывания неявны между строками кода. Кстати, на Haskell форматирование в блоках do можно заменить запятыми и точками с запятой. Это дает основание для того, чтобы охарактеризовать монаду как способ перегрузки точки с запятой.

Обратите внимание, что вложение лямбда-выражений и операторов связывания при десахаризации do нотации влияет на выполнение остальной части блока do по результатам каждой строки. Это свойство может использоваться для введения сложных управляющих структур, например, для имитации исключений.

Интересно, что эквивалент do нотации нашел свое применение в императивных языках, в частности в C++. Я говорю о возобновляемых функциях или сопрограммах. Не секрет, что фьючерсы на C++ образуют монаду[1] . Примером является монада продолжения, о которой мы поговорим вкратце. Проблема с продолжениями заключается в том, что их очень сложно сконструировать. В Haskell мы используем обозначение do, чтобы превратить спагетти «мой обработчик вызовет вашего обработчика» во что-то похожее на последовательный код. Возобновляемые функции делают такое же преобразование возможным в C++. И тот же механизм может быть применен для превращения спагетти вложенных циклов[2] в списки распознавания или *образующие*, которые по сути являются обозначением для монады списка. Без унифицирующей абстракции монады каждая из этих проблем обычно решается путем предоставления пользовательских расширений для языка. В Haskell все это осуществляется через библиотеки.

[1] https://bartoszmilewski.com/2014/02/26/
[2] https://bartoszmilewski.com/2014/04/21/getting-lazy-with-c/

Глава 21

Монады и эффекты

Теперь, когда мы знаем, для чего предназначена монада — она позволяет нам составлять обогащенные функции, действительно интересный вопрос — почему в функциональном программировании так важны обогащенные функции. Мы уже видели один пример — монаду `Writer`, где обогащение позволяет нам создавать и накапливать журнал для нескольких вызовов функций. Проблема, которая в противном случае была бы решена с использованием нечистых функций (например, путем доступа и изменения некоторого глобального состояния), была закрыта с помощью чистых функций.

21.1 Проблемы

Вот краткий список подобных проблем, скопированный с основополагающей статьи Эудженио Моджи[1], которые традиционно решаются путем отказа от чистоты функций.

- Частичность: вычисления, которые могут не завершиться
- Недетерминизм: вычисления, которые могут возвращать много результатов

[1] https://www.cs.cmu.edu/~crary/819-f09/Moggi91.pdf

- Побочные эффекты: вычисления, которые получают доступ / изменяют состояние
  - состояние только для чтения или среда
  - состояние только для записи или журнал
  - состояние чтения / записи
- Исключения: частичные функции, которые могут завершиться аварийно
- Продолжения: возможность сохранить состояние программы, а затем восстановить ее по требованию
- Интерактивный ввод
- Интерактивный вывод

Оказывается все эти проблемы могут быть решены с использованием одного и того же хитроумного трюка: перехода к обогащенным функциям. Конечно, в каждом случае обогащение будут совершенно разным.

Вы должны понимать, что на данном этапе нет требования, чтобы обогащение было монадическим. Только когда мы настаиваем на композиции — способности разложить одну обогащеннйю функцию на меньшие обогащенные функции — нам нужна монада. Опять же, поскольку каждое из обогащений особенное, монадическая композиция будет реализована по-разному, но общая картина сохраняется. Очень простой пример: композиция, которая ассоциативна и снабжена тождественным морфизмом.

Следующий раздел очень важен для примеров на Haskell. Можете просто просматреть или даже пропустить его, если хотите вернуться к теории категорий, или если вы уже знакомы с реализациями монад на Haskell.

21.2 Решение

Сначала давайте проанализируем, как мы использовали монаду `Writer`. Мы начали с чистой функции, которая выполняла определенную задачу

— на заданных аргументах она создавала определенный вывод. Мы заменили эту функцию другой функцией, которая обогатила первоначальный вывод, соедив его со строкой. Это было нашим решением проблемы журналирования.

Мы не могли остановиться на этом, потому что, в общем, мы не хотим иметь дело с монолитными решениями. Нам нужно было разложить одну функцию, производящую записи для журнала, на более мелкие функции с тем же действием. Композиция этих небольших функций и привела нас к понятию монады.

Что действительно удивительно, так это то, что один и тот же образец обогащения возвращаемых типов функций работает для большого числа проблем, которые обычно требуют отказа от чистоты. Давайте рассмотрим наш список проблем, по очереди, и для каждой из них определим соответствующее обогащение.

Частичность

Мы модифицируем возвращаемый тип каждой функции, которая может не завершиться, превращая его в «поднятый» тип — тип, который содержит все значения исходного типа плюс специальное «нижнее» значение `_|_`. Например, тип `Bool`, как множество, содержит два элемента: `True` и `False`. Поднятый `Bool` будет содержать три элемента. Функции, которые возвращают поднятый `Bool`, могут выдавать `True` или `False`, или выполняться без завершения.

Забавно, что на ленивом языке, таком как Haskell, не завершающаяся функция может фактически вернуть значение, и это значение может быть передано следующей функции. Это специальное значение мы называем дном. Пока это значение явно не требуется (например, для соответствия шаблону или вывода в качестве результата), оно может быть передано без остановки выполнения программы. Поскольку каждая функция Haskell может потенциально не завершаться, предполагается, что все типы в Haskell должны быть подняты. Вот почему мы часто говорим о категории **Hask** (поднятых) типов и функций Haskell, а не о более простой категории **Set**. Неясно, однако, насколько **Hask** является настоящей категорией (см. сообщение Андрэ Бауэра[2]).

[2] http://math.andrej.com/2016/08/06/hask-is-not-a-category/

Недетерминизм

Если функция может возвращать много различных результатов, она также может возвращать их все сразу. Семантически недетерминированная функция эквивалентна функции, которая возвращает список результатов. Это имеет большой смысл на ленивом языке, сборщике мусора. Например, если все, что вам нужно, — это одно значение, вы можете просто взять первый элемент в списке, а остальная часть списка никогда не будет использована. Если вам нужно случайное значение, используйте генератор случайных чисел для выбора n-го элемента списка. Ленивость позволяет возвращать даже бесконечный список результатов.

В монаде списка — реализации недетерминированных вычислений в Haskell — `join` реализована как `concat`. Напомню, что `join` должна сглаживать контейнер контейнеров — `concat` превращает список списков в один список. `return` создает одноэлементный список:

```
instance Monad [] where
   join     = concat
   return x = [x]
```

Оператор связывания для монады списка задается общей формулой: `fmap`, за которой следует `join`, которая в этом случае дает:

```
as >>= k = concat (fmap k as)
```

Здесь, функция `k`, которая производит список, применяется ко всем элементам списка `as`. Результатом является список списков, который сглаживается с помощью `concat`.

С точки зрения программиста работа со списком проще, чем, например, вызов недетерминированной функции в цикле или реализация функции, возвращающей итератор (хотя, в современном C++ [3], возвращение ленивого ряда почти эквивалентно возврату списка в Haskell).

Хорошим примером творческого использования недетерминированности является игровое программирование. Например, когда компьютер играет в шахматы против человека, он не может предсказать следующий

[3] http://ericniebler.com/2014/04/27/range-comprehensions/

ход противника. Однако он может генерировать список всех возможных ходов и анализировать их один за другим. Аналогично, недетерминированный парсер может генерировать список всех возможных разборов заданного выражения.

Хотя мы можем интерпретировать функции, возвращающие списки как недетерминированные, приложения монады списка намного шире. Это потому, что соединение вместе вычислений, которые производят списки, является идеальной функциональной заменой итеративных конструкций — циклов, которые используются в императивном программировании. Одиночный цикл часто можно переписать с помощью `fmap`, который применяет тело цикла к каждому элементу списка. Нотация `do` в монаде списка может использоваться для замены сложных вложенных циклов.

Мой любимый пример — программа, которая генерирует пифагоровы тройки — тройки положительных целых чисел, которые выражают длины сторон прямоугольных треугольников.

```
triples = do
          z <- [1..]
          x <- [1..z]
          y <- [x..z]
          guard (x^2 + y^2 == z^2)
          return (x, y, z)
```

Первая строка сообщает, что `z` получает элемент из бесконечного списка положительных чисел `[1..]`. Тогда `x` получает элемент из (конечного) списка `[1..z]` чисел между `1` и `z`. Наконец, `y` получает элемент из списка чисел между `x` и `z`. В нашем распоряжении есть три числа таких, что `1 <= x <= y <=z`. Функция `guard` принимает выражение типа `Bool` и возвращает список единиц:

```
guard      :: Bool -> [()]
guard True  = [()]
guard False = []
```

Эта функция (которая является членом более крупного класса, называемого `MonadPlus`) используется здесь для фильтрации непифагоровых

троек. В самом деле, если вы посмотрите на реализацию связывания (или оператора `>>`), вы заметите, что при задании пустого списка он порождает пустой список. С другой стороны, если задан непустой список (здесь одноэлементный список, содержащий единицу `[()]`), связывание будет вызывать продолжение, здесь `return(x,y,z)`, которое создает одноэлементный список с прошедшими проверку пифагоровыми тройками. Все эти одноэлементные списки будут объединены вложенными связями для получения окончательного (бесконечного) результата. Конечно, вызывающий `triples` никогда не сможет извлечь весь список, но это не имеет значения, потому что Haskell ленив.

Проблема, которая обычно требовала бы набора из трех вложенных циклов, была значительно упрощена с помощью монады списка и нотации `do`. Как будто этого было недостаточно, Haskell упростит этот код еще больше, используя построитель списка:

```
triples = [(x, y, z) | z <- [1..]
                     , x <- [1..z]
                     , y <- [x..z]
                     , x^2 + y^2 == z^2]
```

Это просто дополнительный синтаксический сахар для монады списка (строго говоря, `MonadPlus`).

Вы можете увидеть подобные конструкции на других функциональных или императивных языках под видом генераторов и сопрограмм.

Состояние только для чтения

Функция, имеющая доступ только по чтению к некоторому внешнему состоянию или среде, всегда может быть заменена функцией, которая принимает эту среду в качестве дополнительного аргумента. Чистая функция `(a,e) -> b` (где `e` — тип среды) не выглядит, на первый взгляд, как стрелка Клейсли. Но как только мы каррируем ее до `a -> (e -> b)`, мы узнаем обогащение, нашего старого знакомого, в виде считывающего функтора:

```
newtype Reader e a = Reader (e -> a)
```

Вы можете интерпретировать функцию, возвращающую `Reader`, как производящую мини-исполнение: действие, которое при заданной среде приводит к желаемому результату. Для выполнения такого действия имеется вспомогательная функция `runReader`:

```
runReader              :: Reader e a -> e -> a
runReader (Reader f) e = f e
```

Она может привести к различным результатам для различных значений среды.

Заметьте, что и функция, возвращающая `Reader`, и сама операция `Reader` являются чистыми.

Чтобы реализовать связывание для монады `Reader`, сначала обратите внимание на то, что вам нужно создать функцию, которая принимает среду `e` и создает `b`:

```
ra >>= k = Reader (\e -> ... )
```

Внутри лямбда-выражения мы можем выполнить действие `ra` для создания `a`:

```
ra >>= k = Reader (\e ->
     let a = runReader ra e
     in ... )
```

Затем мы можем передать `a` в продолжение `k`, чтобы получить новое действие `rb`:

```
ra >>= k = Reader (\e ->
     let a  = runReader ra e
         rb = k a
     in ... )
```

Наконец, мы можем запустить действие `rb` со средой `e`:

```
ra >>= k = Reader (\e ->
      let a  = runReader ra e
          rb = k a
      in runReader rb e)
```

Чтобы реализовать `return`, мы создаем действие, которое игнорирует среду и возвращает неизмененное значение.

Собирая все это вместе, после нескольких упрощений, мы получим следующее определение:

```
instance Monad (Reader e) where
   ra >>= k = Reader (\e -> runReader
                  (k (runReader ra e)) e)
   return x = Reader (\e -> x)
```

Состояние только для записи

Это только пример начала журнализации. Обогащение обеспечивается функтором `Writer`:

```
newtype Writer w a = Writer (a, w)
```

Для полноты имеется также тривиальный помощник `runWriter`, который распаковывает конструктор данных:

```
runWriter  :: Writer w a -> (a, w)
runWriter (Writer (a, w)) = (a, w)
```

Как мы видели ранее, для того, чтобы сделать `Writer` компонуемым, `w` должен быть моноидом. Вот пример монады для `Writer`, определенной в терминах оператора связывания:

```
instance (Monoid w) => Monad (Writer w) where
   (Writer (a, w)) >>= k =
            let (a', w') = runWriter (k a)
            in Writer (a', w `mappend` w')
   return a = Writer (a, mempty)
```

Состояние

Функции, которые имеют доступ на чтение/запись к состоянию, сочетают приемы от `Reader` и `Writer`. Вы можете думать о них как о чистых функциях, которые принимают состояние в качестве дополнительного аргумента и производят в качестве результата пару значение/состояние: `(a,s) -> (b,s)`. После каррирования, мы получим их в виде стрелок Клейсли `a -> (s -> (b,s))`, с обогащением, растворенном в функторе `State`:

```
newtype State s a = State (s -> (a, s))
```

Опять же, мы можем рассматривать стрелку Клейсли как возвращающую действие, которое может быть выполнено с помощью вспомогательной функции:

```
runState  :: State s a -> s -> (a, s)
runState (State f) s = f s
```

Различные начальные состояния могут порождать не только разные результаты, но также и разные конечные состояния.

Реализация связывания для монады `State` очень похожа на реализацию для монады `Reader`, за исключением того, что необходимо поддерживать правильное состояние на каждом шаге:

```
sa >>= k = State (\s ->
   let (a, s') = runState sa s
       sb      = k a
   in runState sb s')
```

Вот полное определение:

```
instance Monad (State s) where
   sa >>= k = State
      (\s -> let (a, s') = runState sa s
             in runState (k a) s')
   return a = State (\s -> (a, s))
```

Имеются также две вспомогательные стрелки Клейсли, которые могут использоваться для управления состоянием. Одна из них получает состояние для проверки:

```
get :: State s s
get  = State (\s -> (s, s))
```

а другая заменяет его совершенно новым состоянием:

```
put   :: s -> State s ()
put s' = State (\s -> ((), s'))
```

Исключения

Императивная функция, которая сбрасывает исключение, на самом деле является частичной функцией — это функция, которая не определена для некоторых значений ее аргументов. Простейшая реализация исключений в терминах чистых тотальных функций использует функтор `Maybe`. Частичная функция расширяется до тотальной функции, которая возвращает `Just`, когда это имеет смысл, и `Nothing`, когда это не так. Если мы хотим также вернуть некоторую информацию о причине сбоя, мы можем вместо этого использовать функтор `Either` (с фиксированным первым типом, например, с `String`).

Вот экземпляр `Monad` для `Maybe`:

```
instance Monad Maybe where
   Nothing >>= k = Nothing
   Just a  >>= k = k a
   return a  = Just a
```

Обратите внимание, что монадическая композиция для `Maybe` корректно замыкает вычисление (продолжение `k` никогда не вызывается) при обнаружении ошибки. Такого поведения мы и ожидаем от исключений.

Продолжения

Это ситуация «Не звоните нам, мы вам сами позвоним!», с которой вы можете столкнуться после собеседования. Вместо получения прямого ответа вы должны предоставить обработчик, функцию, которая будет вызвана с результатом в качестве параметра. Этот стиль программирования особенно полезен, когда результат неизвестен во время вызова, потому что, например, он оценивается другим потоком или доставляется с удаленного веб-сайта. Стрелка Клейсли в этом случае возвращает функцию, которая принимает обработчик, представляющий «остальную часть вычисления»:

```
data Cont r a = Cont ((a -> r) -> r)
```

Обработчик `a -> r`, когда он в конце концов вызывается, выдает результат типа `r`, и этот результат возвращается в конце вычисления. Тип продолжения параметризуется типом результата (на практике это часто является некоторым видом индикатора статуса).

Существует также вспомогательная функция для выполнения действия, возвращаемого стрелкой Клейсли. Она принимает обработчик и передает его в продолжение:

```
runCont  :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h
```

Композиция продолжений, как известно, сложна, поэтому ее обработка с использованием монады и, в частности, нотации `do`, имеет огромное преимущество.

Давайте выясним реализацию связывания. Сначала посмотрим на сокращенную сигнатуру:

```
(>>=) :: ((a -> r) -> r) ->
         (a -> (b -> r) -> r) ->
         ((b -> r) -> r)
```

Наша цель состоит в том, чтобы создать функцию, которая принимает обработчик `(b -> r)` и выдает результат `r`. Итак, вот наша отправная точка:

```
ka >>= kab = Cont (\hb -> ... )
```

Внутри лямбда-выражения мы хотим вызвать функцию `ka` с соответствующим обработчиком, который представляет остальную часть вычисления. Мы реализуем этот обработчик тоже как лямбда-выражение:

```
runCont ka (\a -> ... )
```

В этом случае остальная часть вычисления включает в себя сначала вызов `kab` с `a`, а затем передачу `hb` в результирующее действие `kb`:

```
runCont ka (\a -> let kb = kab a
                  in runCont kb hb)
```

Как видите, продолжения сложены "наизнанку". Последний обработчик `hb` вызывается из самого внутреннего уровня вычисления. Вот полный код:

```
instance Monad (Cont r) where
   ka >>= kab = Cont
      (\hb -> runCont ka
         (\a -> runCont (kab a) hb))
   return a = Cont (\ha -> ha a)
```

Интерактивный ввод

Это самая сложная проблема и источник путаницы. Ясно, что функция, подобная `getChar`, если она возвращает символ, напечатанный на клавиатуре, не может быть чистой. Но что, если она возвращает символ внутри контейнера? Пока нет способа извлечь символ из этого контейнера, можно утверждать, что функция чиста. Каждый раз, когда вы

вызываете getChar, она возвращает точно такой же контейнер. Концептуально этот контейнер будет содержать суперпозицию всех возможных символов.

Если вы знакомы с квантовой механикой, вам не составит труда понять эту аналогию. Это как коробка с кошкой Шредингера внутри, за исключением того, что нет способа открыть или заглянуть внутрь коробки. Коробка определяется с помощью специального встроенного функтора IO. В нашем примере getChar может быть объявлена как стрелка Клейсли:

```
getChar :: () -> IO Char
```

(Фактически, поскольку функция от единичного типа эквивалентна выбору значения возвращаемого типа, объявление getChar упрощается до getChar :: IO Char.)

Будучи функтором, IO позволяет вам манипулировать его содержимым с помощью fmap. И, как функтор, он может хранить содержимое любого типа, а не только символ. Реальная полезность этого подхода обнаруживается, когда вы считаете, что IO в Haskell является монадой. Это означает, что вы можете создавать стрелки Клейсли, которые создают объекты ввода-вывода.

Вы можете думать, что композиция Клейсли позволит вам заглянуть в содержимое объекта IO (таким образом, «сворачивая волновую функцию», если бы мы продолжили квантовую аналогию). В самом деле, вы можете составить getChar с другой стрелкой Клейсли, которая принимает символ и, скажем, преобразует его в целое число. Загвоздка в том, что эта вторая стрелка Клейсли могла бы вернуть это целое число только как IO Int. Опять же, вы получите суперпозицию всех возможных целых чисел. И так далее. Кошка Шредингера никогда не выходит из мешка. Как только вы окажетесь внутри монады IO, выхода из нее не будет. Для монады IO нет эквивалента runState или runReader. Нет runIO!

Итак, что вы можете сделать с результатом стрелки Клейсли, объекта ввода-вывода, кроме того, что скомпоновать его с другой стрелкой Клейсли? Ну, вы можете вернуть его из main. На Haskell main имеет сигнатуру:

```
main :: IO ()
```

и вы можете думать о ней как о стрелке Клейсли:

```
main :: () -> IO ()
```

С этой точки зрения, программа на Haskell — это просто одна большая стрелка Клейсли в монаде `IO`. Вы можете скомпоновать ее из меньших стрелок Клейсли, используя монадическую композицию. От исполняющей системы зависит, что можно сделать с полученным объектом ввода-вывода (также называемым действием `IO`).

Обратите внимание, что стрелка сама по себе является чистой функцией — это чистые функции на всем пути вглубь. Грязная работа отводится системе. Когда она наконец выполнит действие `IO`, возвращаемое из `main`, она выполняет всевозможные неприятные вещи, такие как чтение пользовательского ввода, изменение файлов, печать неприятных сообщений, форматирование диска и т.д. Программа на Haskell никогда не «пачкает свои руки» (ладно, за исключением случаев, когда она вызывает `unsafePerformIO`, но это другая история).

Конечно, из-за ленивости Haskell, `main` возвращает свое значение почти сразу и грязная работа начинается немедленно. Во время выполнения действия `IO` результаты чистых вычислений запрашиваются и осуществляются по требованию. Таким образом, в действительности выполнение программы представляет собой чередование чистого (Haskell) и грязного (системного) кода.

Существует альтернативная интерпретация монады `IO`, которая является еще более странной, но имеет смысл в качестве математической модели. Она рассматривает весь универсум (Вселенную) как объект в программе. Обратите внимание: концептуально императивная модель рассматривает универсум как внешний глобальный объект, поэтому процедуры, которые выполняют ввод/вывод, имеют побочные эффекты благодаря взаимодействию с этим объектом. Они могут читать и изменять состояние универсума.

Мы уже знаем, как обращаться с состоянием в функциональном программировании — мы используем монаду состояния. Однако, в отличие от простого состояния, состояние универсума не может быть легко

описано с использованием стандартных структур данных. Но мы и не должны этого делать, если никогда напрямую не взаимодействуем с ним. Достаточно предположить, что существует тип `RealWorld` и каким-то чудом всеобъемлющей инженерии среда выполнения может предоставить объект этого типа. Действие `IO` — это всего лишь функция:

```
type IO a = RealWorld -> (a, RealWorld)
```

Или, в теминах монады `State`:

```
type IO = State RealWorld
```

Однако, `>=>` и `return` для монады `IO` должны быть встроены в язык.

Интерактивный вывод

Для инкапсуляции интерактивного вывода используется та же самая монада `IO`. Предполагается, что `RealWorld` будет содержать все устройства вывода. Вы можете удивиться, почему мы не можем просто вызывать функции вывода из Haskell и делать вид, что они ничего не делают. Например, почему мы имеем:

```
putStr :: String -> IO ()
```

а не более простое

```
putStr :: String -> ()
```

Две причины: Haskell ленив, поэтому он никогда не будет вызывать функцию, результат которой — здесь, единичный объект — ни для чего не используется. И, даже если бы он не был ленивым, он все же мог бы свободно изменять порядок подобных вызовов и таким образом искажать результат. Единственным способом, заставляющим последовательно выполнять две функции в Haskell, является зависимость по данным. Входные значения для одной функции должны зависеть от результата, выдаваемого другой функцией. После пропускания `RealWorld` между действиями `IO` обеспечивается последовательность действий.

Концептуально, в программе:

```
main :: IO ()
main = do
       putStr "Hello "
       putStr "World!"
```

действие, которое печатает "World!", получает в качестве входных данных универсум, в котором "Hello " уже отображено на экране. Оно выводит новый универсум, на экране отображается "Hello World!".

21.3 Заключение

Конечно, я только прикоснулся к поверхности монадического программирования. Монады не только выполняют с чистыми функциями то, что обычно делается с побочными эффектами в императивном программировании, но они также делают это с высокой степенью контроля и безопасности типов. Тем не менее, они не лишены недостатков. Главная жалоба на монады заключается в том, что их трудно компоновать друг с другом. Конечно, вы можете комбинировать большинство основных монад, используя библиотеку трансформеров монад. Относительно просто создать стек монады, который сочетает в себе, скажем, состояние с исключениями, но нет рецепта для объединения произвольных монад друг с другом.

Глава 22

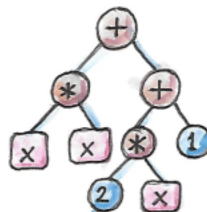
Монады с категорной точки зрения

Если вы рассказываете о монадах программисту, то, вероятно, в конечном итоге будете говорить об эффектах. Для математиков монады относятся к алгебрам. Мы поговорим об алгебрах позже — они играют важную роль в программировании, — но сначала я хотел бы дать вам немного интуитивных соображений об их отношении к монадам. На данный момент, это немного ругательный аргумент, но потерпите.

Алгебра — это создание, манипулирование и оценка выражений. Выражения строятся с помощью операторов. Рассмотрим простое выражение:

$$x^2 + 2x + 1$$

Подобное выражение формируется с использованием переменных типа x и констант, подобных 1 или 2, связанных операторами типа «сложить» или «умножить». Как программисты, мы часто думаем о выражениях как о деревьях.



Деревья являются контейнерами, поэтому, в более общем плане, выражение представляет собой контейнер для хранения переменных. В теории категорий мы представляем контейнеры как эндофункторы. Если мы присвоим тип a переменной x, наше выражение будет иметь тип $m\, a$, где m — эндофунктор, который строит деревья выражений (нетривиальные выражения ветвления обычно создаются с использованием рекурсивно определенных эндофункторов).

Какова наиболее распространенная операция, которая может быть выполнена над выражением? Это замена: замена переменных выражениями. Например, в нашем примере мы могли бы заменить x на $y - 1$, чтобы получить:

$$(y-1)^2 + 2(y-1) + 1$$

Вот что получилось: мы взяли выражение типа $m\, a$ и применили преобразование типа $a \to m\, b$ (b представляет тип y). Результатом является выражение типа $m\, b$. Позвольте мне представить это так:

$$m\, a \to (a \to m\, b) \to m\, b$$

Да, это сигнатура монадического связывания.

Это было краткое обсуждение мотивации. Теперь рассмотрим математику монады. Математики используют более разнообразные обозначения, по сравнению с программистами. Они предпочитают использовать букву T для эндофунктора, греческие буквы: μ для `join` и η для `return`. И `join` и `return` являются полиморфными функциями, поэтому мы можем догадаться, что они соответствуют естественным преобразованиям.

Поэтому в теории категорий монада определяется как эндофунктор T, снабженный парой естественных преобразований μ и η.

μ — это естественное преобразование от квадрата T^2 функтора T обратно к T. Квадрат — это просто функтор, составленный сам с собой, $T \circ T$ (мы можем возводить в квадрат только эндофункторы).

$$\mu :: T^2 \to T$$

Компонента этого естественного преобразования при объекте a является морфизмом:

$$\mu_a :: T\,(T\, a) \to T\, a$$

который в **Hask** напрямую преобразуется в наше определение `join`.

η — естественное преобразование между тождественным функтором I и T:

$$\eta :: I \to T$$

Учитывая, что действие I на объект a есть просто a, компонента η при a задается морфизмом:

$$\eta_a :: a \to T\,a$$

что напрямую переводится в наше определение `return`.

Эти естественные преобразования должны удовлетворять некоторым дополнительным законам. Один из способов рассмотреть их состоит в том, что эти законы позволяют определить категорию Клейсли для эндофунктора T. Напомним, что стрелка Клейсли между a и b определяется как морфизм $a \to T\,b$. Композицию двух таких стрелок (я обозначаю ее символом $\circ$ с индексом T) можно реализовать, используя μ:

$$g \circ_T f = \mu_c \circ (T\,g) \circ f$$

где

$$f :: a \to T\,b$$
$$g :: b \to T\,c$$

Здесь T, являющийся функтором, можно применить к морфизму g. Может быть, проще распознать эту формулу в нотации Haskell:

```
f >=> g = join . fmap g . f
```

или, в компонентах:

```
(f >=> g) a = join (fmap g (f a))
```

В терминах алгебраической интерпретации мы просто составляем две последовательные подстановки.

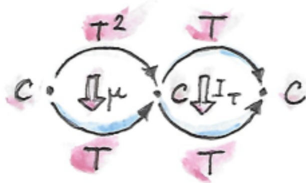
Для стрелок Клейсли для образования категории мы хотим, чтобы их композиция была ассоциативной, а η_a — тождественной стрелкой Клейсли в точке a. Это требование можно перевести в монадические законы

для μ и η. Но есть другой способ вывести эти законы, который заставляет их больше походить на законы для моноида. На самом деле, μ часто называют умножением, а η — единицей.

Грубо говоря, закон ассоциативности утверждает, что два способа сведения T^3 к T должны дать один и тот же результат. Два закона для единицы (левый и правый) гласят, что когда η применяется к T, а результат затем обрабатывается с помощью μ, мы возвращаемся к T.

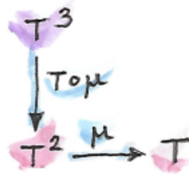
Все немного запутано, потому что мы соединяем естественные преобразования и функторы. Так что, немного освежим знания о горизонтальной композиции. Например, T^3 можно рассматривать как композицию T и T^2. Мы можем применить к ней горизонтальную композицию двух естественных преобразований:

$$I_T \circ \mu$$

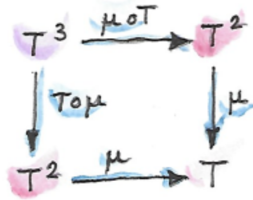


и получить $T \circ T$, которое может быть далее сведено к T применением μ. I_T — это естественное естественное преобразование от T к T. Вы часто будете видеть обозначения этого типа горизонтальной композиции, $I_T \circ \mu$, укороченной до $T \circ \mu$. Это обозначение однозначное, поскольку нет смысла составлять функтор с естественным преобразованием, поэтому в этом контексте T должен означать I_T.

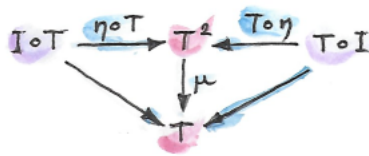
Мы можем также изобразить диаграмму в категории (эндо-) функторов $[\mathbf{C},\mathbf{C}]$:



В качестве альтернативы, мы можем рассматривать T^3 как композицию $T^2 \circ T$ и применить к ней $\mu \circ T$. Результатом является $T \circ T$, который, опять же, можно свести к T, используя μ. Мы требуем, чтобы оба способа давали один и тот же результат.



