функционального программирования

Бартош Милевски

Данная книга содержит изложение теоретических и прикладных основ функционального программирования, базирующихся на понятиях теории категорий. Назначение книги (как и предыдущей книги автора, «Теория категорий для программистов», множества его публикаций и видео-трансляций) — подготовить заинтересованного читателя к более глубокому погружению в эти области.

Подготовка и оформление перевода произведены полностью с использованием системы LyX.

Перевод и редактирование:
Геннадий Чернышев
(https://henrychern.wordpress.com/)

The Dao of Functional Programming

Bartosz Milewski

(Last updated: May 13, 2023)

Оглавление

Предисловие

Большинство обучающих примеров по программированию, следуя Брайну Кернигану (Brian Kernighan), демонстрируют вывод фразы «Hello World!». Естественно желание получить немедленное удовлетворение от того, что компьютер выполняет ваши команды и печатает эти знаменитые слова. Но настоящее мастерство компьютерного программирования кроется глубже, а такой простой результат может дать вам только ложное ощущение силы, когда в действительности вы просто повторяете известное. Если вы стремитесь просто освоить полезный, хорошо оплачиваемый навык, то обязательно начните с «Hello World!»-программы. Существует множество книг и курсов, которые учат как записывать код на любом языке. Однако, если ваша цель — добраться до сути программирования, то нужно проявить терпение и настойчивость.

Теория категорий — это раздел математики, предоставляющий абстракции, соответствующие практическому опыту программирования. Перефразируя фон Клаузевица: программирование — это просто продолжение математики другими средствами. Многие сложные идеи теории категорий становятся очевидными для программистов, когда они объясняются в терминах типов данных и функций. В этом смысле теория категорий может быть более доступной для программистов, чем для профессиональных математиков.

Столкнувшись с новыми категорными концепциями, я часто искал их в Википедии или nLab или перечитывал главу из МакЛейна или Келли. Это отличные источники, но они требуют предварительного знакомства с темами и способности заполнить пробелы. Одна из целей этой книги — обеспечить начальную загрузку для продолжения изучения теории категорий.

В теории категорий и компьютерных науках содержится много фольклорных знаний, которых нет в литературе. Очень трудно приобрести полезную интуицию, проходя через сухие определения и теоремы. Я пытался, насколько это возможно, восполнить недостающие интуитивные представления и объяснить, не только «что», но и «почему».

Название этой книги навеяно книгой Бенджамина Хоффа «Дао Пуха» (Benjamin Hoff. "The Tao of Pooh") и книгой Роберта Пирсига «Дзен и искусство ухода за мотоциклом» (Robert Pirsig. "Zen and the Art of Motorcycle Maintenance"), которые представляют собой попытки жителей Запада усвоить элементы пасхальной философии. Грубо говоря, идея

состоит в том, что теория категорий относится к программированию так же, как Дао[1] относится к западной философии. Многие определения теории категорий не имеют особого смысла при первом чтении, но со временем вы научитесь ценить их глубокую мудрость. Если бы теорию категорий можно было резюмировать одним расхожим рассуждением, это было бы: «Вещи определяются их отношением ко Вселенной».

Теория множеств

Традиционно теория множеств считалась основой математики, хотя, в последнее время, на эту роль претендует теория типов. В определенном смысле, теория множеств — это язык ассемблера математики, и поэтому она содержит множество деталей реализации, которые часто затемняют представление идей более высокого уровня.

Теория категорий не пытается заменить теорию множеств и часто используется для построения абстракций, которые позже моделируются с помощью множеств. Фактически, фундаментальная теорема теории категорий, лемма Йонеды, связывает категории с их моделями в теории множеств. Можно найти полезную интуицию в компьютерной графике, где мы строим абстрактные миры и манипулируем ими только для того, чтобы в последний момент спроецировать и подготовить их для цифрового дисплея.

Не обязательно свободно владеть теорией множеств, чтобы изучать теорию категорий. Но, некоторое знакомство с основами необходимо. Например, понимание того, что множества содержат элементы. Для заданных, множества S и элемента a, имеет смысл поинтересоваться, является ли a элементом S или нет. Эта формулировка записывается как $a \in S$ (a является элементом S). Также возможно «наличие» пустого множество, не содержащего элементов.

Важным свойством элементов множества является то, что их можно сравнивать на равенство. Имея два элемента $a \in S$ и $b \in S$, можно спросить: равны ли a и b? Или можно ввести условие, что $a = b$, в случае, если a и b являются результатом двух разных рецептов выбора элементов множества S. Равенство элементов множества — суть всех коммутативных диаграмм в теории категорий.

[1] Дао — более современное написание Тао.

Декартово произведение двух множеств $S \times T$ определяется как множество всех пар элементов $\langle s, t \rangle$, таких что $s \in S$ и $t \in T$.

Функция $f : S \to T$, от исходного множества, называемого *областью* f, к целевого множеству, называемого *ко-областью*, также определяется как множество. Это пары вида $\langle s, t \rangle$, где $t = fs$. Здесь, fs — результат действия функции f на аргумент s. Обычно, более знакомым является обозначение $f(s)$, для применения функции к аргументу, но здесь будем следовать соглашению Haskell об опускании круглых скобок (и запятых, для функций с несколькими переменными).

В процедурном программировании принято, что функции определяются последовательностью инструкций (алгоритмом). В начале представляется аргумент s, затем описываются применения инструкций, чтобы в конечном итоге получить результат t. При этом, зачастую, приходится оценивать, сколько времени потребуется для получения результата и, вообще, завершится ли алгоритм. В математике предполагается, что для любого заданного аргумента $s \in S$, результат $t \in T$ доступен немедленно, и что он единственен. В программировании такие функции называются чистыми и тотальными.

Соглашения

Я постараюсь сохранять единую нотацию на протяжении всей книги. Она основана на стиле, преобладающем в nLab. В частности, я решил использовать строчные буквы, такие как a или b, для именования объектов в категории, а буквы в верхнем регистре, такие как S, будут использоваться для множеств (даже несмотря на то, что множества являются объектами в категории множеств и функций). Категории, в которых мы будем абстрагироваться от их специфических особенностей (скажем, структурных), будут иметь имена такие, как $\mathcal{C}$ или $\mathcal{D}$, тогда как конкретные категории будут обозначаться, например, как $\mathbf{Set}$ или $\mathbf{Cat}$.

Примеры программирования будут иллюстрироваться на Haskell. Хотя данная книга и не является руководством по Haskell, введение языковых конструкций будет происходить достаточно постепенно, чтобы помочь читателю уверенно ориентироваться в коде. Тот факт, что синтаксис Haskell основан на математической нотации, является дополнительным преимуществом. Фрагменты программы будут записываться в следующем формате:

```
apply        :: (a -> b , a) -> b
apply (f , x) = f x
```

Глава 1

Основы

Программирование начинается с типов и функций. Вероятно, у читателя, не очень глубоко погруженному в данную область, имеются некоторые предубеждения насчет типов и функций: избавьтесь от них! Они препятствуют критическому осмыслению этих понятий, прежде всего, с теоретических позиций.

Не стоит задумываться (по крайней мере, пока) о том, как вычислительные процессы реализованы на аппаратном уровне. То, что представляют собой работающие компьютеры, — это лишь одна из многих моделей вычислений. Мы не должны к этому привязываться. Вы можете выполнять вычисления в уме или с ручкой и бумагой. Физический субстрат не имеет отношения к идее программирования.

1.1 Типы и функции

Перефразируя Лао-цзы (Lao Tzu[1]): *тип, который можно описать, не является вечным типом*. Другими словами, тип — это примитивное понятие, он не может быть определен.

Тип можно было бы также назвать *объектом* или *высказыванием* — словами, которые используются для его описания в разных областях математики (в теории типов, теории категорий и в логике, соответственно).

[1]Современное написание Lao Tzu — Laozi, но здесь будет использоваться традиционное. Лао-цзы был полулегендарным автором Дао Дэ Цзин (или Даодэцзин), классического текста о даосизме.

Возможно существование более одного типа, поэтому необходимо договориться о способе их обозначения. Так что, будем оперировать типами a, b, c; или `Int`, `Bool`, `Double` и т.п. (это просто имена).

Тип, сам по себе, не имеет значения. Что делает его особенным, так это то, как он соединяется с другими типами. Соединения будем изображать стрелками. Стрелка имеет один тип в качестве источника и один тип в качестве цели. Цель может быть такой же, как и источник, и в этом случае стрелка имеет замкнутый вид.

Стрелка между типами называется *функцией*. Стрелка между объектами называется *морфизмом*. Стрелка между высказываниями называется *следованием* или *импликацией*. Это всего лишь термины, которые используются для описания стрелок в разных областях математики, так что их можно использовать как синонимы.

В логике, высказывание — это то, что может быть истинным или ложным, а стрелка между двумя объектами a и b интерпретируется так, как если бы a влекло за собой b, или b было выводимо из a.

Между двумя типами может быть более одной стрелки, поэтому их также нужно именовать. Например, ситуация, когда стрелка f направлена от типа a к типу b, выглядит так:

$$a \xrightarrow{f} b$$

Возможны разные способы интерпретации такой записи. Например, можно сказать, что функция f принимает аргумент типа a и возвращает результат типа b, или, f является доказательством того, что если a истинно, то b также истинно.

Замечание. Связь между теорией типов, лямбда-исчислением (которое является основой функционального программирования), логикой и теорией категорий известна как соответствие Карри-Ховарда-Ламбека (Curry-Howard-Lambek).

1.2 Инь и Янь

Объект характеризуется своими связями. Стрелка есть доказательство, или свидетельство, того факта, что два объекта связаны. Иногда, когда доказательство отсутствует, объекты не связаны; иногда (различных) доказательств много и, соответственно, столько же стрелок; а иногда

имеется *единственное* доказательство — одна стрелка между двумя объектами.

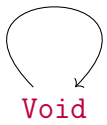
Использование характеристики «единственное» означает, что если можно найти два таких, то они должны быть равны.

Объект, у которого имеется единственная исходящая стрелка к любому объекту, называется *инициальным объектом*.

Двойственное понятие — объект, который имеет единственную входящую стрелку от любого объекта, называется *терминальным объектом*.

В математике инициальный объект часто обозначается 0, а терминальный объект — 1.

Инициальный объект является источником всего. Как тип, он известен в Haskell под обозначением `Void`, а в математике — как 0, и символизирует «хаос, из которого все возникает». Так как имеется стрелка от `Void` к каждому объекту, то имеется и стрелка от `Void` к самому себе.



Таким образом, `Void` порождает `Void` и все остальное.

Терминальный объект объединяет все. Как тип, он обозначается `Unit`, в математике — 1, и символизирует высший порядок.

В логике терминальный объект означает абсолютную истину, обозначаемый символом T, или $\top$. Тот факт, что к нему ведет стрелка от любого объекта, означает, что $\top$ истинно независимо от каких-либо предположений.

Двойственно, инициальный объект означает логическую ложь, противоречие или контр-фактуальное высказывание. Он записывается как `False` и обозначается символом F, или $\bot$. То, что от него идет стрелка к любому объекту, означает, что можно доказать все, что угодно, начиная с ложных посылок.

В английском языке есть специальная грамматическая конструкция для контр-фактуальных импликаций. Когда говорят: «Если желания были бы лошадьми, нищие ездили бы на них» (If wishes were horses, beggars would ride), то имеется в виду, что равенство между желаниями и лошадьми подразумевает, что нищие могут ездить верхом. Но мы

знаем, что это предположение ложно.

Язык программирования позволяет нам общаться друг с другом и с компьютерами. Одни языки предназначены для компьютера, другие ближе к теории. Мы будем использовать Haskell в качестве компромисса.

В Haskell инициальный объект соответствует типу `Void`. Обозначение для терминального типа есть `()`, или `Unit`. Эти обозначения имеют определенный смысл, который будет раскрыт позже.

В Haskell может быть бесконечно много типов (с учетом определямых новых), и к каждому из них ведет единственная стрелка/функция от `Void`. Все эти функции известны под общим названием: `absurd`.

Программирование	Теория категорий	Логика
тип	объект	высказывание
функция	морфизм (стрелка)	импликация
`Void`	инициальный объект, 0	ложь, $\bot$
`()`	терминальный объект, 1	истина, $\top$

1.3 Элементы

Объект не имеет определяемых составных частей, но может иметь структуру. Структура определяется стрелками, указывающими на объект, так что исследование объекта заключается в исследовании этих стрелок.

В программировании и в логике обычно требуется, чтобы инициальный объект не имел структуры. Будем предполагать, что у такого объекта нет входящих стрелок (кроме той, которая замкнута на него). По этой причине `Void` не имеет структуры.

Терминальный объект имеет простейшую структуру. От любого объекта к нему имеется только одна входящая стрелка, т.е. есть только один способ исследовать его по любому направлению. В этом отношении терминальный объект ведет себя как обособленная точка. Его единственное свойство состоит в том, что он существует, и стрелка от любого другого объекта к нему подтверждает это (является доказательством этого).

Поскольку терминальный объект настолько прост, то его можно использовать для исследования других, более сложных объектов.

Если от терминального объекта к некоторому объекту a имеется более одной стрелки, это означает, что a имеет некоторую структуру: его можно интерпретировать более чем одним способом. Поскольку терми-

нальный объект ведет себя как точка, можно рассматривать каждую стрелку от него как выбор другой точки или элемента своей цели.

В теории категорий говорят, что x является *глобальным элементом* для a, если он является стрелкой

$$1 \xrightarrow{x} a$$

Мы часто будем называть его просто элементом (опуская «глобальный»).

В теории типов, $x : A$ означает, что x является типом A.

Для этого в Haskell используется нотация с двойным двоеточием:

```
x :: A
```

(Haskell использует имена с заглавными буквами для конкретных типов, а имена со строчными буквами — для переменных того или иного типа.)

Будем считать, что `x` — терм типа `A`, но, категорно, будем интерпретировать его как стрелку $x : 1 \to A$, глобальный элемент `A`.[2]

В логике, такой x называется доказательством A, так как он соответствует импликации $\top \to A$ (если **True** истинно, то **A** также истинно). Отметим, что может быть много разных доказательств A.

Поскольку мы определили, что стрелок от любого другого объекта к `Void` нет, то нет и стрелок от терминального объекта к нему. Поэтому `Void` не имеет элементов. Вот почему мы воспринимаем `Void` как пустой тип.

Терминальный объект имеет только один элемент, так как от него к самому себе идет единственная стрелка, $1 \to 1$; поэтому иногда его называют синглетоном.

Замечание. В теории категорий нет ограничений на инициальный объект, к которому идут стрелки от других объектов. Однако в декартово замкнутых категориях, которые мы будем изучать, этого не допускается.

1.4 Объект стрелок

Стрелки между любыми двумя объектами образуют множество[3]. Вот почему знание основ теории множеств является необходимым условием

[2] Система типов Haskell различает `x :: A` и `x :: () -> A`. Однако, в категорной семантике эти выражения обозначают одно и то же.

[3] Строго говоря, это верно только в *локально малой* категории.

для изучения теории категорий.

В программировании мы говорим о *типах* функций от `a` к `b`. На Haskell, запись

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

означает, что `f` имеет тип «функция от `a` к `b`». Здесь, `a -> b` — это просто обозначение, которое мы приписываем этому типу.

Если мы хотим, чтобы функциональные типы обрабатывались так же, как и другие типы, нам нужен объект, который будет представлять собой набор стрелок от `a` к `b`.

Чтобы полностью определить этот объект, необходимо описать его отношение к другим объектам, в частности, к `a`, и к `b`. Пока у нас нет соответствующих инструментов, но мы к этому придем.

А пока, зафиксируем следующее различие: с одной стороны, имеется стрелка, которая соединяет два объекта, `a` и `b`. Эти стрелки образуют множество. С другой стороны, имеется *объект стрелок* от `a` к `b`. «Элемент» этого объекта определяется как стрелка от терминального объекта `()` к объекту, обозначенному как `a -> b`.

Нотация, которая используется в программировании, имеет тенденцию стирать это различие. Вот почему в теории категорий объект стрелок называется *экспоненциалом* и записывается как b^a (исходный объект находится в показателе). Итак, запись

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

эвивалентна

$$1 \xrightarrow{f} b^a .$$

В логике, стрелка $A \to B$ является импликацией: она констатирует тот факт, что «если A, то B». Экспоненциальный объект B^A является соответствующим высказыванием. Оно может быть истинным, но может быть и ложным; заранее это неизвестно и нуждается в доказательстве. Такое доказательство является элементом B^A.

Предъявите элемент из B^A, и тогда можно будет сказать, что B следует из A.

Рассмотрим снова высказывание: «Если желания были бы лошадьми, нищие ездили бы на них» — на этот раз, как объект. Это не пустой объект, потому что можно предъявить его доказательство — что-то вроде:

«Человек, у которого есть лошадь, передвигается на ней. У нищих есть желания. Поскольку желания — это лошади, то у нищих есть лошади. Поэтому ездят и нищие». Но, даже если имеется доказательство этого высказывания, оно бесполезно, потому что не удастся доказать его посылку: «желание = лошадь».

Глава 2

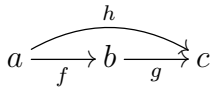
Композиция

2.1 Композиция

Программирование — это о композиции. Перефразируя Витгенштейна (Wittgenstein), можно было бы сказать: «О том, что нельзя разложить, не следует говорить». Это не запрет, это констатация факта. Процессы изучения, понимания и описания базируются на процессе декомпозиции; и язык объектов и стрелок отражает это.

Словарь объектов и стрелок построен именно для того, чтобы выразить идею композиции.

Имея стрелку f от a к b и стрелку g от b к c, их композиция представляет собой стрелку, идущую непосредственно от a к c. Другими словами, если имеются две стрелки, цель одной из которых совпадает с источником другой, то всегда можно скомпоновать их, чтобы получить третью стрелку.



В математике композицию обычно обозначают символом $\circ$. Тогда приведенное изображение можно записать в виде

$$h = g \circ f\,,$$

что можно прочесть как: «h равно g после f». Порядок композиции может показаться непривычным, но это потому, что зачастую полагают, что функции принимают аргументы справа. В Haskell в качестве символа композиции используется точка:

```
h = g . f
```

Чтобы выполнить функцию `h`, она, если не является непосредственно выполнимой, должна быть декомпозирована на более простые части, `f` и `g`. Они, в свою очередь, могут быть подвергнуты дальнейшей декомпозиции и так далее. Это относится к любой программе.

Теперь предположим, что удалось разложить g на $j \circ k$. Тогда

$$h = (j \circ k) \circ f$$

Мы хотим, чтобы это разложение было таким же, как и

$$h = j \circ (k \circ f)$$

При этом должна существовать возможность заявить, что h была декомпозирована на три более простые функции

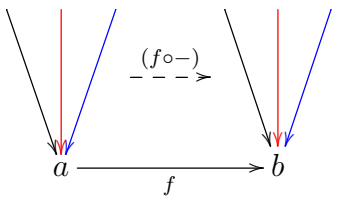
$$h = j \circ k \circ f$$

и не нужно отслеживать, какая декомпозиция была первой. Это свойство называется *ассоциативностью* композиции, и мы будем его придерживаться.

Композиция является источником двух отображений стрелок, называемых пред-композицией и пост-композицией.

Когда стрелка f является пост-компонуемой со стрелкой h, получается стрелка $f \circ h$. Конечно, f можно пост-компоновать только со стрелками, целью которых является источник для f. Пост-композиция посредством f записывается как $(f \circ -)$, оставляя пустóту для h. Как сказал бы Лао-цзы: «Полезность пост-композиции исходит из того, чего нет».

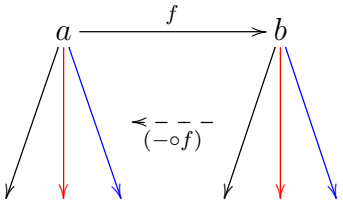
Таким образом, стрелка $f : a \to b$ индуцирует отображение стрелок $(f \circ -)$, которое отображает стрелки, идущие к a, в стрелки, идущие к b.



Поскольку объекты не имеют внутренней структуры, то когда мы говорим, что f преобразует a в b, то имеется в виду именно это.

Пост-композиция позволяет смещать фокус с одного объекта на другой.

Двойственным образом, можно пред-композировать f, или применить $(- \circ f)$ к стрелкам, исходящим из b, сопоставляя их со стрелками, исходящими из a.



Пред-композиция позволяет смещать перспективу от одного наблюдателя к другому. Обратите внимание, что исходящие стрелки отображаются в направлении, противоположному стрелке f.

Пред- и пост-композиция — это отображение стрелок. Поскольку стрелки образуют множества, это функции между множествами.

Другой способ взглянуть на пред- и пост-композицию состоит в том, что они являются результатом частичного применения оператора композиции с двумя пустотами $(- \circ -)$, в котором мы предварительно замещаем одну пусто́ту, или другую, стрелкой.

В программировании, исходящая стрелка интерпретируется как извлечение данных из источника. Входящая стрелка интерпретируется как создание цели. Исходящие стрелки определяют интерфейс, входящие стрелки определяют конструкторы.

Выполните следующие упражнения, чтобы убедить себя в том, что смещение фокуса и перспективы можно комбинировать.

Упражнение 2.1.1. *Предположим, имеются стрелки, $f : a \to b$ и $g : b \to c$. Их композиция $g \circ f$ индуцирует отображение стрелок $((g \circ f) \circ -)$. Покажите, что результат будет таким же, если сначала применить $(f \circ -)$, а затем $(g \circ -)$. Формально*:

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

Подсказка: *выберите произвольный объект x и стрелку $h : x \to a$, и посмотрите, получится ли один и тот же результат. Обратите*

внимание, что $\circ$ здесь перегружен. Это означает регулярную функциональную композицию, когда она помещается между двумя посткомпозициями.

Упражнение 2.1.2. *Проверьте, что композиция из предыдущего упражнения ассоциативна. Подсказка*: *начните с трех композируемых стрелок.*

Упражнение 2.1.3. *Покажите, что пред-композиция $(- \circ f)$ компонуема, но порядок композиции является обратным*:

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

2.2 Применение функции

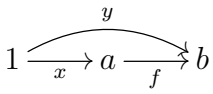
Все готово для того, чтобы написать первую программу. Есть поговорка: «Путь в тысячу ли начинается с одного шага». Наше путешествие — от 1 к b. Первый шаг — это стрелка от терминального объекта 1 к (типу) a. Это какой-то элемент типа a. Можно записать это как

$$1 \xrightarrow{x} a$$

Оставшийся путь — это стрелка

$$a \xrightarrow{f} b$$

Эти две стрелки являются компонуемыми (они соприкасаются на объекте a), и их композиция — это стрелка y от 1 к b. Другими словами, y является *элементом* из b



Можно записать это так:

$$y = f \circ x$$

Мы используем f для отображения элемента из a к *элементу* из b. Поскольку такое действие встречается довольно часто, оно имеет специальное название — *применение* функции f к x; будем использовать для этого сокращенное обозначение

$$y = fx$$

Теперь переведем это на Haskell. Начнем с элемента x из a (сокращение для `() -> a`)

```
x :: a
```

Объявим функцию f как элемент «объекта стрелок» от a к b

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

с пониманием (о чем будет сказано ниже), что оно соответствует стрелке от `a` к `b`. Результатом является элемент из b

```
y :: b
```

определяемый как

```
y = f x
```

Это называется применением функции к аргументу, но мы смогли выразить это исключительно в терминах композиции функции (примечание: в других языках программирования применение функции требует использования круглых скобок, например, `y = f(x)`).

2.3 Тождественность

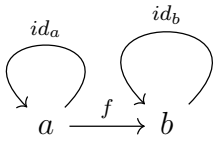
Можно думать о стрелках как о представляющих изменения: объект a становится объектом b. Стрелка, зацикленная на объекте, представляет собой изменение самого объекта. Но изменение имеет свою двойственность: отсутствие изменений или бездействие.

Каждый объект имеет специальную стрелку, называемую *тождественной* (или *тождественностью*), которая оставляет объект неизменным. Это означает, что когда вы компонуете эту стрелку с любой другой стрелкой, входящей или исходящей, то получаете эту же стрелку. Тождественная стрелка ничего не делает с другой стрелкой.

Тождественная стрелка на объекте a обозначается как id_a. Так что, если имеется стрелка $f : a \to b$, то можно скомпоновать ее с тождественностями с обеих сторон

$$\mathsf{id}_b \circ f = f = f \circ \mathsf{id}_a$$

или, схематично:



Просто проверить, что тождественность делает с элементами. Возьмем элемент $x : 1 \to a$ и скомпонуем его с id_a. Результат:

$$\mathsf{id}_a \circ x = x$$

который означает, что тождественность оставляет элементы неизменными.

В Haskell используется одно и то же имя `id` для всех тождественных функций (оно не индексируется типом, с которым работает). Приведенное выше уравнение, определяющее действие id на элементы, переводится непосредственно в:

```
id x = x
```

и становится определением функции `id`.

Мы уже видели, что, и инициальный объект, и терминальный объект, имеют единственные стрелки, возвращающиеся к ним. Теперь мы говорим, что каждый объект имеет тождественную стрелку, замкнутую на нем. Помните, что говорилось выше об единственности: если можно найти две таких стрелки, то они должны совпадать. Можно заключить, что эти единственные замкнутые стрелки, о которых шла речь, должны быть тождественными стрелками. Теперь можно пометить следующие диаграммы:

В логике тождественная стрелка понимается как тавтология. Доказательство того, что «если a истинно, то a истинно», тривиально. Его также называют *правилом тождественности*.

Если тождественность ничего не делает, то почему надо беспокоится о ней? Представим, что вы отправляетесь в путешествие, компонуете несколько стрелок и снова оказываетесь в исходной точке. Вопрос

в следующем: вы что-нибудь сделали или зря потратили время? Единственный способ ответить на этот вопрос — сравнить свой путь с тождественной стрелкой.

Некоторые поездки туда и обратно вызывают изменения, другие — нет.

Упражнение 2.3.1. *Что делает* $(\mathsf{id}_a \circ -)$ *со стрелками, оканчивающимися на* a*? Что делает* $(- \circ \mathsf{id}_a)$ *со стрелками, исходящими из* a*?*

Глава 3

Изоморфизмы

Когда утверждается, что:

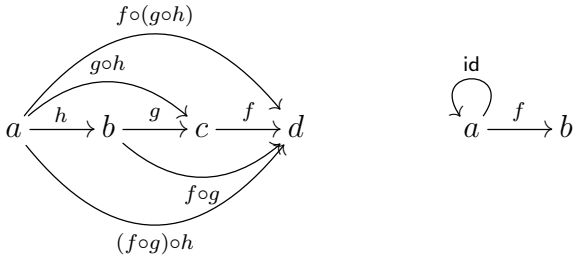
$$f \circ (g \circ h) = (f \circ g) \circ h$$

или:

$$f = f \circ \mathsf{id}$$

то провозглашается *равенство* стрелок. Стрелка слева является результатом одной операции, а стрелка справа — результат другой. Но результаты *равны*.

Такие равенства часто иллюстрируются изображениями коммутативных диаграмм, например,



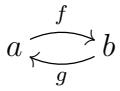
Подобным образом сравниваются стрелки.

Объекты же не сравниваются на совпадение, поскольку объекты воспринимаются как места состыковки стрелок, поэтому, если необходимо сравнить два объекта, надо анализировать стрелки.

3.1 Изоморфные объекты

Простейшее отношение между двумя объектами — это стрелка.

Самый простой круговой маршрут — это композиция из двух стрелок, идущих в противоположных направлениях.



Здесь имеются два возможных круговых маршрута. Одним из них является $g \circ f$, который идет от a к a. Другой — это $f \circ g$, идущий от b к b.

Если оба приводят к тождеству, то говорят, что стрелка g является обратной к стрелке f

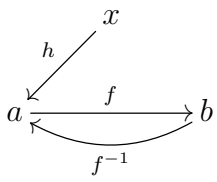
$$g \circ f = \mathsf{id}_A$$
$$f \circ g = \mathsf{id}_B$$

и записывают это как $g = f^{-1}$ (произносится, как *обратная* к f). Стрелка f^{-1}, другими словами, отменяет результат действия стрелки f.

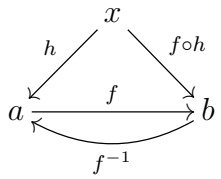
Такая пара стрелок называется *изоморфизмом*, а два объекта, которые так связаны, называются *изоморфными*.

О чем говорит существование изоморфизма, соединяющего два объекта?