Аналогично, мы можем применить горизонтальную композицию $\eta \circ T$ к композиции тождественного функтора I после T для получения T^2, которая затем может быть сведена далее с помощью μ. Результат должен быть таким же, как если бы мы применили тождественное естественное преобразование непосредственно к T. И, по аналогии, то же самое должно быть справедливо и для $T \circ \eta$.



Вы можете убедиться, что рассмотренные законы гарантируют, что композиция стрелок Клейсли действительно удовлетворяет требованиям категории.

Сходство между монадой и моноидом поражает. Имеются: умножение μ, единица η, ассоциативность и законы для единицы. Но наше определение моноида слишком узкое, чтобы описывать монаду как моноид, так что, давайте обобщим понятие моноида.

22.1 Моноидальные категории

Вернемся к обычному определению моноида. Это множество с бинарной операцией и специальным элементом, называемым единицей. На Haskell

моноид может быть выражен в виде ти́пового класса:

```
class Monoid m where
   mappend :: m -> m -> m
   mempty  :: m
```

Бинарная операция `mappend` должна быть ассоциативной и унитальной.

Обратите внимание на то, что в Haskell определение `mappend` каррировано. Его можно интерпретировать как отображение каждого элемента `m` в функцию:

```
mappend :: m -> (m -> m)
```

Именно такая интерпретация и порождает определение моноида как категории одного объекта, где эндоморфизмы `(m -> m)` представляют элементы моноида. Но поскольку каррирование встроено в Haskell, мы могли бы начать с другого определения умножения:

```
mu :: (m,m) -> m
```

Здесь декартово произведение `(m,m)` становится источником пар, подлежащих умножению.

Это определение предполагает иной путь к обобщению: замену декартова произведения категорным произведением. Мы могли бы начать с категории, где произведения определены глобально, выбрать там объект m и определить умножение как морфизм:

$$\mu :: m \times m \to m$$

У нас есть одна проблема: в произвольной категории мы не можем заглянуть внутрь объекта, что связано с тем, как мы выбираем единичный элемент? Есть одна хитрость. Помните, что выбор элемента эквивалентен функции от одноэлементного множества? На Haskell мы могли бы заменить определение `mempty` функцией:

```
eta :: () -> m
```

Синглетон — это терминальный объект в **Set**, поэтому естественно обобщить это определение на любую категорию, у которой есть терминальный объект t:

$$\eta :: t \to m$$

Это позволяет нам выбрать единичный «элемент» без необходимости говорить об элементах.

В отличие от нашего предыдущего определения моноида как категории с одним объектом, моноидальные законы здесь не выполняются автоматически — мы должны навязывать их. Но для того, чтобы их сформулировать, необходимо установить моноидальную структуру самого базового категорного произведения. Напомним, как в Haskell работает моноидальная структура.

Начнем с ассоциативности. В Haskell соответствующий эквациональный закон имеет вид:

```
mu (x (mu y z) = mu (mu x y) z)
```

Прежде чем мы сможем обобщить его на другие категории, мы должны переписать его как равенство функций (морфизмов). Мы должны абстрагировать его от действий над отдельными переменными — другими словами, мы должны использовать бесточечную нотацию. Известно, что декартово произведение является бифунктором, и мы можем написать левую часть так:

```
(mu . bimap id mu)(x, (y, z))
```

а правую часть в виде:

```
(mu . bimap mu id)((x, y), z)
```

Это почти то, что нам требуется. К сожалению, декартово произведение не является строго ассоциативным — `(x,(y,z))` не совпадает с `((x,y),z)`, поэтому мы не можем просто записать в бесточечной форме:

```
mu . bimap id mu = mu . bimap mu id
```

С другой стороны, два вложения пар изоморфны. Существует обратимая функция, называемая ассоциатором, которая преобразует их:

```
alpha            :: ((a, b), c) ->
                        (a, (b, c))
alpha ((x, y), z) = (x, (y, z))
```

С помощью ассоциатора мы можем написать бестотечный закон ассоциативности для `mu`:

```
mu . bimap id mu . alpha = mu . bimap mu id
```

Мы можем применить подобный трюк к законам единицы, которые в новых обозначениях принимают вид:

```
mu (eta (), x) = x
mu (x, eta ()) = x
```

Они могут быть переписаны так:

```
(mu . bimap eta id) ((), x) = lambda ((), x)
(mu . bimap id eta) (x, ()) = rho (x, ())
```

Изоморфизмы `lambda` и `rho` называются соответственно левым и правым уединителем. Они свидетельствуют о том, что единица `()` является тождественностью декартова произведения с точностью до изоморфизма:

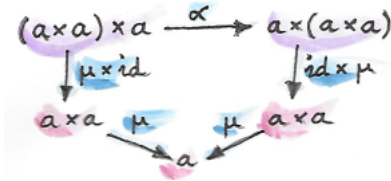
```
lambda        :: ((), a) -> a
lambda ((), x) = x

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

Таким образом, бесточечные версии законов единицы имеют вид:

```
mu . bimap id eta = rho
mu . bimap eta id = lambda
```

Мы сформулировали бесточечные моноидальные законы для `mu` и `eta`, используя тот факт, что основное декартово произведение действует как моноидальное умножение в категории типов. Имейте в виду, что ассоциативность и единичные законы для декартова произведения справедливы только с точностью до изоморфизма.

Оказывается, эти законы можно обобщить на любую категорию с произведениями и терминальным объектом. Категорные произведения действительно ассоциативны с точностью до изоморфизма, а терминальный объект является единицей, также с точностью до изоморфизма. Ассоциатор и два уединителя являются естественными изоморфизмами. Законы могут быть представлены коммутативными диаграммами.



Обратите внимание: поскольку произведение является бифунктором, он может поднять пару морфизмов — на Haskell это делается с помощью `bimap`.

Мы могли бы остановиться здесь и сказать, что можем определить моноид поверх любой категории с категорными произведениями и терминальным объектом. Пока мы можем выбрать объект m и два морфизма μ и η, которые удовлетворяют моноидальным законам, мы имеем моноид. Но мы можем добиться большего. Нам не нужно полномасштабное категорное произведение для формулировки законов для μ и η. Напомним, что произведение определяется универсальной конструкцией, использующей проекции. Мы не использовали никаких проекций при формулировке моноидальных законов.

Бифунктор, который ведет себя как произведение, не будучи произведением, называется *тензорным произведением*, часто обозначаемым инфиксным оператором $\otimes$. Определение тензорного произведения вообще несколько сложно, но мы не будем об этом беспокоиться. Мы просто перечислим его свойства — наиболее важным является ассоциативность с точностью до изоморфизма.

Аналогично, нам не нужно, чтобы объект t был терминальным. Мы никогда не использовали его терминальное свойство, а именно существование единственного морфизма от любого объекта к нему. Нам нужно, чтобы он хорошо взаимодействовал с тензорным произведением. Это означает, что мы хотим, чтобы он был единицей тензорного произведения, опять же с точностью до изоморфизма. Давайте соберем все вместе.

Моноидальная категория — это категория $\mathbf{C}$, снабженная бифунктором, называемым тензорным произведением:

$$\otimes :: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$$

и особым объектом i, называемым единичным объектом, вместе с тремя естественными изоморфизмами, называемыми, соответственно, ассоциатором и левым и правым уединителями:

$$\alpha_{a\,b\,c} :: (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$
$$\lambda_a :: i \otimes a \to a$$
$$\rho_a :: a \otimes i \to a$$

(Существует также условие когерентности для упрощения квадрупольного тензорного произведения.)

Важно то, что тензорное произведение описывает много знакомых бифункторов. В частности, оно работает на произведении, копроизведении и, как мы вскоре увидим, на композиции эндофункторов (а также на некоторых других эзотерических произведениях, таких как дневная свертка). Моноидальные категории будут играть существенную роль в формулировании обогащенных категорий.

22.2 Моноид в моноидальной категории

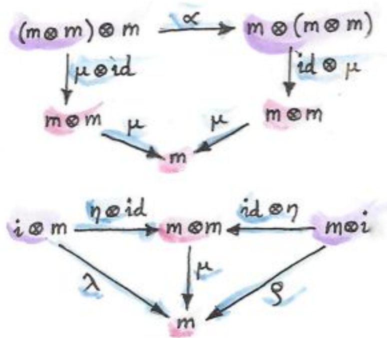
Теперь мы можем определить моноид в более общей ситуации с моноидальной категорией. Начнем с выбора объекта m. Используя тензорное

произведение, мы можем образовать степени m. Квадрат m равен $m \otimes m$. Существует два способа формирования куба m, но они изоморфны через ассоциатор. Аналогично для высших степеней m (именно здесь нам нужны условия когерентности). Чтобы сформировать моноид, нам нужно выбрать два морфизма:

$$\mu :: m \otimes m \to m$$
$$\eta :: i \to m$$

где i — единичный объект для нашего тензорного произведения.

Эти морфизмы должны удовлетворять законам ассоциативности и единицы, которые могут быть выражены в терминах следующих коммутативных диаграмм:



Заметим, и это существенно, что тензорное произведение является бифунктором, потому что нам нужно поднять пары морфизмов, чтобы обра-

зовать такие произведения, как $\mu \otimes \mathrm{id}$ или $\eta \otimes \mathrm{id}$. Эти диаграммы представляют собой простое обобщение наших предыдущих результатов для категорных произведений.

22.3 Монады как моноиды

Моноидальные структуры появляются в неожиданных местах. Одним из таких мест является категория функторов. Если вы посмотрите пристальнее, вы сможете разглядеть в функторной композиции форму умножения. Проблема состоит в том, что не могут быть скомпонованы никакие два функтора — целевая категория одного должна быть исходной категорией другого. Это обычное правило составления морфизмов, а как известно, функторы действительно являются морфизмами в категории $\mathbf{Cat}$. Но то, как эндоморфизмы (морфизмы, которые возвращаются к тому же объекту, от которого вышли) всегда могут быть скомпонованы, относится так же и к эндофункторам. Для любой заданной категории $\mathbf{C}$ эндофункторы от $\mathbf{C}$ к $\mathbf{C}$ образуют категорию функторов $[\mathbf{C},\mathbf{C}]$. Его объекты — эндофункторы, а морфизмы — естественные преобразования между ними. Мы можем взять любые два объекта из этой категории, скажем эндофункторы F и G, и создать третий объект $F \circ G$ — эндофунктор, который является их композицией.

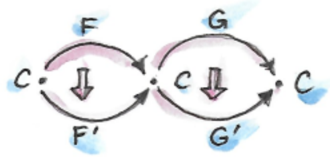
Является ли композиция эндофунторов хорошим кандидатом на тензорное произведение? Во-первых, мы должны установить, что это бифунктор. Может ли это быть использовано для поднятия пары морфизмов — здесь, естественных преобразований? Сигнатура аналога `bimap` для тензорного произведения будет выглядеть примерно так:

$$bimap :: (a \to b) \to (c \to d) \to (a \otimes c \to b \otimes d)$$

Если вы заменяете объекты эндофункторами, стрелки — естественными преобразованиями, а тензорные произведения — композицией, то получаете сигнатуру:

$$(F \to F') \to (G \to G') \to (F \to G \to F' \circ G')$$

которую можете признать особым случаем горизонтальной композиции.

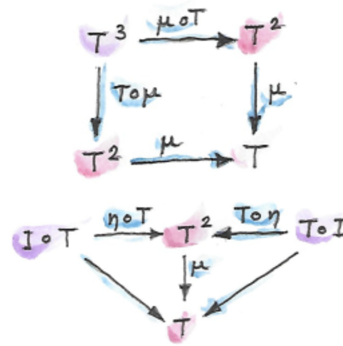


У нас также имеется единичный эндофунктор I, который может служить тождественным для композиции эндофунторов — нашего нового тензорного произведения. Кроме того, функторная композиция ассоциативна. На самом деле ассоциативность и законы единицы строгие — нет никакой необходимости в ассоциаторе или двух уединителях. Таким образом, эндофункторы образуют строгую моноидальную категорию с композицией функторов как тензорным произведением.

Что является моноидом в этой категории? Это объект — эндофунктор T и два морфизма — естественные преобразования:

$$\mu :: T \circ T \to T$$
$$\eta :: I \to T$$

естественно, с законами моноида:



Они — точь-в-точь законы монады, с которыми мы встречались ранее. Теперь вы поймете знаменитую цитату Сондерса Мак Лейна:

> *В общей сложности, монада — это просто моноид в категории эндофункторов.*

Возможно, вы видели ее в виде украшения на некоторых футболках на конференциях по функциональному программированию.

22.4 Монады из сопряжений

Сопряжение[1] $L \dashv R$ является парой функторов, идущих встречными курсами между двумя категориями $\mathbf{C}$ и $\mathbf{D}$. Существует два способа их композиции, дающих начало двум эндофункторам $R \circ L$ и $L \circ R$. В соответствии с сопряжением эти эндофункторы связаны с тождественными функторами посредством двух естественных преобразований, называемых единицей и коединицей:

$$\eta :: I_{\mathbf{D}} \to R \circ L$$
$$\varepsilon :: L \circ R \to I_{\mathbf{C}}$$

Сразу же мы видим, что единица сопряжения выглядит точно так же, как единица монады. Оказывается, что эндофунктор $R \circ L$ действительно является монадой. Все, что нам нужно, это определить подходящее μ для перехода с η. Это естественное преобразование между квадратом нашего эндофунктора и самим эндофунктором или, в терминах сопряженных функторов:

$$R \circ L \circ R \circ L \to R \circ L$$

И, действительно, мы можем использовать коединицу, чтобы свернуть $L \circ R$ в середине этого выражения. Точная формула для μ дается горизонтальной композицией:

$$\mu = R \circ \varepsilon \circ L$$

Монадические законы следуют из тождеств, которым удовлетворяют единица и коединица сопряжения, и закона чередования.

Мы не встречаем много монад, полученных из сопряжений, в Haskell, потому что сопряжение обычно охватывает две категории. Однако исключениями являются определения экспоненциала и функционального объекта. Вот два эндофунктора, которые образуют это сопряжение:

$$L\, z = z \times s$$
$$R\, b = s \Rightarrow b$$

[1] См. главу 18 о сопряжениях.

Вы можете узнать в их композиции знакомую монаду состояния:

$$R(L\,z) = s \Rightarrow (z \times s)$$

Мы уже сталкивались с этой монадой на Haskell:

```
newtype State s a = State (s -> (a, s))
```

Давайте также преобразуем сопряжение к нотации, принятой в Haskell. Левый функтор является функтором произведения:

```
newtype Prod s a = Prod (a, s)
```

а правый функтор — считывающим функтором:

```
newtype Reader s a = Reader (s -> a)
```

Они образуют сопряжение:

```
instance Adjunction (Prod s) (Reader s) where
   counit (Prod (Reader f, s)) = f s
   unit a                      = Reader
                                  (\s -> Prod
                                        (a, s))
```

Вы можете легко убедиться в том, что композиция считвающего функтора с функтором произведения действительно эквивалентна функтору состояния:

```
newtype State s a = State (s -> (a, s))
```

Как и ожидалось, `unit` сопряжения эквивалентна функции `return` монады состояния. `counit` действует, оценивая функцию, действующую на его аргумент. В этом можно узнать некаррированную версию функции `runState`:

```
runState             :: State s a -> s
                                  -> (a, s)
runState (State f) s = f s
```

(некаррированную, потому что в `counit` она действует на пару).

Теперь мы можем определить `join` для монады состояния как компонент естественного преобразования μ. Для этого нам нужна горизонтальная композиция трех естественных преобразований:

$$\mu = R \circ \varepsilon \circ L$$

Другими словами, нам нужно протащить коединицу ε через уровень считывающего функтора. Мы не можем вызвать `fmap` напрямую, потому что компилятор использует ее для функтора `State`, а не функтора `Reader`. Но напомним, что `fmap` для считывающего функтора — это просто левая композиция функций. Поэтому мы будем использовать функциональную композицию напрямую.

Мы должны сначала раскрыть конструктор данных `State`, чтобы сделать доступной функцию внутри функтора `State`. Это осуществляется с помощью `runState`:

```
ssa          :: State s (State s a)
runState ssa :: s -> (State s a, s)
```

Затем слева мы компонуем его с коединицей, что определяется некаррированной `runState`. И наконец, оборачиваем его обратно в конструктор данных `State`:

```
join    :: State s (State s a) -> State s a
join ssa = State (uncurry runState . runState ssa)
```

Это действительно является реализацией `join` для монады `State`.

Оказывается, что не только всякое сопряжение порождает монаду, но верно и обратное: каждая монада может быть разложена в композицию из двух сопряженных функторов. Однако такая факторизация не является однозначной.

О другом эндофункторе $L \circ R$ мы поговорим в следующем разделе.

Глава 23

Комонады

Теперь, когда мы рассмотрели монады, мы можем воспользоваться преимуществами двойственности и получить комонады без особых усилий, просто изменив направление стрелок на противоположное и работая в двойственной категории.

Напомним, что на базовом уровне монады описывают стрелки Клейсли:

```
a -> m b
```

где `m` — функтор, являющийся монадой. Если мы используем букву `w` (перевернутую `m`) для комонад, мы можем определить кострелки Клейсли как морфизм типа:

```
w a -> b
```

Аналог оператора `>=>` для ко-стрелок Клейсли определяется как:

```
(=>=) :: (w a -> b) -> (w b -> c)
                    -> (w a -> c)
```

Для ко-стрелок Клейсли, чтобы сформировать категорию, мы также должны иметь тождественную ко-стрелку Клейсли, которая обозначается `extract`:

```
extract :: w a -> a
```

Она двойственна `return`. Мы также должны ввести законы ассоциативности, а также левую и правую идентичность. Объединив все вместе, мы могли бы определить комонаду на Haskell как:

```
class Functor w => Comonad w where
   (=>=)   :: (w a -> b) -> (w b -> c)
                         -> (w a -> c)
   extract :: w a -> a
```

Как мы вскоре увидим, на практике используются несколько другие примитивы.

Вопрос в том, какова польза от комонад в программировании?

23.1 Программирование с помощью комонад

Давайте сравним монаду с комонадой. Монада обеспечивает способ помещения значения в контейнер с помощью `return`. Она не дает вам доступ к значению или значениям, хранящимся внутри. Конечно, структуры данных, которые реализуют монады, могут обеспечить доступ к их содержимому, но это считается бонусом. Нет общего интерфейса для извлечения значений из монады. И мы видели пример монады `IO`, которая никогда не раскрывает свое содержимое.

С другой стороны, комонада предоставляет средства для извлечения из него одного значения. Она не предоставляет средств для ввода значений. Поэтому, если вы хотите думать о комонаде как о контейнере, то надо иметь в виду, что она всегда приходит с предварительно заполненным содержимым, и это позволяет заглянуть в него.

Подобно тому, как стрелка Клейсли принимает значение и производит некоторый обогащенный результат — она обогащает его контекстом — ко-стрелка Клейсли принимает значение вместе со всем контекстом и выдает результат. Это воплощение контекстных вычислений.

23.2 Комонада Product

Помните считывающую монаду? Мы представили ее для решения проблемы реализации вычислений, которые нуждаются в доступе к некоторой среде `e` только для чтения. Такие вычисления можно представить в виде чистых функций вида:

```
(a, e) -> b
```

Мы использовали карринг, чтобы превратить их в стрелки Клейсли:

```
a -> (e -> b)
```

Но заметьте, что эти функции уже имеют форму ко-стрелок Клейсли. Давайте представим их аргументы в более удобной форме функтора:

```
data Product e a = Prod e a
   deriving Functor
```

Мы можем легко определить оператор композиции, сделав одно и то же окружение доступным для стрелок, которые мы компонуем:

```
(=>=)  :: (Product e a -> b) ->
            (Product e b -> c) ->
                (Product e a -> c)
f =>= g = \(Prod e a) -> let b = f (Prod e a)
                             c = g (Prod e b)
                         in  c
```

Реализация `extract` просто игнорирует среду:

```
extract (Prod e a) = a
```

Неудивительно, что комонаду результата (Product) можно использовать для выполнения точно таких же вычислений, что и считывающую монаду. В некотором смысле, совместная реализация окружения более естественна — она следует духу «вычислений в контексте». С другой стороны, монады поставляются с удобным синтаксическим сахаром do нотации.

Связь между считывающей монадой и комонадой результата идет глубже, что связано с тем, что функтор Reader является правым сопряженным к функтору Product. В общем, однако, комонады охватывают больше различных ньюансов вычислений, чем монады. Мы увидим еще примеры позже.

Легко обобщить комонаду Product на произвольные типы результатов, включая кортежи и записи.

23.3 Анализ композиции

Продолжая процесс дуализации, мы могли бы продолжить дуализировать монадическое связывание и соединение. Альтернативно, мы можем повторить процесс, который использовали с монадами, где изучалась анатомия оператора >=>. Этот подход кажется более поучительным.

Отправной точкой является осознание того, что оператор композиции должен создать ко-стрелку Клейсли, которая принимает w a и производит c. Единственный способ получить c — это применить вторую функцию к аргументу типа w b:

```
(=>=)  :: (w a -> b) -> (w b -> c)
                     -> (w a -> c)
f =>= g = g ...
```

Но как мы можем получить значение типа w b, которое можно было бы передать в g? В нашем распоряжении есть аргумент типа w a и функция f :: w a -> b. Решением является определение двойного связывания, которое называется продолжением (extend):

```
extend :: (w a -> b) -> w a -> w b
```

Используя `extend`, мы можем реализовать композицию:

```
f =>= g = g . extend f
```

Можем ли мы следующим шагом расщепить `extend`? Возможно, у вас возникнет соблазн сказать, почему бы просто не применить функцию `w a -> b` к аргументу `w a`, но тогда вы быстро поймете, что у вас не будет способа преобразовать результирующий `b` в `w b`. Помните, что комонада не предоставляет средств для поднятия значений. На этом этапе в аналогичной конструкции для монад мы использовали `fmap`. Единственный способ, которым мы могли бы использовать `fmap` здесь, был бы, если бы мы имели в распоряжении что-то типа `w (w a)`. Если бы мы только могли превратить `w a` в `w (w a)`. И, как ни странно, это было бы в точности двойственно к `join`. Обозначим это преобразование `duplicate`:

```
duplicate :: w a -> w (w a)
```

Итак, как и в случае с определениями монады, мы имеем три эквивалентных определения комонады: используя кострелки Kleisli, `extend` или `duplicate`. Вот определение Haskell, взятое непосредственно из библиотеки `Control.Comonad`:

```
class Functor w => Comonad w where
   extract   :: w a -> a
   duplicate :: w a -> w (w a)
   duplicate  = extend id
   extend    :: (w a -> b) -> w a -> w b
   extend f   = fmap f . duplicate
```

При том, что `extend` реализуется при помощи `duplicate` и наоборот, вам нужно переопределить только один из них.

Интуиция этих функций основана на идее, что, в общем, комонаду можно рассматривать как контейнер, заполненный значениями типа `a` (комонада результата была частным случаем только одного значения). Существует понятие «текущее» значение, которое легко доступно через

extract. Кострелка Kleisli выполняет некоторые вычисления, ориентированные на текущее значение, но имеет доступ ко всем соседним значениям. Подумайте об игре в жизнь Конвэя. Каждая ячейка в ней содержит значение (обычно просто True или False). Комонада, соответствующая игре в жизнь, представляет собой сетку ячеек, сфокусированных на «текущей» ячейке.

Так что же делает duplicate? Она принимает комонадический контейнер w a и создает контейнер с контейнерами w (w a). Идея состоит в том, что каждый из этих контейнеров ориентирован на «свое» a внутри w a. В игре в жизнь вы получите сетку сеток, где каждая ячейка внешней сетки содержит внутреннюю сетку, которая фокусируется на «своей» ячейке.

Теперь рассмотрим extend. Эта функция берет ко-стрелку Клейсли и комонадический контейнер w a, заполненный экземплярами a. Она применяет вычисления ко всем этим a, заменяя их на b. В результате получается комонадический контейнер, заполненный значениями b. extend выполняет это, переставляя фокус с одного a на другое и применяя ко-стрелку Клейсли к каждому из них по очереди. В игре в жизнь ко-стрелка Клейсли вычисляет новое состояние текущей ячейки. Чтобы сделать это, необходимо проанализировать свой контекст — предположительно своих ближайших соседей. По умолчанию реализация extend иллюстрирует этот процесс. Сначала мы вызываем duplicate для создания всех возможных фокусов, а затем применяем f к каждому из них.

23.4 Комонада Stream

Процесс смещения фокуса с одного элемента контейнера на другой лучше всего иллюстрируется на примере бесконечного потока. Такой поток подобен списку, за исключением того, что у него нет пустого конструктора:

```
data Stream a = Cons a (Stream a)
```

Это тривиально Functor:

```
instance Functor Stream where
   fmap f (Cons a as) = Cons (f a) (fmap f as)
```

Фокусом потока является его первый элемент, поэтому здесь приводится реализация `extract`:

```
extract (Cons a _) = a
```

`duplicate` создает поток потоков, каждый из которых сфокусирован на своем элементе.

```
duplicate (Cons a as) = Cons (Cons a as)
                             (duplicate as)
```

Первый элемент — это исходный поток, второй элемент — хвост исходного потока, третий элемент — его хвост и т.д. до бесконечности.

Вот полный экземпляр:

```
instance Comonad Stream where
   extract (Cons a _)    = a
   duplicate (Cons a as) = Cons (Cons a as)
                                (duplicate as)
```

Это настоящий функциональный способ рассматрения потоков. На императивном языке мы, вероятно, начнем с метода `advance`, который сдвигает поток на одну позицию. Здесь же `duplicate` производит все сдвинутые потоки одним махом. Ленивость Haskell делает это возможным и даже желательным. Конечно, чтобы практиковать `Stream`, нам бы также нужно реализовать аналог `advance`:

```
tail             :: Stream a -> Stream a
tail (Cons a as) = as
```

но это, конечно, не является частью комонадического интерфейса.

Если у вас есть опыт работы с цифровой обработкой сигналов, вы сразу увидите, что ко-стрелка Клейсли для потока — это просто цифровой фильтр, а `extend` — отфильтрованный поток.

В качестве простого примера давайте реализуем фильтр скользящего среднего. Вот функция, суммирующая `n` элементов потока:

```
sumS      :: Num a => Int -> Stream a -> a
sumS n (Cons a as) = if n <= 0 then 0
                     else a + sumS (n - 1) as
```

Вот функция, вычисляющая среднее из первых `n` элементов потока:

```
average     :: Fractional a => Int ->
                              Stream a -> a
average n stm = (sumS n stm) / (fromIntegral n)
```

Частичное применение `average n` — это ко-стрелка Клейсли, поэтому мы можем применить `extend` ко всему потоку:

```
movingAvg  :: Fractional a => Int ->
                             Stream a ->
                               Stream a
movingAvg n = extend (average n)
```

Результатом является поток средних значений.

Поток — это пример однонаправленной одномерной комонады. Ее легко можно сделать двунаправленной или расширить до второй или более размерностей.

23.5 Комонада с категорной точки зрения

Определение комонады в теории категорий — это простое упражнение по двойственности. Как и в случае монады, мы начинаем с эндофунктора T. Два естественных преобразования, η и μ, которые определяют

монаду, просто обращаются для комонады:

$$\varepsilon :: T \to I$$
$$\delta :: T \to T^2$$

Компоненты этих преобразований соответствуют `extract` и `duplicate`. Законы для комонады — зеркальное отражение законов монады. Здесь нет неожиданностей.

Затем, существует вывод монады из сопряжения. Двойственность меняет сопряжение: левая сопряженная становится правой сопряженной и наоборот. И, так как композиция $R \circ L$ определяет монаду, $L \circ R$ должна определять комонаду. Коединица сопряжения:

$$\varepsilon :: L \circ R \to I$$

это то же ε, что присутствует в определении комонады, или в компонентах, как `extract` из Haskell. Мы также можем использовать единицу сопряжения:

$$\eta :: I \to R \circ L$$

чтобы вставить $R \circ L$ в середину $L \circ R$ и получить $L \circ R \circ L \circ R$. Обозначаем $T \to T^2$ через δ, и этим завершаем определение комонады.

Мы также видели, что монада является моноидом. Двойник этого утверждения потребует использования комоноида, так что же это такое? Исходное определение моноида, как категории с одним объектом, не дуализуется к чему-либо, вызывающему интерес. Когда вы меняете направление всех эндоморфизмов, вы получаете другой моноид. Напомню, однако, что в нашем подходе к монаде мы использовали более общее определение моноида как объекта в моноидальной категории. Конструкция была основана на двух морфизмах:

$$\mu :: m \otimes m \to m$$
$$\eta :: i \to m$$

Обращение этих морфизмов приводит к появлению комоноида в моноидальной категории:

$$\delta :: m \to m \otimes m$$
$$\varepsilon :: m \to i$$

На Haskell можно так записать определение комоноида:

```
class Comonoid m where
   split   :: m -> (m, m)
   destroy :: m -> ()
```

Но это довольно тривиально. Очевидно, `destroy` игнорирует свой аргумент.

```
destroy _ = ()
```

`split` — это всего лишь пара функций:

```
split x = (f x, g x)
```

Теперь рассмотрим законы комоноида, двойственные моноидальным законам единицы.