Уже было акцентировано, что объекты характеризуются их взаимодействием с другими объектами. Так что, рассмотрим, как выглядят два изоморфных объекта с точки зрения объекта-наблюдателя x. Выберем стрелку h, идущую от x к a.



Существует соответствующая стрелка, идущая от x к b. Это просто композиция $f \circ h$, или действие $(f \circ -)$ на h.



Точно так же, для любой стрелки, идущей к b, существует соответствующая стрелка, идущая к a. Это задается действием $(f^{-1} \circ -)$.

Мы можем перемещать фокус назад и вперед между a и b, используя сопоставления $(f \circ -)$ и $(f^{-1} \circ -)$.

Можно комбинировать эти два отображения (см. упражнение 2.1.1), чтобы сформировать круговой маршрут. Результат такой же, как если бы мы применили композицию $((f^{-1} \circ f) \circ -)$. Но это равно $(\mathsf{id}_A \circ -)$, что, как следует из упражнения 2.3.1, оставляет стрелки без изменений.

Точно так же, круговой маршрут, вызванный $f \circ f^{-1}$, оставляет стрелки $x \to b$ неизменными.

Это создает «взаимную поддержку» между двумя группами стрелок. Представьте себе, что каждая стрелка отправляет сообщение своему напарнику, как это определено с помощью f или f^{-1}. Тогда каждая стрелка получит ровно одно сообщение, и это будет сообщение от ее напарника. Ни одна стрелка не останется обделенной вниманием, и ни одна стрелка не получит более одного сообщения. Математики называют такую систему напарников *биекцией* или взаимно-однозначным соответствием.

Следовательно, стрелка за стрелкой, два объекта a и b выглядят совершенно одинаково с точки зрения x. По-стрелочно, между этими двумя объектами нет никакой разницы.

В частности, если заменить x терминальным объектом 1, то будет ясно, что эти два объекта имеют одинаковые элементы. Каждому элементу $x : 1 \to a$ соответствует элемент $y : 1 \to b$, а именно, $y = f \circ x$, и наоборот. Существует биекция между элементами изоморфных объектов.

Такие неразличимые объекты называются *изоморфными*, потому что они имеют «одинаковую форму». «Увидеть» один означает «увидеть» их все.

Запишем этот изоморфизм как:

$$a \cong b$$

Имея дело с объектами, используют изоморфизм вместо равенства.

В программировании, два изоморфных типа имеют одинаковое внешнее поведение. Один тип может быть реализован через другой, и наоборот. Одно можно заменить другим без изменения поведения системы (за исключением, возможно, производительности).

В классической логике, если B следует из A, а A следует из B, то A и B логически эквивалентны. Часто говорят, что B истинно тогда и только тогда, когда истинно A. Однако, в отличие от предыдущих параллелей между логикой и теорией типов, здесь не все так просто, если считать доказательства релевантными. На самом деле, это привело к развитию нового раздела фундаментальной математики, гомотопической теории типов, или HoTT.

Упражнение 3.1.1. *Приведите аргумент, что существует биекция между стрелками, исходящими из двух изоморфных объектов. Изобразите соответствующие схемы.*

Упражнение 3.1.2. *Докажите, что каждый объект изоморфен самому себе.*

Упражнение 3.1.3. *Если имеются два терминальных объекта, покажите, что они изоморфны.*

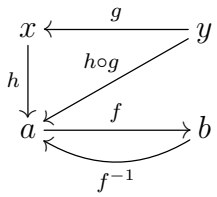
Упражнение 3.1.4. *Покажите, что изоморфизм из предыдущего упражнения единственен.*

3.2 Естественность

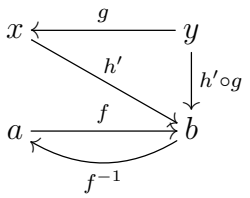
Как мы уже знаем, когда два объекта изоморфны, то можно переключить фокус с одного на другой с помощью пост-композиции: либо $(f \circ -)$, либо $(f^{-1} \circ -)$.

И наоборот, для переключения между разными наблюдателями будет использоваться пред-композиция.

В самом деле, стрелка h ведущая к a от x, связана со стрелкой $h \circ g$, идущей к тому же объекту от y.



Точно так же, стрелка h', ведущая к b от x, соответствует стрелке $h' \circ g$, направленной туда же от y.



В обоих случаях мы меняем перспективу с x на y, применяя пред-композицию $(- \circ g)$.

Важным наблюдением является то, что изменение точки зрения сохраняет взаимную поддержку, установленную изоморфизмом. Если две стрелки были напарниками с точки зрения x, они по-прежнему остаются напарниками и с точки зрения y. Это так же просто, как сказать, что не имеет значения, делаете ли вы сначала пред-композицию с помощью g (переключение точки зрения), а затем пост-композицию с помощью f (переключение фокуса), или сначала пост-композицию с помощью f, а затем пред-композицию с g. Символически это записывается так:

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

и мы называем это условием *естественности*.

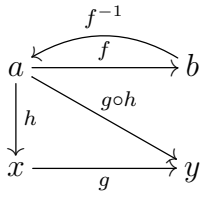
Смысл этого равенства раскрывается, если применить его к морфизму $h : x \to a$. Обе стороны оцениваются как $f \circ h \circ g$.

$$y \xrightarrow{g} x \xrightarrow{h} a \xrightarrow{f} b$$

Здесь, условие естественности — выполняется автоматически из-за ассоциативности, но вскоре мы увидим его обобщение на менее тривиальные обстоятельства.

Стрелки используются для передачи информации об изоморфизме. Естественность отражает то, что все объекты получают единообразное представление об этом изоморфизме, независимо от пути.

Можно также поменять местами роли наблюдателей и субъектов. Например, используя стрелку $h : a \to x$, объект a может исследовать произвольный объект x. Если имеется стрелка $g : x \to y$, она может переключить фокус на y. Переключение точки зрения на b осуществляется пред-композицией с помощью f^{-1}.



Снова имеем условие естественности, на этот раз с точки зрения изоморфной пары:

$$(- \circ f^{-1}) \circ (g \circ -) = (g \circ -) \circ (- \circ f^{-1})$$

Такая ситуация, когда нужно сделать два шага, чтобы переместиться из одного места в другое, типична для теории категорий. Здесь, операции пред-композиции и пост-композиции можно выполнять в любом порядке — мы говорим, что они компонуемы. Но в целом, порядок, согласно которому делаются шаги, приводит к разным результатам. Мы часто накладываем условия компоновки и говорим, что одна операция совместима с другой, если эти условия выполняются.

Упражнение 3.2.1. *Покажите, что обе части условия естественности для* f^{-1}, *при воздействии на* h, сводятся к:

$$b \xrightarrow{f^{-1}} a \xrightarrow{h} x \xrightarrow{g} y$$

3.3 Рассуждения на языке стрелок

Мастер Йонеда говорит: «Смотри на стрелки!»

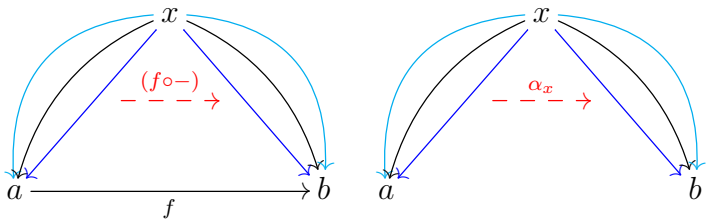
Если два объекта изоморфны, они имеют одно и то же множество входящих стрелок.

Если два объекта изоморфны, они, к тому же, имеют одно и то же множество исходящих стрелок.

Если вы хотите понять, изоморфны ли два объекта, осмотрите их стрелки!

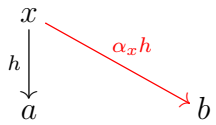
Если два объекта a и b изоморфны, то любой изоморфизм f индуцирует взаимно однозначное отображение $(f \circ -)$ между соответствующими множествами стрелок.

Предположим, что неизвестно, изоморфны ли объекты, но существует обратимое отображение, α_x, между множествами входящих стрелок к a и к b от каждого объекта x. Другими словами, для каждого x, α_x является биекцией стрелок? Ответ — «да», если семейство α_x удовлетворяет условию естественности.



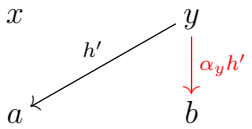
Биекция стрелок слева порождается изоморфизмом f. Справа, биекция стрелок задается посредством α_x. Означает ли это, что эти два объекта справа изоморфны? Можно ли построить изоморфизм f по семейству отображений α_x?

Вот демонстрация действия α_x на конкретную стрелку h:



Это отображение вместе с его обратным α_x^{-1}, которое переводит стрелки $x \to b$ в стрелки $x \to a$, будет играть роль $(f \circ -)$ и $(f^{-1} \circ -)$, если действительно существовал изоморфизм f. Семейство отображений α описывает «искусственный» способ переключения фокуса между двумя нашими объектами.

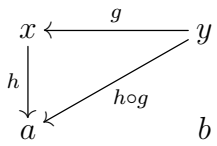
Та же ситуация, с точки зрения другого наблюдателя y:



Заметим, что y использует отображение α_y для переключения фокуса с a на b.

Эти два отображения, α_x и α_y, действуют всякий раз, когда существует морфизм $g : y \to x$. В этом случае пред-композиция с g позволяет

переключить перспективу с x на y (обратите внимание на направление)



Мы отделили переключение фокуса от переключения перспективы. Первое выполняется с помощью α, второе — с помощью пред-композиции. Естественность налагает условие совместимости между этими двумя действиями.

Действительно, начиная с некоторого h, можно применить $(-\circ g)$ для переключения на точку зрения y, а затем применить α_y для переключения фокуса на b:

$$\alpha_y \circ (- \circ y)$$

или можно сначала предоставить x переключить фокус на b с помощью α_x, а затем переключить перспективу с помощью $(- \circ g)$:

$$(- \circ g) \circ \alpha_x$$

В обоих случаях обозреваетсяt b из y. Мы уже производили такое действие ранее, когда рассматривался изоморфизм между a и b, и было обнаружено, что результаты одинаковы. Мы назвали это условием естественности.

Если необходимо, чтобы эти α были источником изоморфизма, мы должны наложить эквивалентное условие естественности:

$$\alpha_y \circ (- \circ g) = (- \circ g) \circ \alpha_x$$

Мы хотим, чтобы все α были совместимы с пред-композицией. Или, при воздействии на некоторую стрелку $h : x \to a$:

$$\alpha_y(h \circ g) = (\alpha_x h) \circ g$$

Таким образом, если заменить все α на $(f \circ -)$, все будет работать.

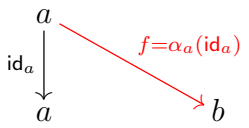
И действительно, это так! И называется трюком Йонеды. Это есть способ восстановить f из всех α. Вот как это работает:

Так как α_x определен для любого объекта x, он также определен и для a. По определению, α_a переводит морфизм $a \to a$ в морфизм $a \to b$.

Известно, что существует, по крайней мере, один морфизм $a \to a$, а именно, тождественность id_a. Оказывается, что искомый изоморфизм f определяется выражением:

$$f = \alpha_a(\mathsf{id}_a)$$

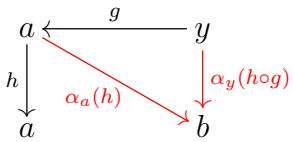
или, схематично:



Проверим это. Если f — изоморфизм, то α_x должен быть равен $(f \circ -)$ для любого x. Чтобы убедиться в этом, перепишем условие естественности, заменив x на a. Тогда получаем:

$$\alpha_y(h \circ g) = (\alpha_a h) \circ g$$

что иллюстрирует диаграмма:



Поскольку и источником, и целью h, является a, это равенство должно выполняться и для $h = \mathsf{id}_a$

$$\alpha_y(\mathsf{id}_a \circ g) = (\alpha_a(\mathsf{id}_a)) \circ g$$

Но $\mathsf{id}_a \circ g$ равно g, а $\alpha_a(\mathsf{id}_a)$ — это f, поэтому, получаем:

$$\alpha_y g = f \circ g = (f \circ -)g$$

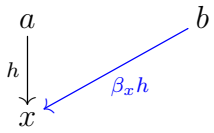
Другими словами, $\alpha_y = (f \circ -)$, для каждого объекта y и любого морфизма $g : y \to a$.

Заметим, что, несмотря на то, что α_x был определен для каждого x и каждой стрелки $x \to a$ в отдельности, оказалось, что он полностью определяется своим значением на одной единичной стрелке. Это — проявление мощи естественности!

Обращение стрелок

Как сказал бы Лао-цзы, двойственность между наблюдателем и наблюдаемым не может быть полной, если наблюдателю не будет позволено поменяться ролями с наблюдаемым.

Опять же, мы хотим показать, что два объекта a и b изоморфны, но на этот раз мы хотим обращаться с ними как с наблюдателями. Стрелка $h : a \to x$ «исследует» произвольный объект x с точки зрения a. Ранее, когда мы узнали, что два объекта изоморфны, мы смогли переключить точку зрения на b, используя $(- \circ f^{-1})$. На этот раз, в нашем распоряжении имеется преобразование β_x. Оно устанавливает биекцию между стрелками, приходящими к x.



Если мы хотим наблюдать за другим объектом, y, мы будем использовать β_y для переключения точки зрения между a и b, и так далее.

Если два объекта x и y соединены стрелкой $g : x \to y$, то имеется возможность переключения фокуса с помощью $(g \circ -)$. Если мы хотим переключить, и точку зрения, и фокус, есть два способа сделать это. Естественность требует, чтобы результаты были равны:

$$(g \circ -) \circ \beta_x = \beta_y \circ (g \circ -)$$

В самом деле, если заменить β на $(- \circ f^{-1})$, то восстановится условие естественности для изоморфизма.

Упражнение 3.3.1. *Используйте трюк с тождественным морфизмом, чтобы восстановить f^{-1} из семейства отображений β.*

Упражнение 3.3.2. *Используя f^{-1} из предыдущего упражнения, оцените $\beta_y g$ для произвольного объекта y и произвольной стрелки $g : a \to y$.*

Как сказал бы Лао-цзы: «Чтобы предъявить изоморфизм, часто проще определить естественное преобразование между десятью тысячами стрелок, чем найти пару стрелок между двумя объектами».

Глава 4

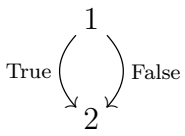
Тип-сумма

4.1 Bool

Мы научились компоновать стрелки. А как компоновать объекты?

Выше мы определили 0 (инициальный объект) и 1 (терминальный объект). Но, что такое 2, если не 1 плюс 1?

2 — это объект с двумя элементами: двумя стрелками, исходящими из 1. Назовем одну стрелку `True`, а другую — `False`. Не путайте эти имена с логическими интерпретациями инициальных и терминальных объектов. Эти два элемента — *стрелки*.

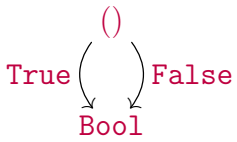


Эта простая идея может быть немедленно выражена на Haskell[1] как определение типа, традиционно называемого `Bool`, в честь автора, Джорджа Буля (George Boole).

```
data Bool where
  True  :: () -> Bool
  False :: () -> Bool
```

[1]Этот стиль определения называется обобщенными алгебраическими типами данных, или GADT, в Haskell.

Это соответствует такой же диаграмме (только с некоторыми переименованиями, согласно Haskell):



Как было представлено выше, для элементов существует сокращенная запись, так что, более компактная версия имеет вид:

```
data Bool where
  True  :: Bool
  False :: Bool
```

Теперь можно определить терм типа `Bool`, например

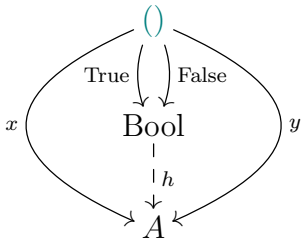
```
x :: Bool
x  = True
```

Первая строка объявляет `x` элементом `Bool` (скрытая функция `() -> Bool`), а вторая строка содержит присвоение `x` одного из двух возможных значений.

Функции `True` и `False`, которые использованы в определении `Bool`, называются *конструкторами данных*. Их можно использовать для создания конкретных термов, как в примере выше. Кстати, в Haskell имена функций начинаются со строчных букв, за исключением случаев, когда они являются конструкторами данных.

Наше определение типа `Bool` еще не завершено. Понятно, как построить терм `Bool`, но мы не знаем, что с ним делать. Должна существовать возможность определять стрелки, выходящие из `Bool` — *отображения от* `Bool`.

Первое наблюдение состоит в том, что если имеется стрелка `h` от `Bool` к некоторому конкретному типу `A`, то мы автоматически получаем две стрелки `x` и `y` от единицы к `A`, просто композицией. Следующие два (искаженных) треугольника являются коммутативными:



Другими словами, каждая функция `Bool -> A` производит пару элементов типа `A`.

Для конкретного типа `A`:

```
h :: Bool -> A
```

имеем

```
x = h True
y = h False
```

где

```
x :: A
y :: A
```

Обратите внимание на использование сокращенной записи для применения функции к элементу:

```
h True      -- означает: h . True
```

Теперь мы готовы завершить наше определение `Bool`, добавив условие, что любая функция от `Bool` к `A` не только производит, но и *эквивалентна* паре элементов из `A`. Другими словами, пара элементов однозначно определяет функцию от `Bool`.

Это означает, что мы можем интерпретировать приведенную выше диаграмму двумя способами. Для заданного `h`, можно легко получить `x` и `y`. Но верно и обратное: пара элементов `x` и `y` однозначно *определяет* `h`.

Здесь работает биекция. На этот раз это однозначное сопоставление между парой элементов (x, y) и стрелкой h.

На Haskell, это определение `h` инкапсулировано в конструкции `if ... then ... else ...` . Для

```
x :: A
y :: A
```

мы определяем *отображение-вне*

```
h  :: Bool -> A
h b = if b then x else y
```

Здесь, `b` является термом типа `Bool`.

Как правило, тип данных создается с использованием правил *введения* и деконструируется с использованием правил *исключения*. Тип данных `Bool` имеет два правила введения: одно использует `True`, а другое — `False`. Конструкция `if ... then ... else ...` определяет правило исключения.

То, что для выше определенного `h`, можно получить два терма, которые были использованы для его определения, называется правилом *вычисления*. Оно определяет, как вычислить результат `h`. Если мы вызовем `h` с `True`, результатом будет `x`; если вызов производится с `False`, результатом будет `y`.

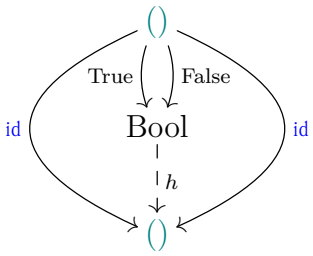
Мы никогда не должны упускать из виду цель программирования: разложить сложные проблемы на ряд более простых. Определение `Bool` иллюстрирует эту идею. Всякий раз, когда нам нужно построить отображение-вне `Bool`, мы разбиваем его на две меньшие задачи по созданию пары элементов целевого типа. Итак, мы разложили одну ситуацию на две, более простые.

Примеры

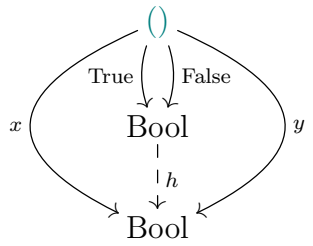
Приведем несколько примеров. Мы еще не ввели достаточного количества типов, поэтому ограничимся отображениями-вне `Bool` к `Void`, `()` и `Bool`. К тому же, такие "пограничные" случаи могут дать новое понимание уже известных результатов.

Мы поняли, что не может быть никаких функций (кроме тождественной) с `Void` в качестве цели, поэтому, очевидно, не существует каких-то бы ни было функций от `Bool` к `Void`. И действительно, в `Void` элементы отсутствуют.

А как насчет функций от `Bool` к `()`? Так как `()` — терминальный тип, к нему может вести только одна функция от `Bool`. И действительно, эта функция соответствует единственной возможной паре функций от `()` к `()`, обе — тождественные.



Интересным случаем являются функции от `Bool` к `Bool`. Поместим `Bool` вместо `A`:



Сколько пар (x, y) функций от `()` к `Bool` имеется в нашем распоряжении? Исходных функций в `Bool` всего две, `True` и `False`, поэтому можно образовать четыре пары. Это $(True, True)$, $(False, False)$, $(True, False)$ и $(False, True)$. Так что, существуют только четыре функции от `Bool` к `Bool`.

Можно записать их на Haskell, используя конструкцию `if ... then ... else ...`. Например, последняя из четырех функций, которую назовем `not`, определяется как:

```
not  :: Bool -> Bool
not b = if b then False else True
```

Также можно рассматривать функции от `Bool` к `A` как элементы объекта стрелок или экспоненциальный объект A^2, где 2 — это объект `Bool`. Согласно нашим подсчетам, имеется: ноль элементов в 0^2, один элемент в 1^2 и четыре элемента в 2^2. Это именно то, чего мы ожидаем от элементарной алгебры, где числа, на самом деле, означают числа.

Упражнение 4.1.1. *Составьте реализации трех других функций* `Bool -> Bool`.

4.2 Перечисления

Ну а что дальше, после 0, 1 и 2? — Объект с тремя конструкторами данных. Например:

```
data RGB where
  Red   :: RGB
  Green :: RGB
  Blue  :: RGB
```

Если вас утомляет избыточный синтаксис, есть сокращение для подобного типа определения:

```
data RGB = Red | Green | Blue
```

Это правило введения позволяет строить термы типа RGB, например:

```
c :: RGB
c  = Blue
```

Чтобы определить отображения от RGB, необходим более общий шаблон исключения. Точно так же, как функция от Bool определялась двумя элементами, функция от RGB к A определяется тройкой элементов типа A: x, y и z. Запишем такую функцию, используя синтаксис шаблона *сопоставления с образцом*:

```
h       :: RGB -> A
h Red   = x
h Green = y
h Blue  = z
```

Это всего лишь одна функция, определение которой разбито на три случая.

Такой же синтаксис можно использовать и для Bool, вместо if ... then ... else ...:

```
h       :: Bool -> A
h True  = x
h False = y
```

На самом деле, имеется и третий способ записать то же самое, используя case:

```
h c = case c of
          Red   -> x
          Green -> y
          Blue  -> z
```

или даже

```
h  :: Bool -> A
h b = case b of
          True  -> x
          False -> y
```

Вы можете использовать любой из них по своему усмотрению.

Эти шаблоны также будут работать для типов с четырьмя, пятью и более конструкторами данных. Например, десятичная цифра является одной из:

```
data Digit = Zero | One | Two | Three | ... | Nine
```

Существует огромное перечисление символов Unicode под названием Char. Их конструкторам даются специальные имена: вы записываете само имя в апострофах, например,

```
c :: Char
c  = 'a'
```

Как сказал бы Лао-Цзы, на создание шаблона из десяти тысяч элементов ушло бы много лет, поэтому люди придумали шаблон подстановочного знака, символ подчеркивания, который соответствует всему.

Поскольку шаблоны сопоставляются по порядку «сверху-вниз», шаблон подстановочного знака следует располагать последним:

```
yesno  :: Char -> Bool
yesno c = case c of
             'y' -> True
             'Y' -> True
              _  -> False
```

Но почему мы должны на этом останавливаться? Тип `Int` можно рассматривать как перечисление целых чисел в диапазоне от -2^{29} до 2^{29} (или больше, в зависимости от реализации). Конечно, об исчерпывающем сопоставлении с образцом на таких диапазонах не может быть и речи, но принцип сохраняется.

На практике, в язык встроены типы `Char` для символов Unicode, `Int` для целых чисел фиксированной точности, `Double` для чисел с плавающей запятой двойной точности и некоторые другие.

Это не бесконечные типы. Их элементы можно перечислить, даже если на это уйдут тысячелетия. Однако, тип `Integer` бесконечен.

Отступление от синтаксиса Haskell

Так как мы собираемся записывать большое количество кода на Haskell, нужно установить некоторые предварительные условия. Чтобы определять типы данных с помощью функций, нужно использовать языковую прагму, называемую `GADTs` (Generalized Algebraic Data Types — обобщенные алгебраические типы данных). Прагма должна быть помещена в начало исходного файла. Например:

```
{- # language GADTs # -}

data Bool where
  True  :: () -> Bool
  False :: () -> Bool
```

Тип данных `Void` может быть определен как:

```
data Void where
```

с пустым предложением `where` (без конструктора данных!).

Функция `absurd` работает с любым типом в качестве цели (это *полиморфная* функция), поэтому она параметризуется *переменной типа*. В отличие от конкретных типов переменные типа должны начинаться со строчной буквы. Вот подобная переменная типа:

```
absurd  :: Void -> a
absurd v = undefined
```

Здесь используется `undefined`, чтобы формально удовлетворить требования компилятора. В этом случае мы абсолютно уверены, что функция `absurd` никогда не может быть вызвана, потому что невозможно сконструировать аргумент типа `Void`.

Вы можете использовать `undefined`, когда вызывает интерес только компиляция, а не запуск вашего кода. Например, вам может понадобиться подключить функцию `f` для проверки, работают ли ваши определения совместно:

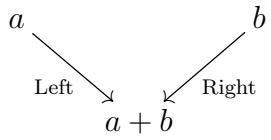
```
f :: a -> x
f  = undefined
```

Если возникает необходимость поэкспериментировать с определением своих собственных версий стандартных типов, таких как `Either`, нужно сообщить компилятору, чтобы он скрывал оригиналы, определенные в стандартной библиотеке, которая называется PRELUDE. Для этого случая, поместите следующую строку в начало файла после языковых прагм:

```
import Prelude hiding (Either, Left, Right)
```

4.3 Типы сумм

Тип `Bool` можно рассматривать как сумму $2 = 1 + 1$. Но ничто не мешает заменить 1 другим типом или даже заменить каждую из единиц разными типами, скажем, a и b. Можно определить новый тип $a + b$ с помощью двух стрелок. Назовем их `Left` и `Right`. Следующая определяющая диаграмма является правилом введения:

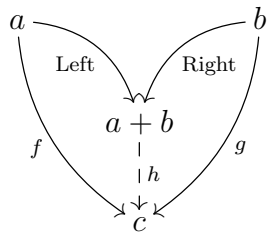


На Haskell тип $a+b$ называется `Either a b`. По аналогии с `Bool`, можно определить его как

```
data Either a b where
  Left  :: a -> Either a b
  Right :: b -> Either a b
```

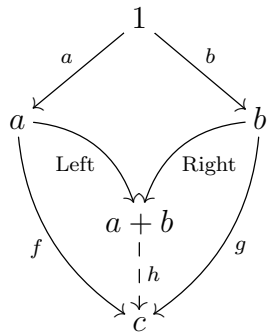
(Обратите внимание на использование строчных букв для переменных типа.)

Точно так же, отображение-вне от $a + b$ к некоторому типу c определяется следующей коммутативной диаграммой:



Имея функцию h и компонуя ее с `Left` and `Right`, получаем пару функций f и g. Обратно, такая пара функций однозначно определяет h. Это правило исключения.

Если требуется перевести эту диаграмму на Haskell, то надо выбрать элементы двух типов. Это можно сделать, определив стрелки a и b от терминального объекта.



Следуя по стрелкам на этой диаграмме, получаем:

$$h \circ \text{Left} \circ a = f \circ a$$
$$h \circ \text{Right} \circ b = g \circ b$$

Синтаксис Haskell позволяет почти буквально повторить эти уравнения, что приводит к следующему выражению сопоставления с образцом для определения `h`:

```
h           :: Either a b -> c
h (Left a)  = f a
h (Right b) = g b
```

(Опять же, обратите внимание на использование строчных букв для переменных типа и тех же букв для термов этого типа. В отличие от людей, компиляторов это не смущает.)

Если просмотреть эти уравнения справа налево, то обнаружатся правила вычисления для тип-сумм. Две функции, которые использовались для определения `h`, могут быть восстановлены путем применения `h` к `(Left a)` и `(Right b)`.

Также можно использовать синтаксис `case` для определения `h`:

```
h e = case e of
  Left  a -> f a
  Right b -> g b
```

Итак, в чем сущность типа данных? Это всего лишь рецепт манипулирования стрелками.