```
lambda . bimap destroy id . split = id
rho . bimap id destroy . split    = id
```

Здесь `lambda` и `rho` являются левым и правым уединителями, соответственно (см. определение моноидальных категорий). Применяя левую часть первого закона к аргументу, получаем:

```
lambda (bimap destroy id (split x))
= lambda (bimap destroy id (f x, g x))
= lambda ((), g x)
= g x
```

т.е. `g = id`. Аналогично, второй закон сводится к `f = id`. А тогда:

```
split x = (x, x)
```

откуда следует, что в Haskell (и, в общем, в категории **Set**) каждый объект является тривиальным ко-моноидом.

К счастью, есть и другие, более интересные моноидальные категории, в которых можно определить комоноиды. Одним из них является категория эндофункторов. И оказывается, что, подобно монаде, являющейся моноидом в категории эндофункторов,

Комонада является комоноидом в категории эндофункторов.

23.6 Комонада Store

Другим важным примером комонады является двойственная к монаде состояния. Она называется комонадой косостояния или, в качестве альтернативы, комонадой хранения.

Ранее мы видели, что монада состояния генерируется сопряжением, которое определяет экспоненциалы:

$$L\,z = z \times s$$
$$R\,a = s \Rightarrow a$$

Мы будем использовать это же сопряжение для определения комонады косостояния. Комонада определяется композицией $L \circ R$:

$$L\,(R\,a) = (s \Rightarrow a) \times s$$

Переводя это на Haskell, мы начнем с сопряжения между функтором `Prod` слева и функтором `Reader` справа. Композиция `Prod` после `Reader` эквивалентна следующему определению:

```
data Store s a = Store (s -> a) s
```

Коединица сопряжения при объекте a, есть морфизм:

$$\varepsilon_a :: ((s \Rightarrow a) \times s) \to a$$

или, в нотации Haskell:

```
counit (Prod (Reader f, s)) = f s
```

А это становится нашим `extract`:

```
extract (Store f s) = f s
```

Единица сопряжения:

```
unit  :: a -> Reader s (Product a s)
unit a = Reader (\s -> Prod a s)
```

может быть переписана как частично примененный конструктор данных:

```
Store f :: s -> Store f s
```

Мы конструируем δ, или `duplicate`, как горизонтальную композицию:

$$\delta :: L \circ R \to L \circ R \circ L \circ R$$
$$\delta = L \circ \eta \circ R$$

Нам нужно протащить η через самый левый L, который является функтором `Product`. Это означает действие с η или `Store f` на левом компоненте пары (это то, что будет делать `fmap` для `Product`). Мы получаем:

```
duplicate (Store f s) = Store (Store f) s
```

(напомню, что в формуле для δ, L и R обозначают тождественные естественные преобразования, компоненты которых являются тождественными морфизмами).

Вот полное определение комонады `Store`:

```
instance Comonad (Store s) where
   extract (Store f s)   = f s
   duplicate (Store f s) = Store (Store f) s
```

Вы можете размышлять о части `Reader` из `Store` как обобщенном контейнере, так как это ключ, использующий элементы типа `s`. Например, если `s` — `Int`, то `Reader Int a` — бесконечный двунаправленный поток элементов `a`. `Store` связывает этот контейнер со значением типа ключа. Например, `Reader Int a` сопряжен с `Int`. В этом случае `extract` использует это целое число для индексации в бесконечном потоке. Вы можете думать о втором компоненте `Store` как о текущей позиции.

Продолжая этот пример, `duplicate` создает новый бесконечный поток, проиндексированный `Int`. Этот поток содержит потоки в качестве своих элементов. В частности, в текущей позиции он содержит исходный поток. Но если вы используете какое-либо другое значение типа `Int` (положительное или отрицательное) в качестве ключа, вы получите сдвинутый поток, расположенный в этом новом индексе.

В общем, вы можете удостовериться, что когда `extract` действует на сдублированный `Store`, он производит исходный `Store` (на самом деле, закон идентификации для комонады утверждает, что `extract . duplicate = id`).

Комонада `Store` играет важную роль в качестве теоретической основы библиотеки `Lens`. Концептуально, комонада `Store s a` инкапсулирует идею «фокусировки» (наподобие объектива) на определенную подструктуру типа данных `a`, используя тип `s` в качестве индекса. В частности, функция типа:

```
a -> Store s a
```

эквивалентна паре функций:

```
set :: a -> s -> a
get :: a -> s
```

Если `a` — тип произведения, `set` может быть реализована как установка поля типа `s` внутри `a` при возврате модифицированной версии `a`. Аналогично, `get` может быть реализована для чтения значения поля `s` из `a`. Мы рассмотрим эти идеи более подробно в следующем разделе.

Упражнения

1. Реализуйте «Игру в жизнь» Конвэя, используя комонаду `Store`. Подсказка: какой тип вы выберете для `s`?

Глава 24

F-алгебры

Мы рассмотрели несколько формулировок моноида: как множество, как категорию с одним объектом, как объект в моноидальной категории. Что еще мы можем выжать из этой простой концепции?

Давайте попробуем. Возьмем в качестве определения моноида множество m с парой функций:

$$\mu :: m \times m \to m$$
$$\eta :: 1 \to m$$

Здесь 1 — это терминальный объект в одноэлементном множестве (синглетоне). Первая функция определяет умножение (она принимает пару элементов и возвращает их произведение), вторая выбирает единичный элемент из m. Не любой выбор двух функций с этими сигнатурами приводит к моноиду. Для этого нам необходимо наложить дополнительные условия: ассоциативность и законы для единицы. Но давайте забудем об этом на минутку и просто рассмотрим «потенциальные моноиды». Пара функций — это элемент декартова произведения двух множеств функций. Мы знаем, что эти множества могут быть представлены как экспоненциальные объекты:

$$\mu \in m^{m \times m}, \quad \eta \in m^1$$

Декартовым произведением этих двух множеств является:

$$m^{m\times m} \times m^1$$

Используя алгебру старших классов школы (которая работает в каждой декартовой замкнутой категории), мы можем переписать его как:

$$m^{m\times m+1}$$

Знак «+» означает копроизведение в нашем синглетоне. Мы только что заменили пару функций одной функцией — элементом множества:

$$m\times m + 1 \to m$$

Любой элемент этого множества функций является потенциальным моноидом.

Красота этой формулировки состоит в том, что она приводит к интересным обобщениям. Например, как бы мы описали группу, использующую этот язык? Группа — это моноид с одной дополнительной функцией, которая назначает инверсию каждому элементу. Последнее является функцией типа $m \to m$. Например, целые числа образуют группу со сложением в качестве бинарной операции, нулем в качестве единицы, и отрицанием в качестве обратной операции. Чтобы определить группу, мы начали бы с тройки функций:

$$\begin{aligned} m \times m &\to m \\ m &\to m \\ 1 &\to m \end{aligned}$$

Как и выше, мы можем объединить все такие тройки в один набор функций:

$$m \times m + m + 1 \to m$$

Мы начали с одного бинарного оператора (сложения), одного унарного оператора (отрицания) и одного нульарного оператора (тождественного, здесь ноль). Мы объединили их в одну функцию. Все функции с этой сигнатурой определяют потенциальные группы.

Мы можем продолжать в таком же духе. Например, чтобы определить кольцо, мы добавили бы еще один бинарный оператор и один нульарный оператор и т.д. Каждый раз мы получаем функциональный тип, левая часть которого представляет собой сумму степеней (возможно, включая нулевую мощность — для конечного объекта), а правая часть — это само множество.

Так мы можем помешаться на обобщениях. Прежде всего, мы можем заменить множества объектами, а функциями морфизмами. Мы можем определить n-арные операторы как морфизмы от n-арных произведений. Это означает, что нам нужна категория, которая поддерживает конечные произведения. Для нульарных операторов требуется существование терминального объекта. Поэтому нам требуется декартова категория. Чтобы объединить эти операторы, нам нужны экспоненциалы, так что это должна быть декартово замкнутая категория. Наконец, нам нужны копроизведения, чтобы завершать наши алгебраические манипуляции.

Кроме того, мы можем просто забыть о том, как мы получали наши формулы и сосредоточиться на полном произведении. Сумма произведений в левой части нашего морфизма определяет эндофунктор. Что если мы возьмем для этого произвольный эндофунктор F? В этом случае мы не должны накладывать каких-либо ограничений на нашу категорию. То, что мы получим, называется **F**-алгеброй.

F-алгебра представляет собой тройку, состоящую из эндофунктора F, объекта a и морфизма

$$F\,a \to a$$

Объект часто называют носителем, базовым объектом или, в контексте программирования, типом носителя. Морфизм часто называют оценочной функцией или структурным отображением. Можно считать функтор F формирующими выражениями, а морфизм их оценкой.

Вот определение **F**-алгебры на Haskell:

```
type Algebra f a = f a -> a
```

Оно идентифицирует алгебру с ее оценочной функцией.

В примере моноида, рассматриваемый функтор есть:

```
data MonF a = MEmpty | MAppend a a
```

Это определение на Haskell для $1 + a \times a$ (вспомните алгебраические структуры данных).

Кольцо могло бы быть определено с помощью следующего функтора:

```
data RingF a = RZero
             | ROne
             | RAdd a a
             | RMul a a
             | RNeg a
```

который, как алгебраическая структура, есть $1 + 1 + a \times a + a \times a + a$.

Примером кольца является множество целых чисел. Мы можем выбрать `Integer` в качестве типа оператора и определить оценочную функцию как:

```
evalZ :: Algebra RingF Integer
evalZ RZero      = 0
evalZ ROne       = 1
evalZ (RAdd m n) = m + n
evalZ (RMul m n) = m * n
evalZ (RNeg n)   = -n
```

Существуют и другие **F**-алгебры, основанные на одном и том же функторе `RingF`. Например, многочлены образуют кольцо и, следовательно, квадратные матрицы.

Как вы можете видеть, роль функтора заключается в генерации выражений, которые могут быть вычислены с помощью оценщика алгебры. До сих пор мы сталкивались только с очень простыми выражениями. Нас же часто интересуют более сложные выражения, которые можно определить с помощью рекурсии.

24.1 Рекурсия

Одним из способов генерации деревьев произвольных выражений является замена переменной `a` внутри определения функтора рекурсией. Например, произвольное выражение в кольце генерируется следующей древовидной структурой данных:

```
data Expr = RZero
          | ROne
          | RAdd Expr Expr
          | RMul Expr Expr
          | RNeg Expr
```

Мы можем заменить исходную оценочную функцию кольца рекурсивной версией:

```
evalZ             :: Expr -> Integer
evalZ RZero        = 0
evalZ ROne         = 1
evalZ (RAdd e1 e2) = evalZ e1 + evalZ e2
evalZ (RMul e1 e2) = evalZ e1 * evalZ e2
evalZ (RNeg e)     = -(evalZ e)
```

Это все еще не очень практично, поскольку мы вынуждены представлять все целые числа в виде суммы единиц, но это будет выполняться в крайнем случае.

Но как мы можем описать деревья выражений, используя язык **F**-алгебр? Нам нужно как-то формализовать процесс замены переменной свободного типа в определении нашего функтора, рекурсивно, с результатом замены. Представьте, что вы делаете это пошагово. Сначала определим дерево глубины 1 как:

```
type RingF1 a = RingF (RingF a)
```

Мы заполняем дыры в определении `RingF` деревьями глубины 0, генерируемыми `RingF a`. Аналогично получаются деревья глубины 2:

```
type RingF2 a = RingF (RingF (RingF a))
```

что мы также можем написать так:

```
type RingF2 a = RingF (RingF1 a)
```

Продолжая этот процесс, мы можем написать символическую формулу:

```
type RingFn+1 a = RingF (RingFn a)
```

Концептуально, повторяя этот процесс бесконечно, мы в итоге получим наш `Expr`. Обратите внимание, что `Expr` не зависит от `a`. Начальная точка нашего движения не имеет значения, мы всегда оказываемся в нужном месте. Это не всегда верно для произвольного эндофунктора в произвольной категории, но в категории **Set** все обстоит так, как описано.

Конечно, этот аргумент слабоват, и я сделаю его более строгим позже.

Применяя эндофунктор бесконечное число раз, вы получаете неподвижную точку, объект, определяемый как:

$$Fix\, f = f\,(Fix\, f)$$

Интуиция, лежащая в основе этого определения, состоит в том, что, поскольку мы применяем f бесконечно много раз, чтобы получить $Fix\, f$, применение его еще один раз ничего не меняет. На Haskell определение неподвижной точки имеет следующую форму:

```
newtype Fix f = Fix (f (Fix f))
```

Возможно, было бы более читаемо, если бы имя конструктора отличалось от имени определяемого типа, например:

```
newtype Fix f = In (f (Fix f))
```

но я буду придерживаться принятой нотации. Конструктор `Fix` (или, если вы предпочитаете, `In`) можно рассматривать как функцию:

```
Fix :: f (Fix f) -> Fix f
```

Существует также функция, которая отделяет (уменьшает) один уровень применения функтора:

```
unFix         :: Fix f -> f (Fix f)
unFix (Fix x) = x
```

Две последние функции являются взаимо обратными. Мы будем использовать их позже.

24.2 Категория F-алгебр

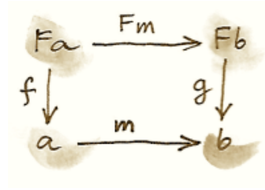
Вот самый испытанный трюк в книге: всякий раз, когда вы придумываете способ построения некоторых новых объектов, посмотрите, образуют ли они категорию. Не удивительно, что алгебры над заданным эндофунктором F образуют категорию. Объектами в этой категории являются пары алгебр, состоящие из носителя a и морфизма $F\,a \to a$, оба из исходной категории $\mathbf{C}$.

Чтобы завершить построение, мы должны определить морфизмы в категории **F**-алгебр. Морфизм должен отображать одну алгебру (a, f) в другую алгебру (b, g). Мы определим его как морфизм m, который отображает носители — от a к b в исходной категории. Не любой морфизм годится для этого: мы хотим, чтобы он был совместим с двумя оценивающими функциями (мы называем такой морфизм, сохраняющий структуру, *гомоморфизмом*). Вот как вы можете определить гомоморфизм **F**-алгебр. Во-первых, заметьте, что мы можем поднять m до отображения:

$$F\,m :: F\,a \to F\,b$$

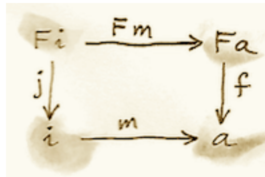
Мы можем затем с помощью g перейти от $F\,b$ к b. Аналогично, мы можем использовать f для перехода от $F\,a$ к a, а затем следовать согласно m. При этом мы хотим, чтобы два пути были равноправны:

$$g \circ F\,m = m \circ f$$



Легко убедиться в том, что это действительно категория (подсказка: тождественные морфизмы из **C** работают просто прекрасно, а композиция гомоморфизмов является гомоморфизмом).

Инициальный объект в категории **F**-алгебр, если он существует, называется *инициальной алгеброй*. Обозначим носитель этой инициальной алгебры через i, а ее оценивающую функцию через $j :: F\, i \to i$. Оказывается, что оценивающая функция j инициальной алгебры является изоморфизмом. Этот результат известен как теорема Ламбека. Доказательство опирается на определение инициального объекта, для которого требуется, чтобы от него существовал единственный гомоморфизм m в любую другую **F**-алгебру. Так как m — гомоморфизм, следующая диаграмма должна быть коммутативной:



Теперь давайте построим алгебру, носителем которой является $F\, i$. Оценивающей функцией такой алгебры должен быть морфизм от $F\,(F\, i)$ к $F\, i$. Мы можем легко построить такую функцию просто поднятием j:

$$F\, j :: F\,(F\, i) \to F\, i$$

Поскольку (i, j) является инициальной алгеброй, должен существовать единственный гомоморфизм m от нее к $(F\, i, F\, j)$. Следующая диаграмма должна быть коммутативной:

Но у нас также имеется эта тривиально коммутативная диаграмма (оба пути одинаковы!):

что можно интерпретировать как доказательство того, что j является гомоморфизмом алгебр, отображением $(F\,i, F\,j)$ в (i, j). Мы можем склеить эти две диаграммы, чтобы получить:

Эта диаграмма, в свою очередь, может быть истолкована как показывающая, что $j \circ m$ является гомоморфизмом алгебр. Только в этом случае обе алгебры одинаковы. Более того, поскольку (i, j) является инициальной, от нее может идти только один гомоморфизм к себе, а это тождественный морфизм id_i, который, как мы знаем, является гомоморфизмом алгебр. Поэтому $j \circ m = \text{id}_i$. Аналогично, мы можем показать, что $m \circ j = \text{id}_{F\,i}$, складывая две диаграммы в обратном порядке. Это демонстрирует, что m является инверсией j и, следовательно, j является изоморфизмом между $F\,i$ и i:

$$F\,i \cong i$$

Но это просто говорит о том, что i — неподвижная точка F. Это формальное доказательство первоначальных неформальных рассуждений.

Вернемся к Haskell: мы распознаем i как `Fix f`, j как конструктор `Fix`, а его инверсию как `unFix`. Изоморфизм в теореме Ламбека означает, что для получения инициальной алгебры берется функтор f и его аргумент a заменяется на `Fix f`. Также ясно, почему неподвижная точка не зависит от a.

24.3 Натуральные числа

Натуральные числа также могут быть определены как **F**-алгебра. Отправной точкой является пара морфизмов:

$$zero :: 1 \to N$$
$$succ :: N \to N$$

Первая выбирает нуль, а вторая отображает все числа к своим преемникам. Как и прежде, мы можем объединить эти два определения в одно:

$$1 + N \to N$$

Левая часть определяет функтор, который на Haskell можно записать так:

```
data NatF a = ZeroF | SuccF a
```

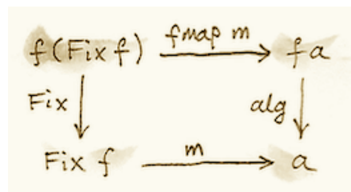
Неподвижная точка этого функтора (инициальная алгебра, которую он порождает) может быть закодирована на Haskell как:

```
data Nat = Zero | Succ Nat
```

Натуральное число может быть либо нулем, либо преемником другого числа. Это называется представлением Пеано для натуральных чисел.

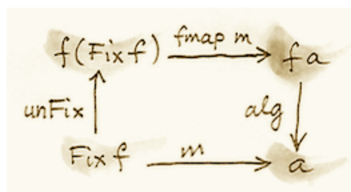
24.4 Катаморфизмы

Перепишем условие инициальности, используя нотацию Haskell. Будем обозначать инициальную алгебру как `Fix f`. Ее оценивающая функция — это конструктор `Fix`. Существует единственный морфизм `m` от инициальной алгебры к любой другой алгебре над тем же функтором. Рассмотрим алгебру, носителем которой является `a`, а оценивающей функцией — `alg`.



Кстати, обратите внимание на то, что `m`: это оценивающая функция для неподвижной точки, оценивающая функция для всего рекурсивного дерева выражений. Давайте найдем общий способ ее реализации.

Из теоремы Ламбека следует, что конструктор `Fix` является изоморфизмом. Обозначим обратный к нему как `unFix`. Тогда мы можем изменить направление одной стрелки на приведенной диаграмме, чтобы получить:



Запишем условие коммутативности этой диаграммы:

```
m = alg . fmap m . unFix
```

Мы можем интерпретировать эту формулу как рекурсивное определение `m`. Рекурсия обязана завершиться для любого конечного дерева, созданного с помощью функтора `f`. Мы видим это, замечая, что `fmap m` действует под верхним слоем функтора `f`. Другими словами, она работает с дочерними элементами исходного дерева.

Вот что происходит, когда мы применяем m к дереву, построенному с помощью Fix f. Действие unFix удаляет конструктор, раскрывая верхний уровень дерева. Затем мы применяем m ко всем дочерним элементам верхнего узла. Это приводит к результатам типа a. Наконец, мы объединяем эти результаты, применяя нерекурсивную оценивающую функцию alg. Ключевым моментом является то, что оценивающая функция alg является простой нерекурсивной.

Так как мы можем сделать это для любой алгебры alg, имеет смысл определить функцию высшего порядка, которая принимает алгебру как параметр и выдает нам функцию, которую мы назвали m. Эта функция высшего порядка называется *катаморфизмом*:

```
cata     :: Functor f => (f a -> a) ->
                              Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

Рассмотрим это на примере. Возьмем функтор, определяющий натуральные числа:

```
data NatF a = ZeroF | SuccF a
```

Давайте выберем (Int, Int) в качестве типа носителя и определим нашу алгебру как:

```
fib                :: NatF (Int, Int) ->
                               (Int, Int)
fib ZeroF           = (1, 1)
fib (SuccF (m, n)) = (n, m + n)
```

Вы можете легко убедиться в том, что катаморфизм для этой алгебры, cata fib, вычисляет числа Фибоначчи.

В общем случае алгебра для NatF определяет рекуррентное соотношение: значение текущего элемента выражается в терминах предыдущего элемента. Затем катаморфизм определяет n-ый элемент этой последовательности.

24.5 Свертки

Список e является инициальной алгеброй следующего функтора:

```
data ListF e a = NilF | ConsF e a
```

Действительно, заменив переменную a результатом рекурсии, которую мы обозначим List e, получим:

```
data List e = Nil | Cons e (List e)
```

Алгебра для функтора списка выбирает конкретный тип носителя и определяет функцию, которая выполняет сопоставление с образцом для двух конструкторов. Его значение для NilF говорит нам, как оценивать пустой список, а его значение для ConsF — как объединить текущий элемент с ранее накопленным значением.

Например, вот алгебра, которая может быть использована для вычисления длины списка (тип носителя — Int):

```
lenAlg            :: ListF e Int -> Int
lenAlg (ConsF e n) = n + 1
lenAlg NilF        = 0
```

В самом деле, полученный катаморфизм cata lenAlg вычисляет длину списка. Обратите внимание, что оценивающая функция представляет собой комбинацию: (1) функции, которая принимает элемент списка и аккумулятор, а возвращает новый аккумулятор, и (2) начальное значение, здесь ноль. Тип значения и тип аккумулятора задаются типом носителя.

Сравните это с традиционным определением на Haskell:

```
length = foldr (\e n -> n + 1) 0
```

Два аргумента функции правой свертки foldr являются в точности двумя компонентами алгебры.

Попробуем другой пример:

```
sumAlg             :: ListF Double Double ->
                                   Double
sumAlg (ConsF e s) = e + s
sumAlg NilF        = 0.0
```

Опять, сравните это с:

```
sum = foldr (\e s -> e + s) 0.0
```

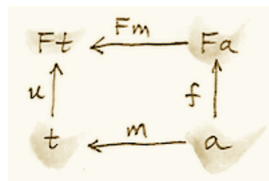
Как вы можете видеть, `foldr` — это просто удобная специализация катаморфизма для списков.

24.6 Коалгебры

Как обычно, мы имеем двойственную конструкцию — **F**-коагебру, в которой, по сравнению с **F**-алгеброй, направление морфизма изменено на обратное:

$$a \to F\, a$$

Коалгебры для данного функтора также образуют категорию с гомоморфизмами, сохраняющими коалгебраическую структуру. Терминальный объект (t, u) в этой категории называется терминальной (или финальной) коалгеброй. Для любой другой алгебры (a, f) существует единственный гомоморфизм m, который превращает следующую диаграмму в коммутативную:



Терминальная коалгебра является неподвижной точкой функтора в том смысле, что морфизм $u :: t \to F\, t$ является изоморфизмом (теорема Ламбека для коалгебр):

$$F\, t \cong t$$

Терминальная коалгебра обычно интерпретируется в программировании как средство для генерирования (возможно, бесконечных) структур данных или переходных систем.

Подобно тому, как катаморфизм может быть использован для оценки инициальной алгебры, анаморфизм может быть использован для оценки терминальной коалгебры:

```
ana      :: Functor f => (a -> f a) ->
                          a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
```

Канонический пример коалгебры основан на функторе, неподвижной точкой которого является бесконечный поток элементов типа `e`. Это функтор:

```
data StreamF e a = StreamF e a
   deriving Functor
```

а это его неподвижная точка:

```
data Stream e = Stream e (Stream e)
```

Коалгебра для `StreamF e` — это функция, которая принимает начальное значение типа `a` и создает пару (`StreamF` — причудливое имя для пары), состоящее из элемента и следующего начального значения.

Вы можете легко создавать простые примеры коалгебр, которые производят бесконечные последовательности, такие как список квадратов или обратных значений.

Более интересным примером является коалгебра, которая создает список простых чисел. Хитрость заключается в использовании бесконечного списка в качестве носителя. Нашим начальным значением будет список `[2..]`. Следующее значение будет хвостом этого списка со всеми удаленными кратными `2` числами. Это список нечетных чисел, начинающийся с `3`. На следующем шаге мы возьмем хвост этого списка и удалим все числа кратные `3` и т.д. Вы можете узнать здесь решето Эратосфена. Эта коалгебра реализуется с помощью следующей функции:

```
era          :: [Int] -> StreamF Int [Int]
era (p : ns) = StreamF p (filter (notdiv p) ns)
               where notdiv p n = n `mod` p /= 0
```

Анаморфизм для этой коалгебры порождает список простых чисел:

```
primes = ana era [2..]
```

Поток — это бесконечный список, поэтому его можно преобразовать в список Haskell. Для этого мы можем использовать тот же функтор `StreamF`, чтобы сформировать алгебру, и уже над ней можно запустить катаморфизм. Пример катаморфизма, который преобразует поток в список:

```
toListC :: Fix (StreamF e) -> [e]
toListC  = cata  al where
           al :: StreamF e [e] -> [e]
           al (StreamF e a) = e : a
```

Здесь одна и та же неподвижная точка одновременно является инициальной алгеброй и терминальной коалгеброй для одного и того же эндофунктора. Это не всегда так в произвольной категории. Вообще говоря, у эндофунктора может быть много (или ни одной) неподвижных точек. Инициальная алгебра является, так называемой, наименьшей неподвижной точкой, а терминальная коалгебра — наибольшей неподвижной точкой. В Haskell, однако, они обе определены по одной и той же формуле (они совпадают).

Анаморфизм для списков называется *разверткой*. Чтобы создать конечные списки, функтор модифицируется, чтобы создать пару `Maybe`:

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

Значение `Nothing` приведет к прекращению генерации списка.

Интересный частный случай коалгебры связан с линзами. Линзу можно представить как пару, состоящую из получателя и установщика:

```
set :: a -> s -> a
get :: a -> s
```

Здесь `a` обычно представляет собой произведение некоторого типа данных с полем типа `s`. Получатель извлекает значение этого поля, а установщик заменяет это поле новым значением. Эти две функции могут быть объединены в одну:

```
a -> (s, s -> a)
```

Далее, можно переписать эту функцию так:

```
a -> Store s a
```

где

```
data Store s a = Store (s -> a) s
```

Заметим, что это не простой алгебраический функтор, построенный из сумм произведений. Он включает в себя экспоненциал a^s.

Линза — это коалгебра для этого функтора с носителем типа `a`. Мы уже видели, что `Store s` также является комонадой. Оказывается, что хорошо действующая линза соответствует коалгебре, которая совместима со структурой комонады. Мы поговорим об этом в следующем разделе.

Упражнения

1. Реализуйте оценочную функцию для кольца многочленов одной переменной. Вы можете представить многочлен в виде списка коэффициентов перед степенями x. Например, $4x^2 - 1$ будет представлен как (начиная с нулевой степени) `[-1,0,4]`.

2. Обобщите предыдущую конструкцию на многочлены многих независимых переменных, например $x^2y - 3y^3z$.

3. Реализуйте алгебру для кольца матриц 2×2.

4. Определите коалгебру, анаморфизм которой создает список квадратов натуральных чисел.

5. Реализуйте получение списка первых n простых чисел, используя `unfoldr`.

Глава 25

Алгебры для монад

Если мы интерпретируем эндофункторы как способы определения выражений, то алгебры позволяют оценивать их, а монады — формировать и манипулировать ими. Комбинируя алгебры с монадами, мы не только повышаем функциональность, но также получаем возможность ответить на несколько интересных вопросов. Один из таких вопросов касается связи между монадами и сопряжениями. Как мы видели, каждое сопряжение определяет монаду (и комонаду). Возникает вопрос: любая ли монада (комонада) может быть получена из сопряжения? Ответ положительный. Существует целое семейство сопряжений, которые генерируют заданную монаду. Я покажу вам два таких сопряжения.

Давайте проанализируем соответствующие определения. Монада является эндофунктором m, снабженным двумя естественными преобразованиями, которые удовлетворяют некоторым условиям когерентности. Компонентами этих преобразований в точке a являются:

$$\eta_a :: a \to m\, a$$
$$\mu_a :: m\,(m\, a) \to m\, a$$

Алгебра для одного и того же эндофунктора представляет собой выбор конкретного объекта — носителя a — вместе с морфизмом:

$$alg :: m\, a \to a$$

Прежде всего заметим, что алгебра действует в противоположном направлении по сравнению с η_a. Интуиция заключается в том, что η_a создает тривиальное выражение из значения типа a. Первое условие когерентности, которое делает алгебру совместимой с монадой, гарантирует, что вычисление этого выражения с использованием алгебры, носителем которой является a, возвращает нам исходное значение:

$$alg \circ \eta_a = \mathrm{id}_a$$

Второе условие возникает из-за того, что существует два способа вычисления дважды вложенного выражения $m\,(m\, a)$. Сначала мы можем применить μ_a для сглаживания выражения, а затем использовать оценивающую функцию алгебры, или мы можем применить поднятую оценивающую функцию для внутренних выражений, а затем применить оценивающую функцию к результату. Хотелось бы, чтобы эти две стратегии были эквивалентны:

$$alg \circ \mu_a = alg \circ m\, alg$$

Здесь $m\, alg$ — это морфизм, возникающий при подъеме alg с помощью функтора m. Следующие коммутативные диаграммы изображают два вышеупомянутых условия (я заменил m на T в предвкушении обсуждения **T**-алгебр в следующем подразделе):

$$\begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \searrow^{=} & \downarrow{\scriptstyle alg} \\ & & a \end{array} \qquad \begin{array}{ccc} T(Ta) & \xrightarrow{T alg} & Ta \\ \downarrow{\scriptstyle \mu_a} & & \downarrow{\scriptstyle alg} \\ Ta & \xrightarrow[alg]{} & a \end{array}$$

Мы также можем выразить эти условия в нотации Haskell:

```
alg . return = id
alg . join   = alg . fmap alg
```

Давайте рассмотрим небольшой пример. Алгебра для эндофунктора списка содержит некоторый тип `a` и функцию, которая производит `a` из списка `a`. Мы можем выразить эту функцию, используя `foldr`, выбирая тип элемента и тип аккумулятора равным `a`:

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```

Эта конкретная алгебра задается функцией `f` с двумя аргументами, один из них — значение `z`. Функтор списка является монадой, при этом `return` превращает значение в одноэлементный список. Композиция алгебры, здесь `foldr f z`, после `return`, использует `x` для:

```
foldr f z [x] = x `f` z
```

где действие `f` записано в инфиксной нотации. Алгебра совместима с монадой, если для любого `x` выполнено следующее условие когерентности:

```
x `f` z = x
```

Если мы посмотрим на `f` как на бинарный оператор, это условие говорит нам, что `z` является правой единицей.

Второе условие когерентности действует в списке списков. Действие `join` заключается в сцеплении отдельных списков. Затем мы можем свернуть полученный список. С другой стороны, мы можем сначала свернуть отдельные вложенные списки, а затем свернуть полученный список. Опять же, если мы интерпретируем `f` как бинарный оператор, это условие говорит нам, что эта операция ассоциативна. Эти условия, разумеется, выполняются, когда `(a,f,z)` является моноидом.

25.1 T-алгебры

Так как математики предпочитают обозначать свои монады буквой T, они называют алгебры, сочетаемые с ними, **T**-алгебрами. **T**-алгебры для

данной монады T в категории $\mathbf{C}$ образуют категорию, называемую категорией Эйленберга-Мура, которую часто обозначают как $\mathbf{C}^T$. Морфизмы в этой категории являются гомоморфизмами алгебр. Это те же гомоморфизмы, которые были определены для $\mathbf{F}$-алгебр.

$\mathbf{T}$-алгебра — это пара, состоящая из объекта — носителя и оценивающей функции (a, f). Существует очевидный забывающий функтор U^T от $\mathbf{C}^T$ к $\mathbf{C}$, который отображает (a, f) в a. Он также отображает гомоморфизм $\mathbf{T}$-алгебр в соответствующий морфизм между объектами-носителями в $\mathbf{C}$. Вы можете вспомнить из нашего обсуждения сопряжений, что левый сопряженный к забывающему функтору называется свободным функтором.

Левый сопряженный к U^T обозначается F^T. Он отображает объект a из $\mathbf{C}$ к свободной алгебре из $\mathbf{C}^T$. Носителем этой свободной алгебры является $T\,a$. Его оценивающая функция является морфизмом от $T\,(T\,a)$ обратно к $T\,a$. Поскольку T является монадой, мы можем использовать монадический μ_a (`join` в Haskell) в качестве оценивающей функции.

Нам еще предстоит показать, что это $\mathbf{T}$-алгебра. Для этого должны выполняться два условия когерентности:

$$alg \circ \eta_{T\,a} = \mathrm{id}_{T\,a}$$
$$alg \circ \mu_a = alg \circ T\,alg$$

Но это как раз монадические законы, если вы включите μ для алгебры.

Как вы помните, каждое сопряжение определяет монаду. Оказывается, что сопряжение между F^T и U^T определяет саму монаду T, которая использовалась при построении категории Эйленберга-Мура. Поскольку мы можем выполнить эту конструкцию для каждой монады, то заключаем, что каждая монада может быть сгенерирована из сопряжения. Позже я покажу, что есть еще одно сопряжение, которое генерирует ту же монаду.

Вот план: сначала я покажу, что F^T действительно является левым сопряженным к U^T. Я сделаю это, определив единицу и коединицу этого сопряжения, и докажу, что соответствующие треугольные тождества выполняются. Затем я покажу, что монада, созданная этим сопряжением, действительно является нашей первоначальной монадой.

Единицей сопряжения является естественное преобразование:

$$\eta :: I \to U^T \circ F^T$$

Вычислим компонент a этого преобразования. Тождественный функтор дает нам a. Свободный функтор порождает свободную алгебру $(T\,a, \mu_a)$, а забывающий функтор сводит ее к $T\,a$. В итоге мы получаем отображение от a к $T\,a$. Мы просто будем использовать единицу монады T в качестве единицы этого сопряжения.

Давайте рассмотрим коединицу:

$$\varepsilon :: F^T \circ U^T \to I$$

Вычислим ее компоненту в некоторой **T**-алгебре (a, f). Забывающий функтор забывает f, а свободный функтор производит пару $(T\,a, \mu_a)$. Итак, чтобы определить компоненту коединицы ε в точке (a, f), нам нужен правый морфизм в категории Эйленберга-Мура или гомоморфизм **T**-алгебр:

$$(T\,a, \mu_a) \to (a, f)$$

Такой гомоморфизм должен отображать носитель $T\,a$ в a. Давайте просто воскресим забытую оценивающую функцию f. На этот раз мы будем использовать ее как гомоморфизм **T**-алгебр. В самом деле, ту же коммутативную диаграмму, которая делает функцию f **T**-алгеброй, можно переинтерпретировать, чтобы показать, что она является гомоморфизмом **T**-алгебр:

$$\begin{array}{ccc} T(Ta) & \xrightarrow{Tf} & Ta \\ {\scriptstyle \mu_a}\downarrow & & \downarrow{\scriptstyle f} \\ Ta & \xrightarrow[f]{} & a \end{array}$$

Таким образом, мы определили компоненту естественного преобразования ε в точке (a, f) (объект в категории **T**-алгебр) как f.

Для завершения сопряжения нам также нужно показать, что единица и коединица удовлетворяют треугольным тождествам:

$$\begin{array}{ccc} Ta & \xrightarrow{T\eta_a} & T(Ta) \\ & \searrow^{=} & \downarrow{\scriptstyle alg} \\ & & Ta \end{array} \qquad \begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \searrow^{=} & \downarrow{\scriptstyle f} \\ & & a \end{array}$$

Первое следует из закона единицы для монады T. Второе — это просто закон $\mathbf{T}$-алгебры (a, f).

Мы установили, что два функтора образуют сопряжение:

$$F^T \dashv U^T$$

Каждое сопряжение порождает монаду. Круиз

$$U^T \circ F^T$$

является эндофунктором в $\mathbf{C}$, который порождает соответствующую монаду. Давайте посмотрим, каково его действие на объект a. Свободная алгебра, порожденная F^T, есть $(T\,a, \mu_a)$. Забывающий функтор U^T отбрасывает оценивающую функцию. Итак, на самом деле, мы получаем:

$$U^T \circ F^T = T$$

Как и ожидалось, единицей сопряжения является единица монады T.

Вы можете помнить, что коединица сопряжения производит монадическое умножение по формуле:

$$\mu = R \circ \varepsilon \circ L$$

Это горизонтальная композиция трех естественных преобразований, два из которых представляют собой отображение тождественных естественных преобразований, соответственно, L к L и R к R. Коединица, находящаяся в середине, является естественным преобразованием, компонентой которой в алгебре (a, f) является f.

Вычислим компоненту μ_a. Сначала по горизонтали компонуем ε после F^T, что приводит к компоненте ε в $F^T\,a$. Так как F^T применяет a к алгебре $(T\,a, \mu_a)$, а ε выбирает оценивающую функцию, мы получаем μ_a. Горизонтальная композиция слева с U^T ничего не меняет, так как действие U^T на морфизмах тривиально. Таким образом, действительно, μ, полученное из сопряжения, совпадает с μ исходной монады T.

25.2 Категория Клейсли

Мы уже встречались с категорией Клейсли. Это категория, построенная из другой категории $\mathbf{C}$ и монады T. Мы будем обозначать эту категорию $\mathbf{C}_T$. Объекты в категории $\mathbf{C}_T$ являются объектами $\mathbf{C}$, но морфизмы отличаются. Морфизм $f_{\mathbf{K}}$ от a к b в категории Клейсли соответствует морфизму f от a к $T\,b$ в исходной категории. Мы называем этот морфизм стрелкой Клейсли от a к b.

Композиция морфизмов в категории Клейсли определяется в терминах монадической композиции стрелок Клейсли. К примеру, давайте составим композицию $g_{\mathbf{K}}$ после $f_{\mathbf{K}}$. В категории Клейсли мы имеем:

$$f_{\mathbf{K}} :: a \to b$$
$$g_{\mathbf{K}} :: b \to c$$

что в категории $\mathbf{C}$ соответствует:

$$f :: a \to T\,b$$
$$g :: b \to T\,c$$

Мы определяем композицию:

$$h_{\mathbf{K}} = g_{\mathbf{K}} \circ f_{\mathbf{K}}$$

как стрелку Клейсли в $\mathbf{C}$

$$h :: a \to T\,c$$
$$h = \mu \circ (T\,g) \circ f$$

На Haskell мы запишем ее как:

```
h = join . fmap g . f
```

Существует функтор F от $\mathbf{C}$ к $\mathbf{C}_T$, который действует тривиально на объекты. На морфизмах он отображает f из $\mathbf{C}$ к морфизмам из $\mathbf{C}_T$, создавая стрелку Клейсли, которая обогащает возвращаемое значение f. При заданном морфизме:

$$f :: a \to b$$

он создает морфизм в $\mathbf{C}_T$ с соответствующей стрелкой Клейсли:

$$\eta \circ f$$

На Haskell мы запишем его как:

```
return . f
```

Мы также можем определить функтор G от $\mathbf{C}_T$ обратно к $\mathbf{C}$. Он принимает объект a из категории Клейсли и отображает его к объекту $T\,a$ из $\mathbf{C}$. Его действие на морфизм $f_{\mathbf{K}}$, соответствующий стрелке Клейсли:

$$f :: a \to T\,b$$

есть морфизм в $\mathbf{C}$:

$$T\,a \to T\,b$$

задаваемый сначала поднятием f, а затем применением μ:

$$\mu_{T\,b} \circ T\,f$$

В нотации Haskell это будет выглядеть следующим образом:

```
G f_T = join . fmap f
```

Вы можете признать это определением монадического связывания в терминах `join`.

Легко видеть, что два функтора образуют сопряжение:

$$F \dashv G$$

а их композиция $G \circ F$ воспроизводит исходную монаду T.

Итак, это второе сопряжение, которое производит ту же монаду. Фактически существует целая категория сопряжений $\mathbf{Adj}(\mathbf{C}, T)$, которые приводят к одной и той же монаде T на $\mathbf{C}$. Сопряжением Клейсли, которое мы только что рассмотрели, является инициальный объект в этой категории, а сопряжением Эйленберга-Мура является терминальный объект.

25.3 Коалгебры для комонад

Аналогичные построения могут быть выполнены для любой комонады W. Мы можем определить категорию коалгебр, совместимых с комонадой. Они делают следующие диаграммы коммутативными:

$$\begin{array}{ccc} a & \xleftarrow{\epsilon_a} & W a \\ & \diagdown\!\!\diagdown & \uparrow{\scriptstyle coa} \\ & & a \end{array} \qquad \begin{array}{ccc} W(W a) & \xleftarrow{W coa} & W a \\ {\scriptstyle \delta_a}\uparrow & & \uparrow{\scriptstyle coa} \\ W a & \xleftarrow[coa]{} & a \end{array}$$

где coa — морфизм кооценки коалгебры, носителем которой является a:

$$coa :: a \to W\ a$$

а ε и δ — два естественных преобразования, определяющие комонаду (в Haskell их компоненты называются `extract` и `duplicate`).

Существует очевидный забывающий функтор U^W от категории этих коалгебр к $\mathbf{C}$. Он просто забывает о ко-оценке. Рассмотрим его правый сопряженный F^W.

$$U^W \dashv F^W$$

Правый сопряженный к забывающему функтору, называется ко-свободным функтором. F^W генерирует косвободные коалгебры. Он приписывает объекту a из $\mathbf{C}$ коалгебру $(W\ a, \delta_a)$. Сопряжение воспроизводит исходную комонаду как составную $U^W \circ F^W$.

Аналогично, мы можем построить ко-категорию Клейсли с ко-стрелками Клейсли и воссоздать комонаду из соответствующего сопряжения.

25.4 Линзы

Вернемся к обсуждению линз. Линзу можно записать в виде коалгебры:

$$coalg_s :: a \to Store\, s\, a$$

для функтора $Store\, s$:

```
data Store s a = Store (s -> a) s
```

Эта коалгебра также может быть выражена в виде пары функций:

$$set :: a \to s \to a$$
$$get :: a \to s$$

(Напомню, что a обозначает «все», а s — «небольшую» часть). В терминах этой пары имеем:

$$coalg_s\, a = Store\,(set\, a)\,(get\, a)$$

Здесь a — значение типа a. Обратите внимание, что частичное применение set является функцией $s \to a$.

Мы также знаем, что $Store\, s$ является комонадой:

```
instance Comonad (Store s) where
   extract (Store f s)   = f s
   duplicate (Store f s) = Store (Store f) s
```

Вопрос: при каких условиях линза является коалгеброй для этой комонады? Первое условие когерентности:

$$\varepsilon_a \circ coalg = \mathrm{id}_a$$

транслируется в:

$$set\, a\,(get\, a) = a$$

Это закон линзы, который выражает тот факт, что если вы установите поле структуры a в его предыдущее значение, ничего не изменится.

Второе условие:

$$fmap\, coalg \circ coalg = \delta_a \circ coalg$$

требует немного больше работы. Во-первых, вспомним определение `fmap` для функтора `Store`:

```
fmap g (Store f s) = Store (g . f) s
```

Применение $fmap\,coalg$ к результату $coalg$ дает нам:

```
Store (coalg . set a) (get a)
```

С другой стороны, применение `duplicate` к результату $coalg$ приводит к:

$$Store\,(Store\,(set\,a))\,(get\,a)$$

Для того чтобы эти два выражения были равны, обе функции в $Store$ должны быть равны при действии на произвольное s:

$$coalg\,(set\,a\,s) = Store\,(set\,a)\,s$$

Раскрывая $coalg$, получаем:

$$Store\,(set\,(set\,a\,s))\,(get\,(set\,a\,s)) = Store\,(set\,a)\,s$$

Это эквивалентно двум оставшимся законам линзы. Первый из них:

$$set\,(set\,a\,s) = set\,a$$

говорит, что установка значения поля дважды имеет такой же эффект, как и установка его один раз. Второй закон:

$$get\,(set\,a\,s) = s$$

говорит, что получение значения поля, которое было установлено в s, дает s.

Другими словами, хорошо действующая линза действительно является комонадой коалгебры для функтора $Store$.

Упражнения

1. Каково действие свободного функтора $F :: C \to C^T$ на морфизмах. Подсказка: используйте условие естественности для монадического μ.

2. Определите сопряжение:

$$U^W \dashv F^W$$

3. Докажите, что вышеприведенное сопряжение воспроизводит исходную комонаду.

Глава 26

Концы и ко-концы

Существует множество интуиций, которые мы можем соотнести с морфизмами в категории, но все мы можем согласиться с тем, что если существует морфизм от объекта a к объекту b, то эти два объекта каким-то образом «связаны». Морфизм в этом смысле является доказательством этой связи. Это хорошо видно в любой категории упорядоченных множеств, где морфизм является отношением. В общем, может быть много «доказательств» одного и того же отношения между двумя объектами. Эти доказательства образуют множество, которое мы называем $\hom$-множеством. Когда мы меняем объекты, мы получаем отображение пар объектов на множества «доказательств». Это отображение функториально-контравариантно в первом аргументе и ковариантно во втором. Мы можем рассматривать это как установление глобальных отношений между объектами в категории. Эта связь описывается гомоморфизмом:

$$\mathbf{C}(-,=) :: \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$$

В общем, любой подобный функтор может быть истолкован как установление отношения между объектами в категории. Отношение может также включать две разные категории $\mathbf{C}$ и $\mathbf{D}$. Функтор, который описывает такое отношение, называется профунктором и имеет следующую сигнатуру:

$$p :: \mathbf{D}^{op} \times \mathbf{C} \to \mathbf{Set}$$

Математики говорят, что это профунктор от $\mathbf{C}$ к $\mathbf{D}$ (обратите внимание на инверсию) и используют в качестве символа для этого сокращенную

стрелку:

$$\mathbf{C} \nrightarrow \mathbf{D}$$

Вы можете думать о профункторе как о релевантном доказательстве отношения между объектами $\mathbf{C}$ и объектами $\mathbf{D}$, где элементы множества $\mathbf{Set}$ символизируют доказательства отношения. Всякий раз, когда $p\,a\,b$ пусто, связь между a и b отсутствует. Следует иметь в виду, что отношения не обязательно должны быть симметричными.

Другая полезная интуиция — это обобщение идеи о том, что эндофунктор является контейнером. Значение профунктора типа $p\,a\,b$ может тогда считаться контейнером, содержащим все b, которые вводятся с помощью элементов типа a. В частности, элемент hom-профунктора является функцией от a к b.

В Haskell, профунктор определяется как конструктор типа `p` с двумя аргументами, оснащенный методом, называемым `dimap`, который поднимает пару функций, причем первая идет в «неправильном» направлении:

```
class Profunctor p where
   dimap :: (c -> a) -> (b -> d) -> p a b
                                  -> p c d
```

Функториальность профуктора означает, что если у нас есть доказательство того, что `a` связано с `b`, то мы получаем доказательство того, что `c` связано с `d`, если существует морфизм от `c` к `a` и морфизм от `b` к `d`. Или же, мы можем думать о первой функции как о переводе новых ключей в старые ключи, а о второй функции — как о модификации содержимого контейнера.

Для профункторов, действующих в рамках одной категории, мы можем извлечь довольно много информации из диагональных элементов типа $p\,a\,a$. Мы можем доказать, что b связано с c, если мы имеем пару морфизмов $b \to a$ и $a \to c$. Более того, мы можем использовать один морфизм для достижения внедиагональных значений. Например, если мы имеем морфизм $f :: a \to b$, мы можем поднять пару $\langle f, \mathrm{id}_b \rangle$, чтобы перейти от $p\,b\,b$ к $p\,a\,b$:

```
dimap f id (p b b) :: p a b
```

или можем поднять пару $\langle \text{id}_a, f\rangle$, чтобы перейти от $p\,a\,a$ к $p\,a\,b$:

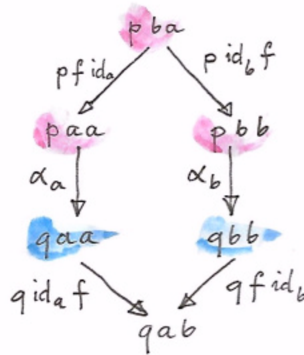
```
dimap id f (p a a) :: p a b
```

26.1 Диестественные преобразования

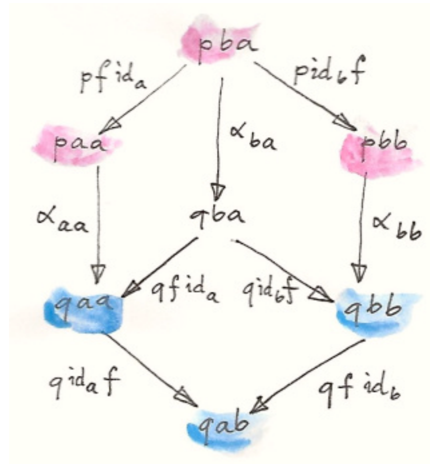
Поскольку профункторы являются функторами, мы можем стандартным образом определить естественные преобразования между ними. Во многих случаях, однако, достаточно определить отображение между диагональными элементами двух профукторов. Такое преобразование называется диестественным преобразованием, если оно удовлетворяет коммутирующим условиям, которые отражают два способа соединения диагональных элементов с недиагональными. Диестественное преобразование между двумя профункторами p и q, которые являются членами категории функторов $[\mathbf{C}^{op}\times\mathbf{C}, \mathbf{Set}]$, является семейством морфизмов:

$$\alpha_a :: p\,a\,a \to q\,a\,a$$

для которого следующая диаграмма коммутативна, для любой $f :: a \to b$:



Заметьте, что это строго слабее условия естественности. Если бы α было естественным преобразованием в $[\mathbf{C}^{op}\times\mathbf{C}, \mathbf{Set}]$, приведенная выше диаграмма могла бы быть построена из двух квадратов естественности и одного условия функториальности (профунктор q, сохраняющий композицию):



Обратите внимание, что компонент естественного преобразования α из $[\mathbf{C}^{op} \times \mathbf{C}, \mathbf{Set}]$ индексируется парой объектов $\alpha_{a\,b}$. С другой стороны, диестественное преобразование индексируется одним объектом, поскольку оно отображает только диагональные элементы соответствующих профункторов.

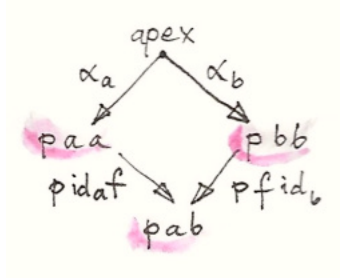
26.2 Концы

Теперь мы готовы перейти от «алгебры» к тому, что можно считать «исчислением» в теории категорий. Исчисление концов (и ко-концов) заимствует идеи и даже некоторые обозначения из традиционного исчисления. В частности, ко-конец может пониматься как бесконечная сумма или интеграл, тогда как конец подобен бесконечному произведению. Есть даже нечто, напоминающее дельта-функцию Дирака.

Конец — это обобщение предела, где функтор заменяется профунктором. Вместо конуса мы имеем клин. Основание клина образовано диагональными элементами профунктора p. Вершина клина — это объект (в данном случае множество, поскольку мы рассматриваем $\mathbf{Set}$-значащие профункторы), а стороны — это семейство функций, отображающих вершину на множества в базе. Вы можете рассматривать это семейство как одну полиморфную функцию — функцию, полиморфную по типу возвращаемого значения:

$$\alpha :: \forall\, a\,.\, apex \to p\; a\; a$$

Внутри клина мы не рассматриваем какие-либо функции, которые соединяли бы вершины базы. Однако, как мы видели ранее, для любого морфизма $f :: a \to b$ в $\mathbf{C}$ мы можем связать и $p\,a\,a$ и $p\,b\,b$ с общим множеством $p\,a\,b$. Поэтому мы утверждаем, что следующая диаграмма коммутативна:



Это называется условием клина. Его можно записать так:

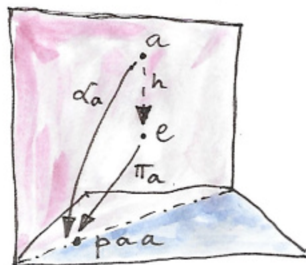
$$p\,\mathrm{id}_a\,f \circ \alpha_a = p\,f\,\mathrm{id}_b \circ \alpha_b$$

Или, используя нотацию Haskell:

```
dimap id f . alpha = dimap f id . alpha
```

Теперь мы можем исходить из универсальной конструкции и определить конец p как универсальный клин — множество e вместе с семейством функций π таких, что для любого другого клина с вершиной a и семейства α существует единственная функция $h :: a \to e$, что делает все треугольники коммутативными:

$$\pi_a \circ h = \alpha_a$$



Символом для конца является знак интеграла, с «переменной интегрирования» в позиции нижнего индекса:

$$\int_c p\,c\,c$$

Компоненты π называются проекционными отображениями для конца:

$$\pi_a :: \int_c p\,c\,c \to p\,a\,a$$

Заметим, что если $\mathbf{C}$ — дискретная категория (нет других морфизмов, кроме тождественных), то конец — это всего лишь глобальное произведение всех диагональных элементов p всей категории $\mathbf{C}$. Позже я покажу вам, что в более общем случае существует отношение между концом и этим произведением через уравнитель.

На Haskell формула конца переводится непосредственно в квантор всеобщности:

```
forall a. p a a
```

Строго говоря, это просто произведение всех диагональных элементов p, но условие клина выполняется автоматически из-за параметричности[1]. Для любой функции $f :: a \to b$ условие клина записывается так:

```
dimap f id . pi = dimap id f . pi
```

или, с аннотациями типа:

$$\texttt{dimap f id}_b \texttt{ . pi}_b \texttt{ = dimap id}_a \texttt{ f . pi}_a$$

где обе стороны формулы имеют тип:

```
Profunctor p => (forall c. p c c) -> p a b
```

[1] https://bartoszmilewski.com/2017/04/11/profunctor-parametricity/

а pi — полиморфная проекция:

```
pi  :: Profunctor p => forall c.
                (forall a. p a a) -> p c c
pi e = e
```

Здесь вывод типа автоматически выбирает правый компонент e.

Подобно тому, как мы смогли выразить весь набор условий коммутирования для конуса как одного естественного преобразования, аналогично можно сгруппировать все условия клина в одно естественное преобразование. Для этого нам понадобится обобщение постоянного функтора Δ_c на константный профунктор, который отображает все пары объектов на один объект c, а все пары морфизмов — в тождественный морфизм для этого объекта. Клин — это естественное преобразование от этого функтора к профунктору p. Действительно, шестиугольник диестественности сжимается до алмаза клина, когда мы понимаем, что Δ_c поднимает все морфизмы до одной тождественной функции.

Концы также могут быть определены для целевых категорий, отличных от **Set**, но здесь мы рассматриваем только **Set**-значащие профункторы и их концы.

26.3 Концы как уравнители

Условие коммутирования в определении конца может быть записано с помощью уравнителя. Сначала давайте определим две функции (я использую нотацию Haskell, потому что в этом случае математическая нотация кажется менее удобной для пользователя). Эти функции соответствуют двум сходящимся ветвям условия клина:

```
lambda      :: Profunctor p => p a a ->
                             (a -> b) -> p a b
lambda paa f = dimap id f paa

rho      :: Profunctor p => p b b ->
                         (a -> b) -> p a b
rho pbb f = dimap f id pbb
```

Обе функции отображают диагональные элементы профунктора `p` в полиморфные функции типа:

```
type ProdP p = forall a b. (a -> b) -> p a b
```

Эти функции имеют разные типы. Тем не менее, мы можем унифицировать их типы, если сформируем один большой тип произведения, собирая вместе все диагональные элементы `p`:

```
newtype DiaProd p = DiaProd (forall a. p a a)
```

Функции `lambda` и `rho` вызывают два отображения из этого типа произведения:

```
lambdaP :: Profunctor p => DiaProd p ->
                                      ProdP p
lambdaP (DiaProd paa) = lambda paa

rhoP :: Profunctor p => DiaProd p -> ProdP p
rhoP (DiaProd pbb) = rho pbb
```

Конец `p` — это уравнитель этих двух функций. Помните, что уравнитель выбирает наибольшее подмножество, в котором две функции равны. В этом случае он выбирает подмножество произведения всех диагональных элементов, для которых клиновые диаграммы коммутативны.

26.4 Естественные преобразования как концы

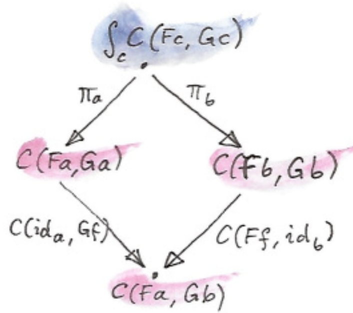
Самым важным примером конца является множество естественных преобразований. Естественным преобразованием между двумя функторами F и G является семейство морфизмов, отобранных из $\hom$-множеств вида $\mathbf{C}(F\,a, G\,a)$. Если бы не условие естественности, множество естественных преобразований было бы просто произведением всех этих $\hom$-множеств. Фактически, на Haskell это:

```
forall a. f a -> g a
```

Причина, по которой это работает в Haskell, заключается в том, что естественность следует из параметричности. Однако за пределами Haskell не все диагональные сечения на таких hom-множествах приводят к естественным преобразованиям. Но обратите внимание, что отображение:

$$\langle a, b\rangle \to \mathbf{C}\,(F\,a, G\,b)$$

является профунктором, поэтому имеет смысл изучить конец этого отображения. Вот условие клина:



Давайте просто выберем один элемент из множества $\int_c \mathbf{C}(F\,a, G\,a)$. Следующие две проекции сопоставят этот элемент с двумя компонентами конкретного преобразования, назовем их:

$$\tau_a :: F\,a \to G\,a$$
$$\tau_b :: F\,b \to G\,b$$

В левой ветви мы поднимаем пару морфизмов $\langle \mathrm{id}_a, G\,f\rangle$, используя hom-функтор. Вы можете вспомнить, что такой подъем осуществляется как одновременный пред- и посткомпозиционный. При воздействии на τ_a поднятая пара дает нам:

$$G\,f \circ \tau_a \circ \mathrm{id}_a$$

Другая ветвь диаграммы дает нам:

$$\mathrm{id}_b \circ \tau_b \circ F\,f$$

Их равенство, требуемое условием клина, есть не что иное, как условие естественности для τ.

26.5 Ко-концы

Как и ожидалось, двойственный к концу называется *ко-концом*. Его построение основано на двойственном понятии клина, называемому *ко-клином*.

Обозначением для ко-конца является знак интеграла с «переменной интегрирования» в позиции надстрочного индекса:

$$\int^c p\, c\, c$$

Подобно тому, как конец связан с произведением, ко-конец связан с копроизведением или суммой (в этом отношении он напоминает интеграл, который является пределом суммы). Вместо проекций, мы имеем инъекции, идущие от диагональных элементов профунктора к ко-концу. Если бы не условия ко-клина, мы могли бы сказать, что ко-конец профунктора p — это либо $p\, a\, a$, либо $p\, b\, b$, либо $p\, c\, c$, и т.д. Или мы могли бы сказать, что существует такое a, для которого ко-конец — это просто множество $p\, a\, a$. Универсальный квантификатор, который мы использовали в определении конца, превращается в квантор существования для ко-конца.

Вот почему на псевдо-Haskell мы определяем ко-конец так:

```
exists a. p a a
```

Стандартный способ кодирования экзистенциальных кванторов в Haskell заключается в использовании универсальных квантифицированных конструкторов данных. Таким образом, можно определить:

```
data Coend p = forall a. Coend p a a
```

Логика этого состоит в том, что должна существовать возможность построить ко-конец, используя значение любого из семейства типов `p a a`, независимо от того, какой `a` мы выбрали.