Maybe

Очень полезный тип данных, `Maybe`, определяется как алгебраическая сумма $1 + a$, для любого a. Его определение на Haskell:

```
data Maybe a where
  Nothing :: () -> Maybe a
  Just    :: a  -> Maybe a
```

Конструктор данных `Nothing` представляет собой стрелку от единичного типа, а `Just` конструирует `Maybe a` из `a`. `Maybe a` изоморфно `Either () a`. Его также можно определить с помощью сокращенной нотации

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

`Maybe`, в основном, используется для кодирования возвращаемого типа частичных функций: тех, которые не определены для некоторых значений их аргументов. В этом случае, вместо аварийного завершения, такие функции возвращают `Nothing`. В других языках программирования частичные функции часто реализуются с помощью механизма исключений.

Логика

В логике, высказывание $A + B$ называется альтернативой или *логическим или*. Это можно доказать, предоставив доказательство A или доказательство B, любого из них будет достаточно.

Если требуется доказать, что C следует из $A + B$, то надо быть готовым к двум возможностям при доказательстве $A + B$: либо доказать A (а B может быть и ложным), либо доказать B (A может быть ложным). В первом случае надо будет показать, что C следует из A. Во втором случае надо доказать, что C следует из B. Это в точности стрелки в правиле исключения для $A + B$.

4.4 Ко-декартовы категории

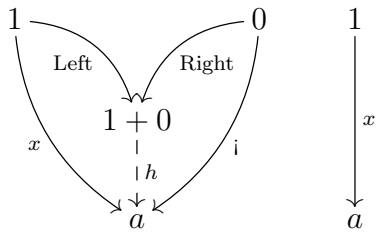
На Haskell можно определить сумму любых двух типов, используя `Either`. Категория, в которой существуют все суммы и существует инициальный объект, называется *ко-декартовой*, а сумма называется *ко-произведением*. Можно заметить, что типы суммы имитируют сложение чисел. Получается, что инициальный объект играет роль нуля.

Один плюс ноль

Сначала покажем, что $1 + 0 \cong 1$, то есть сумма терминального объекта и инициального объекта, изоморфна терминальному объекту. Стандартной процедурой для такого рода доказательств является использование трюка Йонеды. Поскольку типы сумм определяются путем отображения-вне, то мы должны сравнивать исходящие стрелки для обеих сторон.

Аргумент Йонеды допускает, что два объекта изоморфны, если существует биекция β_a между множествами стрелок, исходящих от них к произвольному объекту a, и эта биекция естественна.

Посмотрим на определение суммы $1 + 0$ и на ее отображение-вне к произвольному объекту a. Это отображение определяется парой $(x, ¡)$, где x — элемент из a, а ¡ — единственная стрелка от инициального объекта к a (функция `absurd` в Haskell).



Мы хотим установить взаимно-однозначное отображение между стрелками, исходящими из $1 + 0$, и стрелками, исходящими из 1. Поскольку h определяется парой $(x, ¡)$, можно просто сопоставить ее со стрелкой x, исходящей из 1. А поскольку ¡, единственная, упомянутое отображение является биекцией.

Мы определяем β_a для отображения любого h, определенного а парой $(x, ¡)$, к x. И наоборот, β_a^{-1} отображает x к паре $(x, ¡)$. Но является ли это естественным преобразованием?

Чтобы ответить на этот вопрос, нужно рассмотреть, что происходит, когда мы меняем фокус от a к некоторому b, которое связано с ним стрелкой $g : a \to b$. Теперь имеем два варианта.

- Заставим h переключить фокус, скомпоновав x и ¡ с помощью g. Получим новую пару $(y = g \circ x, ¡)$ и проследуем за ней через β_b.

- Используем β_a для отображения $(x, ¡)$ к x и проследуем за ним с пост-композицией $(g \circ -)$.

В обоих случаях мы получаем одну и ту же стрелку $y = g \circ x$. Так что, преобразование β является естественным. Следовательно, $1+0$ изоморфно 1.

На Haskell можно определить две функции, являющиеся изоморфными, но нет способа напрямую выразить тот факт, что они являются обратными друг другу.

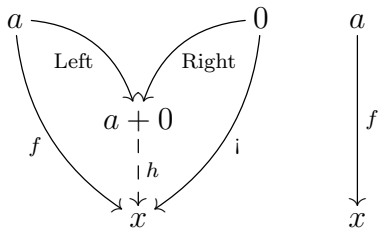
```
f          :: Either () Void -> ()
f (Left ()) = ()
f (Right _) = ()

f_1        :: () -> Either () Void
f_1 _       = Left ()
```

Подстановочный знак подчеркивания в определении функции означает, что аргумент игнорируется. Второе определение `f` избыточно, так как нет термов типа `Void`.

Что-то плюс ноль

Очень похожее рассуждение можно использовать, чтобы показать, что $a + 0 \cong a$. Следующая диаграмма поясняет это.



Можно перевести этот аргумент на Haskell, реализуя (полиморфную) функцию `h`, которая работает для любого типа `a`.

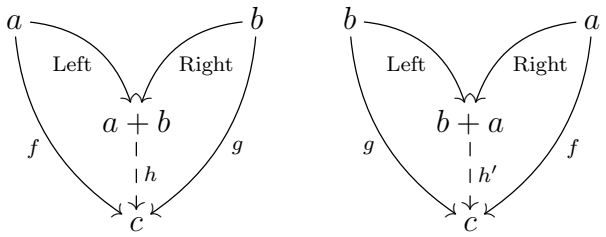
Упражнение 4.4.1. *Реализуйте на Haskell две функции, образующие изоморфизм между* `Either a Void` *и* `a`.

Мы могли бы использовать аналогичный аргумент, чтобы показать, что $0 + a \cong a$, но имеется более общее свойство типов суммы, которое позволяет избежать этого.

Коммутативность

В диаграммах, определяющих тип суммы, имеется хорошая лево-правая симметрия, которая предполагает, что он удовлетворяет правилу коммутативности, $a + b \cong b + a$.

Рассмотрим отображения с обеих сторон. Очевидно, что для каждой h, определяемой парой (f, g) слева, существует соответствующая h', задаваемая парой (g, f) справа. Это устанавливает биекцию стрелок.



Упражнение 4.4.2. *Покажите, что определенная выше биекция естественна. Подсказка: и f, и g, меняют фокус пост-композицией с $k : x \to y$.*

Упражнение 4.4.3. *Реализуйте на Haskell функцию, свидетельствующую об изоморфизме между* `Either a b` *и* `Either b a`. *Обратите внимание, что эта функция является обратной самой себе.*

Ассоциативность

Как и в арифметике, сумма, которую мы определили, ассоциативна:

$$(a + b) + c \cong a + (b + c)$$

Легко написать отображение-вне для левой стороны:

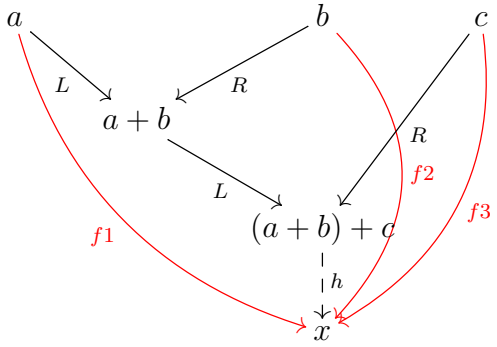
```
h                  :: Either a (Either b c) -> x
h (Left (Left a))   = f1 a
h (Left (Right b))  = f2 b
h (Right c)         = f3 c
```

Обратите внимание на использование вложенных шаблонов, таких как `(Left (Left a))`, и т.д. Отображение полностью определяется тройкой функций. Эти же функции можно использовать для определения отображения-вне правой стороны:

```
h'                   :: Either (Either a b) c -> x
h' (Left a)           = f1 a
h' (Right (Left b))   = f2 b
h' (Right (Right c)) = f3 c
```

Это устанавливает взаимно-однозначное отображение между тройками функций, которые определяют два отображения. Это отображение естественно, потому что все изменения фокуса выполняются с помощью пост-композиции. Следовательно, две стороны изоморфны.

Этот код также может быть отображен в виде диаграммы. Вот схема левой части указанного изоморфизма:



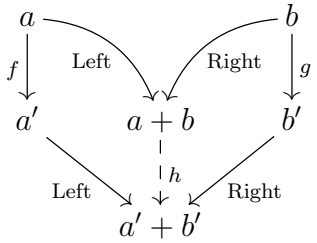
Функториальность

Поскольку сумма определяется свойством отображения-вне, легко понять, что происходит, когда мы меняем фокус: он меняется «естественным образом» вместе с фокусами стрелок, определяющих произведение. Но что происходит, если перемещать источники этих стрелок?

Предположим, что имеются стрелки, которые отображают a и b к некоторым a' и b':

$$f : a \to a'$$
$$g : b \to b'$$

Композицию этих стрелок с конструкторами Left и Right, соответственно, можно использовать для определения отображения между суммами:



Пара стрелок, $(\mathrm{Left} \circ f, \mathrm{Right} \circ g)$ однозначно определяет стрелку $h : a + b \to a' + b'$.

Это свойство суммы называется *функториальностью*. Можно представлять себе, что это позволяет преобразовывать два объекта *внутри* суммы и получить новую сумму. Мы также говорим, что функториальность позволяет нам *поднять* пару стрелок, чтобы оперировать суммами.

Упражнение 4.4.4. *Покажите, что функториальность сохраняет композицию. Подсказка: возьмите две компонуемые стрелки, $g : b \to b'$ и $g' : b' \to b''$, и покажите, что применение $g' \circ g$ дает тот же результат, что и первое применение g для преобразования $a + b$ в $a + b'$, а затем применяя g' для преобразования $a + b'$ к $a + b''$.*

Упражнение 4.4.5. *Покажите, что функториальность сохраняет тождественность. Подсказка*: *используйте* id_b *и покажите, что она отображается к* id_{a+b}.

Симметричная моноидальная категория

Когда ребенок учится складывать, мы называем это арифметикой. Когда взрослый учится сложению, мы называем это ко-декартовой категорией.

Складываем ли мы числа, компонуем стрелки или конструируем суммы объектов, мы повторно используем одну и ту же идею декомпозиции составных вещей на их более простые компоненты.

Как сказал бы Лао-цзы, когда вещи собираются вместе, чтобы сформировать новую вещь, и такая операция ассоциативна и имеет нейтральный элемент, мы знаем, как обращаться с десятками тысяч вещей.

Тип-сумма, который мы определили, удовлетворяет следующим свойствам:

$$a + 0 \cong a$$
$$a + b \cong b + a$$
$$(a + b) + c \cong a + (b + c)$$

и она функториальна. Категория с таким типом операции называется *симметричной моноидальной*. Когда операция представляет собой сумму (ко-произведение), она называется *ко-декартовой*. В следующей главе мы увидим еще одну моноидальную структуру, называемую *декартовой*.

Глава 5

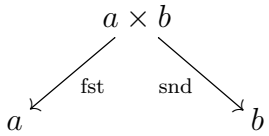
Тип-произведение

Мы можем использовать тип-сумму для перечисления возможных значений данного типа, но кодирование может быть расточительным. Нам понадобилось десять конструкторов только для кодирования чисел от нуля до девяти.

```
data Digit = Zero | One | Two | Three | ... | Nine
```

Но если мы объединим две цифры в единую структуру данных, двузначное десятичное число, мы сможем закодировать сотню чисел. Или, как сказал бы Лао-Цзы, всего четырьмя цифрами можно закодировать десять тысяч чисел.

Тип данных, объединяющий два типа подобным образом, называется произведением или *декартовым произведением*. Его определяющим качеством является правило исключения: две стрелки исходят из $a \times b$; один, с именем «fst», идет к a, а другой, с именем «snd», идет к b. Они называются *проекциями* и позволяют получить a и b из их произведения $a \times b$.

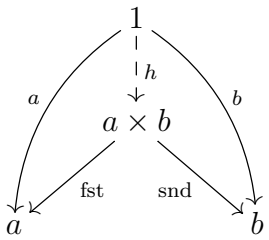


Предположим, что имеется элемент произведения, то есть стрелка h от терминального объекта 1 к $a \times b$. Тогда можно легко получить пару элементов, просто используя композицию: элемента a, заданного как

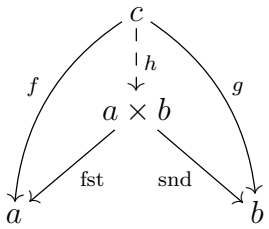
$$a = \text{fst} \circ h$$

и элемента b, заданного как

$$b = \mathrm{snd} \circ h$$



Фактически, для стрелки от произвольного объектат c к $a \times b$ можно определить композицией пару стрелок $f : c \to a$ и $g : c \to b$



Как это было сделано выше с тип-суммой, можно обратить эту идею и использовать эту диаграмму для *определения* тип-произведения: пара функций f и g должна находиться во взаимно-однозначном соответствии с *отображением-внутрь* от c к $a \times b$. Это есть правило *введения* для произведения.

В частности, отображение-вне терминального объекта используется на Haskell для определения тип-произведения. По двум элементам, `a :: A` и `b :: B`, построим произведение

```
(a , b) :: (A , B)
```

Встроенный синтаксис для произведений таков: пара скобок и запятая между ними. Он используется как для определения произведения двух типов `(A,B)`, так и для конструктора данных `(a,b)`, который соединяет вместе два элемента.

Мы никогда не должны упускать из виду цель программирования: разложить сложные проблемы на ряд более простых. Мы снова видим это в определении произведения. Всякий раз, когда нужно построить *отображение-внутрь* произведения, мы разбиваем эту задачу

на две меньшие, для построения пары функций, каждая из которых отображает-внутрь один из компонентов произведения. Это так же просто, как сказать, что для реализации функции, возвращающей пару значений, достаточно реализовать две функции, каждая из которых возвращает один из элементов пары.

Логика

В логике, тип-произведение соответствует логической конъюнкции. Чтобы доказать $A \times B$ (A и B), нужно предоставить доказательства, и A, и B. Это стрелки, направленные и к A, и к B. Правило исключения гласит, что если имеется доказательство $A \times B$, то автоматически получается доказательство A (через fst) и доказательство B (через snd).

Кортежи и записи

Как сказал бы Лао-Цзы, произведение десяти тысяч объектов — это всего лишь объект с десятью тысячами проекций.

Используя нотацию кортежа, можно формировать обычные произведения на Haskell. Например, произведение трех типов записывается как `(A,B,C)`. Терм этого типа может быть построен из трех элементов: `(a,b,c)`.

О том, что математики называют «злоупотреблением обозначениями»: произведение нуля типов записывается как `()`, в виде пустого кортежа, который записывается так же, как и терминальный объект или тип единицы. Это связано с тем, что произведение ведет себя очень похоже на умножение чисел, а терминальный объект играет роль единицы.

На Haskell, вместо определения отдельных проекций для всех кортежей, используется синтаксис сопоставления с образцом. Например, чтобы извлечь третий компонент из тройки, мы должны написать

```
thrd           :: (a , b , c) -> c
thrd (_ , _ , c) = c
```

Мы используем символ подчеркивания для компонентов, которые хотим игнорировать.

Лао-цзы говорил, что «Именование есть начало всех отдельных вещей». В программировании сложно отслеживать значение компонентов

конкретного кортежа, не присваивая им имена. Синтаксис записи позволяет давать имена проекциям. Вот определение произведения, записанного в стиле записи:

```
data Product a b = Pair { fst :: a , snd :: b }
```

где `Pair` — это конструктор данных, а `fst` и `snd` — проекции.

Вот как это определение можно использовать для объявления и инициализации конкретной пары:

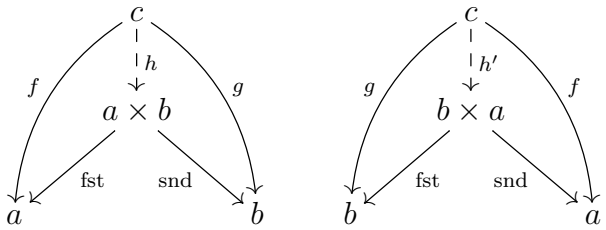
```
ic :: Product Int Char
ic  = Pair 10 'A'
```

5.1 Декартова категория

На Haskell можно определить произведение любых двух типов. Категория, в которой существуют все произведения и существует терминальный объект, называется *декартовой*.

Арифметика кортежей

Тождества, которым удовлетворяет произведение, могут быть получены с использованием свойства отображения. Например, чтобы показать, что $a \times b \cong b \times a$, рассмотрим следующие диаграммы:



Они показывают, что, для любого объекта x, стрелки к $a \times b$ находятся во взаимно однозначном соответствии со стрелками к $b \times a$. Это потому, что каждая из этих стрелок определяется одной и той же парой f и g.

Вы можете проверить, что условие естественности выполнено, поскольку, когда точка зрения сдвигается с помощью $k : x' \to x$, все

стрелки, исходящие из x, сдвигаются в соответствии с пред-композицией $(- \circ k)$.

На Haskell этот изоморфизм может быть реализован как функция, обратная самой себе:

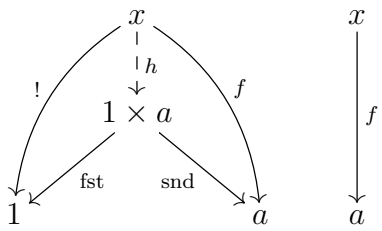
```
swap  :: (a, b) -> (b, a)
swap x = (snd x , fst x)
```

Та же функция, записанная с использованием сопоставления с образцом:

```
swap (x, y) = (y, x)
```

Важно иметь в виду, что произведение симметрично только «с точностью до изоморфизма». Это не означает, что изменение порядка в паре не изменит поведение программы. Симметричность означает, что информационное наполнение исходной пары и пары с переставленными компонентами одинаково, но доступ к ней необходимо модифицировать.

Следующую диаграмму можно использовать для доказательства того, что терминальный объект является единицей произведения, $1 \times a \cong a$.



Единственная стрелка от x к 1 обозначена как '!'. Своей уникальностью отображение h полностью определяется стрелкой f.

Обратимая стрелка, свидетельствующая об изоморфизме между $1 \times a$ и a, называется *левым объединителем* (или *унитором*, unitor):

$$\lambda : 1 \times a \to a$$

Приведем примеры изоморфизмов, записанных на Haskell (без доказательств существования обратного). Ассоциативность:

```
assoc             :: ((a, b), c) -> (a, (b, c))
assoc ((a, b), c) = (a, (b, c))
```

Правая единица:

```
runit        :: (a, ()) -> a
runit (a, _) = a
```

Приведенным двум функциям соответствует *ассоциатор*:

$$\alpha : (a \times b) \times c \to a \times (b \times c)$$

и *правый объединитель*:

$$\rho : a \times 1 \to a$$

Упражнение 5.1.1. *Покажите, что биекция в доказательстве левой единицы естественна. Подсказка: смените фокус с помощью стрелки* $g : a \to b$.

Упражнение 5.1.2. *Постройте стрелку*

$$h : b + a \times b \to (1 + a) \times b$$

Эта стрелка единственная?

Подсказка: *это отображение-внутрь произведения, поэтому оно задается парой стрелок. Эти стрелки, в свою очередь, отображают-вне сумму, каждая из которых задается, поэтому, парой стрелок.*

Подсказка: *отображение* $b \to 1 + a$ *задается как* $(\mathrm{Left} \circ !)$

Упражнение 5.1.3. *Повторите предыдущее упражнение, рассматривая, на этот раз,* h *как отображение-вне суммы.*

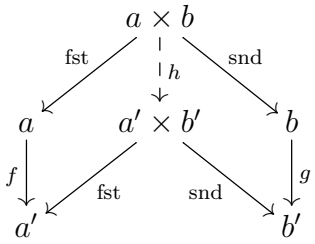
Упражнение 5.1.4. *Реализуйте Haskell-функцию* `maybeAB :: Either b (a, b) -> (Maybe a, b)`. *Является ли эта функция однозначно определенной сигнатурой своего типа или есть некоторая свобода действий*?

Функториальность

Предположим, что имеются стрелки, которые отображают a и b к некоторым a' и b':

$$f : a \to a'$$
$$g : b \to b'$$

Композицию этих стрелок с проекциями fst и snd, соответственно, можно использовать для определения отображения между произведениями:



Сокращенная нотация этой диаграммы:

$$a \times b \xrightarrow{f \times g} a' \times b'$$

Это свойство произведения называется *функториальностью*. Можно представить себе, что оно позволяет трансформировать два объекта *внутри* произведения, чтобы получить новое произведение. Мы также говорим, что функториальность позволяет *поднять* пару стрелок, чтобы оперировать произведениями.

5.2 Двойственность

Когда мы видим стрелку, то ясно, какой ее конец указывает на источник, а какой — на цель.

$$a \to b$$

Но, возможно, это всего лишь привычка. Был бы мир совсем другим, если бы мы назвали b источником, а a — целью?

Мы бы еще могли скомпоновать эту стрелку с

$$b \to c$$

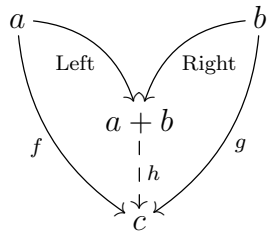
чья «цель» b такая же, как и «источник» для $a \to b$, но результат все равно будет стрелкой

$$a \to c$$

только теперь мы сказали бы, что она направлена от c к a.

В этом двойственном мире, объект, который мы называли «инициальным», будет называться «терминальным», потому что он является «мишенью» уникальных стрелок, исходящих от всех объектов. И наоборот, терминальный объект будет называться инициальным.

Теперь рассмотрим диаграмму, которая была использована для определения объекта суммы:



В новой интерпретации, стрелка h будет направлена «от» произвольного объекта c «к» объекту, который обозначен $a + b$. Эта стрелка однозначно определяется парой стрелок (f, g), источником которых является c. Если переименовать Left на fst, а Right на snd, то получим определяющую диаграмму для произведения.

Произведение — это сумма с обратными стрелками.

И наоборот, сумма — это произведение с обратными стрелками.

Каждая конструкция в теории категорий имеет свою двойственную.

Если направление стрелок — это просто вопрос интерпретации, то что же делает тип-сумм в программировании такими отличными от тип-произведений? Разница восходит к одному допущению, которое мы сделали в начале: инициальный объект не имеет входящих стрелок (кроме тождественной стрелки). Это контрастирует с терминальным объектом, имеющим множество исходящих стрелок, стрелок, которые были использованы для определения (глобальных) элементов. На самом деле, мы предполагаем, что каждый интересующий нас объект содержит элементы, а те, у которых их нет, являются изоморфными к `Void`.

Мы увидим еще более глубокую разницу, когда будем рассматривать функциональные типы.

5.3 Моноидальная категория

Выше мы установили истинность следующих простых правил для произведения:

$$1 \times a \cong a$$
$$a \times b \cong b \times a$$
$$(a \times b) \times c \cong a \times (b \times c)$$

и рассмотрели функториальность.

Категория, в которой определена операция с этими свойствами, называется *симметричной моноидальной*[1]. Мы встречались с подобной структурой выше, при рассмотрении сумм и инициального объекта.

Категория может иметь несколько моноидальных структур одновременно. Если вы не хотите приводить в качестве примера свою моноидальную структуру, то можете заменить знак плюс или знак произведения на знак тензорной операции, а единичный элемент — на букву I. Тогда правила симметричной моноидальной категории можно записать так:

$$I \otimes a \cong a$$
$$a \otimes b \cong b \otimes a$$
$$(a \otimes b) \otimes c \cong a \otimes (b \otimes c)$$

Эти изоморфизмы часто записывают в виде семейств обратимых стрелок, называемых ассоциаторами и объединителями. Если же моноидальная категория не является симметричной, то существуют, раздельно, левый и правый объединители.

$$\alpha : (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$
$$\lambda : I \otimes a \to a$$
$$\rho : a \otimes I \to a$$

Симметрия вводится посредством правила:

$$\gamma : a \otimes b \to b \otimes a$$

[1]Строго говоря, произведение двух объектов определено с точностью до изоморфизма, тогда как произведение в моноидальной категории должно быть определено однозначно. Но, моноидальную категорию можно получить, сделав выбор произведения.

Функториальность позволяет *поднять* пару стрелок,

$$f : a \to a'$$
$$g : b \to b'$$

для работы с тензорными произведениями:

$$a \otimes b \xrightarrow{f \otimes g} a' \otimes b'$$

Можно мыслить тензорное произведение как наименьшей общий знаменатель произведения и суммы. В нем по-прежнему выполняется правило введения, которое требует оба объекта, a и b; но у него нет правила исключения. Однажды созданное, тензорное произведение «забывает», как оно было создано. В отличие от декартова произведения, оно не имеет проекций. Некоторые интересные примеры тензорных произведений даже не симметричны.

Моноиды

Моноиды — это очень простые структуры, снабженные бинарной операцией и единицей. Натуральные числа со сложением и нулем образуют моноид. Так же, моноидом являются натуральные числа с умножением и единицей.

Интуиция подсказывает, что моноид позволяет объединить две сущности, чтобы получить одну. Есть также одна особенность, такая, что комбинирование, которое должно быть ассоциативным, с чем-либо еще дает то же самое. Это — единица.

Что не предполагается, так это то, что комбинирование должно быть симметричным, или, что существует обратный элемент.

Правила, определяющие моноид, напоминают правила категории. Разница в том, что в моноиде любые две сущности можно скомпоновать, тогда как в категории это обычно не так: можете скомпоновать две стрелки только в том случае, если цель одной является источником другой; за исключением случаев, когда категория содержит только один объект, в этом случае все стрелки являются компонуемыми.

Категория с одним объектом называется *моноидом*. Операция компоновки — это композиция стрелок, а единица — тождественная стрелка.

Это совершенно обоснованное определение. На практике, однако, нас часто интересуют моноиды, встроенные в более крупные категории. В

частности, в программировании, желательно иметь возможность определять моноиды внутри категории типов и функций.

В категории мы вынуждены определять операции сразу, а не рассматривать отдельные элементы. Начнем с объекта m. Бинарная операция — это функция двух аргументов. Поскольку элементами произведения являются пары элементов, можно охарактеризовать бинарную операцию как стрелку от произведения $m \times m$ к m:

$$\mu : m \times m \to m$$

Единичный элемент можно определить как стрелку от терминального объекта 1:

$$\eta : 1 \to m$$

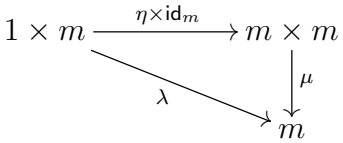
Можно перевести это описание непосредственно на Haskell, определив класс типов, снабженный двумя методами, традиционно называемыми `mappend` и `mempty`:

```
class Monoid m where
  mappend :: (m, m) -> m
  mempty  :: ()     -> m
```

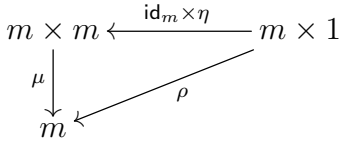
Две стрелки μ и η должны удовлетворять моноидным законам, но, опять же, их надо сформулировать в целом, без какого-либо обращения к элементам.

Чтобы сформулировать закон левой единицы, сначала образуем произведение $1 \times m$. Затем используем η , чтобы «выбрать единичный элемент в m» или, на языке стрелок, превратим 1 в m. Так как мы работаем с произведением $1 \times m$, то должны поднять пару $\langle \eta, \mathsf{id}_m \rangle$, что гарантирует, что мы «не трогаем» m. В завершение, выполним «умножение», используя μ.

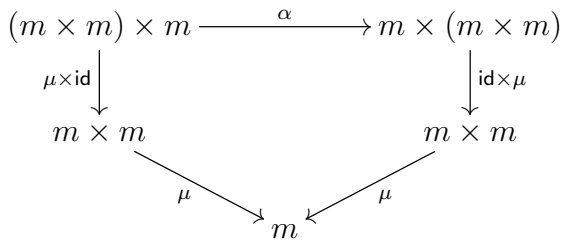
Мы хотим, чтобы результат был таким же, каков исходный элемент из m, но без упоминания элементов. Так что, просто используем левый объединитель λ чтобы перейти от $1 \times m$ к m, не «перемешивая сущностей».



Аналогичный закон для правой единицы:



Чтобы сформулировать закон ассоциативности, начнем с тройного произведения и воздействуем на него, как на единое целое. Здесь, α — это ассоциатор, который перестраивает произведение, не «перемешивая сущности».