Так же, как конец может быть определен с помощью уравнителя, ко-конец может быть описан с помощью ко-уравнителя. Все условия ко-клина можно обобщить, взяв одно огромное копроизведение `p a b` для всех возможных функций `b -> a`. На Haskell это может быть выражено экзистенциальным типом:

```
data SumP p = forall a b. SumP (b -
> a) (p a b)
```

Имеются два способа вычислить этот тип-сумму: поднять функцию с помощью `dimap` и применить ее к профунктору `p`:

```
lambda, rho         :: Profunctor p =>
                              SumP p -> DiagSum p
lambda (SumP f pab) = DiagSum (dimap f id pab)
rho (SumP f pab)    = DiagSum (dimap id f pab)
```

где `DiagSum` — сумма диагональных элементов `p`:

```
data DiagSum p = forall a. DiagSum (p a a)
```

Ко-уравнителем этих двух функций является ко-конец. Ко-уравнитель получается из `DiagSum p` путем определения значений, полученных применением `lambda` или `rho` к одному и тому же аргументу. Здесь, аргумент — это пара, состоящая из функции `b -> a` и элемента `p a b`. Применение `lambda` и `rho` создает два потенциально разных значения типа `DiagSum p`. В ко-конце эти два значения идентифицируются, что автоматически удовлетворяет условию ко-клина.

Процесс идентификации связанных элементов множестве формально известен как взятие частного. Для определения частного можно воспользоваться *отношением эквивалентности* $\sim$, отношением, которое является рефлексивным, симметричным и транзитивным:

$$a \sim a$$
$$\text{если } a \sim b, \text{ то } b \sim a$$
$$\text{если } a \sim b \text{ и } b \sim c, \text{ то } a \sim c$$

Такое отношение разбивает множество на классы эквивалентности. Каждый класс состоит из элементов, которые связаны друг с другом. Мы формируем фактор-множество, выбирая по одному представителю из каждого класса. Классическим примером является определение рациональных чисел, как пар целых чисел со следующим отношением эквивалентности:

$$(a, b) \sim (c, d) \text{ тогда и только тогда, когда } a * d = b * c$$

Легко проверить, что это является отношением эквивалентности. Пара (a, b) интерпретируется как дробь $\frac{a}{b}$, и выделяются дроби, числитель и знаменатель которых имеют общий делитель. Рациональное число — это класс эквивалентности таких дробей.

Вы можете вспомнить из нашего предыдущего обсуждения пределов и копределов, что $\hom$-функтор является непрерывным, т.е. сохраняет пределы. Двойственно, контравариантный $\hom$-функтор превращает копределы в пределы. Эти свойства могут быть обобщены на концы и ко-концы, что является обобщением пределов и копределов соответственно. В частности, мы получаем очень полезную идентификацию для конвертирования ко-концов в концы:

$$\mathbf{Set}(\int^x p\, x\, x, c) \cong \int_x \mathbf{Set}(p\, x\, x, c)$$

На псевдо-Haskell это выглядит так:

```
(exists x. p x x) -> c ≅ forall x. p x x -> c
```

Это показывает нам, что функция, которая принимает экзистенциальный тип, эквивалентна полиморфной функции. И это имеет смысл, потому что такая функция должна быть подготовлена для обработки любого из типов, которые могут быть закодированы в экзистенциальном типе — тот же принцип, согласно которому функция, которая принимает тип-сумму, должна быть реализована как оператор выбора с кортежом обработчиков, по одному для каждого типа, присутствующего в сумме. Здесь тип-сумма заменяется ко-концом, а семейство обработчиков становится концом или полиморфной функцией.

26.6 Лемма ниндзя Йонеды

Множество естественных преобразований, которое фигурирует в лемме Йонеды, может быть закодировано с использованием конца, что приводит к следующей формулировке:

$$\int_z \mathbf{Set}(\mathbf{C}(a, z), F\, z) \cong F\, a$$

Существует также двойственная формула:

$$\int^z \mathbf{C}(a, z) \times F\, z \cong F\, a$$

Это тождество сильно напоминает формулу для дельта-функции Дирака (функцию $\delta(a - z)$, точнее, распределение, имеющее бесконечный пик при $a = z$). Здесь $\hom$-функтор играет роль дельта-функции.

Вместе эти две идентичности иногда называют леммой ниндзя Йонеды.

Чтобы доказать вторую формулу, воспользуемся следствием вложения Йонеды, которое утверждает, что два объекта изоморфны тогда и только тогда, когда их $\hom$-функторы изоморфны. Другими словами, $a \cong b$ тогда и только тогда, когда существует естественное преобразование типа:

$$[\mathbf{C},\mathbf{Set}](\mathbf{C}(a, -),\mathbf{C}(b, =))$$

которое является изоморфизмом.

Сначала мы вставим левую часть тождества, которую хотим доказать, внутрь $\hom$-функтора, который идет к какому-нибудь произвольному объекту c:

$$\mathbf{Set}(\int^z \mathbf{C}(a, z) \times F\, z, c)$$

Используя непрерывность аргумента, мы можем заменить ко-конец на конец:

$$\int_z \mathbf{Set}(\mathbf{C}(a, z) \times F\, z, c)$$

Теперь мы можем воспользоваться сопряжением между произведением и экспоненциалом:

$$\int_z \mathbf{Set}(\mathbf{C}(a, z), c^{(F\,z)})$$

Мы можем «выполнить интегрирование», используя лемму Йонеды, чтобы получить:

$$c^{(F\,a)}$$

(обратите внимание, что была использована контравариантная версия леммы Йонеды, поскольку функтор $c^{(F\,z)}$ контравариантен в z). Этот экспоненциальный объект изоморфен hom-множеству:

$$\mathbf{Set}(F\,a, c)$$

Наконец, мы воспользуемся вложением Ионеды, чтобы прийти к изоморфизму:

$$\int^z \mathbf{C}(a, z) \times F\,z \cong F\,a$$

26.7 Композиция профункторов

Давайте еще раз рассмотрим идею о том, что профунктор описывает отношение — точнее, доказательно-релевантное отношение, а это значит, что множество $p\,a\,b$ представляет собой множество доказательств того, что a связано с b. Если у нас есть два отношения p и q, мы можем попытаться их соединить. Мы будем говорить, что a связано с b через композицию q после p, если существует промежуточный объект c такой, что и $q\,b\,c$, и $p\,c\,a$ непусты. Доказательствами этого нового отношения являются все пары доказательств индивидуальных отношений. Поэтому, понимая, что квантор существования соответствует ко-концу, а декартово произведение двух множеств соответствует «парам доказательств», мы можем определить композицию профункторов, используя следующую формулу:

$$(q \circ p)\,a\,b = \int^c p\,c\,a \times q\,b\,c$$

Эквивалентное определение из `Data.Profunctor.Composition` в Haskell, после некоторого переименования, можно представить в виде:

```
data Procompose q p a b where
   Procompose :: q a c -> p c b ->
                 Procompose q p a b
```

Здесь используется обобщенный алгебраический тип данных, или синтаксис GADT, в котором переменная свободного типа (здесь `c`) автоматически экзистенциально квантифицируется. Конструктор данных (некаррированный) `Procompose`, таким образом, эквивалентен:

```
exists c. (q a c, p c b)
```

Единица таким образом определенной композиции является $\mathbf{hom}$-функтором — это непосредственно следует из леммы ниндзя Йонеды. Поэтому имеет смысл задаться вопросом, существует ли категория, в которой профункторы выступают в роли морфизмов? Ответ положительный, с оговоркой, что оба закона, ассоциативности и идентичности, для композиции профункторов имеют место только с точностью до естественного изоморфизма. Категория, в которой законы действительны с точностью до изоморфизма, называется бикатегорией (которая является более общей, чем 2-категория). Итак, у нас есть бикатегория **Prof**, в которой объекты являются категориями, морфизмы являются профункторами, а морфизмы между морфизмами (также известные как 2-клетки) являются естественными преобразованиями. На самом деле, можно пойти еще дальше, потому что помимо профункторов, у нас также имеются регулярные функторы, в качестве морфизмов между категориями. Категория, которая имеет два типа морфизмов, называется двойной категорией.

Профунторы играют важную роль в библиотеке линз и в библиотеке стрелок Haskell.

Упражнения

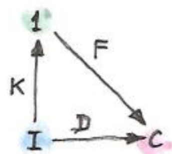
1. Запишите в явном виде условие ко-клина для ко-конца.

Глава 27

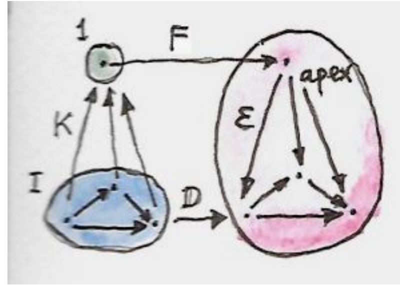
Расширения Кана

До сих пор мы в основном работали с одной категорией или парой категорий. В некоторых случаях это было слишком сложно. Например, при определении предела в категории **C** мы вводили категорию индекса **I** в качестве конструкции для шаблона, который формировал бы базис для конусов. Было бы разумно ввести другую категорию — тривиальную — в качестве шаблона для вершины конуса. Вместо этого мы использовали константный функтор $\Delta_{\mathbf{c}}$ от **I** до **C**.

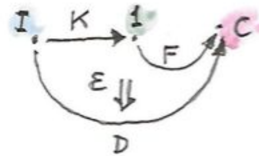
Пришло время исправить эту неловкость. Определим предел по трем категориям. Начнем с функтора D от индексной категории **I** к **C**. Это функтор, который выбирает основание конуса — функтор диаграммы.



Новое добавление — это категория **1**, которая содержит один объект (и один тождественный морфизм). Существует только один возможный функтор K от **I** к этой категории. Он отображает все объекты на единственный объект в **1**, а все морфизмы — на тождественный морфизм. Любой функтор F от **1** к **C** выбирает потенциальную вершину для нашего конуса.

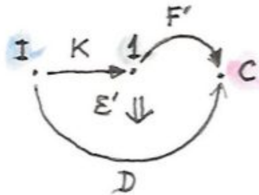


Конус является естественным преобразованием ε от $F \circ K$ к D. Заметим, что $F \circ K$ делает то же, что и исходный $\Delta_{\mathbf{c}}$. На следующем рисунке показано это преобразование.



Теперь мы можем определить универсальное свойство, которое выбирает «лучший» такой функтор F. Этот F отобразит $\mathbf{1}$ в объект, являющийся пределом D в $\mathbf{C}$, а естественное преобразование ε от $F \circ K$ к D обеспечит соответствующие проекции. Этот универсальный функтор называется правым расширением Кана D вдоль K и обозначается $\mathrm{Ran}_K D$.

Сформулируем универсальное свойство. Предположим, у нас есть другой конус — это другой функтор F' вместе с естественным преобразованием ε' от $F' \circ K$ к D.

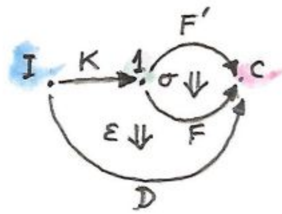


Если существует расширение Кана $F = \mathrm{Ran}_K D$, то к нему должно идти единственное естественное преобразование σ от F', такое, что ε' факто-

ризуется через ε, то есть:

$$\varepsilon' = \varepsilon \mathbin{.} (\sigma \circ K)$$

Здесь $\sigma \circ K$ — горизонтальная композиция двух естественных преобразований (одно из которых является тождественным естественным преобразованием на K). Тогда приведенное преобразование вертикально скомпоновано с ε.



В компонентах, воздействуя на объект i из $\mathbf{I}$, мы получаем:

$$\varepsilon'_i = \varepsilon_i \circ \sigma_{K\,i}$$

В нашем случае σ имеет только одну компоненту, соответствующую единственному объекту из $\mathbf{1}$. Таким образом, на самом деле это единственный морфизм от вершины конуса, определяемой F', к вершине универсального конуса, определенной $\mathrm{Ran}_K D$. Условия коммутации — это те требования, которые требуются в определении предела.

Но, что важно, мы можем заменить тривиальную категорию $\mathbf{1}$ произвольной категорией $\mathbf{A}$, а определение правого расширения Кана останется в силе.

27.1 Правое расширение Кана

Правое расширение Кана функтора $D :: \mathbf{I} \to \mathbf{C}$ вдоль функтора $K :: \mathbf{I} \to \mathbf{A}$ является функтором $F :: \mathbf{A} \to \mathbf{C}$ (обозначаемым $\mathrm{Ran}_K D$) вместе с естественным преобразованием

$$\varepsilon :: F \circ K \to D$$

такими, что для другого функтора $F' :: \mathbf{A} \to \mathbf{C}$ и естественного преобразования

$$\varepsilon' :: F' \circ K \to D$$

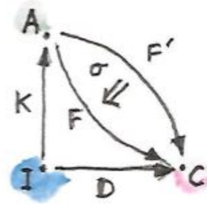
существует единственное естественное преобразование

$$\sigma :: F' \to F$$

факторизующее ε':

$$\varepsilon' = \varepsilon \,.\, (\sigma \circ K)$$

Это довольно не легко воспринимаемое определение, но оно может быть визуализировано в этой элегантной диаграмме:

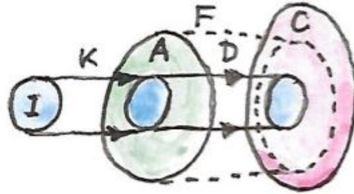


Интересный способ взглянуть на это — заметить, что в некотором смысле расширение Кана действует как обратное к «умножению функторов». Некоторые авторы используют, насколько это возможно, обозначение D/K для $\mathrm{Ran}_K D$. Действительно, в этих обозначениях определение ε, называемое также коединицей правого расширения Кана, выглядит как простое сокращение члена K:

$$\varepsilon :: D/K \circ K \to D$$

Существует другая интерпретация расширений Кана. Предположим, что функтор K вкладывает категорию $\mathbf{I}$ внутрь $\mathbf{A}$. В простейшем случае $\mathbf{I}$ может быть просто подкатегорией $\mathbf{A}$. Имеется функтор D, который отображает $\mathbf{I}$ в $\mathbf{C}$. Можем ли мы расширить D до функтора F, который определен на всем $\mathbf{A}$? В идеале такое расширение сделало бы композицию $F \circ K$ изоморфной D. Другими словами, F будет расширять область действия D до $\mathbf{A}$. Но полномасштабного изоморфизма обычно чересчур много для этого, поэтому мы можем использовать только половину его,

а именно одностороннее естественное преобразование ε от $F \circ K$ к D (левое расширение Кана выбирает другое направление).



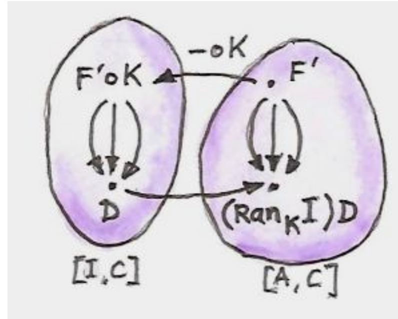
Конечно, изображение внедрения нарушается, когда функтор K неинъективен на объектах или не точен на $\hom$-множествах, как в примере предела. В этом случае расширение Кана прилагает все усилия для экстраполяции потерянной информации.

27.2 Расширение Кана как сопряжение

Предположим теперь, что правое расширение Кана существует для любых D (и фиксированного K). В этом случае $\mathrm{Ran}_K_$ (с подчеркиванием, заменяющим D) является функтором от категории функторов $[\mathbf{I},\mathbf{C}]$ к категории функторов $[\mathbf{A},\mathbf{C}]$. Оказывается, этот функтор является правым сопряженным к функтору предкомпозиции $- \circ K$. Последнее отображает функторы из $[\mathbf{A},\mathbf{C}]$ к функторам из $[\mathbf{I},\mathbf{C}]$. Сопряжением является:

$$[\mathbf{I},\mathbf{C}](F' \circ K, D) \cong [\mathbf{A},\mathbf{C}](F', \mathrm{Ran}_K D)$$

Это всего лишь подтверждение того факта, что каждому естественному преобразованию, которое мы обозначили ε', соответствует единственное естественное преобразование, обозначенное σ.



Более того, если мы выбираем категорию $\mathbf{I}$ такой же, как $\mathbf{C}$, мы можем подставить тождественный функтор $I_{\mathbf{C}}$ для D. Получим следующее тождество:

$$[\mathbf{C},\mathbf{C}](F' \circ K, I_{\mathbf{C}}) \cong [\mathbf{A},\mathbf{C}](F', \mathrm{Ran}_K I_{\mathbf{C}})$$

Теперь мы можем выбрать F' таким же, что и $\mathrm{Ran}_K I_{\mathbf{C}}$. В этом случае правая часть содержит тождественное естественное преобразование и, соответствуя ему, левая часть превращается в следующее естественное преобразование:

$$\varepsilon :: \mathrm{Ran}_K I_{\mathbf{C}} \circ K \to I_{\mathbf{C}}$$

Это очень похоже на коединицу сопряжения:

$$\mathrm{Ran}_K I_{\mathbf{C}} \dashv K$$

В самом деле, правое расширение Кана тождественного функтора вдоль функтора K можно использовать для вычисления левого сопряженного для K. Для этого необходимо еще одно условие: правое расширение Кана должно сохраняться функтором K. Сохранение расширения означает, что если вычислить расширение Кана функтора, предварительно скомпонованного с K, мы получим тот же результат, что и при предположении об исходном расширении Кана с K. В нашем случае это условие упрощается:

$$K \circ \mathrm{Ran}_K I_{\mathbf{C}} \cong \mathrm{Ran}_K K$$

Обратите внимание, что, используя нотацию деления на K, сопряжение можно записать так:

$$I/K \dashv K$$

что подтверждает нашу интуицию, что сопряжение описывает некую обратную форму. Условием сохранения становится:

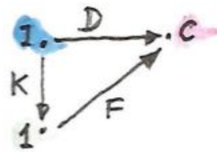
$$K \circ I/K \cong K/K$$

Правое расширение Кана функтора K вдоль самого себя, K/K, называется коплотной монадой.

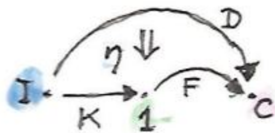
Формула сопряжения является важным результатом, поскольку, как мы вскоре увидим, можно рассчитать расширения Кана, используя концы (ко-концы), что предоставляет практические средства для нахождения правых (и левых) сопряженных.

27.3 Левое расширение Кана

Существует двойственная конструкция, которая дает нам левое расширение Кана. Чтобы построить некоторую интуицию, можно начать с определения копредела и реструктурировать его с целью использования одноэлементной категории. Мы строим коконус, используя функтор $D :: \mathbf{I} \to \mathbf{C}$, чтобы сформировать его базу, а функтор $F :: \mathbf{1} \to \mathbf{C}$, чтобы выбрать его вершину.

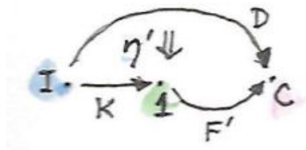


Стороны коконуса, инъекции, являются компонентами естественного преобразования η от D к $F \circ K$.

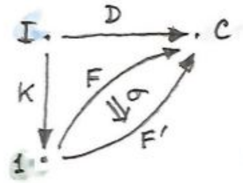


Копредел — универсальный коконус. Таким образом, для любого другого функтора F' и естественного преобразования

$$\eta' :: D \to F' \circ K$$



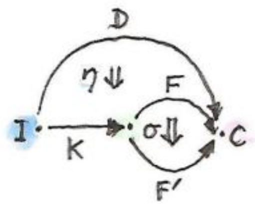
существует единственное естественное преобразование σ от F к F'



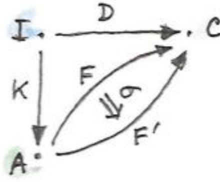
такое, что

$$\eta' = (\sigma \circ K) \,.\, \eta$$

Это проиллюстрировано на следующей диаграмме:



Заменой одноэлементной категории $\mathbf{1}$ на $\mathbf{A}$ это определение естественно обобщается на определение левого расширения Кана, обозначаемого $\mathrm{Lan}_K D$.



Естественное преобразование:

$$\eta :: D \to \mathrm{Lan}_K D \circ K$$

называется единицей левого расширения Кана.

Как и ранее, мы можем переписать взаимно однозначное соответствие между естественными преобразованиями:

$$\eta' = (\sigma \circ K) \;.\; \eta$$

в обозначениях сопряжения:

$$[\mathbf{A},\mathbf{C}](\mathrm{Lan}_K D, F') \cong [\mathbf{I},\mathbf{C}](D, F' \circ K)$$

Другими словами, левое расширение Кана является левым сопряженным, а правое расширение Кана является правым сопряженным к предкомпозиции с K.

Точно так же, как правое расширение Кана тождественного функтора может быть использовано для вычисления левого сопряженного к K, левое расширение Кана тождественного функтора оказывается правым сопряженным к K (с η в качестве единицы сопряжения):

$$K \dashv \mathrm{Lan}_K I_{\mathbf{C}}$$

Объединяя два результата, получаем:

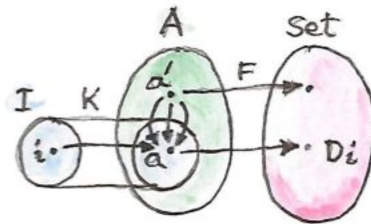
$$\mathrm{Ran}_K I_{\mathbf{C}} \dashv K \dashv \mathrm{Lan}_K I_{\mathbf{C}}$$

27.4 Расширения Кана как концы

Действительная мощность расширений Кана исходит из того факта, что они могут быть расчитаны с использованием концов (и ко-концов). Для простоты мы ограничимся рассмотрением случая, когда целевой категорией $\mathbf{C}$ является $\mathbf{Set}$, но получившиеся формулы могут быть скорректированы для любой категории.

Вернемся к идее, что расширение Кана можно использовать для расширения действия функтора вне его первоначальной области. Предположим, что K вкладывает $\mathbf{I}$ внутрь $\mathbf{A}$. Функтор D отображает $\mathbf{I}$ в $\mathbf{Set}$. Мы могли бы просто сказать, что для любого объекта a в образе K, то есть $a = K\,i$, расширенный функтор отображает a в $D\,i$. Проблема в том, что делать с теми объектами в $\mathbf{A}$, которые находятся вне образа K? Идея состоит в том, что каждый такой объект потенциально связан через множество морфизмов с каждым объектом в образе K. Функтор должен сохранять эти морфизмы. Совокупность морфизмов от объекта a к образу K характеризуется hom-функтором:

$$\mathbf{A}(a, K\,-)$$



Обратите внимание, что этот hom-функтор является композицией двух функторов:

$$\mathbf{A}(a, K\,-) = \mathbf{A}(a,\,-) \circ K$$

Правое расширение Кана является правым сопряженным к композиции функторов:

$$[\mathbf{I},\mathbf{Set}](F' \circ K, D) \cong [\mathbf{A},\mathbf{Set}](F', \mathrm{Ran}_K D)$$

Давайте посмотрим, что произойдет, если мы заменим F' на hom-функтор:

$$[\mathbf{I},\mathbf{Set}](\mathbf{A}(a, -) \circ K, D) \cong [\mathbf{A},\mathbf{Set}](\mathbf{A}(a, -), \mathrm{Ran}_K D)$$

а затем заменим композицию:

$$[\mathbf{I},\mathbf{Set}](\mathbf{A}(a, K\, -), D) \cong [\mathbf{A},\mathbf{Set}](\mathbf{A}(a, -), \mathrm{Ran}_K D)$$

Правая часть может быть сокращена с помощью леммы Йонеды:

$$[\mathbf{I},\mathbf{Set}](\mathbf{A}(a, K\, -), D) \cong \mathrm{Ran}_K D\, a$$

Теперь мы можем перезаписать множество естественных преобразований как конец, чтобы получить эту очень удобную формулу для правого расширения Кана:

$$\mathrm{Ran}_K D\, a \cong \int_i \mathbf{Set}(\mathbf{A}(a, K\, i), D\, i)$$

Существует аналогичная формула для левого расширения Кана в терминах ко-конца:

$$\mathrm{Lan}_K D\, a \cong \int^i \mathbf{A}(K\, i, a) \times D\, i$$

Чтобы убедиться, что это так, мы покажем, что это действительно левая сопряженная к функторной композиции:

$$[\mathbf{A},\mathbf{Set}](\mathrm{Lan}_K D, F') \cong [\mathbf{I},\mathbf{Set}](D, F' \circ K)$$

Подставим нашу формулу в левую часть:

$$[\mathbf{A},\mathbf{Set}](\int^i \mathbf{A}(K\, i, -) \times D\, i, F')$$

Это множество естественных преобразований, поэтому его можно заменить концом:

$$\int_a \mathbf{Set}(\int^i \mathbf{A}(K\, i, a) \times D\, i, F'\, a)$$

Используя непрерывность $\hom$-функтора, мы можем заменить ко-конец на конец:

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(K\,i, a) \times D\,i, F'\,a)$$

Мы можем использовать умноженно-экспоненциальное (product-exponential) сопряжение:

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(K\,i, a), (F'\,a)^{D\,i})$$

Экспоненциал изоморфен соответствующему $\hom$-множеству:

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(K\,i, a), \mathbf{A}(D\,i, F'\,a))$$

Существует теорема, называемая теоремой Фубини, которая позволяет переставлять два конца:

$$\int_i \int_a \mathbf{Set}(\mathbf{A}(K\,i, a), \mathbf{A}(D\,i, F'\,a))$$

Внутренний конец представляет собой множество естественных преобразований между двумя функторами, поэтому мы можем использовать лемму Йонеды:

$$\int_i \mathbf{A}(D\,i, F'\,(K\,i))$$

Это действительно множество естественных преобразований, составляющих правую часть сопряжения, которое доказываем:

$$[\mathbf{I},\mathbf{Set}](D, F' \circ K)$$

Эти виды вычислений, использующих концы, ко-концы и лемму Йонеды, довольно типичны для «исчисления» концов.

27.5 Расширения Кана на Haskell

Формулы конца / ко-конца для расширений Кана могут быть легко переведены на Haskell. Начнем с правого расширения:

$$\mathrm{Ran}_K D\, a \cong \int_i \mathbf{Set}(\mathbf{A}(a, K\, i), D\, i)$$

Мы заменяем конец квантором всеобщности, а hom-множества — функциональными типами:

```
newtype Ran k d a = Ran (forall i.
                         (a -> k i) -> d i)
```

Если рассмотреть это определение, то ясно, что `Ran` должно содержать значение типа `a`, к которому может применяться функция, и естественное преобразование между двумя функторами `k` и `d`. Например, предположим, что `k` — это функтор дерева, а `d` — функтор списка, и вам было предоставлено выражение `Ran Tree [] String`. Если вы передадите ему функцию:

```
f :: String -> Tree Int
```

вы получите список из `In`, и т.д. Правое расширение Кана будет использовать вашу функцию для создания дерева, а затем переупаковать его в список. Например, вы можете передать ему парсер, который генерирует дерево парсинга из строки, и вы получите список, который соответствует обходу этого дерева в глубину.

Правое расширение Кана можно использовать для вычисления левого сопряженного к данному функтору, заменяя функтор `d` на тождественный функтор. Это приводит к левому сопряженному функтору `k`, представленному множеством полиморфных функций типа:

```
forall i. (a -> k i) -> i
```

Предположим, что `k` — забывающий функтор от категории моноидов. Квантор всеобщности перебирает все моноиды. Конечно, в Haskell мы не можем выразить моноидальные законы, но следующее является подходящим приближением результирующего свободного функтора (забывающий функтор `k` является тождественным на объектах):

```
type Lst a = forall i. Monoid i => (a -> i)
                                        -> i
```

Как и ожидалось, он генерирует свободные моноиды или списки Haskell:

```
toLst    :: [a] -> Lst a
toLst as  = \f -> foldMap f as

fromLst  :: Lst a -> [a]
fromLst f = f (\a -> [a])
```

Левое расширение Кана является ко-концом:

$$\mathrm{Lan}_K D\, a \cong \int^i \mathbf{A}(K\, i, a) \times D\, i$$

поэтому оно преобразуется в квантор существования. В символьном виде:

```
Lan k d a = exists i. (k i -> a, d i)
```

Это может быть закодировано на Haskell с использованием GADT или с помощью квантора всеобщности конструктора данных:

```
data Lan k d a = forall i.
                 Lan (k i -> a) (d i)
```

Интерпретация этой структуры данных заключается в том, что она содержит функцию, которая принимает контейнер с некоторыми неопределенными `i` и создает `a`, также содержащий контейнер из этих же `i`. Поскольку вы не знаете, что такое `i`, единственное, что вы можете сделать

с этой структурой данных, — это извлечь контейнер с `i`, перепаковать его в контейнер, определенный функтором `k`, используя естественное преобразование, и вызвать функцию для получения `a` , Например, если `d` — дерево, а `k` — это список, вы можете сериализовать дерево в список, вызвать функцию с полученным списком и получить `a`.

Левое расширение Кана можно использовать для вычисления правого сопряженного функтора. Мы знаем, что правый сопряженный к функтору произведения — экспоненциал, поэтому попробуем реализовать его с помощью расширения Кана:

```
type Exp a b = Lan ((,) a) I b
```

Это действительно изоморфно функциональному типу, о чем свидетельствует следующая пара функций:

```
toExp  :: (a -> b) -> Exp a b
toExp f = Lan (f . fst) (I ())

fromExp              :: Exp a b -> (a -> b)
fromExp (Lan f (I x)) = \a -> f (a, x)
```

Обратите внимание, что, как описано ранее в общем случае, мы выполнили следующие шаги:

1. извлекли контейнер `x` (здесь это просто тривиальный контейнер идентичности) и функцию `f`,

2. повторно упаковали контейнер, используя естественное преобразование между тождественным функтором и функтором пары и

3. вызвали функцию `f`.

27.6 Свободный функтор

Интересным применением расширений Кана является построение свободного функтора. Оно является решением следующей практической

задачи: предположим, имеется конструктор типов — это отображение объектов. Можно ли определить функтор на основе этого конструктора типов? Другими словами, возможно ли определить отображение морфизмов, которое расширяло бы этот конструктор типов до полноценного эндофунктора?

Ключевым наблюдением является то, что конструктор типа можно описать как функтор, областью определения которого является дискретная категория. Дискретная категория не имеет морфизмов, кроме тождественных. Для заданной категории $\mathbf{C}$, всегда можно построить дискретную категорию $|\mathbf{C}|$, просто отбрасывая все нетождественные морфизмы. Функтор F от $|\mathbf{C}|$ к $\mathbf{C}$ — это простое сопоставление объектов или то, что является конструктором типа в Haskell. Существует также канонический функтор $\mathcal{J}$, который вкладывает $|\mathbf{C}|$ в $\mathbf{C}$: это тождественность на объектах (и на тождественных морфизмах). Левое расширение Кана F вдоль $\mathcal{J}$, если оно существует, тогда является функтором от $\mathbf{C}$ к $\mathbf{C}$:

$$\mathrm{Lan}_{\mathcal{J}} F a = \int^{i} \mathbf{C}(\mathcal{J} i, a) \times F i$$

Он называется свободным функтором, базирующимся на F.

На Haskell его можно было бы записать так:

```
data FreeF f a = forall i. FMap (i -> a) (f i)
```

Действительно, для любого конструктора типа `f`, `FreeF f` является функтором:

```
instance Functor (FreeF f) where
   fmap g (FMap h fi) = FMap (g . h) fi
```

Ясно что свободный функтор имитирует поднятие функции, записывая как функцию, так и ее аргумент. Он аккумулирует поднятые функции, записывая их композицию. Правила функтора выполняются автоматически. Эта конструкция была использована в статье о свободных монадах[1].

В качестве альтернативы можно использовать правое расширение Кана для той же цели:

[1] http://okmij.org/ftp/Haskell/extensible/more.pdf

```
newtype FreeF f a = FreeF (forall i. (a -
> i) -> f i)
```

Легко проверить, что это действительно функтор:

```
instance Functor (FreeF f) where
   fmap g (FreeF r) = FreeF (\bi -
> r (bi . g))
```

Глава 28

Обогащенные категории

Категория называется *малой*, если ее объекты образуют множество. Но существуют совокупности большие, чем множества. Как известно, множество всех множеств не может быть сформировано в рамках стандартной теории множеств (теория Цермело-Френкеля, возможно дополненная аксиомой выбора). Таким образом, категория всех множеств должна быть большой. Существуют математические трюки, например универсум Гротендика, которые можно использовать для определения коллекций, выходящих за пределы множеств. Эти трюки позволяют говорить о больших категориях.

Категория *локально мала*, если морфизмы между любыми двумя объектами образуют множество. Если они не образуют множества, нам нужно пересмотреть несколько определений. В частности, что означает композиция морфизмов, если мы даже не можем выбрать их из множества? Решение состоит в отдельной настройке, заключающейся в замене hom-множеств, которые являются объектами $\mathbf{Set}$, объектами из другой категории $\mathbf{V}$. Разница состоит в том, что, в общем, объекты не имеют элементов, поэтому нам больше не разрешается говорить об отдельных морфизмах. Мы должны определить все свойства некой *обогащенной* категории в терминах операций, которые могут быть выполнены для hom-объектов в целом. Чтобы сделать это, категория, которая предоставляет hom-объекты, должна иметь дополнительную структуру — она должна быть моноидальной категорией, скажем $\mathbf{V}$. Тогда можно говорить о категории $\mathbf{C}$, обогащенной категорией $\mathbf{V}$.

Помимо причин, связанных с размером, нам может быть интересно обобщить $\hom$-множества на нечто более структурированное, чем простое множество. Например, традиционная категория не имеет понятия расстояния между объектами. Два объекта либо связаны морфизмами, либо нет. Все объекты, которые связаны с данным объектом, являются его соседями. В отличие от реальной жизни, в категории друг моего друга так же близок ко мне, как и мой приятель. В подходящей обогащенной категории мы можем определить расстояния между объектами.

Есть еще одна весьма практическая причина для приобретения некоторого опыта работы с обогащенными категориями, в связи с наличием очень полезного онлайн-источника категорных знаний, nLab[1], излагаемых, в основном, с точки зрения обогащенных категорий.

28.1 Почему моноидальная категория?

При построении обогащенной категории необходимо иметь в виду, что мы должны быть в состоянии восстановить обычные определения, когда заменим моноидальную категорию обратно на **Set**, а $\hom$-объекты на $\hom$-множества. Лучший способ добиться этого состоит в том, чтобы начать с обычных определений и продолжать их переформулировку в бесточечном стиле, то есть без указания элементов множеств.

Начнем с определения композиции. Обычно пару морфизмов: один из $\mathbf{C}(b, c)$, а другой из $\mathbf{C}(a, b)$, она отображает к морфизму из $\mathbf{C}(a, c)$. Другими словами, это отображение:

$$\mathbf{C}(b, c) \times \mathbf{C}(a, b) \to \mathbf{C}(a, c)$$

Это функция между множествами — одно из них является декартовым произведением двух $\hom$-множеств. Эту формулу можно легко обобщить, заменив декартово произведение на нечто более общее. Категорное произведение будет работать, но мы можем пойти еще дальше и использовать совершенно общее тензорное произведение.

[1] https://ncatlab.org/

Что касается тождественных морфизмов, то вместо выбора отдельных элементов из $\hom$-множеств, мы можем определить их, используя функции от одноэлементного множества 1:

$$j_a :: 1 \to \mathbf{C}(a, a)$$

Опять же, мы могли бы заменить одноэлементное множество терминальным объектом, но можно пойти еще дальше, заменив его единицей i тензорного произведения.

Таким образом, ясно, что объекты, взятые из некоторой моноидальной категории $\mathbf{V}$, являются хорошими кандидатами для замены $\hom$-множества.

28.2 Моноидальная категория

Ранее мы говорили о моноидальных категориях, но стоит переформулировать определение. Моноидальная категория определяет тензорное произведение, являющееся бифунктором:

$$\otimes :: \mathbf{V} \times \mathbf{V} \to \mathbf{V}$$

Мы хотим, чтобы тензорное произведение было ассоциативным, и достаточно, чтобы ассоциативность выполнялась с точностью до естественного изоморфизма. Этот изоморфизм называется ассоциатором. Его компоненты:

$$\alpha_{a\,b\,c} :: (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$

должны быть естественными по всем трем аргументам.

Моноидальная категория также должна определять специальный единичный объект i, который служит единицей тензорного произведения, опять-таки, с точностью до естественного изоморфизма. Два изоморфизма называются, соответственно, левым и правым уединителем, а их компонентами являются:

$$\lambda_a :: i \otimes a \to a$$
$$\rho_a :: a \otimes i \to a$$

Ассоциатор и уединители должны удовлетворять условиям когерентности:

$$
\begin{array}{ccc}
((a \otimes b) \otimes c) \otimes d & \xrightarrow{\alpha_{abc} \otimes \mathrm{id}_d} & (a \otimes (b \otimes c)) \otimes d \\
\downarrow{\scriptstyle \alpha_{(a \otimes b)cd}} & & \downarrow{\scriptstyle \alpha_{a(b \otimes c)d}} \\
(a \otimes b) \otimes (c \otimes d) & & a \otimes ((b \otimes c) \otimes d) \\
& \searrow{\scriptstyle \alpha_{ab(c \otimes d)}} \quad \swarrow{\scriptstyle \mathrm{id}_a \otimes \alpha_{bcd}} & \\
& a \otimes (b \otimes (c \otimes d)) &
\end{array}
$$

$$
\begin{array}{ccc}
(a \otimes i) \otimes b & \xrightarrow{\alpha_{aib}} & a \otimes (i \otimes b) \\
& \searrow{\scriptstyle \rho_a \otimes \mathrm{id}_b} \quad \swarrow{\scriptstyle \mathrm{id}_a \otimes \lambda_b} & \\
& a \otimes b &
\end{array}
$$

Моноидальная категория называется *симметричной*, если существует естественный изоморфизм с компонентами:

$$\gamma_{a\,b} :: a \otimes b \to b \otimes a$$

чей «квадрат равен единице»:

$$\gamma_{b\,a} \circ \gamma_{a\,b} = \mathrm{id}_{a \otimes b}$$

и который согласуется с моноидальной структурой.

Интересный факт, связанный с моноидальными категориями состоит в том, что вы можете определить внутренний $\hom$ (функциональный объект) как правый сопряженный к тензорному произведению. Вы можете вспомнить, что стандартное определение функционального объекта, или экспоненты, было дано через правое сопряжение к категорному произведению (категория, в которой такой объект существует для любой пары объектов, называется декартово замкнутой). Вот сопряжение, которое определяет внутренний $\hom$ в моноидальной категории:

$$\mathbf{V}(a \otimes b, c) \sim \mathbf{V}(a, [b, c])$$

Следуя Келли[2], я использую обозначение $[b, c]$ для внутреннего $\hom$. Коединица этого сопряжения является естественным преобразованием,

[2] http://www.tac.mta.ca/tac/reprints/articles/10/tr10.pdf

компоненты которого называются оценочными морфизмами:

$$\varepsilon_{a\,b} :: ([a,b] \otimes a) \to b$$

Заметим, что если тензорное произведение не симметрично, мы можем определить другой внутренний hom, обозначаемый $[[a, c]]$, используя следующее сопряжение:

$$\mathbf{V}(a \otimes b, c) \sim \mathbf{V}(b, [[a, c]])$$

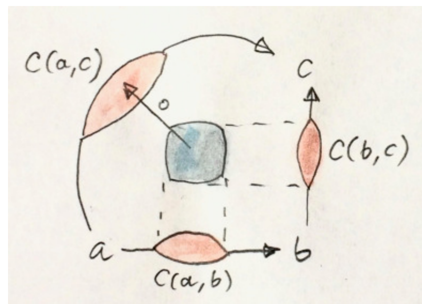
Моноидальная категория, в которой определены оба сопряжения, называется би-замкнутой. Примером категории, не являющейся бизамкнутой, является категория эндофункторов в **Set** с композицией функторов, выступающей в качестве тензорного произведения. Эту категорию мы использовали для определения монад.

28.3 Обогащенная категория

Категория **C**, обогащенная моноидальной категорией **V**, заменяет hom-множества на hom-объекты. Каждой паре объектов a и b из **C** сопоставим объект $\mathbf{C}(a, b)$ в **V**. Мы используем те же обозначения для hom-объектов, что и для hom-множеств, осознавая, что они не содержат морфизмов. С другой стороны, **V** — регулярная (не обогащенная) категория с hom-множествами и морфизмами. Таким образом, мы не полностью избавлены от множеств — мы просто подмели их под коврик.

Так как мы не можем говорить об отдельных морфизмах в **C**, композиция морфизмов заменяется семейством морфизмов в **V**:

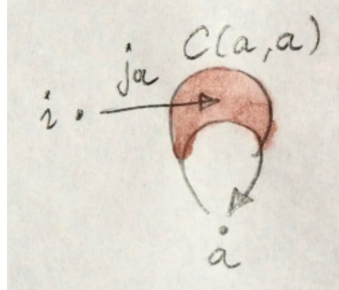
$$\circ :: \mathbf{C}\,(b, c) \otimes \mathbf{C}\,(a, b) \;\to \mathbf{C}\,(a, c)$$



Аналогично тождественные морфизмы заменяются семейством морфизмов в $\mathbf{V}$:

$$j_a :: i \to \mathbf{C}(a, a)$$

где i — единичный тензор в $\mathbf{V}$.



Ассоциативность композиции определяется в терминах ассоциатора в $\mathbf{V}$:

$$\begin{array}{ccccc}
(\mathbf{C}(c,d) \otimes \mathbf{C}(b,c)) \otimes \mathbf{C}(a,b) & \xrightarrow{\circ \otimes \mathrm{id}} & \mathbf{C}(b,d) \otimes \mathbf{C}(a,b) & & \\
\downarrow \alpha & & & \searrow \circ & \\
& & & & \mathbf{C}(a,d) \\
& & & \nearrow \circ & \\
\mathbf{C}(c,d) \otimes (\mathbf{C}(b,c) \otimes \mathbf{C}(a,b)) & \xrightarrow[\mathrm{id} \otimes \circ]{} & \mathbf{C}(c,d) \otimes \mathbf{C}(a,c) & &
\end{array}$$

Законы единицы подобным образом выражаются в терминах уединителей:

$$\begin{array}{ccc}
\mathbf{C}(a,b) \otimes i & \xrightarrow{\mathrm{id} \otimes j_a} & \mathbf{C}(a,b) \otimes \mathbf{C}(a,a) \\
& \rho \searrow \quad \swarrow \circ & \\
& \mathbf{C}(a,b) &
\end{array}$$

$$\begin{array}{ccc}
i \otimes \mathbf{C}(a,b) & \xrightarrow{j_b \otimes \mathrm{id}} & \mathbf{C}(b,b) \otimes \mathbf{C}(a,b) \\
& \lambda \searrow \quad \swarrow \circ & \\
& \mathbf{C}(a,b) &
\end{array}$$

28.4 Предпорядки

Предпорядок определяется как тонкая категория, в которой каждое hom-множество либо пустое, либо одноэлементное. Мы интерпретируем непустое множество $\mathbf{C}(a, b)$ как доказательство того, что a меньше или равно b. Такая категория может быть истолкована как обогащенная очень простой моноидальной категорией, содержащей только два объекта, 0 и 1 (иногда называемые False и True). Помимо обязательных тождественных морфизмов эта категория имеет единственный морфизм от 0 к 1, обозначим его $0 \to 1$. В ней можно установить простую моноидальную структуру, при этом тензорное произведение моделирует простую арифметику 0 и 1 (т.е. единственным ненулевым произведением является $1 \otimes 1$). Тождественным объектом в этой категории является 1. Это строгая моноидальная категория, то есть ассоциатор и уединители являются тождественными морфизмами.

Так как в предпорядке множество hom является пустым или синглетоном, мы можем просто заменить его hom-объектом из нашей тонкой категории. Обогащенный предпорядок $\mathbf{C}$ имеет hom-объект $\mathbf{C}(a, b)$ для любой пары объектов a и b. Если a меньше или равно b, этот объект есть 1, в противном случае — 0.

Давайте рассмотрим композицию. Тензорное произведение любых двух объектов равно 0, если только они не равны 1, и в этом случае это 1. Если это 0, то у нас есть два варианта композиции морфизмов: это может быть либо id_0, либо $0 \to 1$. Но если это 1, тогда единственный вариант — id_1. Если перевести это обратно в отношения, то мы получаем, что если $a \leqslant b$ и $b \leqslant c$, то $a \leqslant c$, что в точности и требуется для закона транзитивности.

А как насчет тождественных морфизмов? Это морфизм от 1 к $\mathbf{C}(a, a)$. Существует только один морфизм, идущий от 1, и это тождественный морфизм id_1, поэтому $\mathbf{C}(a, a)$ должно быть 1. Это означает, что $a \leqslant a$, что является законом рефлексивности для предпорядка. Таким образом, и транзитивность, и рефлексивность автоматически выполняются, если мы реализуем предпорядок как обогащенную категорию.

28.5 Метрические пространства

Интересный пример приведен Ловером[3]. Он заметил, что метрические пространства могут быть определены с использованием обогащенных категорий. Метрическое пространство определяет расстояние между любыми двумя объектами. Расстояние — это неотрицательное действительное число. Удобно включить бесконечность в качестве возможного значения. Если расстояние бесконечно, то существует способ добраться от исходного объекта до целевого объекта.

Имеются некоторые очевидные свойства, которым должны удовлетворять расстояния. Одно из них заключается в том, что расстояние от объекта до самого себя должно быть равно нулю. Другим является неравенство треугольника: прямое расстояние не больше суммы расстояний с промежуточными остановками. Не требуется, чтобы расстояние было симметричным, что может показаться сначала странным, но, как объяснил Ловер, можно себе представить, что в одном направлении вы идете вверх, а в другом вы спускаетесь вниз. В любом случае симметрия может быть введена позднее в качестве дополнительного ограничения.

Итак, как метрическое пространство может быть выражено на категорном языке? Мы должны построить категорию, в которой hom-объекты являются расстояниями. Имейте в виду, что расстояния — это не морфизмы, а hom-объекты. Как hom-объект может быть числом? Только если мы можем построить моноидальную категорию $\mathbf{V}$, в которой эти числа являются объектами. Неотрицательные действительные числа (плюс бесконечность) образуют полный порядок, поэтому их можно рассматривать как тонкую категорию. Морфизм между двумя такими числами x и y существует тогда и только тогда, когда $x \geqslant y$ (примечание: это противоположно тому, что традиционно используется в определении предпорядка). Моноидальная структура задается сложением, с нулем, служащим в качестве единичного объекта. Другими словами, тензорное произведение двух чисел есть их сумма.

Метрическое пространство — категория, обогащенная такой моноидальной категорией. hom-объект $\mathbf{C}(a, b)$ от объекта a до b является неотрицательным (возможно, бесконечным) числом, которое мы будем называть

[3] http://www.tac.mta.ca/tac/reprints/articles/1/tr1.pdf

расстоянием от a до b. Давайте посмотрим, что является идентичностью и композицией в такой категории.

По нашим определениям, морфизм от тензорной единицы, которая является числовым нулем, к hom-объекту $\mathbf{C}(a, a)$, является отношением:

$$0 \geqslant \mathbf{C}(a, a)$$

Так как $\mathbf{C}(a, a)$ — неотрицательное число, это условие говорит нам, что расстояние от a до a всегда равно нулю. Проверьте!

Теперь поговорим о композиции. Начнем с тензорного произведения двух примыкающих hom-объектов, $\mathbf{C}(b, c) \otimes \mathbf{C}(a, b)$. Мы определили тензорное произведение как сумму двух расстояний. Композиция является морфизмом в $\mathbf{V}$ от этого произведения к $\mathbf{C}(a, c)$. Морфизм в $\mathbf{V}$ определяется как отношение больше или равно. Другими словами, сумма расстояний от a до b и от b до c больше или равна расстоянию от a до c. Но это просто стандартное неравенство треугольника. Проверьте!

Рассматривая метрическое пространство в терминах обогащенной категории, мы получили неравенство треугольника и нулевое расстояние «бесплатно».

28.6 Обогащенные функторы

Определение функтора включает в себя отображение морфизмов. В обогащенной среде мы не имеем понятия отдельных морфизмов, поэтому нам приходится иметь дело с hom-объектами в целом. hom-объекты — это объекты в моноидальной категории $\mathbf{V}$, и в нашем распоряжении имеются морфизмы между ними. Поэтому имеет смысл определять обогащенные функторы между категориями, когда они обогащаются по одной и той же моноидальной категории $\mathbf{V}$. Затем мы можем использовать морфизмы в $\mathbf{V}$ для отображения hom-объектов между двумя обогащенными категориями.

Обогащенный функтор F между двумя категориями $\mathbf{C}$ и $\mathbf{D}$, кроме отображения объектов на объекты, также назначает каждой паре объектов из $\mathbf{C}$ морфизм из $\mathbf{V}$:

$$F_{a\,b} :: \mathbf{C}\,(a, b) \to \mathbf{D}\,(F\,a, F\,b)$$

Функтор является сохраняющим структуру отображением. Для обычных функторов это означало сохранение композиции и тождественных морфизмов. В обогащенной обстановке сохранение композиции означает, что следующая диаграмма коммутативна:

$$\begin{array}{ccc} \mathbf{C}(b,c) \otimes \mathbf{C}(a,b) & \xrightarrow{\circ} & \mathbf{C}(a,c) \\ \downarrow{\scriptstyle F_{bc}\otimes F_{ab}} & & \downarrow{\scriptstyle F_{ac}} \\ \mathbf{D}(Fb,Fc) \otimes \mathbf{D}(Fa,Fb) & \xrightarrow[\circ]{} & \mathbf{D}(Fa,Fc) \end{array}$$

Сохранение тождественных морфизмов заменяется сохранением морфизмов в $\mathbf{V}$, которые «выбирают» тождественные морфизмы:

$$\begin{array}{ccc} & i & \\ {\scriptstyle j_a}\swarrow & & \searrow{\scriptstyle j_{Fa}} \\ \mathbf{C}(a,a) & \xrightarrow[F_{aa}]{} & \mathbf{D}(Fa,Fa) \end{array}$$

28.7 Самообогащение

Замкнутая симметричная моноидальная категория может быть сделана самообогащенной, заменой hom-множеств на внутренние hom (см. определение выше). Чтобы добиться этого, мы должны определить закон композиции для внутренних hom. Другими словами, мы должны реализовать морфизм со следующей сигнатурой:

$$[b,c] \otimes [a,b] \to [a,c]$$

Это мало чем отличается от любой другой задачи программирования, за исключением того, что в теории категорий мы обычно используем бесточечные реализации. Мы начинаем с определения множества, элементом которого должен быть указанный морфизм. В этом случае, это член hom-множества:

$$\mathbf{V}([b,c] \otimes [a,b], [a,c])$$

Это hom-множество изоморфно следующему:

$$\mathbf{V}(([b,c] \otimes [a,b]) \otimes a, c)$$

Я просто использовал сопряжение, которое определяло внутренний $\hom$ $[a, c]$. Если мы можем построить морфизм в этом новом множестве, сопряжение укажет нам на морфизм в исходном множестве, который мы затем можем использовать в качестве композиции. Мы построим этот морфизм, составив композицию нескольких имеющихся в нашем распоряжении морфизмов. Для начала мы можем использовать ассоциатор $\alpha_{[b,c]\,[a,b]\,a}$ для повторного связывания выражения слева:

$$([b, c] \otimes [a, b]) \otimes a \to [b, c] \otimes ([a, b] \otimes a)$$

Мы можем следовать этому с коединицей сопряжения $\varepsilon_{a\,b}$:

$$[b, c] \otimes ([a, b] \otimes a) \to [b, c] \otimes b$$

И используем коединицу $\varepsilon_{b\,c}$, чтобы добраться до c. Таким образом, мы построили морфизм:

$$\varepsilon_{b\,c} \,.\, (\mathrm{id}_{[b,c]} \otimes \varepsilon_{a\,b}) \,.\, \alpha_{[b,c]\,[a,b]\,a}$$

то есть элемент $\hom$-множества:

$$\mathbf{V}(([b, c] \otimes [a, b]) \otimes a, c)$$

Сопряжение даст нам закон композиции, который мы искали.

Точно так же, тождественный морфизм:

$$j_a :: i \to [a, a]$$

является членом следующего $\hom$-множества:

$$\mathbf{V}(i, [a, a])$$

которое изоморфно, посредством сопряжения, $\hom$-множеству:

$$\mathbf{V}(i \otimes a, a)$$

Мы знаем, что это $\hom$-множество содержит левый тождественный морфизм λ_a. Мы можем определить j_a как его образ при сопряжении.

Практическим примером самообогащения является категория **Set**, которая служит прототипом для типов в языках программирования. Мы уже видели, что это замкнутая моноидальная категория по отношению к декартовому произведению. В **Set**, hom-множество между любыми двумя множествами само по себе является множеством, и таким образом, оно является объектом в **Set**. Мы знаем, что это изоморфно экспоненциальному множеству, поэтому внешний и внутренние hom эквивалентны. Теперь мы также знаем, что посредством самообогащения мы можем использовать экспоненциальное множество как hom-объект и выразить композицию в терминах декартовых произведений экспоненциальных объектов.

28.8 Связь с 2-категориями

Я говорил о 2-категориях в контексте **Cat** — категории (малых) категорий. Морфизмы между категориями являются функторами, но есть дополнительная структура: естественные преобразования между функторами. В 2-категории объекты часто называют 0-клетками, морфизмы — 1-клетками, а морфизмы между морфизмами — 2-клетками. В **Cat** 0-клетки являются категориями, 1-клетки являются функторами, а 2-клетки — естественными преобразованиями.

Но обратите внимание, что функторы между двумя категориями также образуют категорию, поэтому в **Cat** мы действительно имеем hom-категорию, а не hom-множество. Оказывается, что, так же как **Set** можно рассматривать как категорию, обогащенную категорией **Set**, **Cat** можно рассматривать как категорию, обогащенную категорией **Cat**. В более общем плане, точно так же, как каждая категория может рассматриваться как обогащенная категорией **Set**, каждая 2-категория может считаться обогащенной категорией **Cat**.

Глава 29

Топосы

Я понимаю, что мы, возможно, отходим от программирования и погружаемся в сложную математику. Но вы не знаете, что может произойти в результате следующей крупной революции в программировании, и какая математика может понадобиться для ее понимания. И сейчас имеются интересные идеи, такие как функциональное реактивное программирование с непрерывным временем, пополнение системы типов Haskell зависимыми типами или исследования по применению гомотопической теории типов в программировании.

До сих пор я вскользь определял типы с множествами значений. Это не совсем правильно, потому что такой подход не учитывает того факта, что в программировании мы вычисляем значения, а вычисление — это процесс, который требует времени и в определенных случаях может не завершиться. Дивергентные вычисления являются частью каждого Тьюринг-полного языка.

Существуют также основополагающие причины, по которым теория множеств может оказаться не лучшей в качестве основы для компьютерной науки или даже самой математики. Хорошей аналогией является теория множеств, являющаяся языком ассемблера, привязанным к определенной архитектуре. Если же вы захотите запустить свою математику на разных архитектурах, вам нужно будет использовать более общие инструменты.

Одна из возможностей заключается в использовании пространств вместо множеств. Пространства обладают более богатой структурой и могут

быть определены без использования множеств. Одна из формализаций, обычно связанная с пространствами, — это топология, которая необходима для определения таких свойств, как непрерывность. И традиционный подход к топологии, как вы догадались, использует теорию множеств. В частности, понятие подмножества является центральным в топологии. Неудивительно, что категорные теоретики обобщили эту идею на категории, отличные от **Set**. Тип категории, которая имеет только подходящие свойства, служит заменой теории множеств и называется топосом (во множественном числе: топосы), и обеспечивает, помимо прочего, обобщение понятия подмножества.

29.1 Классификатор подобъектов

Начнем с того, что попытаемся выразить идею подмножества, используя функции, а не элементы. Любая функция f от некоторого множества a к множеству b определяет подмножество в b — образ a под f. Но существует много функций, которые определяют одно и то же подмножество. Мы должны быть более конкретными. Начнем с того, что можно сосредоточиться на функциях, которые являются инъективными, — которые не связывают несколько элементов с одним. Инъективные функции «вкладывают» одно множество в другое. Для конечных множеств вы можете визуализировать инъективные функции в виде параллельных стрелок, соединяющих элементы одного множества с элементами другого. Конечно, первое множество не может быть больше второго множества, иначе стрелки будут обязательно сходиться. Но остается некоторая неоднозначность: может существовать другое множество a' и другая инъективная функция f' от этого множества к b, которая выбирает то же самое подмножество. И вы можете легко убедиться, что такое множество должно быть изоморфно a. Этот факт можно использовать, чтобы определить подмножество как семейство инъективных функций, связанных изоморфизмами их областей. Точнее, мы говорим, что две инъективные функции:

$$f :: a \to b$$
$$f' :: a' \to b$$

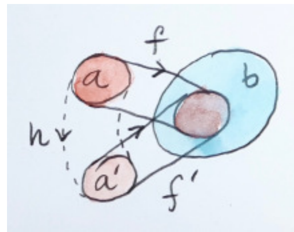
эквивалентны, если существует изоморфизм

$$h :: a \to a'$$

такой, что

$$f = f' \,.\, h$$

Такое семейство эквивалентных инъективных функций определяет подмножество в b.



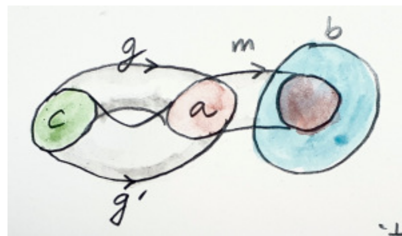
Это определение может быть поднято до произвольной категории, если мы заменим инъективные функции мономорфизмом. Напомним, мономорфизм m от a к b определяется его универсальным свойством. Для любого объекта c и любой пары морфизмов:

$$g :: c \to a$$
$$g' :: c \to a$$

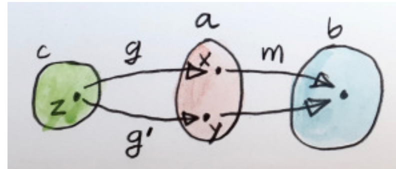
таких, что

$$m \,.\, g = m \,.\, g'$$

должно быть $g = g'$.



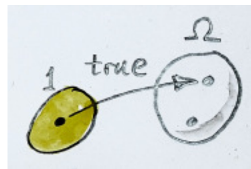
На множествах это определение легче понять, если учесть, что оно означало бы, что функция m не является мономорфизмом. Она отобразила бы два разных элемента из a к одному элементу из b. Тогда мы могли бы найти две функции g и g', которые отличаются только этими двумя элементами. Последующая композиция с m затем замаскирует эту различие.



Существует еще один способ определения подмножества: использование единственной функции, называемой характеристической функцией. Это функция χ от множества b к двухэлементному множеству Ω. Один элемент последнего множества называется «истина», а другой — «ложь». Эта функция присваивает «истина» тем элементам из b, которые являются членами подмножества, а «ложь» — тем, которые в него не входят.

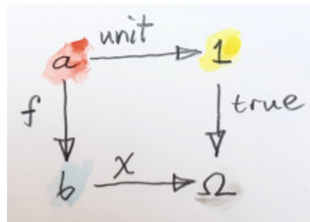
Остается указать, что означает характеризация такого элемента из Ω, как «истина». Мы можем воспользоваться стандартным трюком: использовать функцию от одноэлементного множества к Ω. Будем называть эту функцию $true$:

$$true :: 1 \to \Omega$$



Эти определения могут быть объединены таким образом, чтобы они не только устанавливали, что является подобъектом, но и определяли особый объект Ω, не упоминая элементы. Идея состоит в том, что нам необходим морфизм $true$ для представления *обобщенного* подобъекта. В **Set** он выбирает одноэлементное подмножество из двухэлементного множества Ω. Оно получается обобщенным по построению. И оно, безусловно, является собственным подмножеством, потому что в Ω есть еще один элемент, который *не* находится в этом подмножестве.