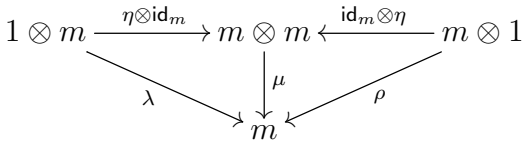
Заметим, что нам не нужно было делать много предположений о категорном произведении, которое было применено к объектам m и 1. В частности, нам не пришлось использовать проекции. Это говорит о том, что приведенное выше определение будет также хорошо работать для тензорного произведения в произвольной моноидальной категории. Оно даже не должно быть симметричным. Все, что нужно предположить, это то, что: существует единичный объект, что произведение функториально и что оно удовлетворяет законам единицы и ассоциативности с точностью до изоморфизма.

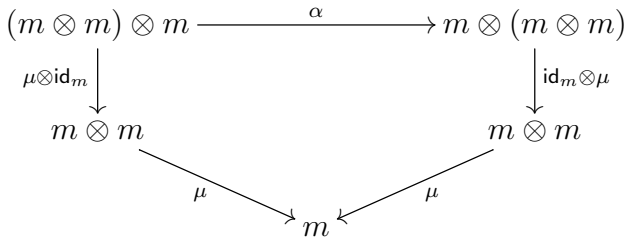
Таким образом, если заменить $\times$ на $\otimes$, а1 — на I, то получим определение моноида в произвольной моноидальной категории.

Моноид в моноидальной категории — это объект m, снабженный двумя морфизмами:

$$\mu : m \otimes m \to m$$
$$\eta : I \to m$$

удовлетворяющих законам единицы и ассоциативности:





Мы использовали функториальность $\otimes$ для поднятия пар стрелок, например, $\eta \otimes \mathsf{id}_m$, $\mu \otimes \mathsf{id}_m$, и т.д.

Глава 6

Функциональные типы

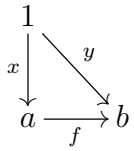
Существует еще один тип композиции, лежащий в основе функционального программирования. Это происходит, когда вы передаете функцию в качестве аргумента другой функции. Затем внешняя функция может использовать этот аргумент как подключаемую часть своего собственного механизма. Он позволяет реализовать, например, общий алгоритм сортировки, который принимает произвольную функцию сравнения.

Если функции моделируются стрелками между объектами, то что значит иметь функцию в качестве аргумента?

Чтобы определить стрелки, которые имеют «объект стрелок» в качестве источника или цели, нужен способ объективации функций. Функция, которая принимает некоторую функцию в качестве аргумента или возвращает функцию, называется функцией *высшего порядка*. Функции высшего порядка — это рабочие лошадки функционального программирования.

Правило исключения

Определяющим качеством функции является то, что ее можно применить к аргументу для получения результата. Определим применение функции в терминах композиции:



Здесь, f представлена в виде стрелки от a к b, но хотелось бы иметь возможность заменить f элементом объекта стрелок или, как это называют математики, экспоненциальным объектом b^a, или, как принято в функциональном программировании, функциональным типом `a -> b`.

Для элемента из b^a и элемента из a, применение функции должно создать элемент из b. Другими словами, для пары элементов:

$$f : 1 \to b^a$$
$$x : 1 \to a$$

должен быть получен элемент:

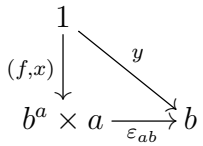
$$y : 1 \to b$$

Имейте в виду, что здесь f обозначает элемент из b^a. Раньше это была стрелка от a к b.

Известно, что пара элементов (f, x) эквивалентна элементу произведения $b^a \times a$. Таким образом, можно определить применение функции как одну стрелку:

$$\varepsilon_{ab} : b^a \times a \to b$$

Таким образом, y, как результат применения функции, определяется коммутативной диаграммой:



Применение функции — это *правило исключения* для функционального типа.

Когда предъявляется элемент функционального объекта, единственное, что можно с ним сделать, это применить его к элементу типа аргумента, используя ε.

Правило введения

Для завершения определения функционального объекта понадобится соответствующее правило введения.

Во-первых, предположим, что существует способ создания функционального объекта b^a из некоторого другого объекта c. Значит, должна существовать стрелка

$$h : c \to b^a$$

Мы знаем, что можем исключить результат h, используя ε_{ab}, но сначала нужно умножить его на a. Итак, умножим сначала c на a и воспользуемся функториальностью, чтобы отобразить его к $b^a \times a$.

Функториальность позволяет применить пару стрелок к произведению для получения другого произведения. Здесь, пара стрелок — это (h, id_a) (мы хотим превратить c в b^a, но нас не интересует изменение a)

$$c \times a \xrightarrow{h \times \mathsf{id}_a} b^a \times a$$

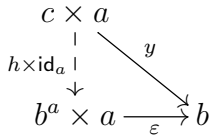
Теперь можно следовать этому с помощью применения функции, чтобы добраться до b

$$c \times a \xrightarrow{h \times \mathsf{id}_a} b^a \times a \xrightarrow{\varepsilon_{ab}} b$$

Эта составная стрелка определяет отображение, которое обозначим f:

$$f : c \times a \to b$$

Соответствующая диаграмма имеет вид



Эта коммутативная диаграмма отражает то, что по заданному h, можно построить f; но также можно потребовать и обратное: каждое отображение-вне произведения, $f : c \times a \to b$, должно однозначно определять отображение-внутрь экспоненты, $h : c \to b^a$.

Можно использовать это свойство, однозначное соответствие между двумя множествами стрелок, чтобы определить экспоненциальный объект. Это — *правило введения* для функционального объекта.

Мы видели, что произведение было определено с использованием его свойства отображения-внутрь. Применение функции, с другой стороны, определяется как *отображение-вне* от произведения.

Карринг

Имеется несколько точек зрения на это определение. Один из них — рассматривать его как пример карринга.

До сих пор мы рассматривали только функции одного аргумента. Это несущественное ограничение, так как всегда можно реализовать функцию двух аргументов как (одноаргументную) функцию от произведения. Такой функцией является `f` в определении функционального объекта:

```
f :: (c, a) -> b
```

С другой стороны, `h` — это функция, которая возвращает функцию (объект)

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

Карринг — это изоморфизм между этими двумя типами.

Этот изоморфизм может быть представлен на Haskell парой функций (высшего порядка). Поскольку на Haskell карринг работает для любых типов, эти функции записаны с использованием переменных типа — они являются *полиморфными*:

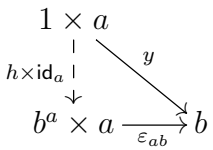
```
curry   :: ((c, a) -> b)   -> (c -> (a -> b))

uncurry :: (c -> (a -> b)) -> ((c, a) -> b)
```

Другими словами, h в определении функционального объекта можно записать как

$$h = curry\, f$$

Конечно, при таком написании, типы `curry` и `uncurry` соответствуют функциональным объектам, а не стрелкам. Это различие обычно не замечают, поскольку между *элементами* экспоненциала и *стрелками*, которые их определяют, существует взаимно однозначное соответствие. Но различие становится очевидным, если заменить произвольный объект терминальным объектом:



В этом случае, h — элемент объекта b^a, а f — стрелка от $1 \times a$ к b. Но известно, что $1 \times a$ изоморфно a, поэтому, фактически, f является стрелкой от a к b.

Поэтому, впредь, будем отождествлять стрелку `->` со стрелкой $\to$, особо не заморачиваясь по этому поводу. Правильная характеристика для такого рода феноменов состоит в том, чтобы сказать, что категория является само-обогащенной.

Будем записывать ε_{ab} как Haskell-функцию `apply`:

```
apply        :: (a -> b, a) -> b
apply (f, x) = f x
```

но это всего лишь синтаксический трюк: применение функции встроено в язык: `f x` означает, что `f` применяется к `x`. Другие языки программирования (но не Haskell) требуют, чтобы аргументы функции были заключены в круглые скобки.

Несмотря на то, что определение приложения функции, как отдельной функции, может показаться излишним, библиотека Haskell предоставляет для этой цели инфиксный оператор `$`:

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

Хитрость, однако, заключается в том, что обычное применение функции привязывается к левому краю, то есть, `f x y` совпадает с `(f x) y`; но знак доллара обеспечивает смещение вправо, так что, `f $ g x` действует как `f (g x)`. В первом случае, `f` должна быть функцией (как минимум) двух аргументов; во втором, это может быть функция одного аргумента.

На Haskell карринг используется повсеместно. Функция с двумя аргументами почти всегда записывается как функция, возвращающая функцию. Поскольку функциональная стрелка `->` привязывается вправо, нет необходимости заключать такие типы в скобки. Например, конструктор пары имеет сигнатуру:

```
pair :: a -> b -> (a, b)
```

Можно думать о ней, как о функции двух аргументов, возвращающей пару, или о функции одного аргумента, возвращающей функцию одного

аргумента, `b -> (a,b)`. Таким образом, можно частично применить такую функцию, в результате чего получится другая функция. Например, можно определить:

```
pairWithTen :: a -> (Int, a)
pairWithTen  = pair 10 -- частичное применение
                       --  пары
```

Связь с лямбда-исчислением

Другой способ взглянуть на определение функционального объекта — интерпретировать c как тип среды, в которой определена f. В этом случае его принято называть средой Γ. Стрелка интерпретируется как выражение, которое использует переменные, определенные в Γ.

Рассмотрим простой пример:

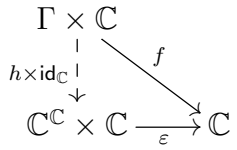
$$ax^2 + bx + c$$

Можно понимать это выражение как параметризованную тройку действительных чисел (a, b, c) с переменной x, принятой, скажем, за комплексное число. Тройка является элементом произведения $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$, которое будет средой Γ для выражения.

Переменная x является элементом из $\mathbb{C}$. Выражение представляет собой стрелку от произведения $\Gamma \times \mathbb{C}$ к типу результата (здесь также $\mathbb{C}$):

$$f : \Gamma \times \mathbb{C} \to \mathbb{C}$$

Это — отображение-вне от произведения, поэтому его можно использовать для создания функционального объекта $\mathbb{C}^{\mathbb{C}}$ и определения отображения $h : \Gamma \to \mathbb{C}^{\mathbb{C}}$



Это новое отображение h можно рассматривать как конструктор функционального объекта. Результирующий функциональный объект представляет все функции от $\mathbb{C}$ к $\mathbb{C}$, которые имеют доступ к среде Γ, то есть, к тройке параметров (a, b, c).

В соответствии с исходным выражением $ax^2 + bx + c$ имеется конкретная функция в $\mathbb{C}^{\mathbb{C}}$, которую запишем как

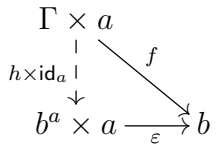
$$\lambda x.\, ax^2 + bx + c$$

или, на Haskell (с обратной косой чертой, обозначающей λ)

```
\x -> a * x^2 + b * x + c
```

Стрелка $h : \Gamma \to \mathbb{C}^{\mathbb{C}}$ однозначно определяется стрелкой f. Это отображение производит функцию, которая обозначается $\lambda x.\, f$.

Вообще, определяющей диаграммой для функционального объекта становится следующая:



Среда Γ, предоставляющая свободные параметры для выражения f, является произведением нескольких объектов, представляющих типы параметров (в нашем примере, это $\mathbb{R}\times\mathbb{R}\times\mathbb{R}$). Пустая среда представляется терминальным объектом 1, единицей произведения. В этом случае, f — это просто стрелка $a \to b$, а h просто выбирает элемент из функционального объекта b^a, который соответствует f.

Важно иметь в виду, что, в общем случае, функциональный объект представляет функции, зависящие от внешних параметров. Такие функции называются *замыканиями*. Замыкания — это функции, которые захватывают значения из своего окружения (среды).

Наш пример, переведенный на Haskell, содержит выражение, соответствующее f:

```
(a :+ 0) * x * x + (b :+ 0) * x + (c :+ 0)
```

При использовании `Double` для аппроксимации $\mathbb{R}$ наша среда является произведением `(Double, Double, Double)`.

Тип `Complex` параметризуется другим типом — здесь снова используется `Double`:

```
type C = Complex Double
```

Преобразование от `Double` к `C` выполняется установкой мнимой части в ноль, как в `(a :+ 0)`.

Соответствующая стрелка h принимает среду и производит замыкание типа `C -> C`:

```
h :: (Double, Double, Double) -> (C -> C)
h (a, b, c) = \x -
> (a :+ 0) * x * x + (b :+ 0) * x + (c :+ 0)
```

Правило вывода

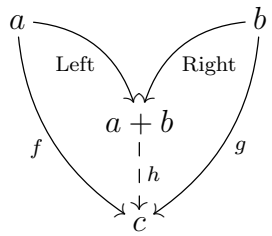
В логике, функциональный объект соответствует импликации. Стрелка от терминального объекта к функциональному объекту является доказательством этой импликации. Применение функции ε соответствует тому, что логики называют modus ponens: если имеется доказательство импликации $A \Rightarrow B$ и доказательство A, то это и составляет доказательство B.

6.1 Еще раз о сумме и произведении

Когда функции получают тот же статус, что и элементы других типов, появляются инструменты для прямого преобразования диаграмм в код.

Тип-сумма

Начнем с определения суммы.



Как было отмечено выше, пара стрелок (f, g) однозначно определяет отображение h из суммы. Это можно записать кратко, используя функцию высшего порядка:

```
h = mapOut (f, g)
```

где:

```
mapOut        :: (a -> c, b -> c)
                 -> (Either a b -> c)
mapOut (f, g) = \aorb -> case aorb of
                           Left  a -> f a
                           Right b -> g b
```

Эта функция принимает пару функций в качестве аргумента и возвращает функцию.

Прежде всего, пара `(f, g)` сопоставляется с образцом, для извлечения `f` и `g`. Далее создается новая лямбда-функцию, которая принимает аргумент `aorb` типа `Either a b`, и выполняет для него анализ случая. Если он был построен с помощью `Left`, то `f` применяется к его содержимому, иначе применяется `g`.

Обратите внимание, что функция, которая возвращается, является замыканием. Она захватывает `f` и `g` из своей среды.

Реализованная функция точно следует схеме, но написана не в обычном стиле Haskell. Программисты на Haskell предпочитают каррировать функции с несколькими аргументами. Также, по возможности, они предпочитают исключать лямбда-выражения.

Вот версия функции `either` из стандартной библиотеки Haskell:

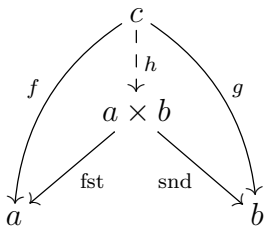
```
either               :: (a -> c) -> (b -> c)
                                 -> Either a b -> c
either f _ (Left x)  = f x
either _ g (Right y) = g y
```

Другое направление биекции, от h к паре (f, g), также следует стрелкам диаграммы.

```
unEither  :: (Either a b -> c) -> (a -> c, b -> c)
unEither h = (h . Left, h . Right)
```

Тип-произведение

Тип-произведение определяются двойственно своим свойством отображения-внутрь.



Прямое следование этой диаграмме на Haskell записывается следующим образом

```
h         :: (c -> a, c -> b) -> (c -> (a, b))
h (f, g) = \c -> (f c, g c)
```

А стилизованная версия, записанная в стиле Haskell, в виде инфиксного оператора, выглядит так: `&&&`

```
(&&&)       :: (c -> a) -> (c -> b)
                        -> (c -> (a, b))
(f &&& g) c = (f c, g c)
```

Другое направление биекции задается выражением:

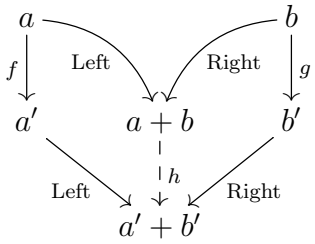
```
fork  :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

которое также следует из приведенной диаграммы.

Еще раз о функториальности

И сумма, и произведение функториальны, что означает, что можно применять функции к их содержимому; их диаграммы можно перевести в код.

Функториальность тип-суммы:



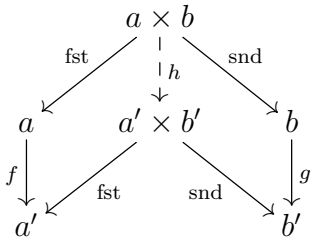
влечет код для h, используя `either`:

```
h f g = either (Left . f) (Right . g)
```

Или расширяя и обозначая как `bimap`,

```
bimap               :: (a -> a') -> (b -> b')
                                 -> Either a b
                                 -> Either a' b'
bimap f g (Left a)  = Left (f a)
bimap f g (Right b) = Right (g b)
```

Аналогично для тип-произведения:



h можно записать как

```
h f g = (f . fst) &&& (g . snd)
```

или, расширяя:

```
bimap            :: (a -> a') -> (b -> b')
                              -> (a, b) -> (a', b')
bimap f g (a, b) = (f a, g b)
```

В обоих случаях используется функция высшего порядка `bimap`, поскольку, в Haskell, и сумма, и произведение являются экземплярами более общего класса, называемого *бифунктором*, `Bifunctor`.

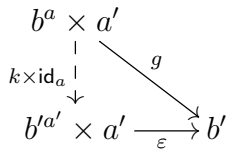
6.2 Функториальность функционального типа

Функциональный тип, или экспоненциал, также является функториальным, но с некоторой особенностью. Нас интересует отображение от b^a к $b'^{a'}$, где объекты a' и b' связаны с a и b некоторыми стрелками, которые необходимо определить.

Экспоненциал определяется своим отображением-внутрь, поэтому, если мы ищем

$$k : b^a \to b'^{a'}$$

то должны изобразить диаграмму, которая содержит k в качестве отображения-внутрь $b'^{a'}$. Эу диаграмму можно получить из исходного определения, заменяя b^a на c, а исходные объекты на штрих-объекты:



Вопрос в том, сможем ли мы найти стрелку g, чтобы завершить эту диаграмму?

$$g : b^a \times a' \to b'$$

Если мы найдем такое g, оно однозначно определит k.

Чтобы достичь этой цели, нужно поразмыслить о том, как можно было бы реализовать g. В качестве аргумента эта функция принимает произведение $b^a \times a'$. Можно мыслить о нем, как о паре, с элементом функционального объекта от a к b и элементом a'. Единственное, что можно сделать с функциональным объектом, это применить его к чему-либо. Но b^a требует аргумента типа a, а в нашем распоряжении есть только a'. Мы ничего не можем сделать, пока не укажем стрелку $a' \to a$. Эта стрелка, примененная к a', сгенерирует аргумент для b^a. Однако результат применения имеет тип b, и предполагается, что g выдает b'. Опять же, требуется стрелка $b \to b'$.

Это может показаться сложным, но суть в том, что нам нужны две стрелки между штрих-объектами и исходными. Особенность в том, что первая стрелка идет от a' к a, что оказывается обратным направлением,

по сравнению с обычным. Так что, чтобы отобразить b^a к $b'^{a'}$, требуется пара стрелок:

$$f : a' \to a$$
$$g : b \to b'$$

Это несколько проще объяснить на Haskell. Наша цель — реализовать функцию `a' -> b'`, принимая во внимание функцию `h :: a -> b`.

Эта новая функция принимает аргумент типа `a'`, поэтому, прежде чем передать его в `h`, нужно преобразовать `a'` в `a`. Вот почему нам нужна функция `f :: a' -> a`.

Так как `h` производит `b`, а надо вернуть `b'`, требуется еще одна функция `g :: b -> b'`. Все это прекрасно вписывается в одну функцию высшего порядка:

```
dimap      :: (a' -> a) -> (b -> b') -> (a -> b)
                                     -> (a' -> b')
dimap f g h = g . h . f
```

Подобно тому, как `bimap` является интерфейсом для `Bifunctor` (класса типов), `dimap` является членом класса типов `Profunctor`.

6.3 Би-декартовы замкнутые категории

Категория, в которой и произведение, и экспоненциал определены для любой пары объектов и которая имеет терминальный объект, называется *декартово замкнутой*. Если она также имеет суммы (копроизведения) и инициальный объект, то называется *би-декартово замкнутой*.

Это минимальная структура для моделирования языков программирования.

Типы данных, построенные с использованием этих операций, называются *алгебраическими типами данных*.

В нашем распоряжении имеются: сложение, умножение и возведение в степень (но не вычитание или деление) типов со всеми знакомыми законами из начальной алгебры. Они выполняются с точностью до изоморфизма. Но имеется еще один алгебраический закон, который мы пока еще не обсуждали.

Дистрибутивность

Умножение чисел распределяется над сложением. Следует ли ожидать того же в би-декартово замкнутой категории?

$$b \times a + c \times a \cong (b + c) \times a$$

Отображение слева направо легко построить, поскольку оно одновременно является отображением-вне суммы и отображением-внутрь произведения. Мы можем построить его, постепенно разделяя на более простые отображения. На Haskell это означает реализацию функции

```
dist :: Either (b, a) (c, a) -> (Either b c, a)
```

Отображение-вне суммы слева можно задать парой стрелок:

$$f : b \times a \to (b + c) \times a$$
$$g : c \times a \to (b + c) \times a$$

Это записывается на Haskell как

```
dist = either f g
  where
    f :: (b, a) -> (Either b c, a)
    g :: (c, a) -> (Either b c, a)
```

Предложение `where` используется для введения определений подфункций.

Теперь нужно реализовать f и g. Они отображаются в произведение, поэтому каждое из них эквивалентно паре стрелок. Например, первая функция задается парой:

$$f' : b \times a \to (b + c)$$
$$f'' : b \times a \to a$$

На Haskell:

```
f   = f' &&& f''
f'  :: (b, a) -> Either b c
f'' :: (b, a) -> a
```

Первую стрелку можно реализовать проекцией первого компонента b, используя затем Left для построения суммы. Вторая стрелка — это просто проекция snd:

$$f' = \text{Left} \circ \text{fst}$$
$$f'' = \text{snd}$$

Объединяя все это вместе, получаем:

```
dist = either f g
  where
    f   = f' &&& f''
    f'  = Left . fst
    f'' = snd
    g   = g' &&& g''
    g'  = Right . fst
    g'' = snd
```

Сигнатуры типов вспомогательных функций:

```
f   :: (b, a) -> (Either b c, a)
g   :: (c, a) -> (Either b c, a)
f'  :: (b, a) -> Either b c
f'' :: (b, a) -> a
g'  :: (c, a) -> Either b c
g'' :: (c, a) -> a
```

Они, также, могут быть встроены, для получения компактной формы:

```
dist = either ((Left  . fst) &&& snd)
              ((Right . fst) &&& snd)
```

Этот стиль программирования называется *бесточечным*, потому что он не использует аргументы (точки). Из соображений удобочитаемости программисты на Haskell предпочитают более явный стиль. Вышеупомянутая функция обычно реализуется так:

```
dist (Left  (b, a)) = (Left  b, a)
dist (Right (c, a)) = (Right c, a)
```

Отметим, что мы использовали только определения сумм и произведений. Другое направление изоморфизма требует использования экспоненты, поэтому оно допустимо только в бидекартово *замкнутой* категории. Это не сразу понятно из этой простой реализации на Haskell:

```
undist              :: (Either b c, a) ->
                           Either (b, a) (c, a)
undist (Left  b, a) = Left  (b, a)
undist (Right c, a) = Right (c, a)
```

но это потому, что в Haskell карринг является неявным.

Бесточечная версия этой функции имеет вид:

```
undist = uncurry (either (curry Left)
                         (curry Right))
```

Это может быть не самая удобочитаемая реализация, но она подчеркивает то, что нам нужна экспоненциальная функция: мы используем как `curry`, так и `uncurry` для реализации отображения.

Мы вернемся к этой тождественности позже, когда будем оснащены более мощными инструментами — сопряжениями.

Упражнение 6.3.1. *Покажите, что*:

$$2 \times a \cong a + a$$

где 2 *— логический тип. Сначала проведите доказательство схематически, а затем реализуйте две Haskell-функции, свидетельствующие об изоморфизме.*

Глава 7

Рекурсия

Если вы находитесь между двумя зеркалами, то видите свое отражение, отражение своего отражения, отражение того отражения и т.д. Каждое отражение определяется в терминах предыдущего отражения, но вместе они производят бесконечность.

Рекурсия — это шаблон декомпозиции, который разбивает одну задачу на множество шагов, число которых потенциально неограниченно.

Рекурсия основана на приостановке неверия. Пусть вы столкнулись с задачей, которая может потребовать сколь угодно много шагов. Вы предварительно предполагаете, что знаете, как ее решить. Затем вы задаете себе вопрос: «Как бы я сделал последний шаг, если бы у меня было решение для всего, кроме последнего шага?»

7.1 Натуральные числа

Объект натуральных чисел N не содержит чисел, поскольку для объектов их внутренняя структура скрыта. Но ее можно определить с помощью стрелок.

Мы можем использовать стрелку от терминального объекта, чтобы определить один специальный элемент. По соглашению, обозначим эту стрелку Z, для «нуля».

$$Z : 1 \to N$$

Но необходимо иметь возможность определения бесконечного количества стрелок, чтобы учесть тот факт, что для каждого натурального числа существует другое число, которое больше него на единицу.

Можно формализовать это утверждение: предположим, что известно, как образовать натуральное число $n : 1 \to N$. Как сделать следующий шаг, определяющий последующее число?

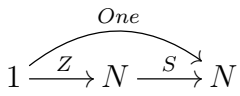
Этот следующий шаг не должен быть более сложным, чем просто пост-композиция n со стрелкой от N к N. Эта стрелка не должна быть тождественностью, потому что необходимо, чтобы преемник числа отличался от этого числа. Но одной такой стрелки, которую обозначим как S, будет достаточно.

Элемент, соответствующий преемнику n, задается композицией:

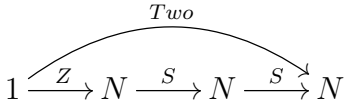
$$1 \xrightarrow{n} N \xrightarrow{S} N$$

(мы иногда изображаем один и тот же объект несколько раз на одной диаграмме, если хотим выпрямить зацикленные стрелки).

В частности, можно определить One в качестве преемника для Z:



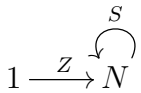
Two в качестве преемника преемника для Z;



и т.д.

Правила введения

Две стрелки, Z и S, служат в качестве правил введения для объекта натуральных чисел N. Особенность в том, что одна из них является рекурсивной: S использует N, и как источник, и как цель.



Два правила введения переводятся на Haskell непосредственно:

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

Их можно использовать для определения произвольных натуральных чисел; например:

```
zero, one, two :: Nat
zero            = Z
one             = S zero
two             = S one
```

Это определение типа натурального числа не очень полезно на практике. Однако, его часто используют для определения натуральных чисел на уровне типов, где каждое число является собственным типом.

Эта конструкция известна под названием *арифметики Пеано*.

Правила исключения

Тот факт, что правила введения являются рекурсивными, немного усложняет ситуацию, когда дело доходит до определения правил исключения. Мы будем следовать схеме из предыдущих глав, сначала предполагая, что дано отображение-вне из N:

$$h : N \to a$$

и посмотрим, что можно из него вывести.

Как выло показано выше, такое h можно разложить на более простые отображения (пары отображений для суммы и произведения, отображение-вне произведения для экспоненциала).

Правила введения для N похожи на правила для суммы, поэтому можно ожидать, что h разбивается на две стрелки. И действительно, можно легко получить первую композицией $h \circ Z$. Это стрелка, которая выбирает элемент из a. Обозначим ее $init$:

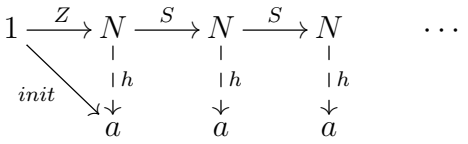
$$init : 1 \to a$$

Но нет очевидного способа найти вторую стрелку.

Чтобы увидеть это, расширим определение N:

$$1 \xrightarrow{Z} N \xrightarrow{S} N \xrightarrow{S} N \qquad \dots$$

подключим h и единицу:



Интуиция подсказывает, что стрелка от N к a представляет собой *последовательность* a_n элементов из a. Нулевой элемент задается посредством $a_0 = init$. Следующий элемент есть

$$a_1 = h \circ S \circ Z$$

за которым следует

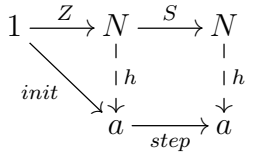
$$a_2 = h \circ S \circ S \circ Z$$

и т.д.

Таким образом, мы заменили одну стрелку h бесконечным числом стрелок a_n. Конечно, новые стрелки проще, так как представляют собой элементы из a, но их бесконечно много.

Проблема в том, что независимо от того, как это смотрится, произвольное отображение-вне N содержит бесконечно много информации.

Необходимо кардинально упростить задачу. Поскольку мы использовали одну стрелку S для генерации всех натуральных чисел, можно попробовать использовать одну стрелку $a \to a$ для генерации всех элементов a_n. Обозначим эту стрелку *step*:



Отображения-вне N, порожденные парами $init$ и $step$, называются *рекурсивными*. Не все отображения-вне N рекурсивны. Фактически, очень немногие; но рекурсивных отображений достаточно, чтобы определить объект натуральных чисел.

Мы используем приведенную выше диаграмму в качестве правила исключения. Положим, что каждое рекурсивное отображение h вне N находится во взаимно однозначном соответствии с парой $init$ и $step$.

Это означает, что *правило вычисления* (извлечение $(init, step)$ для заданной h) нельзя сформулировать для произвольной стрелки $h : N \to a$, а только для тех стрелок, которые были ранее рекурсивно определены с помощью пары $(init, step)$.

Стрелку $init$ всегда можно восстановить с помощью композиции $h \circ Z$. Стрелка $step$ является решением уравнения:

$$step \circ h = h \circ S$$

Если h было определено с помощью некоторых $init$ и $step$, то это уравнение, очевидно, имеет решение.

Важным является то, что требуется, чтобы это решение было *единственным*.

Интуитивно, пара $init$ и $step$ генерирует последовательность элементов $a_0, a_1, a_2, \ldots$ Если две стрелки h и h' заданы одной и той же парой $(init, step)$, это означает, что последовательности, которые они генерируют, одинаковы.

Таким образом, если бы h как-то отличалось от h', это означало бы, что N содержит не только последовательность элементов Z, SZ, $S(SZ)$, ... Например, если добавить -1 к N (то есть, сделать Z чьим-то преемником), то могут быть h и h', отличающиеся на -1, но при этом генерируемые одними и теми же $init$ и $step$. Единственность означает, что перед, после или между числами, сгенерированными Z и S, нет натуральных чисел.

Правило исключения, которое здесь обсуждалось, соответствует *примитивной рекурсии*. Мы увидим более продвинутую версию этого правила, соответствующую принципу индукции, в главе о зависимых типах.

Программирование

Правило исключения может быть реализовано как рекурсивная функция в Haskell:

```
rec          :: a -> (a -> a) -> (Nat -> a)
rec init step = \n ->
  case n of
    Z     -> init
    (S m) -> step (rec init step m)
```

Этой единственной функции, называемой *рекурсором*, достаточно для реализации всех рекурсивных функций натуральных чисел. Например, вот как можно было бы реализовать сложение:

```
plus  :: Nat -> Nat -> Nat
plus n = rec init step
  where
    init = n
    step = S
```

Эта функция принимает `n` в качестве аргумента и создает функцию (замыкание), которая принимает другое число и добавляет к нему `n`.

На практике программисты предпочитают реализовывать рекурсию напрямую — подход, который эквивалентен встраиванию рекурсора `rec`. Следующая реализация, возможно, проще для понимания:

```
plus n m = case m of
  Z     -> n
  (S k) -> S (plus k n)
```

Ее можно понимать так: если `m` равно нулю, то результатом будет `n`. В противном случае, если `m` является преемником некоторого `k`, то результатом будет преемник для `k + n`. Это то же самое, что заявить: `init = n` и `step = S`.

В императивных языках рекурсия часто заменяется итерацией. Концептуально итерацию легче понять, поскольку она соответствует последовательной декомпозиции. Шаги в последовательности обычно следуют некоторому естественному порядку. Это отличается от рекурсивной декомпозиции, где мы предполагаем, что выполнили всю работу до n-го шага, и объединяем этот результат со следующим последовательным шагом.

С другой стороны, рекурсия более естественна при обработке рекурсивно определенных структур данных, таких как списки или деревья.

Эти два подхода эквивалентны, и компиляторы часто преобразуют рекурсивные функции в циклы, что называется *оптимизацией хвостовой рекурсии*.

Упражнение 7.1.1. *Реализуйте каррированную версию сложения как отображение-вне N в функциональный объект N^N. Подсказка: используйте следующие типы в рекурсоре*:

```
init :: Nat  -> Nat
step :: (Nat -> Nat) -> (Nat -> Nat)
```

7.2 Списки

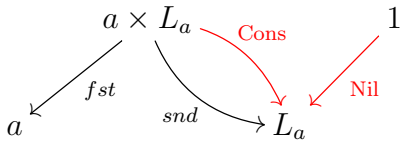
Список сущностей либо пуст, либо за сущностью следует список сущностей.

Это рекурсивное определение преобразуется в два правила введения для типа L_a, списка a:

$$\text{Nil} : 1 \to L_a$$
$$\text{Cons} : a \times L_a \to L_a$$

Элемент *Nil* характеризует пустой список, а *Cons* строит список из головы и хвоста.

На следующей диаграмме показана связь между проекциями и конструкторами списков. Проекции извлекают голову и хвост списка, построенного с помощью *Cons*.

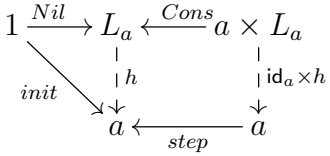


Это описание можно сразу перевести на Haskell:

```
data List a where
  Nil  :: List a
  Cons :: (a, List a) -> List a
```

Правило исключения

Предположим, что имеется отображение-вне, $h : L_a \to c$, от списка a к некоторому произвольному типу c. Вот как мы включим его в определение списка:



Мы использовали функториальность произведения, чтобы применить пару (id_a, h) к произведению $a \times L_a$.

Подобно объекту натурального числа, можно попытаться определить две стрелки, $init = h \circ Nil$ и $step$. Стрелка $step$ является решением для:

$$step \circ (\mathsf{id}_a \times h) = h \circ Cons$$

Опять же, не каждую стрелку h можно свести к такой паре стрелок.

Однако, для $init$ и $step$, можно определить h. Такая функция называется *складкой* или списочным катаморфизмом.

А это рекурсор списка на Haskell:

```
recList           :: c -> ((a, c) -> c) ->
                                  (List a -> c)
recList init step = \as ->
  case as of
    Nil           -> init
    Cons (a, as) -> step (a, recList
                                  init step as)
```

Для `init` и `step`, он создает отображение-вне списка.

Список является настолько базовым типом данных, что в Haskell для него есть встроенный синтаксис. Тип `List` записывается как `[a]`. Конструктор `Nil` представляет собой пустую пару квадратных скобок, `[]`, а конструктор `Cons` представляется инфиксным двоеточием `:`.

Можно применить сопоставление с образцом для этих конструкторов. Общее отображение-вне списка имеет вид:

```
h         :: [a] -> c
h []       = -- случай пустого списка
h (a: as) = -- случай головы и хвоста
            --  не пустого списка
```

В соответствии с рекурсором, имеется сигнатура типа функции `foldr` (складка $right$), которую можно найти в стандартной библиотеке Haskell:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

Вот одна из возможных реализаций:

```
foldr step init = \as ->
  case as of
    []     -> init
    a : as -> step a (foldr step init as)
```

Например, `foldr` можно использовать для вычисления суммы элементов списка натуральных чисел:

```
sum :: [Nat] -> Nat
sum  = foldr plus Z
```

Упражнение 7.2.1. *Подумайте, что произойдет, если заменить a в определении списка терминальным объектом. Подсказка: что такое кодирование натуральных чисел по основанию один?*

Упражнение 7.2.2. *Сколько существует отображений $h : L_a \to 1 + a$? Можно ли получить их все, используя рекурсор списка? Что можно сказать насчет Haskell-функций сигнатуры:*

```
h :: [a] -> Maybe a
```

Упражнение 7.2.3. *Реализуйте функцию, извлекающую третий элемент из списка, если список достаточно длинный. Подсказка: используйте* `Maybe a` *для типа результата.*

7.3 Функториальность

Функториальность означает, грубо говоря, возможность преобразовывать «содержимое» структуры данных. Содержимое списка L_a имеет тип a. Для стрелки $f : a \to b$ нужно определить отображение списков $h : L_a \to L_b$.

Списки определяются свойством отображения-вне, поэтому заменим цель c правила исключения на L_b. Получаем:

$$\begin{array}{ccccc} 1 & \xrightarrow{Nil_a} & L_a & \xleftarrow{Cons_a} & a \times L_a \\ & \searrow^{init} & \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle \mathsf{id}_a \times h} \\ & & L_b & \xleftarrow[step]{} & a \times L_b \end{array}$$

Поскольку здесь мы имеем дело с двумя разными списками, то должны различать их конструкторы. Например, имеется:

$$\mathrm{Nil}_a : 1 \to L_a$$
$$\mathrm{Nil}_b : 1 \to L_b$$

и, аналогично, для *Cons*.

Единственным кандидатом на $init$ является Nil_b, означающее, что h, действуя на пустой список элементов a создает пустой список элементов b:

$$h \circ Nil_a = Nil_b$$

Осталось определить стрелку:

$$step : a \times L_b \to L_b$$

Можно взять:

$$step = \mathrm{Cons}_b \circ (f \times \mathsf{id}_{L_b})$$

Это соответствует функции на Haskell:

```
mapList  :: (a -> b) -> List a -> List b
mapList f = recList init step
  where
    init          = Nil
    step (a, bs) = Cons (f a, bs)
```

или, используя встроенный синтаксис списка и встраивая рекурсор,

```
map           :: (a -> b) -> [a] -> [b]
map f []       = []
map f (a : as) = f a : map f as
```

Можно задаться вопросом, что мешает нам выбрать $step = snd$, в результате чего получается:

```
badMap           :: (a -> b) -> [a] -> [b]
badMap f []       = []
badMap f (a : as) = badMap f as
```

В следующей главе мы увидим, почему это является плохим выбором (подсказка: что произойдет, если мы применим `badMap` к id?).

Глава 8

Функторы

8.1 Категории

До сих пор мы рассматривали, фактически, только одну категорию — типов и функций. Что же нам известно вообще о (произвольной) категории?

Категория — это совокупность объектов и стрелок, которые характеризуются объектом-источником и объектом-целью. Каждая пара стрелок, удовлетворяющих условию — объект-цель одной стрелки совпадает с объектом-источником другой — может быть скомпонована. Такая композиция является ассоциативной операцией, а каждый объект снабжен тождественной стрелкой (тождественностью), для которой этот объект является, и источником, и целью.

То, что типы и функции образуют категорию, можно выразить на Haskell, определив композицию следующим образом:

```
(.)  :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

Композиция двух функций, `g` после `f`, — это новая функция, которая сначала применяет `f` к своему аргументу, а затем применяет `g` к результату.

Тождественность представляет собой полиморфную «ничего не делающую» функцию:

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

Можете легко убедиться, что такая композиция ассоциативна, а композиция любой функции (согласованной) с `id` не изменяет эту функцию.

Основываясь на определении категории, можно придумать всевозможные, в том числе необычные, категории. Например, имеется категория, в которой нет ни объектов, ни стрелок. Она бессодержательно удовлетворяет всем условиям категории. Имеется категория, которая содержит один объект и одну стрелку (можете догадаться, что это за стрелка?). Есть категория с двумя несвязанными объектами и категория с двумя объектами, соединенными одной стрелкой (плюс две тождественные стрелки) и так далее. Это примеры того, что я называю категориями фигурок — категориями с небольшим количеством объектов и стрелок.

Категория множеств

Мы также можем удалить из категории все стрелки (кроме тождественных). Такая категория «обнаженных» объектов называется *дискретной категорией* или множеством[1]. Поскольку мы ассоциируем структуру со стрелками, множество — это категория без структуры.

Множества образуют собственную категорию, обозначаемую **Set**[2]. Объекты в этой категории — множества, а стрелки — функции между множествами. Такие функции определяются как особый вид отношений, которые сами определяются как множества пар.

В этом наименьшем приближении, можно моделировать программирование в категории множеств. Мы часто представляем типы как множества значений, а функции — как теоретико-множественные функции. В этом нет ничего несоответствующего. На самом деле, все категорные конструкции, которые были описали до сих пор, имеют свои теоретико-множественные корни. Категорное произведение — это обобщение декартова произведения множеств, сумма — несвязное объединение и т.д.

Что предлагает теория категорий, так это большую точность: тонкое различие между абсолютно необходимой структурой и излишними деталями.

Например, теоретико-множественная функция не подходит под опре-

[1] Игнорирование проблем «размера».

[2] Опять же, игнорируя проблемы «размера», в частности, отсутствие множества всех множеств.

деление функции, с которой работают программисты. Наши функции должны иметь лежащие в основе алгоритмы, потому что они должны быть вычислимы некоторыми материальными системами, будь то компьютеры или человеческий мозг.

Противоположные категории

В программировании основное внимание уделяется категории типов и функций, но мы можем использовать эту категорию в качестве отправной точки для построения других категорий.

Одна такая категория называется *противоположной* категорией. Это категория, в которой инвертированы (или, обращены) все исходные стрелки: то, что в исходной категории являлось источником стрелки, теперь оказывается ее целью, и наоборот.

Противоположная к категории $\mathcal{C}$ обозначается $\mathcal{C}^{\mathsf{op}}$. Мы сталкивались с этой категорией, когда обсуждали двойственность. Объекты в $\mathcal{C}^{\mathsf{op}}$ такие же, как и в $\mathcal{C}$.

Всякий раз, когда существует стрелка $f : a \to b$ в $\mathcal{C}$, имеется соответствующая стрелка $f^{\mathsf{op}} : b \to a$ в $\mathcal{C}^{\mathsf{op}}$.

Композиция $g^{\mathsf{op}} \circ f^{\mathsf{op}}$ двух таких стрелок, $f^{\mathsf{op}} : a \to b$ и $g^{\mathsf{op}} : b \to c$, задается стрелкой $f \circ g$ (обратите внимание на обратный порядок).

Терминальный объект в $\mathcal{C}$ является инициальным объектом в $\mathcal{C}^{\mathsf{op}}$, произведение в $\mathcal{C}$ является суммой в $\mathcal{C}^{\mathsf{op}}$, и т.д.

Категории произведений

Имея две категории, $\mathcal{C}$ и $\mathcal{D}$, можно построить категорию произведений $\mathcal{C} \times \mathcal{D}$. Объекты в этой категории — это пары объектов $\langle c, d \rangle$, а стрелки — это пары стрелок.

Если имеется стрелка $f : c \to c'$ в $\mathcal{C}$ и стрелка $g : d \to d'$ в $\mathcal{D}$, то существует соответствующая стрелка $\langle f, g \rangle$ в $\mathcal{C} \times \mathcal{D}$. Эта стрелка направлена от $\langle c, d \rangle$ к $\langle c', d' \rangle$, оба являются объектами в $\mathcal{C} \times \mathcal{D}$. Две такие стрелки могут быть скомпонованы, если их компоненты компонуемы, соответственно, в $\mathcal{C}$ и в $\mathcal{D}$. Тождественная стрелка — это пара тождественных стрелок.

Нас больше всего интересуют две категории произведений: $\mathcal{C} \times \mathcal{C}$ и $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$, где $\mathcal{C}$ — знакомая нам категория типов и функций.

В обеих этих категориях объекты представляют собой пары объектов из $\mathcal{C}$. В первой категории, $\mathcal{C}\times\mathcal{C}$, морфизм от $\langle a, b\rangle$ к $\langle a', b'\rangle$ является парой $\langle f : a \to a', g : b \to b'\rangle$. Во второй категории, $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$, морфизм представляет собой пару $\langle f : a' \to a, g : b \to b'\rangle$, в которой первая стрелка идет в противоположном направлении.

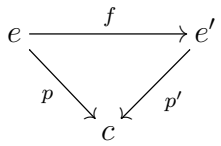
Категории срезов

В четко организованном универсуме объекты всегда остаются объектами, а стрелки — всегда стрелками. За исключением того, что иногда множества стрелок можно рассматривать как объекты. Но категории срезов нарушают это аккуратное разделение: они превращают отдельные стрелки в объекты.

Категория среза $\mathcal{C}/c$ описывает, как конкретный объект c рассматривается с точки зрения его категории $\mathcal{C}$. Это совокупность стрелок, указывающих на c. Но чтобы использовать стрелку, нужно определить оба ее конца. Поскольку один из этих концов есть c, требуется определить только другой.

Объект в *категории среза* $\mathcal{C}/c$ (также известной как над-категория) представляет собой пару $\langle e, p\rangle$, с $p : e \to c$.

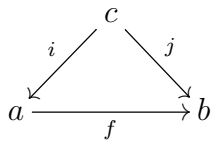
Стрелка между двумя объектами $\langle e, p\rangle$ и $\langle e', p'\rangle$ — это стрелка $f : e \to e'$ из $\mathcal{C}$, которая делает коммутативным следующий треугольник:



Категории ко-срезов

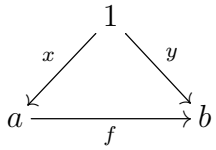
Существует двойственное понятие, категория ко-срезов $c/\mathcal{C}$, также известное как под-категория. Это категория стрелок, исходящих из фиксированного объекта c. Объекты в этой категории представляют собой пары $\langle a, i : c \to \text{a}\rangle$. Морфизмы в $c/\mathcal{C}$ — это стрелки, заставляющие соот-

ветствующие треугольники быть коммутативными.



В частности, если категория $\mathcal{C}$ содержит терминальный объект 1, то ко-срез $1/\mathcal{C}$ имеет в качестве объектов глобальные элементы всех объектов из $\mathcal{C}$.

Морфизмы $1/\mathcal{C}$, соответствующие стрелкам $f : a \to b$, отображают множество глобальных элементов a к множеству глобальных элементов b.



В частности, построение категории ко-срезов из категории типов и функций оправдывает нашу интуицию о типах как о множествах значений, значения которых представлены глобальными элементами типов.

8.2 Функторы

Мы видели примеры функториальности при обсуждении алгебраических типов данных. Идея в том, что такой тип данных «помнит» способ его создания, и мы можем манипулировать этой памятью, применяя стрелку к ее «содержимому».

В некоторых случаях эта интуиция очень убедительна: мы думаем о тип-произведении как о паре, которая «содержит» свои ингредиенты. В конце концов, мы можем получить их с помощью проекций.

Это менее очевидно в случае функциональных объектов. Можно визуализировать функциональный объект, как тайно хранящий все возможные результаты и использующий аргумент функции для их индексации. Функция от `Bool`, очевидно, эквивалентна паре значений, одно для `True` и одно для `False`. Это известный прием программирования для реализации некоторых функций в виде таблиц поиска. Этот механизм

(хранение промежуточных результатов в памяти) называется *мемоизацией*.

Хотя непрактично запоминать функции, которые принимают, скажем, натуральные числа в качестве аргументов, мы все еще можем концептуализировать их как (бесконечные или даже несчетные) таблицы поиска.

Если мыслить тип данных контейнером значений, имеет смысл применить функцию для преобразования всех этих значений и создать преобразованный контейнер. Когда это возможно, мы говорим, что тип данных является *функториальным*.

Опять же, функциональные типы требуют некоторого недоверия. Вы визуализируете функциональный объект как таблицу просмотра с ключами определенного типа. Если вы хотите использовать другой связанный тип в качестве ключа, вам нужна функция, которая переводит новый ключ в исходный ключ. Вот почему для обеспечения функториальности функционального объекта одна из стрелок инвертирована:

```
dimap      :: (a' -> a) -> (b -> b') ->
              (a -> b)  -> (a' -> b')
dimap f g h = g . h . f
```

Вы применяете преобразование к функции `h :: a -> b`, у которой есть «рецептор», реагирующий на значения типа `a`, и вы хотите использовать его для обработки ввода типа `a'`. Это возможно только в том случае, если у вас есть преобразователь из `a'` в `a`, а именно, `f :: a' -> a`.

Идею типа данных, «содержащего» значения другого типа, можно также выразить, сказав, что один тип данных параметризуется другим. Например, тип `List` параметризуется типом `a`.

Другими словами, `List` отображает тип `a` в тип `List a`. `List`, сам по себе, без аргумента, называется *конструктором типа*.

Функторы между категориями

В теории категорий конструктор типов моделируется как отображение объектов к объектам. Это функция на объектах. Ее не следует путать со стрелками между объектами, которые являются частью структуры категории.

Фактически, проще представить отображение *между* категориями. Каждый объект в исходной категории отображается к объекту в целевой

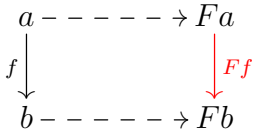
категории. Если a является объектом в $\mathcal{C}$, существует соответствующий объект Fa в $\mathcal{D}$.

Функториальное отображение, или *функтор*, отображает не только объекты, но также и стрелки между ними. Каждая стрелка

$$f : a \to b$$

в исходной категории есть соответствующая стрелка в целевой категории:

$$Ff : Fa \to Fb$$



Мы используем одну и ту же букву, здесь F, для обозначения отображения объектов и отображения стрелок.

Если категории определяют сущность *структуры*, то функторы — это отображения, сохраняющие эту структуру. Объекты, связанные в исходной категории, посредством функтора вызывают такую же связанность соответствующих объектов в целевой категории.

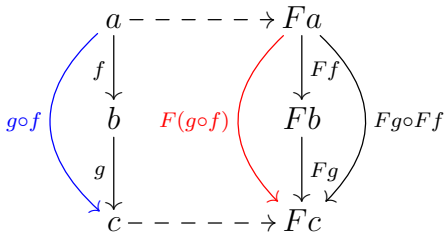
Структура категории определяется стрелками и их композицией. Следовательно, функтор должен сохранять композицию. Что компонуется в одной категории:

$$h = g \circ f$$

должно сохраняться на стрелках в целевой категории:

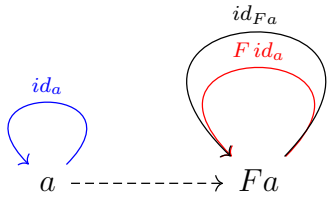
$$Fh = F(g \circ f) = Fg \circ Ff$$

Мы можем либо скомпоновать две стрелки в $\mathcal{C}$ и отобразить результат композиции в $\mathcal{D}$, либо сначала отобразить отдельные стрелки, а затем скомпоновать их в $\mathcal{D}$. Требуется, чтобы результат был одним и тем же.



Наконец, функтор должен сохранять тождественные стрелки:

$$F\,\mathsf{id}_a = \mathsf{id}_{Fa}$$



Все эти условия вместе определяют, что значит для функтора сохранять структуру категории.

Также важно понимать, какие условия *не* являются частью определения. Например, функтору разрешено отображать несколько объектов в один и тот же объект. Он также может отображать несколько стрелок в одну и ту же стрелку, если конечные точки совпадают.

В крайнем случае, любая категория может быть отображена к категории с одним объектом и одной стрелкой.

Кроме того, не все объекты или стрелки в целевой категории должны покрываться функтором. В крайнем случае у нас может быть функтор от одноэлементной категории к любой (непустой) категорию. Такой функтор выбирает один объект вместе с его тождественной стрелкой.

Постоянный функтор Δ_c является примером функтора, который отображает все объекты из исходной категории к одному объекту c в целевой категории, а все стрелки из исходной категории — к одной тождественной стрелке id_c.

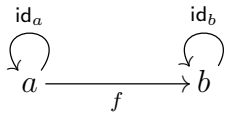
В теории категорий функторы часто используются для создания моделей одной категории внутри другой. Тот факт, что они могут объединять несколько объектов и стрелок в один, означает, что они создают упрощенные представления исходной категории. Они «абстрагируют» некоторые аспекты категории источника.

Тот факт, что они могут охватывать только часть целевой категории, означает, что модели встроены в более обширную среду.

Функторы от некоторых минималистичных категорий фигурок могут использоваться для определения шаблонов в более крупных категориях.

Упражнение 8.2.1. *Опишите функтор, источником которого является категория «шагающая стрелка». Это категория фигурок с двумя объектами и одной стрелкой между ними (плюс обязательные*

тождественные стрелки).



Упражнение 8.2.2. *Категория «ходячий изо» аналогична категории «шагающая стрелка», плюс еще одна стрелка, идущая обратно от b к a. Покажите, что функтор из этой категории всегда выбирает изоморфизм в целевой категории.*

8.3 Функторы в программировании

Эндофункторы — это класс функторов, которые проще всего выразить на языке программирования. Это функторы, отображающие категорию (здесь категорию типов и функций) на себя.

Эндофункторы

Одна часть эндофунктора — это отображение типов к типам. Это делается с помощью конструкторов типов, которые являются функциями уровня типов.

Конструктор списочного типа, `List`, отображает произвольный тип `a` к типу `List a`.

Конструктор типа `Maybe` отображает `a` к `Maybe a`.

Другая часть эндофунктора — это отображение стрелок. Имея функцию `a -> b`, мы хотим иметь возможность определить функцию `List a -> List b`, или `Maybe a -> Maybe b`. Это свойство «функториальности» типов данных, о котором шла речь выше. Функториальность позволяет поднять произвольную функцию до функции между преобразованными типами.

Функториальность может быть выражена на Haskell с помощью *класса типов*. В этом случае класс типов параметризуется конструктором типа `f` (на Haskell мы используем имена в нижнем регистре для переменных конструктора типов). Мы записываем, что `f` есть `Functor`, если существует соответствующее отображение функций, обозначенное `fmap`:

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

Компилятор «понимает», что `f` является конструктором типа, поскольку он применяется к типам, как в `f a` и `f b`.

Чтобы проинформировать компилятор о том, что конкретный конструктор типа есть `Functor`, надо предоставить для него реализацию `fmap`. Это делается путем определения экземпляра класса типов `Functor`. Например:

```
instance Functor Maybe where
  fmap g Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

Функтор также должен удовлетворять некоторым законам: он должен сохранять композицию и тождество. Эти законы не могут быть выражены на Haskell, но должны быть проверены программистом. Ранее мы уже видели определение `badMap`, которое не удовлетворяло законам тождества, но было бы принято компилятором. Это определило бы «недопустимый» экземпляр `Functor` для конструктора типа списка `[]`.

Упражнение 8.3.1. *Покажите, что* `WithInt` *является функтором*

```
data WithInt a = WithInt a Int
```

Некоторые элементарные функторы могут показаться тривиальными, но они служат строительными блоками для других функторов.

Имеется тождественный эндофунктор, который отображает все объекты и все стрелки на себя.

```
data Id a = Id a
```

Упражнение 8.3.2. *Покажите, что* `Id` *есть* `Functor`. *Подсказка*: *реализуйте для него экземпляр* `Functor`.

Имеется также постоянный функтор Δ_c, который отображает все объекты к одному объекту c, а все стрелки — к тождественной стрелке этого объекта. На Haskell — это семейство функторов, параметризованных целевым объектом `c`:

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

Этот конструктор типа игнорирует свой второй аргумент.

Упражнение 8.3.3. *Покажите, что* `(Const c)` *есть* `Functor`. *Подсказка*: *Конструктор типа принимает два аргумента, но здесь он частично применяется к первому аргументу. Он функториален по второму аргументу.*

Бифункторы

Мы также рассматривали конструкторы данных, которые принимают в качестве аргументов два типа: произведение и сумму. Они также были функториальными, но вместо одной функции они поднимали пару функций. В теории категорий мы бы определили их как функторы от категории произведений $\mathcal{C} \times \mathcal{C}$ к $\mathcal{C}$.

Такие функторы отображают пару объектов в объект и пару стрелок в стрелку.

На Haskell мы рассматриваем такие функторы в качестве членов отдельного класса, именуемого `Bifunctor`.