В более общей постановке мы определяем $true$ как мономорфизм от терминального объекта к *классифицирующему объекту* Ω. Но мы должны определить классифицирующий объект. Нам требуется универсальное свойство, связывающее этот объект с характеристической функцией. Оказывается, что в **Set** обратный образ $true$ вдоль характеристической функции χ определяет, как подмножество a, так и инъективную функцию, которая вставляет a в b. Вот диаграмма, связанная с обратным образом:



Давайте проанализируем эту диаграмму. Формула обратного образа есть:

$$true\,.\,unit = \chi\,.\,f$$

Функция $true\,.\,unit$ отображает каждый элемент из a к «истина». Поэтому f должна отображать все элементы из a к тем элементам из b, для которых χ есть «истина». Это, по определению, элементы подмножества, заданные характеристической функцией χ. Таким образом, образ f действительно является подмножеством, о котором идет речь. Универсальность обратного образа гарантирует, что f является инъективной.

Приведенная диаграмма обратного образа может использоваться для определения классифицирующего объекта в категориях, отличных от **Set**. Такая категория должна иметь терминальный объект, который позволит нам определить мономорфизм $true$. Она также должна иметь обратные образы — фактическое требование состоит в том, что она должна содержать все конечные пределы (обратный образ является примером конечного предела). При этих предположениях мы определяем классифицирующий объект Ω, используя то свойство, что для каждого мономорфизма f существует единственный морфизм χ, который замыкает диаграмму обратного образа.

Проанализируем последнее утверждение. Когда мы строим обратный образ, в нашем распоряжении имеются три объекта, Ω, b, 1 и два морфизма, $true$ и χ. Существование обратного образа означает, что мы можем

найти такой наилучший объект a, снабженный двумя морфизмами f и $unit$ (последний однозначно выделяется с помощью определения терминального объекта), которые делают соответствующую диаграмму коммутативной.

Здесь мы решаем другую систему уравнений, для Ω и $true$, изменяя как a, так и b. Для заданных a и b может быть или не быть мономорфизма $f :: a \to b$. Но если он существует, мы хотим, чтобы он был обратным образом некоторой функции χ. Более того, мы хотим, чтобы эта χ однозначно определялась через f.

Мы не можем утверждать, что существует взаимно однозначное соответствие между мономорфизмами f и характеристическими функциями χ, так как обратный образ единственен только с точностью до изоморфизма. Но вспомните наше более раннее определение подмножества как семейства эквивалентных инъективных функций. Его можно обобщить, определив подобъект b как семейство эквивалентных мономорфизмов к b. Это семейство мономорфизмов находится во взаимно однозначном соответствии с семейством эквивалентных обратных образов нашей диаграммы.

Таким образом, мы можем определить множество подобъектов из b, $Sub(b)$, как семейство мономорфизмов, и увидеть, что оно изоморфно множеству морфизмов от b к Ω:

$$Sub(b) \cong C(b, \Omega)$$

Это естественный изоморфизм двух функторов. Другими словами, $Sub(-)$ является представимым (контравариантным) функтором, представление которого является объектом Ω.

29.2 Топос

Топос — это категория, которая

1. декартово замкнута: содержит все произведения, терминальный объект, экспоненциалы (определяемые как правые сопряженные к произведениям),

2. содержит пределы для всех конечных диаграмм,

3. имеет классификатор подобъектов Ω.

Этот набор свойств делает топос похожим на **Set** в большинстве приложений. Он также обладает дополнительными свойствами, которые следуют из его определения. Например, топос имеет все конечные копределы, включая инициальный объект.

Было бы заманчиво определить классификатор подобъектов как копроизведение (сумму) двух копий терминального объекта — это то, что есть в **Set**, — но мы хотим иметь большее обобщение. Топосы же, которые удовлетворяют этому частному случаю, называются булевыми.

29.3 Топосы и логика

В теории множеств характеристическая функция может быть интерпретирована как определяющая свойство элементов множества — предикат, который истинен для некоторых элементов и ложен для других. Предикат $isEven$ выбирает подмножество четных чисел из множества натуральных чисел. В топосе мы можем обобщить идею о том, что предикат является морфизмом от объекта a к Ω. Вот почему Ω иногда называют объектом истинностных значений.

Предикаты являются строительными блоками логики. Топос содержит все необходимые инструменты для исследования логики. Он имеет произведения, которые соответствуют логическим конъюнкциям (логическое *и*), копроизведения для дизъюнкций (логическое *или*) и экспоненциалы для импликаций. Все стандартные аксиомы логики сохраняются в топосе, кроме закона исключения третьего (или, что то же самое, исключения двойного отрицания). Вот почему логика топоса соответствует конструктивной или интуиционистской логике.

Интуиционистская логика неуклонно набирает силу, обретая неожиданную поддержку со стороны компьютерных наук. Классическое понятие исключения третьего основано на убеждении, что существует абсолютная истина: любое утверждение является либо истинным, либо ложным, или, как говорили древние римляне, *tertium non datur* (третьего не дано). Но единственным способом узнать, является ли что-то истинным

или ложным, — является его доказательство или опровержение. Доказательством является процесс, вычисление — и мы знаем, что вычисления требуют времени и ресурсов. В некоторых случаях они никогда не завершатся. Не имеет смысла утверждать, что утверждение истинно, если мы не можем доказать это за конечное время. Топос с его более тонким объектом истинностных значений предоставляет более общую структуру для моделирования интересных логик.

Упражнения

1. Покажите, что функция f, являющаяся обратным образом $true$ вдоль характеристической функции, должна быть инъективной.

Глава 30

Теории Ловера

В настоящее время вы не можете при обучении функциональному программированию не упоминуть про монады. Но есть альтернативная вселенная, в которой, случайно, Евгений Могги обратил свое внимание на теории Ловера, а не на монады. Давайте исследуем эту область.

30.1 Универсальная алгебра

Существует множество способов описания алгебр на разных уровнях абстракции. Мы попытаемся найти общий язык для описания таких понятий, как моноиды, группы или кольца. На простейшем уровне все эти конструкции определяют операции над элементами множества, а также некоторые законы, которые должны выполняться над этими операциями. Например, моноид может быть определен в терминах бинарной операции, которая ассоциативна. Также здесь имеется единичный элемент и законы для него. Но, прибегая к творческой фантазии, мы можем превратить единичный элемент в нульарную операцию — операцию, которая не принимает аргументов и возвращает специальный элемент множества. Если мы говорим о группах, то добавим унарный оператор, который принимает элемент и возвращает его обратный. Для этого есть соответствующие левые и правые законы обратимости. Кольцо определяет два бинарных оператора плюс несколько законов. И так далее.

Цельный образ формируется тем, что алгебра определяется набором n-арных операций для различных значений n и множеством эквациональных тождеств. Эти тождества универсальны. Формула ассоциативности должна выполняться для всех возможных комбинаций трех элементов и т.д.

Между прочим, сказанное исключает поля из рассмотрения по той простой причине, что нуль (единица по сложению) не имеет обратного относительно умножения. Закон обратимости для поля не может быть универсально определен.

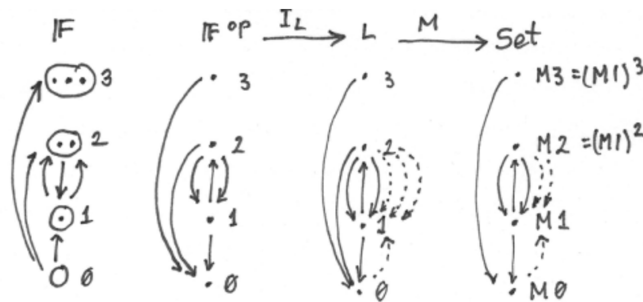
Приведенное определение универсальной алгебры можно распространить на категории, отличные от **Set**, если заменить операции (функции) морфизмами. Вместо множества будем использовать некоторый объект a (называемый общим объектом). Унарная операция — это просто эндоморфизм для a. Но как насчет других арностей (арность — количество аргументов для данной операции)? Бинарная операция (арности 2) может быть определена как морфизм от произведения $a \times a$ обратно к a. Общая n-арная операция является морфизмом от n-й степени a к a:

$$\alpha_n :: a^n \to a$$

Нульарная операция является морфизмом от терминального объекта (нулевой степени объекта a). Итак, вот что нам нужно для определения любой алгебры, — это категория, объекты которой являются степенями одного специального объекта a. Конкретная алгебра кодируется hom-множествами этой категории. Это — теория Ловера в двух словах.

Построение теорий Ловера проходит через многие этапы, поэтому вот последовательность определений:

1. Категория конечных множеств **FinSet**.
2. Ее скелет **F**.
3. Его двойственный $\mathbf{F}^{op}$.
4. Теория Ловера **L**: объект в категории **Law**.
5. Модель M категории Ловера: объект в категории $\mathbf{Mod}(\mathbf{Law},\mathbf{Set})$.



30.2 Теории Ловера

Все теории Ловера имеют общую основу. Все объекты в теории Ловера генерируются только из одного объекта, используя произведения (на самом деле, просто степени). Но как мы определяем эти произведения в общей категории? Оказывается, мы можем определить произведения, используя отображение от более простой категории. На самом деле эта более простая категория может определять копроизведения вместо произведений, и мы будем использовать контравариантный функтор для их встривания в нашу целевую категорию. Контравариантный функтор превращает копроизведения в произведения, а инъекции в проекции.

Естественным выбором для основы категории Ловера является категория конечных множеств **FinSet**. Она содержит пустое множество $\varnothing$, одноэлементное множество 1, двухэлементное множество 2 и т.д. Все объекты этой категории могут быть сгенерированы из одноэлементного набора с использованием копроизведений (обработка пустого множества как частного случая нульарного копроизведения). Например, двухэлементное множество представляет собой сумму двух синглетонов, $2 = 1 + 1$, что выражается на Haskell следующим образом:

```
type Two = Either () ()
```

Тем не менее, хотя естественно считать, что существует только одно пустое множество, может быть много разных одноэлементных множеств. В частности, множество $1 + \varnothing$ отличается от множества $\varnothing + 1$ и отличается от 1, хотя они все изоморфны. Копроизведение в категории

множеств не ассоциативно. Мы можем исправить эту ситуацию, построив категорию, которая идентифицирует все изоморфные множества. Такая категория называется скелетом. Другими словами, основой любой теории Ловера является скелет **F** из **FinSet**. Объекты этой категории могут быть идентифицированы натуральными числами (включая нуль), которые соответствуют количеству элементов в **FinSet**. Копроизведение играет роль сложения. Морфизмы в **F** соответствуют функциям между конечными множествами. Например, существует единственный морфизм от ∅ к n (пустое множество является инициальным объектом), нет морфизмов от n к ∅ (кроме $\varnothing \to \varnothing$), имеется n морфизмов от 1 к n (инъекций), один морфизм от n к 1 и т.д. Здесь n обозначает объект в **F**, соответствующий всем n-элементным множествам в **FinSet**, которые были идентифицированы через изоморфизмы.

Используя категорию **F**, мы можем формально определить *теорию* Ловера как категорию **L**, снабженную специальным функтором

$$I_{\mathbf{L}} :: \mathbf{F}^{op} \to \mathbf{L}$$

Этот функтор должен быть биекцией на объектах и должен сохранять конечные произведения (произведения в $\mathbf{F}^{op}$ совпадают с копроизведениями в **F**):

$$I_{\mathbf{L}}\,(m \times n) = I_{\mathbf{L}}\,m \times I_{\mathbf{L}}\,n$$

Иногда вы можете рассматривать этот функтор как характеризующий тождественность на объектах, что означает одинаковость объектов в **F** и **L**. Поэтому мы будем использовать для них одни и те же имена — обозначать их натуральными числами. Имейте в виду, что объекты в **F** не совпадают с множествами (они являются классами изоморфных множеств).

Hom-множества в **L**, в общем, богаче, чем в $\mathbf{F}^{op}$. Они могут содержать морфизмы, отличные от функций, соответствующих функциям в **FinSet** (последние иногда называются *базовыми операциями произведений*). В этих морфизмах закодированы законы равенства теории Ловера.

Главное наблюдение заключается в том, что одноэлементное множество 1 в **F** отображается на некоторый объект, который мы также называем 1, в **L**, а все остальные объекты в **L** автоматически являются степенями этого объекта. Например, двухэлементное множество 2 в **F** является

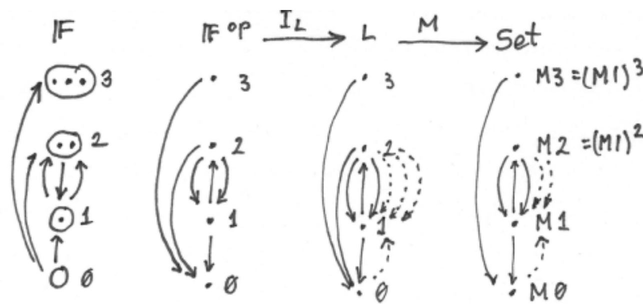
копроизведением $1+1$, поэтому оно должно быть отображено в произведение 1×1 (или 1^2) в **L**. В этом смысле категория **F** ведет себя как логарифм **L**.

Среди морфизмов в **L** имеются те, которые передаются функтором $I_{\mathbf{L}}$ из **F**. Они играют структурообразующую роль в **L**. В частности, инъекции i_k копроизведений становятся проекциями произведений p_k. Полезной интуицией является представление о проекции:

$$p_k :: 1^n \to 1$$

как о прототипе функции от n переменных, которая игнорирует все, кроме k-й переменной. Наоборот, постоянные морфизмы $n \to 1$ в **F** становятся диагональными морфизмами $1 \to 1^n$ в **L**. Они соответствуют дублированию переменных.

Интересными морфизмами в **L** являются те, которые определяют n-арные операции, отличные от проекций. Именно эти морфизмы отличают одну теорию Ловера от другой. Это умножения, сложения, выбор единичных элементов и т.д., которые определяют алгебру. Но чтобы сделать **L** полной категорией, нам также нужны составные операции $n \to m$ (или, что то же самое, $1^n \to 1^m$). Из-за простой структуры категории они оказываются произведениями более простых морфизмов типа $n \to 1$. Это есть обобщение утверждения о том, что функция, возвращающая произведение, является произведением функций (или, как мы видели ранее, что hom-функтор является непрерывным).



Теория Ловера **L** основана на $\mathbf{F}^{op}$, из которой она наследует «скучные» морфизмы,
которые определяют произведения. Она добавляет «интересные» морфизмы, описывающие n-арные операции (пунктирные стрелки).

Теории Ловера образуют категорию $\mathbf{Law}$, в которой морфизмы являются функторами, сохраняющими конечные произведения, и коммутативными с функторами I. Для двух таких теорий $(\mathbf{L}, I_{\mathbf{L}})$ и $(\mathbf{L}', I'_{\mathbf{L}'})$ морфизмом между ними является Функтор $F :: \mathbf{L} \to \mathbf{L}'$ такой, что:

$$F\,(m \times n) = F\,m \times F\,n$$
$$F \circ I_{\mathbf{L}} = I'_{\mathbf{L}'}$$

Морфизмы между теориями Ловера заключают в себе идею интерпретации одной теории внутри другой. Например, групповое умножение может интерпретироваться как моноидное умножение, если мы игнорируем инверсии.

Простейшим тривиальным примером категории Ловера является сама $\mathbf{F}^{op}$ (соответствующий выбору функтора тождества для $I_{\mathbf{L}}$). Эта теория Ловера, которая не имеет никаких операций или законов, является инициальным объектом в $\mathbf{Law}$.

Здесь было бы очень полезно представить нетривиальный пример теории Ловера, но было бы трудно объяснить его, не разобравшись, прежде всего в том, что такое модели.

30.3 Модели теорий Ловера

Ключом к пониманию теорий Ловера является осознание того, что одна такая теория обобщает множество отдельных алгебр, которые имеют одну и ту же структуру. Например, теория Ловера для моноидов описывает сущность того, что делает моноид моноидом. Это должно подходить для всех моноидов. Данный абстрактный моноид становится моделью такой теории. Модель определяется как функтор от теории Ловера $\mathbf{L}$ к категории множеств $\mathbf{Set}$ (существуют обобщения теорий Ловера, которые используют другие категории для моделей, но здесь я просто сконцентрируюсь на множестве). Поскольку структура $\mathbf{L}$ сильно зависит от произведений, мы требуем, чтобы такой функтор сохранял конечные произведения. Модель $\mathbf{L}$, также называемая алгеброй над теорией Ловера $\mathbf{L}$, поэтому определяется функтором:

$$M :: \mathbf{L} \to \mathbf{Set}$$
$$M\,(a \times b) \cong M\,a \times M\,b$$

Заметим, что мы требуем сохранения произведений только *с точностью до изоморфизма*. Это очень важно, потому что строгое сохранение произведений исключит из рассмотрения большинство интересных теорий.

Сохранение произведений согласно моделям означает, что образ M в **Set** представляет собой последовательность множеств, порожденных степенями множества $M\,1$ — образ объекта 1 из **L**. Обозначим это множество буквой a (его иногда называют сортом, и такая алгебра называется односортной; существуют обобщения теорий Ловера на многосортные алгебры). В частности, бинарные операции из **L** сопоставляются с функциями:

$$a \times a \to a$$

Как и для любого функтора, возможно, что несколько морфизмов в **L** свернуты в одну и ту же функцию в **Set**.

Между прочим, тот факт, что все законы универсально оцениваются равенствами, означает, что каждая теория Ловера имеет тривиальную модель: постоянный функтор, отображающий все объекты к одноэлементному множеству, а все морфизмы — к тождественной функции на нем.

Общий морфизм в **L** вида $m \to n$ отображается в функцию:

$$a^m \to a^n$$

Если у нас есть две разные модели: M и N, естественное преобразование между ними — это семейство функций, индексированных по n:

$$\mu_n :: M\,n \to N\,n$$

или, эквивалентно:

$$\mu_n :: a^n \to b^n$$

где $b = N\,1$.

Заметим, что условие естественности гарантирует сохранение n-арных операций:

$$N\,f \circ \mu_n = \mu_1 \circ M\,f$$

где $f :: n \to 1$ — n-арная операция в **L**.

Функторы, определяющие модели, образуют категорию моделей $\mathbf{Mod}(\mathbf{L},\mathbf{Set})$ с естественными преобразованиями в качестве морфизмов.

Рассмотрим модель для тривиальной категории Ловера $\mathbf{F}^{op}$. Такая модель полностью определяется ее значением на 1, $M\,1$. Так как $M\,1$ может быть любым множеством, то существует столько моделей, сколько есть множеств в $\mathbf{Set}$. Более того, каждый морфизм в $\mathbf{Mod}(\mathbf{F}^{op},\mathbf{Set})$ (естественное преобразование между функторами M и N) однозначно определяется его компонентой в $M\,1$. Наоборот, каждая функция $M\,1 \to N\,1$ индуцирует естественное преобразование между двумя моделями M и N. Поэтому $\mathbf{Mod}(\mathbf{F}^{op},\mathbf{Set})$ эквивалентна $\mathbf{Set}$.

30.4 Теория моноидов

Простейший нетривиальный пример теории Ловера описывает структуру моноидов. Это единая теория, которая осуществляет перегонку структуры всех возможных моноидов в том смысле, что модели этой теории охватывают всю категорию моноидов $\mathbf{Mon}$. Мы уже встречали универсальную конструкцию, которая показала, что каждый моноид может быть получен из соответствующего свободного моноида путем идентификации подмножества морфизмов. Таким образом, один свободный моноид уже обобщает множество моноидов. Однако существует бесконечное число свободных моноидов. Теория Ловера для моноидов $\mathbf{L}_{\mathbf{Mon}}$ объединяет все их в одной элегантной конструкции.

Каждый моноид должен иметь единицу, поэтому мы должны иметь особый морфизм η в $\mathbf{L}_{\mathbf{Mon}}$, который направлен от 0 к 1. Отметим,что в $\mathbf{F}$ не может быть соответствующего морфизма. Такой морфизм шел бы в обратном направлении от 1 к 0, который в $\mathbf{FinSet}$ был бы функцией от одноэлементного множества к пустому множеству. Такой функции не существует.

Далее, рассмотрим морфизмы $2 \to 1$, являющиеся членами $\mathbf{L}_{\mathbf{Mon}}(2, 1)$, которые должны содержать прототипы всех бинарных операций. При построении моделей в $\mathbf{Mod}(\mathbf{L}_{\mathbf{Mon}},\mathbf{Set})$ эти морфизмы будут отображаться в функции от декартова произведения $M\,1 \times M\,1$ к $M\,1$; другими словами, в функции двух аргументов.

Возникает вопрос: сколько функций от двух аргументов можно реализовать, используя только моноидальный оператор. Обозначим два аргумента как a и b. Существует одна функция, которая игнорирует оба аргумента и возвращает моноидальную единицу. Тогда существуют две проекции, которые возвращают a и b, соответственно. Затем следуют функции, которые возвращают ab, ba, aa, bb, aab и т.д. На самом деле таких функций от двух аргументов столько, сколько существуют элементов в свободном моноиде с образующими a и b. Обратите внимание, что $\mathbf{L_{Mon}}(2, 1)$ должен содержать все эти морфизмы, потому что одна из моделей является свободным моноидом. В свободном моноиде они соответствуют различным функциям. Другие модели могут свернуть несколько морфизмов в $\mathbf{L_{Mon}}(2, 1)$ до одной функции, но не к свободному моноиду.

Если обозначить свободный моноид с n образующими через n^*, то можно отождествить hom-множество $\mathbf{L_{Mon}}(2, 1)$ с hom-множеством $\mathbf{Mon}(1^*, 2^*)$ в $\mathbf{Mon}$, категории моноидов. В общем случае мы подбираем $\mathbf{L_{Mon}}(m, n)$ как $\mathbf{Mon}(n^*, m^*)$. Другими словами, категория $\mathbf{L_{Mon}}$ двойственна категории свободных моноидов.

Категория моделей теории Ловера для моноидов, $\mathbf{Mod}(\mathbf{L_{Mon}}, \mathbf{Set})$, эквивалентна категории всех моноидов, $\mathbf{Mon}$.

30.5 Теории Ловера и монады

Как вы помните, алгебраические теории могут быть описаны с использованием монад — в частности, алгебры для монад. Поэтому неудивительно, что существует связь между теориями Ловера и монадами.

Во-первых, давайте рассмотрим, как теория Ловера индуцирует монаду. Она делает это через сопряжение между забывающим функтором и свободным функтором. Забывающий функтор U назначает множество каждой модели. Это множество задается оценивающим функтором M от $\mathbf{Mod}(\mathbf{L}, \mathbf{Set})$ при объекте 1 в $\mathbf{L}$.

Другим способом получения U является использование того факта, что $\mathbf{F}^{op}$ является инициальным объектом в $\mathbf{Law}$. Это означает, что для любой теории Ловера $\mathbf{L}$ существует единственный функтор $\mathbf{F}^{op} \to \mathbf{L}$. Этот функтор индуцирует на моделях противоположный функтор (так как

модели являются функторами от теорий к множествам):

$$\mathbf{Mod}(\mathbf{L},\mathbf{Set}) \to \mathbf{Mod}(\mathbf{F}^{op},\mathbf{Set})$$

Но, как мы уже говорили, категория моделей от $\mathbf{F}^{op}$ эквивалентна **Set**, поэтому мы получаем забывающий функтор:

$$U :: \mathbf{Mod}(\mathbf{L},\mathbf{Set}) \to \mathbf{Set}$$

Можно показать, что так определенный U всегда имеет левый сопряженный, свободный функтор F.

Это легко увидеть для конечных множеств. Свободный функтор F порождает свободные алгебры. Свободная алгебра — это особая модель в $\mathbf{Mod}(\mathbf{L},\mathbf{Set})$, которая порождается из конечного множества n образующих. Мы можем реализовать F как представимый функтор:

$$\mathbf{L}(n, _) :: \mathbf{L} \to \mathbf{Set}$$

Чтобы показать, что он действительно свободный, все, что нам нужно сделать, это доказать, что он является левым сопряжением к забывающему функтору:

$$\mathbf{Mod}(\mathbf{L}(n, _), M) \cong \mathbf{Set}(n, U(M))$$

Давайте упростим правую часть

$$\mathbf{Set}(n, U(M)) \cong \mathbf{Set}(n, M\,1) \cong (M\,1)\,n \cong M\,n$$

(Я использовал тот факт, что множество морфизмов изоморфно экспоненциалу, который в этом случае является просто повторяющимся произведением.) Сопряжение является результатом леммы Йонеды:

$$[\mathbf{L},\mathbf{Set}](\mathbf{L}(n, _), M) \cong M\,n$$

Собирая все вместе: забывающий и свободный функторы определяют монаду $T = U \circ F$ на **Set**. Таким образом, каждая теория Ловера порождает монаду.

Оказывается, что категория алгебр для этой монады эквивалентна категории моделей.

Вы можете вспомнить, что монадные алгебры определяют способы оценки выражений, которые формируются с использованием монад. Теория Ловера определяет n-арные операции, которые могут использоваться для генерации выражений. Модели предоставляют средства для оценки этих выражений.

Однако связь между монадами и теориями Ловера не идет в обоих направлениях. Только конечномерные монады приводят к теориям Ловера. Конечномерная монада основана на конечномерном функторе. Конечномерный функтор на **Set** полностью определяется его действием на конечные множества. Его действие на произвольное множество a можно оценить, используя следующий ко-конец:

$$F\,a = \int^n a^n \times (F\,n)$$

Поскольку ко-конец обобщает копроизведение или сумму, приведенная формула является обобщением разложения в ряд по степеням. Или мы можем использовать интуицию, связанную с тем, что функтор является обобщенным контейнером. В этом случае конечномерный контейнер может быть охарактеризован как сумма форм и содержимого. Здесь $F\,n$ представляет собой множество форм для хранения n элементов, а содержимое представляет собой кортеж из n элементов, сам по себе являющийся элементом. Например, список (как функтор) является конечномерным, с одной формой для каждой арности. Дерево имеет больше форм с одной арностью и т.д.

Во-первых, все монады, которые генерируются из теорий Ловера, являются конечномерными, и они могут быть выражены в виде ко-концов:

$$T_{\mathbf{L}}\,a = \int^n a^n \times \mathbf{L}(n, 1)$$

И наоборот, для любой данной конечномерной монады T на **Set**, мы можем построить теорию Ловера. Начнем с построения категории Клейсли для T. Как вы помните, морфизм в категории Клейсли от a к b задается морфизмом в базовой категории:

$$a \to T\,b$$

Если ограничиваться конечными множествами, то это переходит в:

$$m \to T\, n$$

Категорией, противоположной этой категории Клейсли, $\mathbf{Kl}_T^{op}$, ограниченной конечными множествами, является рассматриваемая теория Ловера. В частности, hom-множество $\mathbf{L}(n, 1)$, описывающее n-арные операции в $\mathbf{L}$, задается hom-множеством $\mathbf{Kl}_T(1, n)$.

Оказывается, большинство монад, с которыми мы сталкиваемся в программировании, являются конечномерными, за исключением монады продолжения. Можно расширить понятие теории Ловера за пределы конечномерных операций.

30.6 Монады как ко-концы

Давайте исследуем формулу ко-конца.

$$T_{\mathbf{L}}\, a = \int^n a^n \times \mathbf{L}(n, 1)$$

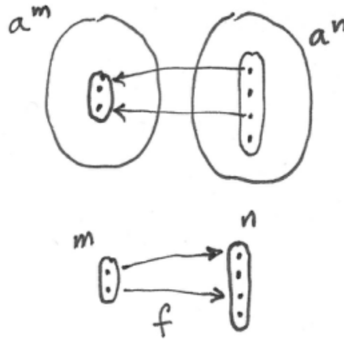
Начнем с того, что ко-конец берется над профунктором P в $\mathbf{F}$, определяемым как:

$$P\, n\, m = a^n \times \mathbf{L}(m, 1)$$

Этот профунктор контравариантен по первому аргументу n. Задумайтесь, как он поднимает морфизмы. Морфизм в $\mathbf{FinSet}$ является отображением конечных множеств $f :: m \to n$. Такое отображение характеризует выбор m элементов из n-элементного множества (допускаются повторения). Он может быть поднят до отображения степеней a, а именно (обратите внимание на направление):

$$a^n \to a^m$$

Подъем просто выбирает m элементов из кортежа, содержащего n элементов $(a_1, a_2, \ldots, a_n)$ (возможно, с повторениями).



Например, рассмотрим функцию $f_k :: 1 \to n$ — выбор k-го элемента из n-элементного набора. Она поднимается до функции, которая принимает n-элементный кортеж из a и возвращает его k-й элемент.

Или возьмем $f :: m \to 1$ — постоянную функцию, которая отображает все m элементов в единицу. Ее подъем — это функция, которая принимает один элемент из a и дублирует его m раз:

$$\lambda x \to (\underbrace{x, x, \ldots, x}_{m})$$

Вы могли заметить, хотя это не очевидно, что рассматриваемый профунктор является ковариантным по второму аргументу. hom-функтор $\mathbf{L}(m, 1)$ фактически контравариантен по m. Тем не менее, мы выбираем ко-конец не из категории $\mathbf{L}$, а из категории $\mathbf{F}$. Переменная ко-конца n пересчитывает конечные множества (или их скелеты). Категория $\mathbf{L}$ содержит противоположности содержимого $\mathbf{F}$, поэтому морфизм $m \to n$ в $\mathbf{F}$ является членом $\mathbf{L}(n, m)$ в $\mathbf{L}$ (вложение задается функтором $I_{\mathbf{L}}$).

Проверим функториальность $\mathbf{L}(m, 1)$, как функтора от $\mathbf{F}$ к $\mathbf{Set}$. Мы хотим поднять функцию $f :: m \to n$, поэтому наша цель — реализовать функцию от $\mathbf{L}(m, 1)$ к $\mathbf{L}(n, 1)$. В соответствии с функцией f существует морфизм в $\mathbf{L}$ от n к m (обратите внимание на направление). Предварительная компоновка этого морфизма с $\mathbf{L}(m, 1)$ дает нам подмножество в $\mathbf{L}(n, 1)$.

$$\mathbf{L}(m, 1) \longrightarrow \mathbf{L}(n, 1)$$

$$m \bullet \xrightarrow{\quad f \quad} \bullet n$$

Заметим, что, поднимая функцию $1 \to n$, можно перейти от $\mathbf{L}(1,1)$ к $\mathbf{L}(n,1)$. Мы будем использовать этот факт позже.

Произведением контравариантного функтора a^n и ковариантного функтора $\mathbf{L}(m,1)$ является профунктор $\mathbf{F}^{op} \times \mathbf{F} \to \mathbf{Set}$. Напомним, что ко-конец может быть определен как копроизведение (непересекающаяся сумма) всех диагональных членов профунктора, в котором идентифицированы некоторые элементы. Идентификация соответствует условиям ко-клина.

Здесь ко-конец начинается как дизъюнктная сумма множеств $a^n \times \mathbf{L}(n,1)$ по всем n. Идентификация может быть сгенерирована путем выражения ко-конца в качестве ко-уравнителя. Начнем с недиагонального члена $a^n \times \mathbf{L}(m,1)$. Чтобы перейти к диагонали, мы можем применить морфизм $f :: m \to n$ либо к первому, либо ко второму компоненту произведения. Оба результата затем идентифицируются.

$$
\begin{array}{ccccc}
 & & a^n \times \mathbf{L}(m,1) & & \\
 & \swarrow^{\langle f, \mathrm{id} \rangle} & & \searrow^{\langle \mathrm{id}, f \rangle} & \\
a^n \times \mathbf{L}(m,1) & & \sim & & a^n \times \mathbf{L}(n,1)
\end{array}
$$

$$f :: m \to n$$

Ранее я показал, что подъем $f :: 1 \to n$ приводит к следующим двум преобразованиям:

$$a^n \to a$$

и

$$\mathbf{L}(1,1) \to \mathbf{L}(n,1)$$

Поэтому, начиная с $a^n \times \mathbf{L}(1,1)$, мы можем достичь их обоих:

$$a \times \mathbf{L}(1,1)$$

когда мы поднимаем $\langle f, \mathrm{id} \rangle$, и

$$a^n \times \mathbf{L}(n,1)$$

когда мы поднимаем $\langle \mathrm{id}, f \rangle$. Это не означает, однако, что все элементы из $a^n \times \mathbf{L}(n,1)$ можно отождествить с $a \times \mathbf{L}(1,1)$. Это связано с тем, что не

все элементы $\mathbf{L}(n, 1)$ могут быть достигнуты от $\mathbf{L}(1, 1)$. Напомню, что мы можем только поднять морфизмы из $\mathbf{F}$. Нетривиальная n-арная операция в $\mathbf{L}$ не может быть построена путем поднятия морфизма $f :: 1 \to n$.

Другими словами, мы можем только идентифицировать все слагаемые в формуле ко-конца, для которых $\mathbf{L}(n, 1)$ можно получить от $\mathbf{L}(1, 1)$ с помощью применения основных морфизмов. Все они эквивалентны $a \times \mathbf{L}(1, 1)$. Основными морфизмами являются те, которые являются образами морфизмов в $\mathbf{F}$.

Посмотрим, как это работает в простейшем случае теории Ловера, самой $\mathbf{F}^{op}$. В такой теории каждый $\mathbf{L}(n, 1)$ можно получить от $\mathbf{L}(1, 1)$. Это связано с тем, что $\mathbf{L}(1, 1)$ является синглетоном, содержащим только тождественный морфизм, а $\mathbf{L}(n, 1)$ содержит только морфизмы, соответствующие инъекциям $1 \to n$ в $\mathbf{F}$, которые являются основными морфизмами. Поэтому все слагаемые в копроизведении эквивалентны, и мы получаем:

$$T\, a = a \times \mathbf{L}(1, 1) = a$$

что является тождественной монадой.

30.7 Теория Ловера побочных эффектов

Поскольку существует такая сильная связь между монадами и теориями Ловера, естественно спросить, можно ли использовать теории Ловера в программировании в качестве альтернативы монадам. Основная проблема с монадами заключается в том, что они плохо компануются. Не существует общего рецепта построения монадных преобразователей. Теории Ловера имеют преимущество в этой области: они могут быть скомпонованы с использованием копроизведений и тензорных произведений. С другой стороны, только конечномерные монады могут быть легко преобразованы в теории Ловера. Исключением здесь является монада продолжения. В этой области ведутся исследования (см. Библиографию).

Чтобы дать вам представление о том, как теория Ловера может быть использована для описания побочных эффектов, я рассмотрю простой случай исключений, которые традиционно реализуются с помощью монады `Maybe`.

Монада `Maybe` строится с помощью теории Ловера с одной нульарной операцией $0 \to 1$. Модель этой теории является функтором, который отображает 1 в некоторое множество a и отображает нульарную операцию в функцию:

```
raise :: () -> a
```

Мы можем восстановить монаду `Maybe`, используя формулу ко-конца. Рассмотрим, к чему приводит добавление нульарной операции к hom-множествам $\mathbf{L}(n, 1)$. Помимо создания нового $\mathbf{L}(0, 1)$ (который отсутствует в $\mathbf{F}^{op}$), также добавляются новые морфизмы в $\mathbf{L}(n, 1)$. Это результаты композиции морфизма типа $n \to 0$ с нашим $0 \to 1$. Такие добавления отождествляются с $a^0 \times \mathbf{L}(0, 1)$ в формуле ко-конца, поскольку они могут быть получены из:

$$a^n \times \mathbf{L}\,(0, 1)$$

подъемом $0 \to n$ двумя различными способами.

$$\begin{array}{ccc} & a^0 \times \mathbf{L}\,(0,1) & \\ \swarrow^{\langle f, \mathrm{id} \rangle} & & \searrow^{\langle \mathrm{id}, f \rangle} \\ a^0 \times \mathbf{L}\,(0,1) & \sim & a^n \times \mathbf{L}\,(n,1) \end{array}$$

$$f :: 0 \to n$$

Этот ко-конец сводится к:

$$T_{\mathbf{L}}\, a = a^0 + a^1$$

или, используя нотацию Haskell:

```
type Maybe a = Either () a
```

что эквивалентно

```
data Maybe a = Nothing | Just a
```

Обратите внимание, что эта теория Ловера поддерживает только поднятие исключений, а не их обработку.

Упражнения

1. Перечислите все морфизмы между 2 и 3 в **F** (скелет **FinSet**).
2. Покажите, что категория моделей для теории моноидов Ловера эквивалентна категории монадных алгебр для монады списка.
3. Теория моноидов Ловера порождает монаду списка. Покажите, что ее бинарные операции могут быть сгенерированы с использованием соответствующих стрелок Клейсли.
4. **FinSet**является подкатегорией **Set** и существует функтор, который встраивает его в **Set**. Любой функтор к Set может быть ограничен до **FinSet**. Покажите, что конечномерный функтор является левым расширением Кана его собственного ограничения.

Библиография

1. Gordon Plotkin and John Power. *Notions of computation determine monads*.[1]
2. F. William Lawvere. *Functorial Semantics of Algebraic Theories*.[2]

[1] http://homepages.inf.ed.ac.uk/gdp/publications/Comp_Eff_Monads.pdf
[2] http://www.tac.mta.ca/tac/reprints/articles/5/tr5.pdf

Глава 31

Монады, моноиды и категории

Не существует лучшей страницы, претендующей на завершающую, для книги по теории категорий. Всегда найдется что еще изложить. Теория категорий — обширная область. В то же время очевидно, что одни и те же темы, концепции и шаблоны постоянно появляются снова и снова. Существует мнение, что все понятия являются расширениями Кана и, действительно, вы можете использовать расширения Кана для получения пределов, копределов, сопряжений, монад, леммы Йонеды и т.д. Само понятие категории возникает на всех уровнях абстракций, также как и понятия моноида и монады. Какое из них самое основополагающее? Как оказалось, все они взаимосвязаны, одно ведет к другому в бесконечном цикле абстракций. Я решил, что демонстрация этих взаимосвязей может быть хорошим ходом для завершения этой книги.

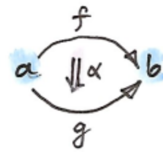
31.1 Бикатегории

Одним из самых сложных аспектов теории категорий является постоянное переключение перспектив. Возьмем, например, категорию множеств. Мы используем определение множеств в терминах элементов. Пустое множество не имеет элементов. Одноэлементный множество содержит один элемент. Декартовым произведением двух множеств является множество пар и т.д. Но говоря о категории **Set**, я попросил вас забыть о содержании множеств и вместо этого сосредоточиться на морфизмах

(стрелках) между ними. Вам разрешалось время от времени заглядывать под обертки, чтобы увидеть, какая конкретная универсальная конструкция в **Set** описана с точки зрения элементов. Терминальный объект оказался множеством с одним элементом и т.д. Но это были просто проверки здравомыслия.

Функтор определяется как отображение категорий. Естественно рассматривать отображение как морфизм в категории. Функтор оказался морфизмом в категории категорий (малых категорий, если мы хотим избежать вопросов о размере). Рассматривая функтор как стрелку, мы теряем информацию о его действии на внутренностях категории (ее объектах и морфизмах), так же, как мы теряем информацию о действии функции на элементах множества, когда мы рассматриваем ее как стрелку в **Set**. Но функторы между любыми двумя категориями также образуют категорию. На этот раз вас попросят рассмотреть то, что было стрелкой в одной категории, в качестве объекта в другом. В категории функторов, функторы — это объекты, а естественные преобразования — морфизмы. Мы обнаружили, что одно и то же может быть стрелкой в одной категории и объектом в другом. Наивное рассмотрение объектов как существительных и стрелок как глаголов не имеет силы.

Вместо переключения между двумя точками зрения, мы можем попытаться объединить их в одно целое. Так мы получаем понятие 2-категории, в которой объекты называются 0-клетками, морфизмы являются 1-клетками, а морфизмы между морфизмами — это 2-клетки.



0-клетки a, b; 1-клетки f, g; 2-клетка α.

Категория категорий **Cat** является подходящим примером. В ней категории являются 0-клетками, функторы — 1-клетками, а естественные преобразования — 2-клетками. Законы 2-категории утверждают, что 1-клетки между любыми двумя 0-клетками образуют категорию (другими словами, $\mathbf{C}(a, b)$ является hom-категорией, а не hom-множеством). Это хорошо согласуется с нашим предыдущим утверждением о том, что

функторы между любыми двумя категориями образуют категорию функторов.

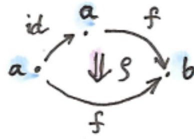
В частности, 1-клетки от любой 0-клетки обратно к себе также образуют категорию, hom-категорию $\mathbf{C}(a, a)$; эта категория имеет даже более развитую структуру. Члены $\mathbf{C}(a, a)$ можно рассматривать как стрелки в $\mathbf{C}$ или как объекты в $\mathbf{C}(a, a)$. Как стрелки, они могут быть скомпонованы друг с другом. Но когда мы рассматриваем их в качестве объектов, композиция становится отображением от пары объектов к объекту. Фактически это очень похоже на произведение — точнее — тензорное произведение. Это тензорное произведение имеет единицу: тождественную 1-ячейку. Оказывается, что в любой 2-категории hom-категория $\mathbf{C}(a, a)$ автоматически является моноидальной категорией с тензорным произведением, определяемым как композиция 1-клеток.Законы ассоциативности и единицы просто следуют из соответствующих законов категории.

Давайте рассмотрим, что это означает в нашем каноническом примере 2-категории $\mathbf{Cat}$. hom-категория $\mathbf{Cat}(a, a)$ является категорией эндофункторов на a. Композиция эндофунторов в ней играет роль тензорного произведения. Тождественный функтор является единицей относительно этого произведения. Мы уже знаем, что эндофункторы образуют моноидальную категорию (мы использовали этот факт в определении монады), но теперь мы видим, что это более общий феномен: эндо-1-клетки в любой 2-категории образуют моноидальную категорию. Мы вернемся к этому, когда будем обобщать монады.

Вы могли бы вспомнить, что в общей моноидальной категории мы не настаивали на однозначном выполнении моноидных законов. Часто бывает достаточно, чтобы единичные законы и законы ассоциативности выполнялись с точностью до изоморфизма. В 2-категории моноидальные законы в $\mathbf{C}(a, a)$ следуют из законов композиции для 1-клеток. Эти законы строги, поэтому мы всегда будем получать строгую моноидальную категорию. Однако, можно также и смягчить эти законы. Можно сказать, например, что композиция тождественной 1-клетки id_a с другой 1-клеткой $f :: a \to b$ изоморфна, но не равна, f. Изоморфизм 1-клеток определяется с помощью 2-клеток. Другими словами, существует 2-клетка:

$$\rho :: f \circ \mathrm{id}_a \to f$$

которая имеет обратную.



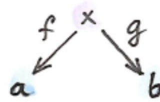
Закон тождественности выполняется в бикатегории с точностью до изоморфизма (обратимая 2-клетка ρ).

Мы можем проделать то же самое для законов левой тождественности и левой ассоциативности. Такая разновидность расслабленной 2-категории называется бикатегорией (существуют некоторые дополнительные законы связности, которые я здесь опускаю).

Как и ожидалось, эндо-1-клетки в бикатегории образуют общую моноидальную категорию с нестрогими законами.

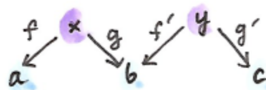
Интересным примером бикатегории является категория пролетов **Span**. Пролет между двумя объектами a и b представляет собой объект x и пару морфизмов:

$$f :: x \to a$$
$$g :: x \to b$$



Вы можете вспомнить, что мы использовали интервалы в определении категорного произведения. Здесь мы хотим рассматривать интервалы в виде 1-клеток в бикатегории. Первый шаг — определить композицию интервалов. Предположим, что имеется примыкающий интервал:

$$f' :: y \to b$$
$$g' :: y \to c$$



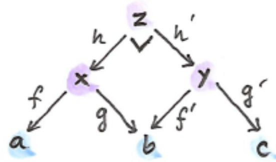
Композицией будет третий интервал с некоторой вершиной z. Наиболее естественным выбором для него является обратный образ g вдоль f'. Напомню, что обратный образ — это объект z вместе с двумя морфизмами:

$$h :: z \to x$$
$$h' :: z \to y$$

такими, что

$$g \circ h = f' \circ h'$$

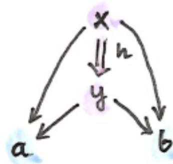
который является универсальным среди всех подобных объектов.



Пока же, давайте сосредоточимся на интервалах над категорией множеств. В этом случае обратный образ является всего лишь множеством пар (p, q) из декартова произведения $x \times y$, для которого:

$$g\, p = f'\, q$$

Морфизм между двумя интервалами, которые имеют одни и те же конечные точки, определяется как морфизм h между их вершинами так, что соответствующие треугольники являются коммутативными.



2-ячейка в **Span**

Подводя итог, в бикатегории **Span**: 0-клетки — это множества, 1-клетки — это интервалы, 2-клетки — это морфизмы над интервалами. Тождественная 1-клетка представляет собой вырожденный интервал, в котором все три объекта одинаковы, а два морфизма являются тождественными.

Мы встречались с еще одним примером бикатегории: бикатегорией **Prof** профункторов, в которой 0-клетки являются категориями, 1-клетки — это профункторы, а 2-клетки — естественные преобразования. Композиция профункторов задается посредством ко-конца.

31.2 Монады

К настоящему моменту вы должны быть хорошо знакомы с определением монады как моноида в категории эндофункторов. Давайте перейдем к этому определению с новым пониманием того, что категория эндофункторов — это всего лишь одна небольшая hom-категория эндо-1-клеток в бикатегории **Cat**. Мы знаем, что это моноидальная категория: тензорное произведение происходит из композиции эндофункторов. Моноид определяется как объект в моноидальной категории — здесь он будет эндофунктором T — вместе с двумя морфизмами. Морфизмы между эндофунторами являются естественными преобразованиями. Один из морфизмов отображает моноидальную единицу — тождественный эндофунктор — к T:

$$\eta :: I \to T$$

Еще один морфизм отображает тензорное произведение $T \otimes T$ на T. Тензорное произведение задается эндофункторной композицией, поэтому получаем:

$$\mu :: T \circ T \to T$$

Мы признаем их в качестве двух операций, определяющих монаду (они называются `return` и `join` в Haskell), и мы знаем, что моноидные законы обращаются к монадным законам.

Теперь давайте удалим все упоминания об эндофункторах из этого определения. Начнем с бикатегории **C** и возьмем в ней 0-клетку a. Как мы видели ранее, hom-категория $\mathbf{C}(a, a)$ является моноидальной категорией. Поэтому мы можем определить моноид в $\mathbf{C}(a, a)$, выбирая 1-клетку, T, и две 2-клетки:

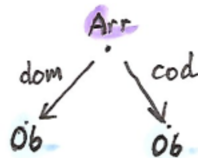
$$\eta :: I \to T$$
$$\mu :: T \circ T \to T$$

удовлетворяющих моноидным законам. Мы называем *это* монадой.

Это гораздо более общее определение монады, использующее только 0-клетки, 1-клетки и 2-клетки. Оно сводится к обычной монаде при применении к бикатегории **Cat**. Но посмотрим, что произойдет в других бикатегориях.

Давайте построим монаду в **Span**. Мы выбираем 0-клетку, которая представляет собой множество, которое по причинам, которые скоро станут понятными, я обозначу Ob. Затем мы выбираем эндо-1-клетку: пролет от Ob обратно к Ob. Он имеет множество на вершине, который я буду обозначать Ar, снабженный двумя функциями:

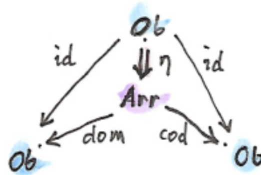
$$dom :: Ar \to Ob$$
$$cod :: Ar \to Ob$$



Назовем элементы множества Ar «стрелками». Если я также скажу вам, что вы называете элементы Ob «объектами», вы можете получить подсказку, к которой все это приводит: две функции dom и cod присваивают домен и кодомен «стрелке».

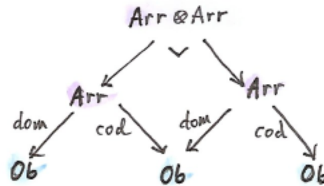
Чтобы превратить наш пролет в монаду, нужны две 2-клетки, η и μ. В этом случае моноидальная единица представляет собой тривиальный пролет от Ob до Ob с вершиной в Ob и двумя тождественными функциями. 2-клетка η является функцией между вершинами Ob и Ar. Другими словами, η каждому «объекту» назначает «стрелку». 2-клетка в **Span** должна удовлетворять условиям коммутативности — в этом случае:

$$dom \circ \eta = \mathrm{id}$$
$$cod \circ \eta = \mathrm{id}$$



В компонентах это принимает вид: $dom\,(\eta\, ob) = ob = cod\,(\eta\, ob)$, где ob — «объект» в Ob. Другими словами, η делает назначение каждому «объекту» и «стрелке», чей домен и кодомен представляют собой «объект». Эту специальную «стрелку» назовем «стрелкой идентификации».

Вторая 2-клетка μ действует на композицию пролета Ar с собой. Композиция определяется как обратный образ, поэтому ее элементами являются пары элементов из Ar — пары «стрелок» (a_1, a_2). Условие обратного образа: $cod\, a_1 = dom\, a_2$. Стрелки a_1 и a_2 являются «компонуемыми», т.к. область одной есть кообласть другой.



2-клетка μ — это функция, которая отображает пару компонуемых стрелок (a_1, a_2) к одной стрелке a_3 из Ar. Другими словами, μ определяет композицию стрелок.

Легко проверить, что законы монады соответствуют законам тождественности и ассоциативности для стрелок. Мы только что определили категорию (малую категорию, в которой, заметьте, объекты и стрелки образуют множества). Итак, все говорит о том, что категория — это просто монада в бикатегории пролетов.

Удивительное в этом результате то, что категории ставятся на ту же основу, что и другие алгебраические структуры, такие как монады и моноиды. Нет ничего особенного в том, чтобы быть категорией. Это всего лишь два множества и четыре функции. Фактически даже нет необходимости в отдельном множестве для объектов, потому что объекты могут быть идентифицированы с помощью тождественных стрелок (они находятся во взаимно однозначном соответствии). Так что категория, на самом деле, одно множество и немного функций. Учитывая ключевую роль, которую играет теория категорий во всей математике, это очень скромная реализация.

Упражнения

1. Выведите законы единицы и ассоциативности для тензорного произведения, определяемого композицией эндо-1-клеток в бикатегории.
2. Проверьте, что монадные законы для монады в **Span** соответствуют законам тождественности и ассоциативности в результирующей категории.
3. Покажите, что монада в **Prof** является функтором тождественности на объектах.
4. Что такое монадная алгебра для монады в **Span**?

Библиография

1. Paweł Sobociński's blog[1].

[1] https://graphicallinearalgebra.net/2017/04/16/

Литература

[1] Артамонов В.А. *Универсальные алгебры*. В кн. Общая алгебра. Т.2. Под ред. Л.А. Скорнякова. М.: Наука, 1991, с.295-367.

[2] Власов В.Н. *Haskell: введение в функциональное программирование* (https://wiki.nsunc.com/haskell).

[3] Гельфанд С.И., Манин Ю.И. *Методы гомологической алгебры*. Т.1. Введение в теорию когомологий и производные категории. М.: Наука, 1988.

[4] Голдблатт Р. *Топосы. Категорный анализ логики*: Пер. с англ. М.: Мир, 1983. 488 с.

[5] Душкин Р. В. *14 занимательных эссе о языке Haskell и функциональном программировании*. М.: ДМК Пресс, 2011. 140 с.

[6] Душкин Р. В. *Другие 14 эссе о языке Haskell и функциональном программировании — серьёзные*. 2012. 365 с.

[7] Душкин Р. В. *Практика работы на языке Haskell*. М.: ДМК Пресс, 2010. 288 с.

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

[9] Душкин Р. В. *Функциональное программирование на языке Haskell*. М.: ДМК Пресс, 2007. 608 с.

[10] Ершов А.В. *Категории и функторы*. Учебное пособие. Саратов: ООО Издательский центр «Наука», 2012. 86 с.

[11] Ершов А.В. *Функторные морфизмы*. Учебное пособие. Саратов: ООО Издательский центр «Наука», 2012. 92 с.

[12] Захаров В.К., Михалев А.В. *Локальная теория классов и множеств как основание для теории категорий*. В кн. Математические методы и приложения. Труды девятых математических чтений МГСУ. М.: МГСУ, 2002, с.91-94.

[13] Кацов Е.Б. *Тензорное произведение модулей* // Сиб. матем. журнал, т.19, №2, с.318-327 (1978).

[14] Кашу А.И. *Функторы и кручения в категориях модулей*. Кишинев: Академия Наук республики Молдова. Институт математики, 1997.

[15] Курош А.Г., Лифшиц А.Х., Шульгейфер Е.Г. *Основы теории категорий* // Успехи матем. наук, 1960, т.15, №6, с.3-52.

[16] Лифшиц А.Х., Цаленко М.Ш., Шульгейфер Е.Г. *Теория категорий*. Алгебра. Топология. Итоги науки. М.:ВИНИТИ, 1962, с.90-106.

[17] Маклейн С. *Категории для работающего математика*. М.: ФИЗМАТЛИТ, 2004. 352 с.

[18] Мальцев А.И. *Алгебраические системы*. М.: Наука, 1970.

[19] Пирс Б. *Типы в языках программирования*. Перевод с англ. М.: Издательство «Лямбда пресс»: «Добросвет», 2011. 656+xxiv с.

[20] Плоткин Б.И. *Универсальная алгебра, алгебраическая логика и базы данных*. М.: Наука, 1991. 448 с.

[21] Полин С.В. *Категория функторов в многообразии универсальных алгебр*. В сб. Матем. исследования. Кишинев, т.8, вып. 1(27), Штиинца, 1973, с.130-140.

[22] Полин С.В. *Морита-эквивалентность категорий* // Вестник Москов. ун-та, сер. матем., 1974, №2, с.41-45

[23] Журнал «*Практика функционального программирования*» (сайт журнала: http://fprog.ru/[2]).

[24] Филд А., Харрисон П. *Функциональное программирование*. М.: Мир, 1993. 637 с.

[25] Цаленко М.Ш. *Моделирование семантики в базах данных*. М.: Наука, 1989. 288 с.

[26] Цаленко М.Ш. Шульгейфер Е.Г. *Категории*. Алгебра. Топология. Геометрия, 1967. Итоги науки. М.: ВИНИТИ, 1969, с.9-57.

[27] Цаленко М.Ш. Шульгейфер Е.Г. *Основы теории категорий*. М.: Наука, 1974.

[28] Цаленко М.Ш. Шульгейфер Е.Г. *Категории*. Алгебра. Топология. Геометрия. Итоги науки и техники. М.: ВИНИТИ, 1975, с.51-147.

[29] Шульгейфер Е.Г. *Категории*. В кн. Общая алгебра. Т.2. Под ред. Л.А. Скорнякова. М.: Наука, 1991, с.368-460.

[2]Через Google недоступен, работает доступ через Яндекс с игнорированием предупреждения (другие поисковики не проверялись)

Предметный указатель

0-,1-,2-клетка, 396
2-категория, 156, 396

F-алгебра, 321, 323
F-коалгебра, 334

hom-множество, 26
hom-функтор, 204

T-алгебра, 341

алгебра типов, 75
алгебраические типы данных, 78
арность, 406
асимметрия, 61
ассоциативность композиции, 6
ассоциатор, 300, 387
базовые опрации произведений, 408
бесплатные теоремы, 147
би-категория, 426
биекция, 63
бикатегории, 423
бикатегория, 365
бифунктор, 103
вертикальная композиция, 154
вложение Йонеды, 225
вывод типов, 13
гомоморфизм
  F-алгебр, 327
  моноидный, 198
горизонтальная композиция, 159
двойственность, 51
денотационная семантика, 17
диестественное преобразование, 353
дизъюнктивное объединение, 59
дно, 15
естественное преобразование, 142, 358
закон исключения третьего, 403
изоморфизм, 48, 51
  hom-множества, 237
  Карри-Говарда, 20, 137
  естественный, 236
инициальная алгебра, 328
карринг, 129
катаморфизм, 332
категория, 3
  Cat, 101, 103, 424
  F-алгебр, 327
  Hask, 16
  Law, 410
  Span, 426, 429
  Клейсли, 45, 267, 345
  би-декартово замкнутая, 134
  двойная, 365
  двойственная, 51
  декартово замкнутая, 133
  дискретная, 356
  локально малая, 385
  малая, 101, 385
  множеств, 14
  мноидальная, 300
  моноидальная, 387
  обогащенная, 203, 385
  предмоноидальная, 104
  самообогащенная, 394

свободная, 26
тонкая, 26, 227
функторов, 153
класс типов, 90
классификатор подобъектов, 398
клин, 354
универсальный, 355
условие, 355
ко-клин, 360
ко-конец, 360
ко-конус, 189
ко-пролет, 185
ко-уравнитель, 360
комонада, 307
Product, 309
Store, 317
Stream, 312
композиция, ix, 4
ассоциативность, 6
вертикальная, 154, 157
горизонтальная, 159, 302
конец, 354
конус, 175
универсальный, 176
копредел, 189
копроизведение объектов, 59
кострелка Клейсли, 308
лемма
Йонеды, 213
ко-Йонеды, 222
ниндзя Йонеды, 363
линза, 347
метрическое пространство, 392
множество
hom-, 123
внешнее hom-, 124
внутреннее hom-, 124
множество hom-, 26
модель теории Ловера, 410
монада, 268, 428
ко-плотная, 373
моноид, 27
свободный, 201
морфизм, 3
неподвижная точка, 326
нотация
бесточечная, 30
точечная, 30
образующие, 274
объект, 3
инициальный, 48
терминальный, 49
операционная семантика, 17
отношение
предпорядка, 26
отношение эквивалентности, 361
полиморфизм
параметрическмй, 146
специализированный, 146
полукольцо, 76
порядок
полный, 26
частичный, 26
предел, 178
предикат, 23
предпорядок, 391
предпучок, 181
произведение
декартово, 104
объектов, 58
пролет, 190, 426, 429
профунктор, 119

развертка, 336
расширение Кана
как конец, 376
как сопряжение, 371
левое, 373
правое, 368
рекурсия, 325

свертка, 333
свободная конструкция, 196
сигнатура типа, 28
сопоставление с образцом, 74
сопряжение, 304
стиль программирования
    декларативный, 165
    императивный, 165
стрелка, 3
тензорное произведение, 300
теория Ловера, 408
теория моноидов, 412
тип-произведения, 65
тип-сумма, 71
тождественность, 6
топос, 398, 402
транспортировка, 144
уединитель, 300
универсальная конструкция, 47
уравнитель, 183
условие естественности, 143
факторизатор, 58
функтор, 81
    hom-, 118
    List, 93
    Maybe, 84
    Reader, 95
    Writer, 114
    вполне точный, 225
    забывающий, 199
    ко-свободный, 347
    ковариантный, 118
    контравариантный, 117, 119
    левый сопряженный, 243
    обогащенный, 393
    постоянный, 83
    правый сопряженный, 243
    представимый, 207
    свободный, 381
функториальность, 104
функция
    запоминающая, 23
    инъективная, 63
    незавершаюшаяся, 15
    обогащенная, 114
    применение (оценка), 125
    с побочными эффектами, 35
    связывания, 270
    сюръективная, 63
    частичная, 16
    чистая, 19
эквивалентное преобразование, 86
экспоненциал, 132
экстенсиональное равенство, 29
эндофунктор, 84

Благодарности

Я хотел бы поблагодарить Edward Kmett и Gershom Bazerman за проверку моей математики и логики, а также многих волонтеров, которые исправляли мои ошибки и описки, улучшая книгу.

Я благодарен Andrew Sutton за переписывание кода моей концепции моноида C++ в соответствии с его и Bjarne Stroustrup последним предложением.

Выражаю благодарность Eric Niebler за то, что он прочитал черновик и предоставил умную реализацию компоновки, которая использует расширенные возможности C++14 для вывода типов. В этой связи я смог избавиться от большой части старомодной магии шаблонов, которая делала то же самое, используя черты типа.