```
class Bifunctor f where
  bimap :: (a -> a') -> (b -> b') ->
                            (f a b -> f a' b')
```

Опять же, компилятор выводит, что `f` является конструктором типа с двумя аргументами (поскольку применяется к двум типам, например, `f a b`).

Чтобы предоставить компилятору информацию о том, что конкретный конструктор типа есть `Bifunctor`, мы должны определить экземпляр. Например, би-функциональность пары можно определить так:

```
instance Bifunctor (,) where
  bimap g h (a, b) = (g a, h b)
```

Упражнение 8.3.4. *Покажите, что* `MoreThanA` *является бифунктором.*

```
data MoreThanA a b = More a (Maybe b)
```

Контравариантные функторы

Функторы из противоположной категории $\mathcal{C}^{\mathrm{op}}$ называются *контравариантными*. Они обладают свойством подъема стрелок, идущих в противоположном направлении. Регулярные функторы иногда называют *ковариантными*.

На Haskell контравариантные функторы образуют `Contravariant` — класс типов:

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

Часто удобно думать о функторах как о производителях и потребителях. В таком видении (ковариантный) функтор является производителем. Производителя всех `a` можно превратить в производителя всех `b`, применив (используя `fmap`) функцию `a -> b`. И наоборот, чтобы превратить потребителя всех `a` в потребителя всех `b`, необходима функция, действующая в противоположном направлении, `b -> a`.

Пример: предикат — это функция, возвращающая `True` или `False`:

```
data Predicate a = Predicate (a -> Bool)
```

Понятно, что это контравариантный функтор:

```
instance Contravariant Predicate where
  contramap f (Predicate h) = Predicate (h . f)
```

Единственными нетривиальными примерами контравариантных функторов являются вариации на тему функциональных объектов.

Один из способов узнать, является ли данный тип функции ковариантным или контравариантным в одном из аргументов типа, — это присвоить полярности типам, используемым в определении. Говорят, что возвращаемый тип функции находится в *позитивной* позиции, поэтому он ковариантен; а тип аргумента находится в *негативной* позиции, поэтому он контравариантен. Но если поместить весь функциональный объект в негативную позицию другой функции, то его полярность изменится на противоположную.

Рассмотрим тип данных:

```
data Tester a = Tester ((a -> Bool) -> Bool)
```

Он содержит `a` в дважды негативной, а следовательно, в позитивной позиции. Вот почему это ковариантный `Functor`, являющийся производителем всех `a`:

```
instance Functor Tester where
  fmap f (Tester g) = Tester g'
    where g' h = g (h . f)
```

Заметим, что здесь важны скобки. Аналогичная функция `a -> Bool -> Bool` содержит `a` в *негативной* позиции. Это потому, что эта функция от `a` возвращает функцию `(Bool -> Bool)`. Эквивалентно, можно отменить ее каррирование, чтобы получить функцию, которая принимает пару: `(a, Bool) -> Bool`. В любом случае, `a` оказывается в негативной позиции.

Профункторы

Мы знаем, что функциональный тип является функториальным. Он одновременно поднимает две функции, как и `Bifunctor`, за исключением того, что одна из функций действует в противоположном направлении.

В теории категорий это соответствует функтору от произведения двух категорий, одна из которых является противоположной категорией: это функтор от $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$. Функторы от $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$ к **Set** называются *профункторами*.

На Haskell профункторы образуют класс типов:

```
class Profunctor f where
  dimap :: (a' -> a) -> (b -> b') ->
                          (f a b -> f a' b')
```

Можно представлять себе профунктор как тип, который одновременно является и производителем, и потребителем. Он потребляет один тип и производит другой.

Функциональный тип, который может быть записан как инфиксный оператор `(->)`, является экземпляром для `Profunctor`

```
instance Profunctor (->) where
  dimap f g h = g . h . f
```

Это соответствует нашей интуиции о том, что функция `a -> b` использует аргументы типа `a` и выдает результаты типа `b`.

В программировании все нетривиальные профункторы являются вариациями функционального типа.

8.4 Hom-функторы

Стрелки между любыми двумя объектами образуют множество. Это множество называется hom-*множеством* и обычно записывается с использованием имени категории, за которым следуют имена объектов:

$$\mathcal{C}(a, b)$$

Можно интерпретировать hom-множество $\mathcal{C}(a, b)$ как совокупность способов наблюдения за b с позиции a.

Другой способ взглянуть на hom-множества состоит в том, чтобы сказать, что они определяют отображение, которое назначает множество $\mathcal{C}(a, b)$ каждой паре объектов. Сами множества являются объектами категории **Set**. Таким образом, имеется отображение между категориями.

Это отображение функториально. Чтобы убедиться в этом, давайте рассмотрим, что происходит, когда мы преобразуем объекты a и b. Нас интересует преобразование, которое отображает множество $\mathcal{C}(a, b)$ к множеству $\mathcal{C}(a', b')$. Стрелки в **Set** являются обычными функциями, поэтому достаточно определить их действие на отдельные элементы множества.

Элементом $\mathcal{C}(a, b)$ является стрелка $h : a \to b$, а элементом $\mathcal{C}(a', b')$ — стрелка $h' : a' \to b'$. Известно как преобразовать одно в другое: нужно предварительно скомпоновать h со стрелкой $g' : a' \to a$, а после этого, результат — со стрелкой $g : b \to b'$.

Другими словами, отображение, переводящее пару $\langle a, b\rangle$ в множество $\mathcal{C}(a, b)$ является *профунктором*:

$$\mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathbf{Set}$$

Часто нас интересует изменение только одного из объектов, оставляя другой фиксированным. Когда фиксируется исходный объект и меняется цель, результатом является функтор, который записывается следующим образом:

$$\mathcal{C}(a, -) : \mathcal{C} \to \mathbf{Set}$$

Действие этого функтора на стрелку $g : b \to b'$ записывается как:

$$\mathcal{C}(a, g) : \mathcal{C}(a, b) \to \mathcal{C}(a, b')$$

и задается пост-композицией:

$$\mathcal{C}(a, g) = (g \circ -)$$

Изменение b означает переключение фокуса с одного объекта на другой, поэтому полный функтор $\mathcal{C}(a, -)$ объединяет все стрелки, исходящие из a в когерентное представление категории с точки зрения a. Это — «мир по a».

И наоборот, когда мы фиксируем цель и меняем источник hom-функтора, то получаем контравариантный функтор:

$$\mathcal{C}(-, b) : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$$

действие которого на стрелку $g' : a' \to a$ записывается как:

$$\mathcal{C}(g', b) : \mathcal{C}(a, b) \to \mathcal{C}(a', b)$$

и задается пред-композицией:

$$\mathcal{C}(g', b) = (- \circ g')$$

Функтор $\mathcal{C}(-, b)$ организует все стрелки, указывающие на b, в одно целостное представление. Это картина для b, «как ее видит мир».

Теперь можно переформулировать результаты главы об изоморфизмах. Если объекты a и b изоморфны, то изоморфны и их hom-множества. В частности,

$$\mathcal{C}(a, x) \cong \mathcal{C}(b, x)$$

и

$$C(x, a) \cong C(x, b)$$

Мы обсудим условия естественности в следующей главе.

Другой способ взглянуть на hom-функтор $\mathcal{C}(a, -)$ — это оракул, дающий ответы на вопрос: «Связан ли a со мной?» Если множество $\mathcal{C}(a, x)$ пусто, ответ отрицательный: «a не связан с x», иначе, каждый элемент

множества $\mathcal{C}(a, x)$ является доказательством того, что такая связь существует.

И наоборот, контравариантный функтор $\mathcal{C}(-, a)$ отвечает на вопрос: «Связан ли я с a?»

В совокупности, профунктор $\mathcal{C}(x, y)$ устанавливает релевантное для доказательства отношение между объектами. Каждый элемент множества $\mathcal{C}(x, y)$ является доказательством того, что x связан с y. Если это множество пусто, то соответствующие два объекта не связаны.

8.5 Композиция функторов

Точно так же, как мы можем компоновать функции, можно компоновать и функторы. Два функтора компонуемы, если целевая категория одного является исходной категорией другого.

На объектах, функторная композиция «G после F» сначала применяет F к объекту, а затем применяет G к результату. Так же она действует и на стрелках.

Очевидно, что составлять композицию можно только на компонуемых функторах. В то же время, все *эндофункторы* компонуемы, поскольку их целевая категория совпадает с исходной категорией.

На Haskell функтор — это параметризованный тип данных, поэтому композиция двух функторов снова является параметризованным типом данных. На объектах мы определяем:

```
data Compose g f a = Compose (g (f a))
```

Здесь, компилятор определяет, что `f` и `g` должны быть конструкторами типов, потому что они применяются к типам: `f` применяется к параметру типа `a`, а `g` применяется к результирующему типу.

В качестве альтернативы, можно сообщить компилятору, что первые два аргумента `Compose` являются конструкторами типов. Это можно сделать, предоставляя *сигнатуру вида*, для которой требуется языковое расширение `KindSignatures`, и которое надо поместить в начале файла с программой:

```
{-# language KindSignatures #-}
```

Также необходимо импортировать библиотеку `Data.Kind`, в которой определяется `Type`:

```
import Data.Kind
```

Сигнатура вида похожа на сигнатуру типа, за исключением того, что ее можно использовать для описания функций, работающих с типами.

Обычные типы имеют вид `Type`. Конструкторы типов имеют вид `Type -> Type`, поскольку они отображают типы в типы.

`Compose` принимает два конструктора типов и создает конструктор типа, поэтому его сигнатура вида есть:

```
(Type -> Type) -> (Type -> Type) -> (Type -> Type)
```

а полное определение таково:

```
data Compose :: (Type -> Type) -> (Type -> Type)
                               -> (Type -> Type)
  where
    Compose  :: (g (f a))      -> Compose g f a
```

Любые два конструктора типов могут быть скомпонованы таким образом. На данный момент нет требования, чтобы они были функторами.

Однако, если требуется поднять функцию, используя композицию конструкторов типов, `g` после `f`, то они должны быть функторами. Это требование закодировано как ограничение в объявлении экземпляра:

```
instance (Functor g, Functor f) =>
          Functor (Compose g f) where
  fmap h (Compose gfa) = Compose
                         (fmap (fmap h) gfa)
```

Ограничение `(Functor g, Functor f)` выражает условие, что оба конструктора типов являются экземплярами класса `Functor`. За ограничениями следует стрелка `=>`.

Конструктор типов, функториальность которого мы устанавливаем, — это `Compose f g`, который является частичным применением `Compose` к двум функторам.

В реализации `fmap`, мы используем механизм сопоставления с образцом в конструкторе данных `Compose`. Его аргумент `gfa` имеет тип `g (f a)`. Мы используем `fmap`, чтобы «попасть под» `g`. Затем мы используем `(fmap h)`, чтобы «попасть под» `f`. Компилятор «понимает», какой `fmap` надо использовать, анализируя типы.

Можно представлять себе составной функтор как контейнер контейнеров. Например, композиция `[]` с `Maybe` представляет собой список необязательных значений.

Упражнение 8.5.1. *Определите композицию*

«`Functor` *после* `Contravariant`»

Подсказка: *вы можете повторно использовать* `Compose`, *но указав объявление другого экземпляра.*

Категория категорий

Можно рассматривать функторы как стрелки между категориями. Как мы только что видели, функторы компонуемы, и просто проверяется, что эта композиция ассоциативна. Также существует тождественный (эндо-) функтор для каждой категории. Таким образом, сами категории образуют категорию, обозначаемую **Cat**.

И здесь математики начинают беспокоиться о проблемах «размера», поскольку вокруг скрываются парадоксы. Так что, правильное определение формулирует, что **Cat** — это категория *малых* категорий. Но пока мы не занимаемся доказательствами существования, можно игнорировать проблемы размера.

Глава 9

Естественные преобразования

Мы знаем, что если объекты a и b изоморфны, то они порождают биекции между множествами стрелок, которые теперь можно выразить как изоморфизмы между hom-множествами:

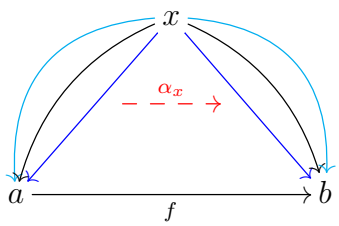
$$\mathcal{C}(a, x) \cong \mathcal{C}(b, x)$$
$$\mathcal{C}(x, a) \cong \mathcal{C}(x, b)$$

Однако, обратное неверно. Изоморфизм между hom-множествами не приводит к изоморфизму между объектами, *пока не* выполнены дополнительные условия естественности. Теперь мы переформулируем эти условия естественности в более общей обстановке.

9.1 Естественные преобразования между hom-функторами

Одним из способов установления изоморфизма между двумя объектами является непосредственное предоставление двух взаимно обратных стрелок. Но довольно часто проще сделать это косвенно, определив биекции между стрелками, либо подходящими к этим двум объекта, либо исходящими от этих двух объектов.

Например, как мы видели ранее, для каждого x может быть обратное отображение стрелок α_x.



Другими словами, для каждого x, существует отображение hom-множеств:

$$\alpha_x : \mathcal{C}(x, a) \to \mathcal{C}(x, b)$$

Когда мы варьируем x, два hom-множества становятся двумя (контравариантными) функторами, $\mathcal{C}(-, a)$ и $\mathcal{C}(-, b)$, а α можно рассматривать как отображение между ними. Такое отображение функторов, называемое преобразованием, на самом деле представляет собой семейство отдельных отображений α_x, по одному на каждый объект x в категории $\mathcal{C}$.

Функтор $\mathcal{C}(-, a)$ выражает то, как «со стороны видится» a, а функтор $\mathcal{C}(-, b)$ — как «со стороны видится» b.

Преобразование α переключается, по направлению, между этими двумя представлениями. Каждый компонент α, биекция α_x, показывает, что представление a от x изоморфно представлению b от x.

Условие естественности, которое обсуждалось ранее, было условием:

$$\alpha_y \circ (- \circ g) = (- \circ g) \circ \alpha_x$$

Оно связывает компоненты α, взятые на разных объектах. Другими словами, оно связывает представления двух разных наблюдателей x и y, которые соединены стрелкой $g : y \to x$.

Обе части этого уравнения действуют на hom-множество $\mathcal{C}(x, a)$. Результат находится в hom-множестве $\mathcal{C}(y, b)$. Можно переписать две стороны этого условия в виде:

$$\mathcal{C}(x, a) \xrightarrow{(-\circ g)} \mathcal{C}(y, a) \xrightarrow{\alpha_y} \mathcal{C}(y, b)$$
$$\mathcal{C}(x, a) \xrightarrow{\alpha_x} \mathcal{C}(x, b) \xrightarrow{(-\circ g)} \mathcal{C}(y, b)$$

Пред-композиция с $g : y \to x$ также является отображением hom-множеств. На самом деле это поднятие g контравариантным hom-функтором. Мы можем записать его как $\mathcal{C}(g, a)$ и $\mathcal{C}(g, b)$, соответственно.

$$\mathcal{C}(x,a) \xrightarrow{\mathcal{C}(g,a)} \mathcal{C}(y,a) \xrightarrow{\alpha_y} \mathcal{C}(y,b)$$
$$\mathcal{C}(x,a) \xrightarrow{\alpha_x} \mathcal{C}(x,b) \xrightarrow{\mathcal{C}(g,b)} \mathcal{C}(y,b)$$

Следовательно, условие естественности можно переписать в виде:

$$\alpha_y \circ \mathcal{C}(g,a) = \mathcal{C}(g,b) \circ \alpha_x$$

Это может быть проиллюстрировано коммутативной диаграммой:

$$\begin{array}{ccc} \mathcal{C}(x,a) & \xrightarrow{\mathcal{C}(g,a)} & \mathcal{C}(y,a) \\ {\scriptstyle \alpha_x}\downarrow & & \downarrow{\scriptstyle \alpha_y} \\ \mathcal{C}(x,b) & \xrightarrow[\mathcal{C}(g,a)]{} & \mathcal{C}(y,b) \end{array}$$

Теперь можно утверждать, что обратимое преобразование α между функторами $\mathcal{C}(-,a)$ и $\mathcal{C}(-,b)$, удовлетворяющее условию естественности, эквивалентно изоморфизму между a и b.

Можно следовать точно таким же рассуждениям для исходящих стрелок. На этот раз начнем с преобразования β, компоненты которого есть:

$$\beta_x : \mathcal{C}(a,x) \to \mathcal{C}(b,x)$$

Два (ковариантных) функтора $\mathcal{C}(a,-)$ и $\mathcal{C}(b,-)$ описывают окружающую обстановку с точки зрения a и b, соответственно. Обратимое преобразование β сообщает нам, что эти два взгляда эквивалентны, а условие естественности

$$(g \circ -) \circ \beta_x = \beta_y \circ (g \circ -)$$

говорит нам, что они ведут себя хорошо, когда мы переключаем фокус.

Коммутативная диаграмма, иллюстрирующая условие естественности, имеет вид:

$$\begin{array}{ccc} \mathcal{C}(a,x) & \xrightarrow{\mathcal{C}(a,g)} & \mathcal{C}(a,y) \\ {\scriptstyle \beta_x}\downarrow & & \downarrow{\scriptstyle \beta_y} \\ \mathcal{C}(b,x) & \xrightarrow[\mathcal{C}(b,g)]{} & \mathcal{C}(b,y) \end{array}$$

И здесь, такое обратимое естественное преобразование β устанавливает изоморфизм между a и b.

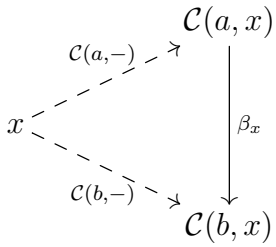
9.2 Естественное преобразование между функторами

Два hom-функтора из предыдущего раздела, это:

$$Fx = \mathcal{C}(a, x)$$
$$Gx = \mathcal{C}(b, x)$$

Каждый из них отображает категорию $\mathcal{C}$ к **Set**, поскольку именно там находятся hom-множества. Можно сказать, что они создают две разные *модели* $\mathcal{C}$ внутри **Set**.

Естественным преобразованием является, сохраняющее структуру, отображение между такими моделями.



Эта идея естественным образом распространяется на функторы между любой парой категорий. Любые два функтора

$$F : \mathcal{C} \to \mathcal{D}$$
$$G : \mathcal{C} \to \mathcal{D}$$

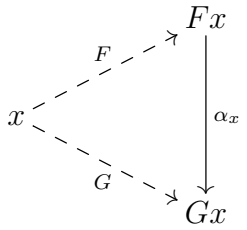
можно рассматривать в качестве двух разных моделей $\mathcal{C}$ внутри $\mathcal{D}$.

Чтобы преобразовать одну модель в другую, мы соединяем соответствующие точки стрелками в $\mathcal{D}$.

Для каждого объекта x из $\mathcal{C}$ мы выбираем стрелку, которая направлена от Fx к Gx:

$$\alpha_x : Fx \to Gx$$

Таким образом, естественное преобразование отображает объекты на стрелки.

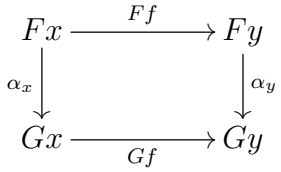


Однако, структура модели так же связана с объектами, как и со стрелками, так что рассмотрим, что происходит со стрелками. Для каждой стрелки $f : x \to y$ в $\mathcal{C}$, имеются две соответствующие стрелки в $\mathcal{D}$:

$$Ff : Fx \to Fy$$
$$Gf : Gx \to Gy$$

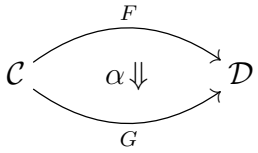
Они являются двумя поднятиями f. Их можно использовать для перемещения в пределах каждой из двух моделей. Кроме того, имеются компоненты α, которые позволяют переключаться между моделями.

Естественность означает, что не должно иметь значения, двигаетесь ли вы сначала внутри первой модели, а затем перемещаетесь во вторую, или сначала перемещаетесь во вторую, а затем двигаетесь внутри нее. Это иллюстрируется коммутативным *квадратом естественности*:



Такое семейство стрелок α_x, удовлетворяющее условию естественности, называется *естественным преобразованием*.

Диаграмма, которая изображает пару категорий, два функтора между ними и естественное преобразование α между функторами, имеет вид:



Поскольку каждой стрелке в $\mathcal{C}$ соответствует квадрат естественности, можно сказать, что естественное преобразование отображает объекты в стрелки, а стрелки — в коммутативные квадраты.

Если каждая компонента α_x естественного преобразования является изоморфизмом, то α называется *естественным изоморфизмом*.

Теперь можно переформулировать основной результат об изоморфизмах: два объекта изоморфны тогда и только тогда, когда существует естественный изоморфизм между их hom-функторами (либо ковариантными, либо контравариантными — подойдет любой из них).

Естественные преобразования обеспечивают очень удобный высокоуровневый способ выражения коммутативных условий в различных ситуациях. Мы будем использовать их в этом качестве, чтобы переформулировать определения алгебраических типов данных.

9.3 Естественные преобразования на Haskell

Естественное преобразование представляет собой семейство стрелок, параметризованных объектами. В программировании это соответствует семейству функций, параметризованных типами, то есть *полиморфной функции*.

Тип аргумента естественного преобразования описывается с использованием одного функтора, а возвращаемый тип — с помощью другого.

На Haskell можно определить тип данных, который принимает два конструктора типов, представляющих два функтора, и производит тип естественных преобразований:

```
data Natural :: (Type -> Type) ->
                (Type -> Type) -> Type where
     Natural :: (forall a. f a -> g a) ->
                                  Natural f g
```

Квантор `forall` сообщает компилятору, что функция является полиморфной, то есть она определена для каждого типа `a`. Поскольку `f` и `g` — функторы, это выражение определяет естественное преобразование.

Однако типы, определяемые `forall`, очень специфичны. Они полиморфны в смысле *параметрического полиморфизма*. Это означает, что для всех типов используется единая формула. Мы видели пример тождественной функции, которую можно записать так:

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

Тело этой функции очень простое, всего лишь переменная `x`. Неважно, каков тип `x`, формула остается той же.

Это отличается от *специального полиморфизма*. Специальная полиморфная функция может использовать разные реализации для разных типов. Примером такой функции является `fmap`, функция-член класса типов `Functor`. Существует одна реализация `fmap` для списков, другая — для `Maybe`, и т.д., для каждого конкретного случая.

Стандартное определение (параметрического) естественного преобразования в Haskell использует *синоним типа*:

```
type Natural f g = forall a. f a -> g a
```

Декларация `type` вводит псевдоним, сокращение, для правой стороны.

Оказывается, ограничение типа естественного преобразования параметрическим полиморфизмом имеет далеко идущие последствия. Такая функция автоматически удовлетворяет условиям естественности. Это пример параметрического создания так называемых *бесплатных теорем*.

Невозможно выразить равенства стрелок в Haskell, но можно использовать естественность для преобразования программ. В частности, если `alpha` является естественным преобразованием, то можно заменить:

```
fmap h . alpha
```

на

```
alpha . fmap h
```

Здесь, компилятор автоматически определит, какие версии `fmap` и какие компоненты `alpha` использовать.

Также, можно использовать более продвинутые языковые опции, чтобы сделать выбор явным. Можно выразить естественность с помощью пары функций:

```
oneWay        ::
  forall f g a b. (Functor f, Functor g) =>
      Natural f g -> (a -> b) -> f a -> g b
oneWay alpha h = fmap @g h . alpha @a

otherWay        ::
  forall f g a b. (Functor f, Functor g) =>
      Natural f g -> (a -> b) -> f a -> g b
otherWay alpha h = alpha @b . fmap @f h
```

Аннотации `@a` и `@b` задают компоненты параметрически полиморфной функции `alpha`, а аннотации `@f` и `@g` задают функторы, для которых создается конкретная полиморфная `fmap`.

Следующие расширения Haskell должны быть указаны в начале файла:

```
{- # language RankNTypes # -}
{- # language TypeApplications # -}
{- # language ScopedTypeVariables # -}
```

Приведем пример полезной функции, представляющей собой естественное преобразование между функтором списка и функтором `Maybe`:

```
safeHead         :: Natural [] Maybe
safeHead []       = Nothing
safeHead (a : as) = Just a
```

(функция `head` стандартной библиотеки «небезопасна» в том смысле, что она аварийно завершается, когда ей подается пустой список).

Другой пример — функция `reverse`, которая обращает список. Вот естественное преобразование от функтора списка к функтору списка:

```
reverse         :: Natural [] []
reverse []       = []
reverse (a : as) = reverse as ++ [a]
```

Между прочим, это очень неэффективная реализация. Фактическая библиотечная функция использует оптимизированный алгоритм.

Полезная интуиция для понимания естественных преобразований основана на идее, что функторы действуют как контейнеры для данных.

Есть две совершенно ортогональные вещи, которые можно делать с контейнером: можно преобразовывать содержащиеся в нем данные, не изменяя форму контейнера. Это то, что делает `fmap`. Или, можно перенести данные, не изменяя их, в другой контейнер. Вот что делает естественное преобразование: это процедура перемещения сущностей между контейнерами без знания того, что они собой представляют.

Другими словами, естественные преобразования переупаковывают содержимое одного контейнера в другой контейнер. Эти действия не зависят от типа содержимого, что означает, что они не могут проверять, создавать или изменять это содержимое. Все, что можно проделать, кроме перемещения, — игнорировать содержимое.

Условие естественности обеспечивает ортогональность этих двух операций. Неважно, или вы сначала измените данные, а затем переместите их в другой контейнер; или сначала переместить их, а потом будете модифицировать.

Это еще один пример успешного разложения сложной проблемы на последовательность более простых. Имейте в виду, однако, что не каждая операция с контейнерами данных может быть декомпозирована таким образом. Фильтрация, например, требует как проверки данных, так и изменения размера или даже формы контейнера.

С другой стороны, почти каждая параметрически полиморфная функция является естественным преобразованием. В некоторых случаях, возможно, придется рассматривать тождественный или константный функтор либо как источник, либо как цель. Например, полиморфную тождественную функцию можно рассматривать как естественное преобразование между двумя тождественными функторами.

Вертикальная композиция естественных преобразований

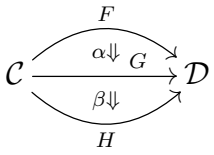
Естественные преобразования могут быть определены только между *параллельными* функторами, то есть функторами, имеющими одну и ту же исходную категорию и одну и ту же целевую категорию. Такие параллельные функторы образуют *категорию функторов*. Стандартное обозначение категории функторов между двумя категориями $\mathcal{C}$ и $\mathcal{D}$ — это $[\mathcal{C}, \mathcal{D}]$ — имена двух категорий в квадратных скобках.

Объекты в $[\mathcal{C}, \mathcal{D}]$ — это функторы, а стрелки — естественные преоб-

разования.

Чтобы показать, что это действительно категория, мы должны определить композицию естественных преобразований. Это несложно, если иметь в виду, что компоненты естественных трансформаций — это обычные стрелки в целевой категории. Эти стрелки компонуемы.

Действительно, предположим, что имеется естественное преобразование α между двумя функторами F и G. Необходимо скомпоновать его с другим естественным преобразованием β, направленным от G к H.



Рассмотрим компоненты этих преобразований на некотором объекте x

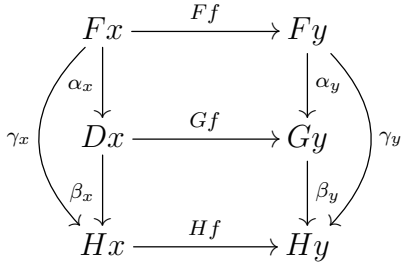
$$\alpha_x : Fx \to Gx$$
$$\beta_x : Gx \to Hx$$

Существуют лишь две стрелки в $\mathcal{D}$, которые можно скомпоновать. Так что, можно определить составное естественное преобразование γ следующим образом:

$$\gamma : F \to H$$
$$\gamma_x = \beta_x \circ \alpha_x$$

Это называется *вертикальной композицией* естественных преобразований и записывается с использованием точки $\gamma = \beta \cdot \alpha$ или как простое соединение $\gamma = \beta\alpha$.

Условие естественности для γ можно показать, соединив вместе (по вертикали) два квадрата естественности, для α и β:



В Haskell, вертикальная композиция естественных преобразований — это обычная композиция функций, применяемая к полиморфным функциям. Используя интуицию о том, что естественные преобразования перемещают элементы между контейнерами, вертикальная композиция комбинирует два таких перемещения одно за другим.

Категории функторов

Поскольку композиция естественных преобразований определяется в терминах композиции стрелок, она автоматически ассоциативна.

Существует также тождественное естественное преобразование id_F, определенное для каждого функтора F. Его компонентом при x является обычная тождественная стрелка на объекте Fx:

$$(\mathsf{id}_F)_x = \mathsf{id}_{Fx}$$

Подводя итог, для каждой пары категорий $\mathcal{C}$ и $\mathcal{D}$ существует категория функторов $[\mathcal{C}, \mathcal{D}]$ с естественными преобразованиями в качестве стрелок.

hom-множество в этой категории представляет собой множество естественных преобразований между двумя функторами F и G. Следуя стандартному соглашению об обозначениях, запишем его как:

$$[\mathcal{C}, \mathcal{D}](F, G)$$

с именами категорий, за которым следуют имена двух объектов (здесь функторы) в круглых скобках.

В теории категорий объекты и стрелки изображаются по-разному. Объекты — это точки, а стрелки — линии с наконечниками.

В Cat, категории категорий, функторы изображаются стрелками. Но, в категории функторов [C; D], функторы — точки, а естественные преобразования — стрелки.

То, что является стрелкой в одной категории, может быть объектом в другой.

Упражнение 9.3.1. *Докажите условие естественности композиции естественных преобразований*:

$$\gamma_y \circ Ff = Hf \circ \gamma_x$$

Подсказка: *используйте определение* γ *и условия естественности для* α *и* β.

Горизонтальная композиция естественных преобразований

Второй вид композиции естественных преобразований индуцируется композицией функторов. Предположим, имеется пара компонуемых функторов

$$F : \mathcal{C} \to \mathcal{D} \qquad G : \mathcal{D} \to \mathcal{E}$$

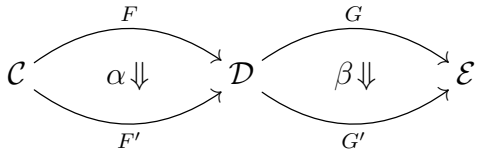
и, параллельно, еще одна пара компонуемых функторов:

$$F' : \mathcal{C} \to \mathcal{D} \qquad G' : \mathcal{D} \to \mathcal{E}$$

Также имеются два естественных преобразования:

$$\alpha : F \to F' \qquad \beta : G \to G'$$

В графическом представлении:



Горизонтальная композиция $\beta \circ \alpha$ отображает $G \circ F$ к $G' \circ F'$.

Выберем произвольный объект x в $\mathcal{C}$ и и попробуем определить этот компонент компоновки $(\beta \circ \alpha)$ в x. Это должен быть морфизм в $\mathcal{E}$:

$$(\beta \circ \alpha)_x : G(Fx) \to G'(F'x)$$

Мы можем использовать α для отображения x к некоторой стрелке

$$\alpha_x : Fx \to F'x$$

Можно поднять эту стрелку, используя G

$$G(\alpha_x) : G(Fx) \to G(F'x)$$

Чтобы перейти отсюда к $G'(F'x)$, можно использовать соответствующий компонент β:

$$\beta_{F'x} : G(F'x) \to G'(F'x)$$

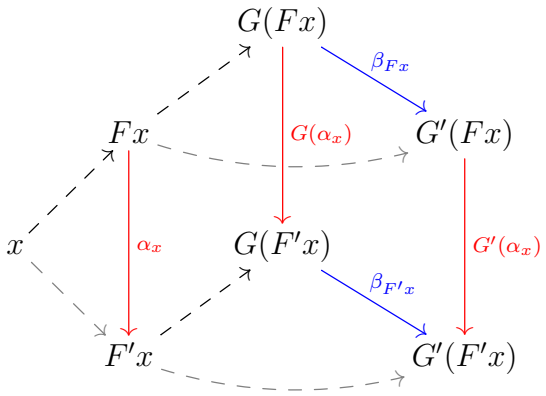
В целом, имеем

$$(\beta \circ \alpha)_x = \beta_{F'x} \circ G(\alpha_x)$$

Но есть и другой, не менее вероятный кандидат:

$$(\beta \circ \alpha)_x = G'(\alpha_x) \circ \beta_{Fx}$$

К счастью, они равноправны из-за естественности β.



Доказательство естественности $\beta \circ \alpha$ оставлено в качестве упражнения для заинтересованного читателя.

Можно перевести это непосредственно на Haskell. Начнем с двух естественных преобразований:

```
alpha :: forall x. F x -> F' x
beta  :: forall x. G x -> G' x
```

Их горизонтальная композиция имеет следующую сигнатуру типа:

```
beta_alpha :: forall x. G (F x) -> G' (F' x)
```

Она имеет две эквивалентные реализации. Первая из них:

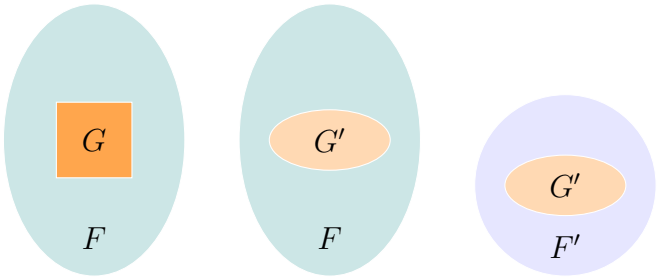
```
beta_alpha = beta . fmap alpha
```

Компилятор автоматически выберет правильную версию `fmap` для функтора `G`. Вторая реализация, это:

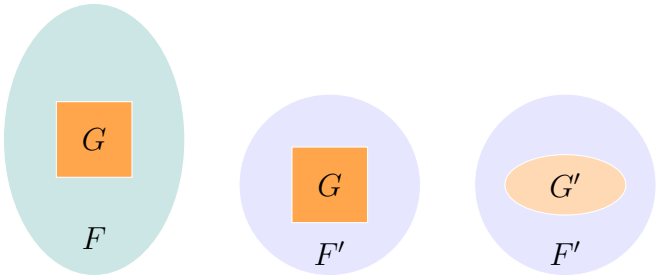
```
beta_alpha = fmap alpha . beta
```

Здесь компилятор выберет версию `fmap` для функтора `G'`.

Какова интуиция для горизонтальной композиции? Мы уже знаем, что естественное преобразование можно рассматривать как переупаковку данных между двумя контейнерами (функторами). Здесь мы имеем дело с вложенными контейнерами и начинаем с внешнего контейнера, описываемого `F`, который заполнен внутренними контейнерами, каждый из которых описывается `G`. Имеются два естественных преобразования: `alpha`, для переноса содержимого `F` в `F'`, и beta, для перемещения содержимого `G` в `G'`. Есть два способа перемещения данных из `G(F x)` в `G'(F' x)`. Можно использовать `fmap alpha`, для переупаковки всех внутренних контейнеров, а затем использовать `beta`, для переупаковки внешнего контейнера.



Или, можно сначала использовать `beta`, для переупаковки внешнего контейнера, а затем применить `fmap alpha`, для переупаковки всех внутренних контейнеров. Конечный результат будет тот же.



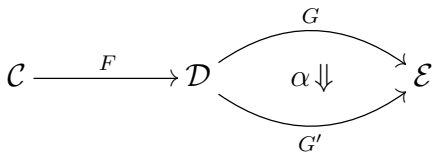
Пример 9.3.2. *Реализуйте две версии горизонтальной композиции* `safeHead` *после* `reverse`. *Сравните их результаты, действующие на различные аргументы.*

Пример 9.3.3. *Сделайте то же самое с горизонтальной композицией* `reverse` *после* `safeHead`.

Вискеринг

Довольно часто используется горизонтальная композиция, при этом одним из естественных преобразований является тождественность. Для такой композиции существует сокращенное обозначение. Например, $\alpha \circ \mathsf{id}_F$ записывается как $\alpha \circ F$.

Из-за характерной формы диаграммы такую композицию называют вискерингом (whisker — усы, бакенбарды).



В компонентах, имеем:

$$(\alpha \circ F)_x = \alpha_{Fx}$$

Как можно было бы перевести это на Haskell? Естественное преобразование является полиморфной функцией. Из-за параметричности оно определяется одной и той же формулой для всех типов. Таким образом, вискеринг справа не изменяет формулу, но меняет сигнатуру функции.

Например, если объявлением `alpha` является:

```
alpha   :: forall x. G x -> G' x
```

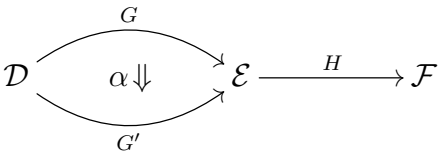
то его вискеринг-версией будет:

```
alpha_f :: forall x. G (F x) -> G' (F x)
alpha_f  = alpha
```

Из-за автоматического вывода типов в Haskell такое изменение часто происходит неявно. Когда вызывается полиморфная функция, не нужно указывать, какой компонент естественного преобразования выполняется — программа проверки типов выясняет это, анализируя тип аргумента.

Интуиция в этом случае такова, что мы переупаковываем внешний контейнер, оставляя внутренние контейнеры нетронутыми.

Точно так же, $\mathsf{id}_H \circ \alpha$ записывается как $H \circ \alpha$.



В компонентах:

$$(H \circ \alpha)_x = H(\alpha_x)$$

На Haskell, поднятие α_x с использованием H выполняется с помощью `fmap`, поэтому для:

```
alpha   :: forall x. G x -> G' x
```

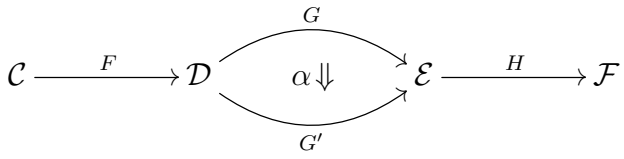
вискеринг-версией было бы:

```
h_alpha :: forall x. H (G x) -> H (G' x)
h_alpha  = fmap alpha
```

Опять же, механизм вывода типов Haskell определяет, какую версию `fmap` использовать (здесь это версия из `Functor`-экземпляра `G`).

Интуиция в этом случае такова, что переупаковывается внешний контейнер, а внутренние контейнеры остаются нетронутыми.

Наконец, во многих приложениях естественное преобразование является вискерингом с обеих сторон:



В компонентах, имеем:

$$(H \circ \alpha \circ F)x = H(\alpha_{Fx})$$

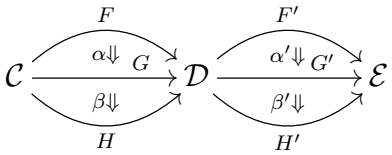
а на Haskell:

```
h_alpha_f :: forall x. H (G (F x)) -> H (G' (F x))
h_alpha_f  = fmap alpha
```

Здесь, интуитивно понятно, что имеется тройной слой контейнеров; преобразуется средний, оставляя внешний контейнер и все внутренние контейнеры нетронутыми.

Закон обмена

Можно комбинировать вертикальную композицию с горизонтальной композицией, как показано на следующей диаграмме:



Закон обмена гласит, что порядок композиции не имеет значения: можно сначала выполнить вертикальную композицию, а затем горизонтальную, или сначала выполнить горизонтальную композицию, а затем вертикальную.

9.4 Переосмысление универсальных конструкций

Лао-цзы говорит, что самый простой узор — самый четкий.

Мы уже встречались с определениями сумм, произведений, экспонент, натуральных чисел и списков.

Подход старой школы к определению таких типов данных заключается в изучении их внутреннего устройства. Это метод теории множеств: мы анализируем, как элементы новых множеств строятся из элементов существующих множеств. Элементом суммы является либо элемент первого множества, либо элемент второго множества. Элемент произведения – это пара элементов. И так далее. Мы смотрим на объекты с инженерной точки зрения.

В теории категорий мы придерживаемся противоположного подхода. Нас не интересует, что внутри объекта или как это реализовано. Нас интересует назначение объекта, как его можно использовать и как он взаимодействует с другими объектами. Мы смотрим на объекты с утилитарной точки зрения.

Оба подхода имеют свои преимущества. Категорный подход появился позже, потому что нужно изучить множество примеров, прежде чем проявятся четкие закономерности. Но как только вы видите закономерности, вы обнаруживаете неожиданные связи между вещами, такие как двойственность между суммами и произведениями.

Определение конкретных объектов через их связи требует рассмотрения, возможно, бесконечного числа объектов, с которыми они взаимодействуют.

«Расскажи мне о своем отношении к Вселенной, и я скажу, кто ты».

Определение объекта его отображениями-вне или отображениями-внутрь по отношению ко всем объектам категории называется *универсальной конструкцией*.

Почему естественные преобразования так важны? Это связано с тем, что большинство категорных конструкций включают коммутативные диаграммы. Если мы сможем преобразовать эти диаграммы в квадраты естественности, мы поднимемся на один уровень вверх по лестнице абстракции и получим новые ценные идеи.

Возможность сжать множество фактов в небольшие элегантные формулы помогает нам увидеть новые закономерности. Мы увидим, например, что естественные изоморфизмы между hom-множествами появляются повсюду в теории категорий и, в конечном итоге, приводят к идее сопряжения.

Но сначала мы более подробно изучим несколько примеров, чтобы получить некоторое представление о лаконичном языке теории категорий. Попытаемся, например, расшифровать утверждение о том, что сумма, или копроизведение, двух объектов определяется следующим естественным изоморфизмом:

$$[\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_x) \cong \mathcal{C}(a + b, x)$$

Выбор объектов

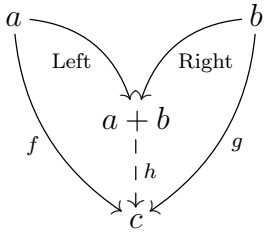
Даже такое простое действие, как указание на предмет, имеет в теории категорий особое толкование. Мы уже видели, что указание на элемент множества эквивалентно выбору функции от одноэлементного множества к нему. Точно так же выбор объекта в категории эквивалентен выбору функтора от категории с одним объектом. Или это можно сделать с помощью постоянного функтора от другой категории.

Довольно часто требуется выбрать пару объектов. Этого можно добиться, выбрав функтор от категории фигурок из двух объектов. Точно так же, выбор стрелки эквивалентен выбору функтора от категории «шагающая стрелка» и т.д.

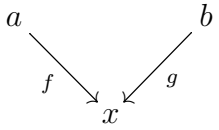
Разумно, выбирая функторы и естественные преобразования между ними, переформулировать все универсальные конструкции, с которые мы уже встречались.

Ко-пролеты как естественные преобразования

Определение суммы требует выбора двух суммируемых объектов и третьего, служащего целью отображения.



Эту диаграмму можно разложить на две более простые формы, называемые *ко-пролётами*:



Чтобы построить ко-пролет, сначала нужно выбрать пару объектов. Для этого, начнем с категории **2**, содержащую два объекта, 1 и 2. Будем использовать функтор

$$D : \mathbf{2} \to \mathcal{C}$$

для выбора объектов, a и b:

$$D1 = a$$
$$D2 = b$$

(D означает «диаграмма», так как два объекта образуют очень простую диаграмму, состоящую из двух точек в $\mathcal{C}$).

Будем, также, использовать постоянный функтор

$$\Delta_x : \mathbf{2} \to \mathcal{C}$$

для выбора объекта x. Этот функтор отображает, как 1, так и 2, к x (и две тождественные стрелки к id_x).

Поскольку оба функтора направлены от $\mathbf{2}$ к $\mathcal{C}$, можно определить между ними естественное преобразование α. В данном случае, это просто пара стрелок:

$$\alpha_1 : D1 \to \Delta_x 1$$
$$\alpha_2 : D2 \to \Delta_x 2$$

Это как раз две стрелки в ко-пролете.

Условие естественности для α тривиально, так как в $\mathbf{2}$ нет стрелок, кроме тождественных.

Может быть много ко-пролетов, разделяющих одни и те же три объекта, что означает: может быть много естественных преобразований между двумя функторами, D и Δ_x. Эти естественные преобразования образуют hom-множество в категории функторов $[\mathbf{2}, \mathcal{C}]$, а именно:

$$[\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_x)$$

Функториальность ко-пролетов

Рассмотрим, что произойдет, если варьировать объект x в ко-пролете. Мы имеем отображение F, принимающее x, к множеству ко-пролетов над x:

$$Fx = [\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_x)$$

Это отображение оказывается функториальным в x.

Чтобы убедиться в этом, рассмотрим стрелку $m : x \to y$. Подъем этой стрелки представляет собой отображение между двумя множествами естественных преобразований:

$$[\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_x) \to [\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_y)$$

Это может выглядеть очень абстрактно, пока вы не вспомните, что у естественных преобразований есть компоненты, и эти компоненты — обычные стрелки. Элемент левой части является естественным преобразованием:

$$\mu : D \to \Delta_x$$

Оно состоит из двух компонентов, соответствующих двум объектам в $\mathbf{2}$. Например, имеем

$$\mu_1 : D1 \to \Delta_x 1$$

или, используя определения D и Δ:

$$\mu_1 : a \to x$$

Это просто стрелка f на нашей диаграмме.

Точно так же, элемент правой части есть естественное преобразование:

$$\nu : D \to \Delta_y$$

Его компонент в 1 является стрелкой

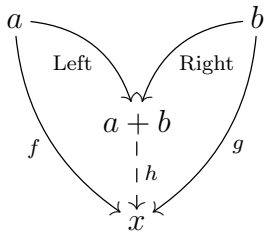
$$\nu_1 : a \to y$$

Можно перейти от μ_1 к ν_1, просто пост-компоновкой с $m : x \to y$. Таким образом, поднятие m — это по-компонентная пост-композиция $(m \circ -)$:

$$\nu_1 = m \circ \mu_1$$
$$\nu_2 = m \circ \mu_2$$

Сумма как универсальный ко-пролет

Из всех ко-пролетов, которые можно построить на паре a и b, тот, в котором стрелки, которые обозначены Left и Right, сходящиеся на $a + b$, является особенным. Существует единственное отображение его на любой другой ко-пролет — отображение, которое делает коммутативными два треугольника.



Теперь можно перевести это условие в утверждение о естественных преобразованиях и hom-множествах. Стрелка h является элементом hom-множества

$$\mathcal{C}(a + b, x)$$

Ко-пролет над x является естественным преобразованием, то есть элементом hom-множества в категории функторов:

$$[\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_x)$$

Оба являются hom-множествами в соответствующих категориях. И то, и другое — просто множества, то есть объекты категории **Set**. Эта категория образует мост между категорией функторов $[\mathbf{2}, \mathcal{C}]$ и «обычной» категорией $\mathcal{C}$, хотя концептуально они кажутся очень разными уровнями абстракции.

Перефразируя Сигмунда Фрейда: «Иногда, множество — это просто множество».

Наша универсальная конструкция есть биекция или изоморфизм множеств:

$$[\mathbf{2}, \mathcal{C}](\mathcal{D}, \Delta_x) \cong \mathcal{C}(a + b, x)$$

Более того, если варьировать объект x, то обе стороны будут вести себя как функторы от $\mathcal{C}$ к **Set**. Поэтому имеет смысл задаться вопросом, является ли это отображение функторов естественным изоморфизмом.

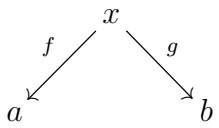
В самом деле, можно показать, что условие естественности этого изоморфизма переводится в коммутативные условия для треугольников в определении суммы. Таким образом, определение суммы можно заменить одним уравнением.

Произведение как универсальный ко-пролет

Аналогичный аргумент можно привести для универсальной конструкции произведения. Опять же, мы начинаем с категории фигурок $\mathbf{2}$ и функтора D. Но на этот раз будем использовать естественное преобразование, идущее в противоположном направлении

$$\alpha : \Delta_x \to D$$

Такое естественное преобразование представляет собой пару стрелок, образующих *пролёт*:



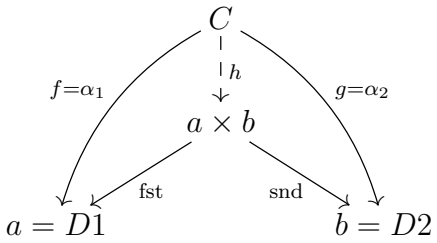
В совокупности эти естественные преобразования образуют hom-множество в категории функторов:

$$[\mathbf{2}, \mathcal{C}](\Delta_x, D)$$

Каждый элемент этого hom-множества находится во взаимно однозначном соответствии с единственным отображением h в произведение $a \times b$. Такое отображение является элементом hom-множества $\mathcal{C}(x, a \times b)$. Это соответствие выражается как изоморфизм:

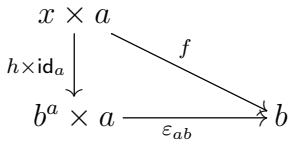
$$[\mathbf{2}, \mathcal{C}](\Delta_x, D) \cong \mathcal{C}(x, a \times b)$$

Можно показать, что естественность этого изоморфизма гарантирует коммутативность треугольников на диаграмме:



Экспоненциалы

Экспоненциалы или функциональные объекты определяются коммутативной диаграммой:



Здесь, f является элементом hom-множества $\mathcal{C}(x \times a, b)$, а h — элементом $\mathcal{C}(x, b^a)$.

Изоморфизм между этими множествами, естественный по x, определяет экспоненциальный объект.

$$\mathcal{C}(x \times a, b) \cong \mathcal{C}(x, b^a)$$

Функция f на приведенной диаграмме — это элемент левой части, а h — соответствующий элемент правой части. Преобразование α_x (которое

также зависит от a и b) отображает f к h.

$$\alpha_x : \mathcal{C}(x \times a, b) \to \mathcal{C}(x, b^a)$$

В Haskell, это обозначается `curry`. Инверсия, α^{-1}, известна как `uncurry`.

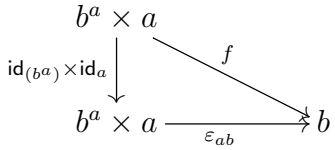
В отличие от предыдущих примеров, здесь оба hom-множества относятся к одной категории, и изоморфизм легко проанализировать более подробно. В частности, хотелось бы выявить, как коммутативное условие:

$$f = \varepsilon_{ab} \circ (h \times \mathsf{id}_a)$$

возникает из естественности.

Стандартный трюк Йонеды состоит в том, чтобы заменить x таким образом, чтобы уменьшить одно из hom-множеств до эндо-hom-множества, то есть hom-множества, источник которого совпадает с целью. Это позволит выбрать канонический элемент этого hom-множества, т.е., тождественную стрелку.

В нашем случае, подстановка b^a вместо x позволяет выбрать $h = \mathsf{id}_{(b^a)}$.



Условие перестановки в этом случае позволяет записать: $f = \varepsilon_{ab}$. Другими словами, мы получаем формулу для ε_{ab} в терминах α:

$$\varepsilon_{ab} = \alpha_x^{-1}(\mathsf{id}_x)$$

где x есть b^a.

Поскольку мы распознаем α^{-1} как `uncurry`, а ε как применение функции, можно записать это на Haskell в виде:

```
apply :: (a -> b, a) -> b
apply  = uncurry id
```

Это может показаться удивительным, пока не станет ясно, что каррингование `(a -> b, a) -> b` приводит к `(a -> b) -> (a -> b)`.

Также можно закодировать обе стороны основного изоморфизма функторами на Haskell:

```
data LeftFunctor  a b x = LF ((x, a) -> b)
data RightFunctor a b x = RF (x -> (a -> b))
```

Они являются контравариантными функторами по переменной типа `x`.

```
instance Contravariant (LeftFunctor a b) where
  contramap g (LF f) = LF (f . bimap g id)
```

Это означает, что поднятие $g : x \to y$ задается следующей пред-композицией

$$\mathcal{C}(y \times a, b) \xrightarrow{(-\circ(g \times \mathsf{id}_a))} \mathcal{C}(x \times a, b)$$

Аналогично

```
instance Contravariant (RightFunctor a b) where
  contramap g (RF h) = RF (h . g)
```

переводится в:

$$\mathcal{C}(y, b^a) \xrightarrow{(-\circ g)} \mathcal{C}(x, b^a)$$

Естественное преобразование α — это всего лишь тонкая инкапсуляция `curry`; и обратного ему `uncurry`:

```
alpha          :: forall a b x. LeftFunctor a b x
                               -> RightFunctor a b x
alpha   (LF f) = RF (curry f)

alpha_1        :: forall a b x. RightFunctor a b x
                                -> LeftFunctor a b x
alpha_1 (RF h) = LF (uncurry h)
```

Используя две формулы поднятия $g : x \to y$, получаем квадрат естественности:

$$\begin{array}{ccc} \mathcal{C}(y \times a, b) & \xrightarrow{(-\circ(g \times \mathsf{id}_a))} & \mathcal{C}(x \times a, b) \\ {\scriptstyle \alpha_y}\downarrow & & \downarrow{\scriptstyle \alpha_x} \\ \mathcal{C}(y, b^a) & \xrightarrow[(-\circ g)]{} & \mathcal{C}(x, b^a) \end{array}$$

Теперь, применим к нему трюк Йонеды и заменим y на b^a. Это также позволяет заменить g — который теперь идет от x к b^a — на h.

$$\begin{array}{ccc} \mathcal{C}(b^a \times a, b) & \xrightarrow{(-\circ(h\times \mathsf{id}_a))} & \mathcal{C}(x \times a, b) \\ \Big\downarrow{\scriptstyle \alpha_{(b^a)}} & & \Big\downarrow{\scriptstyle \alpha_x} \\ \mathcal{C}(b^a, b^a) & \xrightarrow[(-\circ h)]{} & \mathcal{C}(x, b^a) \end{array}$$

Мы знаем, что hom-множество $\mathcal{C}(b^a, b^a)$ содержит, по крайней мере, тождественную стрелку, поэтому можно выбрать элемент $\mathsf{id}_{(b^a)}$ в левом нижнем углу.

Обращая стрелку слева, мы знаем, что α^{-1}, действуя на тождественность, производит ε_{ab} в верхнем левом углу (это `uncurry` id-трюк).

Пред-композиция с h, действующая на тождественность, дает h в правом нижнем углу.

α^{-1}, действуя на h, дает f в правом верхнем углу.

$$\begin{array}{ccc} \varepsilon_{ab} & \xmapsto{(-\circ(h\times \mathsf{id}_a))} & f \\ {\scriptstyle \alpha^{-1}}\Big\uparrow & & \Big\uparrow{\scriptstyle \alpha^{-1}} \\ \mathsf{id}_{(b^a)} & \xmapsto[(-\circ h)]{} & h \end{array}$$

(стрелки $\mapsto$ обозначают действие функций на элементы множеств).

Таким образом, выбор $\mathsf{id}_{(b^a)}$ в левом нижнем углу фиксирует содержимое остальных трех углов. В частности, можно видеть, что верхняя стрелка, примененная к ε_{ab} дает f, что и является условием коммутации:

$$\varepsilon_{ab} \circ (h \times \mathsf{id}_a) = f$$

тем условием, которое мы и намеревались вывести.

9.5 Пределы и копределы

В предыдущем разделе мы определили сумму и произведение с помощью естественных преобразований. Это были преобразования между диаграммами, определенными как функторы от очень простой категории фигурок **2**, одним из этих функторов был постоянный функтор.

Ничто не мешает заменить категорию **2** чем-то более сложным. Например, можно было бы попробовать категории с нетривиальными стрелками между объектами или категории с бесконечным количеством объектов.

Вокруг таких конструкций построена целая лексика.

Мы использовали объекты категории $\mathbf{2}$ для индексации объектов из категории $\mathcal{C}$. Можно заменить $\mathbf{2}$ произвольной индексной категорией $\mathcal{J}$. Диаграмма в $\mathcal{C}$ по-прежнему определяется как функтор $D : \mathcal{J} \to \mathcal{C}$. Он выбирает объекты в $\mathcal{C}$, но также некоторые стрелки между ними.

В качестве второго функтора мы по-прежнему будем использовать постоянный функтор $\Delta_x : \mathcal{J} \to \mathcal{C}$.

Естественное преобразование, являющееся элементом hom-множества

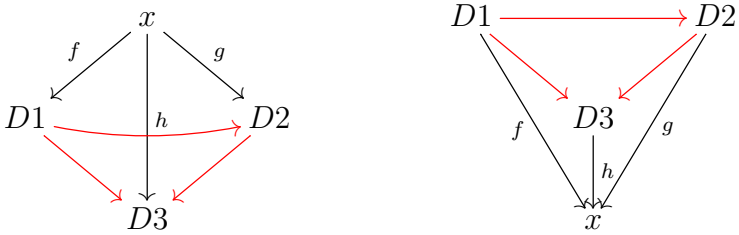
$$[\mathcal{J}, \mathcal{C}](\Delta_x, D)$$

теперь называется *конусом*. Двойственно, элемент

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x)$$

называется *ко-конусом*. Они обобщают пролет и ко-пролет, соответственно.

Схематично, конусы и ко-конусы выглядят следующим образом:



Поскольку индексная категория теперь может содержать стрелки, условия естественности для этих диаграмм больше не являются тривиальными. Постоянный функтор Δ_x сжимает все вершины до одной, так что квадраты естественности превращаются в треугольники. Естественность означает, что теперь все треугольники с x в их вершинах должны быть коммутативными.

Универсальный конус, если он существует, называется *пределом* диаграммы D, и записывается как $\mathrm{Lim} D$. Универсальность означает, что такой конус удовлетворяет следующему, естественному по x, изоморфизму:

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \cong \mathcal{C}(x, \mathrm{Lim} D)$$

Предел $\mathbf{Set}$-значного функтора имеет особенно простую характеристику. Это множество конусов с одноэлементным множеством при вер-

шине. Действительно, элементы предела, то есть функции от одноэлементного множества к нему, находятся во взаимно однозначном соответствии с такими конусами:

$$[\mathcal{J}, \mathcal{C}](\Delta_1, D) \cong \mathcal{C}(1, \mathrm{Lim} D)$$

Двойственно, универсальный ко-конус называется *ко-пределом*, и описывается следующим естественным изоморфизмом:

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(\mathrm{Colim} D, x)$$

Теперь можно сказать, что произведение является пределом (а сумма — копределом) диаграммы от индексной категории $\mathbf{2}$.

Пределы и ко-пределы определяют сущность графического шаблона.

Предел, как и произведение, определяется свойством отображения-внутрь. Ко-предел, как и сумма, определяется своим свойством отображения-вне.

Существует много интересных пределов и ко-пределов, и мы познакомимся с некоторыми из них, когда будем обсуждать алгебры и коалгебры.

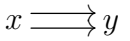
Упражнение 9.5.1. *Покажите, что предел категории «шагающая стрелка», т.е. двухобъектной категории со стрелкой, соединяющей два объекта, имеет те же элементы, что и первый объект на соответствующей диаграмме* («*элементы*» — *это стрелки от терминального объекта*).

Уравнители

Большая часть математики в средней школе включает в себя изучение того, как решать уравнения или системы уравнений. Уравнение уравнивает результаты двух разных способов производства чего-либо. Если имеется возможность производить вычитание, мы обычно собираем все в одну сторону и сводим задачу к вычислению нулей получившегося выражения. В геометрии та же идея выражается как пересечение двух геометрических объектов.

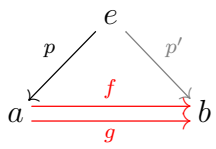
В теории категорий все эти закономерности воплощаются в единой конструкции, называемой уравнителем. Уравнитель — это предел диа-

граммы, шаблон которой задается категорией, содержащей две параллельные стрелки:



Две стрелки представляют два способа производства чего-либо.

Функтор от этой категории выбирает пару объектов и пару морфизмов в целевой категории. Предел этой диаграммы воплощает пересечение двух исходов. Это объект e с двумя стрелками $p : e \to a$ и $p' : e \to b$.



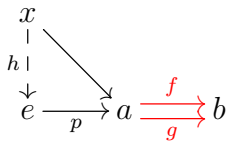
Имеем два коммутирующих условия:

$$p' = f \circ h$$
$$p' = g \circ h$$

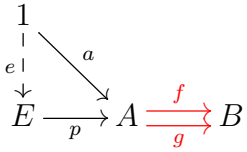
Это означает, что p' полностью определяется одним из уравнений, а другое превращается в ограничения:

$$f \circ p = g \circ p$$

Поскольку эквалайзер является пределом, это есть универсальная пара такая, как показано на диаграмме:



Чтобы развить интуицию для эквалайзеров, полезно рассмотреть, как это работает для множеств. Как обычно, трюк заключается в том, чтобы заменить x на одноэлементное множество 1:

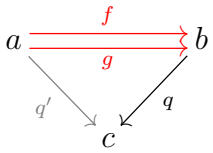


В этом случае a является элементом из A, таким, что $fa = ga$. Это просто способ сказать, что a является решением пары уравнений. Универсальность означает, что существует единственный элемент e из E, такой, что $p \circ e = a$. Другими словами, элементы E находятся во взаимно однозначном соответствии с решениями системы уравнений.

Ко-уравнители

Что двойственно приравниванию или пересечению? Это процесс обнаружения общих черт и организации содержимого в совокупностях. Например, можно распределить целые числа по четным и нечетным накопителям. В теории категорий этот процесс группирования описывается ко-уравнителями.

Ко-уравнитель — это ко-предел той же диаграммы, которая была использована для определения уравнителя:



На этот раз, стрелка q' полностью определяется q; а q должно удовлетворять уравнению:

$$q \circ f = q \circ g$$

Опять же, можно получить некоторую интуицию, рассмотрев ко-уравнитель двух функций, действующих на множествах.

$$A \underset{g}{\overset{f}{\rightrightarrows}} B \xrightarrow{q} C$$

Элемент $x \in A$ отображается к двум элементам fx и gx в B, но затем q отображает их обратно в один элемент из C. Этот элемент представляет совокупность. Универсальность означает, что C является копией B, в которой идентифицированы элементы, произведенные из одного и того же x.

Рассмотрим пример, где A — множество пар целых чисел `(m, n)`, таких, что либо оба четные, либо оба нечетные. Мы хотим уравнять две функции, которые являются двумя проекциями `(fst, snd)`. Множество уравнителя C будет иметь два элемента, соответствующих двум

совокупностям. Представим его как `Bool`. Уравнивающая функция `q` выбирает совокупность:

```
q  :: Int -> Bool
q n = n `mod` 2 == 0
```

Любая функция `q'`, которая не может различать компоненты наших пар, может быть однозначно разложена на множители через функцию `h`:

```
h          :: (Int -> a) -> Bool -> a
h q' True  = g' 0
h q' False = g' 1
```

Упражнение 9.5.2. *Запустите несколько тестов, которые покажут, что в приведенном выше примере факторизация* `(h g') . q` *дает тот же результат, что и* `q'`*, заданная следующим определением*:

```
import Data.Bits

q'  :: Int -> Bool
q' x = testBit x 0
```

Что представляет собой ко-уравнитель пары `(id, reverse)`*, с компонентами типа* `String -> String`? *Проверьте его универсальность, разложив на множители следующую функцию*:

```
q'  :: String -> Maybe Char
q' s = if even len
       then Nothing
       else Just (s !! (len `div` 2))
            where len = length s
```

Существование терминального объекта

Лао-цзы говорит: «Великие дела состоят из маленьких поступков».

До сих пор мы изучали пределы крошечных диаграмм, то есть функторов от простых категорий фигурок. Ничто, однако, не мешает нам определить пределы и ко-пределы там, где шаблоны рассматриваются

как бесконечные категории. Но существует градация бесконечности. Когда объекты в категории образуют правильное множество, такую категорию называют *малой*. К сожалению, в самом простом примере, категория **Set** множеств, не является малой. Мы знаем, что множества всех множеств не существует. **Set** — большая категория. Но, по крайней мере, все hom-множества в **Set** являются множествами. Говорят, что **Set** является *локально малой*. В дальнейшем мы всегда будем работать с локально малыми категориями.

Малый предел — это предел малой диаграммы, то есть функтор из категории, объекты и морфизмы которой образуют множества. Категория, в которой существуют все малые пределы, называется малыми полными, или просто *полными*. В частности, в такой категории существует произведение произвольного множества объектов. Также можно уравнять произвольное множество стрелок между двумя объектами. Если такая категория локально мала, значит, все уравнители существуют.

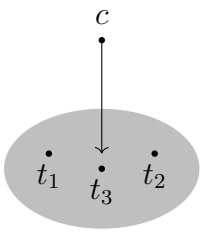
И наоборот, (малая) ко-полная категория имеет все малые ко-пределы. В частности, в такой категории существуют все малые ко-произведения и ко-уравнители.

Категория **Set** является, и полной, и ко-полной.

В ко-полной локально малой категории существует простой критерий существования терминального объекта: достаточно существования слабого терминального множества.

Слабый терминальный объект, как и терминальный объект, имеет стрелку, идущую от любого объекта; за исключением того, что такая стрелка не обязательно единственная.

Слабое терминальное множество — это семейство объектов t_i, индексированное множеством I, такое, что для любого объекта c в $\mathcal{C}$ существует i и стрелка $c \to t_i$. Такое множество также называют *множеством решений*.



В ко-полной категории всегда можно построить ко-произведение $\coprod_{i \in I} t_i$. Этот ко-произведение является слабым терминальным объектом, потому

что к нему ведет стрелка от каждого c. Эта стрелка представляет собой составную часть стрелки к некоторому t_i с последующей инъекцией $\iota_i : t_i \to \coprod_{j \in I} t_j$.

Имея слабый терминальный объект, можно построить (сильный) терминальный объект. Сначала определим подкатегорию $\mathcal{T}$ категории $\mathcal{C}$, объектами которой являются t_i. Морфизмы в $\mathcal{T}$ — это все морфизмы в $\mathcal{C}$, которые связывают объекты в $\mathcal{T}$. Это называется *полной* подкатегорией для $\mathcal{C}$. Следуя нашей конструкции, $\mathcal{T}$ является малой.

Существует очевидный функтор включения F, который вкладывает $\mathcal{T}$ в $\mathcal{C}$. Этот функтор определяет малую диаграмму в $\mathcal{C}$. Оказывается, ко-пределом этой диаграммы является терминальный объект в $\mathcal{C}$.

Двойственно, аналогичную конструкцию можно использовать для определения инициального объекта, как предела слабого инициального множества.

Это свойство множеств решений пригодится при доказательстве теоремы Фрейда о присоединенном функторе.

9.6 Лемма Йонеды

Функтор от некоторой категории $\mathcal{C}$ к категории множеств можно рассматривать как модель этой категории в **Set**. Моделирование, как правило, представляет собой процесс с потерями: часть информации отбрасывается. Постоянный **Set**-значный функтор является крайним примером: он отображает всю категорию к одному множеству и его тождественную функцию.

hom-функтор создает модель категории, рассматриваемую с определенной точки зрения. Например, функтор $\mathcal{C}(a, -)$ предлагает панораму $\mathcal{C}$ с точки зрения a. Он организует все стрелки, выходящие из a, в пакеты, которые соединяются образами стрелок, проходящих между ними, в соответствии с исходной структурой категории-источника.

Некоторые точки зрения лучше, чем другие. Например, вид из инициального объекта довольно разреженный. Каждый объект x отображается в одноэлементное множество $\mathcal{C}(0, x)$, соответствующее единственному отображению $0 \to x$.

Вид из терминального объекта более интересен: он отображает все объекты в их множества (глобальных) элементов $\mathcal{C}(1, x)$.

Лемму Йонеды можно считать одним из самых глубоких утверждений или одним из самых тривиальных утверждений в теории категорий. Начнем с глубокой версии.

Рассмотрим две модели $\mathcal{C}$ в $\mathbf{Set}$: одна задается $\hom$-функтором $\mathcal{C}(a, -)$, то есть панорамным видом $\mathcal{C}$ из точка обзора a; а другая модель задается некоторым произвольным функтором $F : \mathcal{C} \to \mathbf{Set}$. Естественное преобразование между ними встраивает одну модель в другую. Оказывается, множество таких естественных преобразований полностью определяется значением F в точке a.

Множеством естественных преобразований является $\hom$-множество в категории функторов $[\mathcal{C}, \mathbf{Set}]$, так что формальное утверждение леммы Йонеды, это:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F) \simeq Fa$$

Причина, по которой это работает, заключается в том, что все отображения, задействованные в этой теореме, связаны требованиями сохранения структуры категории $\mathcal{C}$ и структуры ее моделей. В частности, условия естественности накладывают огромное множество ограничений на то, как компоненты естественного преобразования распространяются от одной точки к другой.

Доказательство леммы Йонеды начинается с единственной тождественной стрелки и позволяет естественности распространить ее на всю категорию.

Вот набросок доказательства. Он состоит из двух частей. Во-первых, по естественному преобразованию мы строим элемент Fa. Во-вторых, по элементу Fa мы строим соответствующее естественное преобразование.

Сначала выберем произвольный элемент в левой части: естественное преобразование α. Его компонента в точке x является функцией:

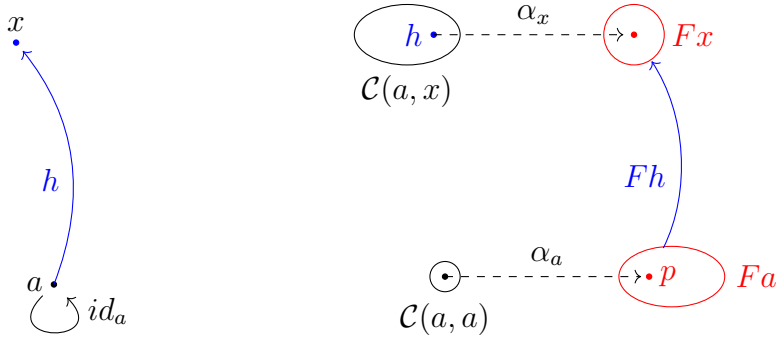
$$\alpha_x : \mathcal{C}(a, x) \to Fx$$

Теперь можно применить прием Йонеды: заменяем x на a:

$$\alpha_a : \mathcal{C}(a, a) \to Fa$$

и выбираем тождественность id_a в качестве канонического элемента $\mathcal{C}(a,a)$. Это предоставит элемент $\alpha_a(\mathsf{id}_a)$ из множества Fa. Это дает нам элемент $\alpha_a(\mathsf{id}_a)$ в множестве Fa. Это определяет отображение в одном направлении, от естественных преобразований к элементам множества Fa.

Глядя на изображение hom-функтора $\mathcal{C}(a,-)$, замечаем, что его действие на самом себе всегда является непустым множеством. Если имеется естественное преобразование от него к F, то Fa тоже должно быть непустым.



Теперь наоборот. По элементу p множества Fa надо построить естественное преобразование α. Во-первых, выберем p как действие α_a на $\mathsf{id}_a \in \mathcal{C}(a,a)$.

Теперь выберем произвольный объект x и произвольный элемент из $\mathcal{C}(a,x)$. Такой элемент соответствует некоторой стрелке $h : a \to x$. Требуемое естественное преобразование должно сопоставить его с элементом из Fx. Это можно сделать, подняв стрелку h с помощью F. Получим функцию:

$$Fh : Fa \to Fx$$

Можно применить эту функцию к p и получить некоторый элемент из Fx. Будем считать этот элемент действием α_x на h:

$$\alpha_x h = (Fh)p$$

Изоморфизм в лемме Йонеды естественен не только в a, но и в F. Другими словами, можно «перейти» от функтора F к другому функтору G, применяя стрелку в категории функторов, что является естественным преобразованием. Это довольно большой скачок в уровнях абстракции,

но все определения функториальности и естественности одинаково хорошо работают в категории функторов, где объекты — это функторы, а стрелки — это естественные преобразования.

Упражнение 9.6.1. *Заполните пробел в доказательстве, когда Fa пусто.*

Упражнение 9.6.2. *Покажите, что отображение*

$$\mathcal{C}(a, x) \to Fx$$

определенное выше, является естественным преобразованием. Подсказка: измените x, используя некоторую функцию $f : x \to y$.

Упражнение 9.6.3. *Покажите, что формулу для α_x можно вывести из предположения, что $\alpha_a(\mathsf{id}_a) = p$ и условия естественности. Подсказка: поднятие h $\hom$-функтором $\mathcal{C}(a, h)$ задается пост-композицией.*

Лемма Йонеды в программировании

Теперь о тривиальной части: доказательство леммы Йонеды переводится непосредственно в код на Haskell. Начнем с типа естественного преобразования между $\hom$-функтором `a -> x` и некоторым функтором `f`, и покажем, что он эквивалентен типу `f`, действующему на `a`.

```
forall x. (a -> x) -> f x. -- это изоморфно (f a)
```

Получим значение типа `f a`, используя стандартный прием Йонеды:

```
yoneda  :: Functor f => (forall x. (a -> x) ->
                                   f x) -> f a
yoneda g = g id
```

Обратное отображение:

```
yoneda_1  :: Functor f => f a ->
                     (forall x. a -> x) -> f x)
yoneda_1 y = \h -> fmap h y
```

Обратите внимание, что мы немного жульничаем, смешивая типы и множества. Лемма Йонеды в настоящей формулировке работает с **Set**-значными функторами. Опять же, правильнее будет сказать, что мы используем расширенную версию леммы Йонеды в самообогащаемой категории.

У леммы Йонеды есть несколько интересных приложений в программировании. Например, рассмотрим, что происходит, когда лемма Йонеды применяется к тождественному функтору. Получается изоморфизм между типом `a` (тождественный функтор, действующий на `a`) и

```
forall x. (a -> x) -> x
```

Мы интерпретируем это как утверждение, что любой тип данных `a` может быть заменен полиморфной функцией высшего порядка. Эта функция принимает другую функцию — называемую обработчиком, обратным вызовом или *продолжением* — в качестве аргумента.

Это стандартное продолжение передачи преобразования, которое часто используется в распределенном программировании, когда значение типа `a` должно быть получено с удаленного сервера. Это также полезно как преобразование программы, которое превращает рекурсивные алгоритмы в хвостовые рекурсивные функции.

Со стилем передачи продолжений трудно работать, потому что композиция продолжений весьма нетривиальна, что приводит к тому, что программисты часто называют «адом обратных вызовов». К счастью, продолжения образуют монаду, а значит, их композицию можно автоматизировать.

Контравариантная лемма Йонеды

Обратив несколько стрелок, можно применить лемму Йонеды и к контравариантным функторам. Это работает над естественными преобразованиями между контравариантным hom-функтором $\mathcal{C}(-, a)$ и контравариантным функтором F:

$$[\mathcal{C}^{\mathsf{op}}, \mathbf{Set}](\mathcal{C}(-, a), F) \cong Fa$$

Реализация отображения на Haskell:

```
coyoneda  :: Contravariant f => (forall x.
                        (x -> a) -> f x) -> f a
coyoneda g = g id
```

А это обратное преобразование:

```
coyoneda_1  :: Contravariant f => f a ->
                   (forall x. (x -> a) -> f x)
coyoneda_1 y = \h -> contramap h y
```

9.7 Вложение Йонеды

В замкнутой категории существуют экспоненциальные объекты, которые служат заменой для hom-множеств. Очевидно, это относится к категории множеств, где hom-множества, будучи множествами, автоматически являются объектами.

Но в категории категорий **Cat**, hom-множества — это множества функторов, и не сразу очевидно, что их можно повысить до объектов — то есть категорий. Но, как мы уже видели, это возможно! Функторы между любыми двумя категориями образуют *категорию функторов*.

Поэтому, функторы можно каррировать так же, как мы каррировали функции. Функтор от категории произведений можно рассматривать как функтор, возвращающий функтор. Другими словами, **Cat** — замкнутая (симметричная) моноидальная категория.

В частности, можно применить карринг к hom-функтору $\mathcal{C}(a, b)$. Это профунктор, или функтор от категории произведений:

$$\mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$$

Но это также и контравариантный функтор в первом аргументе a. Для каждого a в $\mathcal{C}^{\mathsf{op}}$ он производит функтор $\mathcal{C}(a, -)$, который является объектом в категории функторов $[\mathcal{C}, \mathbf{Set}]$. Можно записать это отображение как

$$\mathcal{C}^{\mathsf{op}} \to [\mathcal{C}, \mathbf{Set}]$$

В качестве альтернативы можно сосредоточиться на b и получить контравариантный функтор $\mathcal{C}(-, b)$. Это отображение можно записать как

$$\mathcal{C} \to [\mathcal{C}^{\mathsf{op}}, \mathbf{Set}]$$

Оба отображения являются функториальными, что означает, например, что стрелка в $\mathcal{C}$ отображается к естественному преобразованию в $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$.

Эти $\mathbf{Set}$-значные категории функторов настолько распространены, что имеют специальные названия. Функторы в $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ называются *предпучками*, а в $[\mathcal{C}, \mathbf{Set}]$ — *ко-предпучками*. (названия происходят из алгебраической топологии).

Обратим внимание на следующее прочтение hom-функтора:

$$\mathcal{Y} : \mathcal{C} \to [\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$$

Он принимает объект x и отображает его к предпучку

$$\mathcal{Y}_x = \mathcal{C}(-, x)$$

который можно представить себе как совокупность взглядов на x со всех возможных направлений.

Рассмотрим его действие на стрелки. Функтор $\mathcal{Y}$ поднимает стрелку $f : x \to y$ к отображению предпучков:

$$\alpha : \mathcal{C}(-, x) \to \mathcal{C}(-, y)$$

Компонента этого естественного преобразования при некотором z есть функция между hom-множествами:

$$\alpha_z : \mathcal{C}(z, x) \to \mathcal{C}(z, y)$$

которая реализуется просто как пост-композиция $(f \circ -)$.

Такой функтор $\mathcal{Y}$ можно рассматривать, как создающий модель $\mathcal{C}$ в категории предпучков. Но это не обычная модель — это вложение одной категории в другую. Это конкретное вложение называется *вложением Йонеды*, а функтор $\mathcal{Y}$ называется *функтором Йонеды*.

Прежде всего, каждый объект из $\mathcal{C}$ отображается к другому объекту (предпучку) из $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$. Говорят, что он *инъективен* на объектах. Но это еще не все: каждая стрелка из $\mathcal{C}$ отображается к другой стрелке. Тогда говорят, что функтор вложения является *точным*. Если этого недостаточно, отображение hom-множеств, к тому же, сюръективно, а это означает, что каждая стрелка между объектами в $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ идет от некоторой стрелки из $\mathcal{C}$. Говорят, что такой функтор является *полным*. В целом, рассматриваемое вложение является *вполне точным*.

Последний факт является прямым следствием леммы Йонеды. Известно, что для любого функтора $F : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$, существует естественный изоморфизм:

$$[\mathcal{C}^{\mathsf{op}}, \mathbf{Set}](\mathcal{C}(-, x), F) \simeq Fx$$

В частности, можно заменить F другим hom-функтором $\mathcal{C}(-, y)$:

$$[\mathcal{C}^{\mathsf{op}}, \mathbf{Set}](\mathcal{C}(-, x), \mathcal{C}(-, y)) \cong \mathcal{C}(x, y)$$

Левая часть — это hom-множество в категории предпучков, а правая часть — это hom-множество в $\mathcal{C}$. Они изоморфны, а это доказывает, что вложение является вполне точным.

Рассмотрим подробнее этот изоморфизм. Выберем элемент правого множества $\mathcal{C}(x, y)$ — стрелку f. Изоморфизм отображает его к естественному преобразованию, компонент которого в точке z является функцией:

$$\mathcal{C}(z, x) \to \mathcal{C}(zy)$$

Это отображение реализовано как пост-композиция $(f \circ -)$.

На Haskell мы бы записали это так:

```
toNat  :: (x -> y) -> (forall z.
                          (z -> x) -> (z -> y))
toNat f = \h -> f . h
```

Фактически, работает и такой синтаксис:

```
toNat f = (f . )
```

Обратное отображение:

```
fromNat      :: (forall z. (z -> x)  ->
                           (z -> y)) -> (x -> y)
fromNat alpha = alpha id
```

(опять обратите внимание на использование трюка Йонеды).

Этот изоморфизм отображает тождественность к тождественности, а композицию — к композиции. Это потому, что он реализован как пост-композиция, которая сохраняет и тождественность, и композицию. Мы сталкивались с этим в главе, посвященной изоморфизмам:

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

Поскольку он сохраняет композицию и тождественность, то этот изоморфизм сохраняет и *изоморфизмы*. Итак, если x изоморфен y, то предпучки $\mathcal{C}(-,x)$ и $\mathcal{C}(-,y)$ также изоморфны, и наоборот.

Это именно тот результат, который все время использовался для доказательства многочисленных изоморфизмов в предыдущих главах.

9.8 Представимые функторы

Объекты в категории ко-предпучков — это функторы, которые присваивают множества объектам в $\mathcal{C}$. Некоторые из этих функторов работают, выбирая эталонный объект a и присваивая всем объектам x их hom-множества $\mathcal{C}(a,x)$:

$$Fx = \mathcal{C}(a,x)$$

Такие функторы и все функторы, изоморфные им, называются *представимыми*. Весь функтор «представляется» одним объектом a.

В замкнутой категории функтор, сопоставляющий множество элементов x^a каждому объекту x представлен a, поскольку множество элементов x^a изоморфно $\mathcal{C}(a,x)$:

$$\mathcal{C}(1,x^a) \cong \mathcal{C}(1\times a,x) \cong \mathcal{C}(a,x)$$

С этой точки зрения представляющий объект a подобен логарифму функтора.

Аналогия идет глубже: точно так же, как логарифм произведения является суммой логарифмов, представляющий объект для типа данных произведения является суммой. Например, функтор, возводящий свой аргумент в квадрат с помощью произведения, $Fx = x\times x$, представляется как 2, то есть суммой $1+1$. Действительно, мы уже видели ранее, что $x\times x \cong x^2$.

Представимые функторы играют особую роль в категории **Set**-значных функторов. Отметим, что вложение Йонеды отображает объекты из $\mathcal{C}$ к представимым пред-пучкам. Оно отображает объект x к предпучку, представленного x-ом:

$$\mathcal{Y} : x \mapsto \mathcal{C}(-,x)$$

Можно найти целую категорию $\mathcal{C}$, объектов и морфизмов, вложенных в пред-пучковую категорию в качестве представимых функторов. Вопрос

в том, имеется ли что-нибудь в пред-пучковой категории «между» представимыми функторами?

Как рациональные числа плотны среди действительных чисел, так и представимые «плотны» среди (ко-)предпучков. Каждое действительное число может быть аппроксимировано рациональными числами. Каждый предпучок является ко-пределом представимых (а каждый ко-предпучок — пределом). Мы вернемся к этой теме, когда будем говорить о (ко-)концах.

Упражнение 9.8.1. *Опишите пределы и ко-пределы как представляющие объекты. Какие функторы они представляют*?

Игра в угадывание

Идея в том, что объекты можно описать по тому, как они взаимодействуют с другими объектами, иногда иллюстрируется игрой в воображаемые угадайки. Один категорный теоретик выбирает (секретный) объект в категории, а другой должен угадать, что это за объект (конечно, с точностью до изоморфизма).

Отгадывающему разрешается указывать на объекты и использовать их как «щупы» в секретном объекте. Предполагается, что противник каждый раз отвечает множеством стрелок от зондирующего объекта a к секретному объекту x. Это, конечно же, $\hom$-множество $\mathcal{C}(a, x)$.

Совокупность этих ответов, если противник не жульничает, будет определять предпучок $F : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$, а объект, который они скрывают, является его представляющим объектом.

Но откуда нам знать, что обмана нет? Чтобы проверить это, мы должны уметь задавать вопросы о стрелках. Для каждой выбранной нами стрелки они должны дать нам функцию между двумя множествами — множествами, которые они дали нам для ее крайних концов. Затем можно проверить, все ли тождественные стрелки отображаются на тождественные функции и отображаются ли композиции стрелок к композиции функций. Другими словами, мы сможем проверить, что F — функтор.

Однако, достаточно умный противник все же может нас обмануть. Пред-пучок, который он раскрывают нам, может описывать воображаемый объект, а мы не сможем доказать такую подмену. Оказывается,

такие воображаемые объекты зачастую не менее интересны, чем настоящие.

Представимые функторы в программировании

На Haskell класс представимых функторов определяется использованием двух функции, свидетельствующих об изоморфизме: `tabulate` превращает функцию в таблицу поиска, а `index` использует представляющий тип `Key` для ее индексации.

```
class Representable f where
  type Key f :: Type
  tabulate   :: (Key f -> a) -> f a
  index      :: f a -> (Key f -> a)
```

Алгебраические типы данных, использующие суммы, непредставимы — не существует формулы для логарифмирования суммы. Тип списка определяется как сумма, поэтому и он не представим.

Однако, имеется бесконечный поток. Концептуально такой поток похож на бесконечный кортеж, который технически является произведением. Поток представлен типом натуральных чисел. Другими словами, бесконечный поток эквивалентен отображению натуральных чисел.

```
data Stream a = Stm a (Stream a)
```

Определение экземпляра:

```
instance Representable Stream where
  type Key Stream = Nat
  tabulate g      = tab Z
    where
      tab n = Stm (g n) (tab (S n))
  index stm = \n -> ind n stm
    where
      ind Z (Stm a _)      = a
      ind (S n) (Stm _ as) = ind n as
```

Представимые типы полезны при реализации мемоизации функций.

Упражнение 9.8.2. *Реализуйте экземпляр* `Representable` *для* `Pair`:

```
data Pair x = Pair x x
```

Упражнение 9.8.3. *Является ли постоянный функтор, отображающий все в терминальный объект, представимым? Подсказка: каково значение логарифма от* 1*?*

На Haskell такой функтор может быть реализован как:

```
data Unit a = U
```

Реализуйте для него экземпляр `Representable`*.*

Упражнение 9.8.4. *Функтор списка не представим. Но можно ли его считать суммой или представимыми?*

9.9 2-категория Cat

В категории категорий, **Cat**, the hom-множества — это не просто множества. Каждое из них может быть возведено к категории функторов, а естественные преобразования играют роль стрелок. Такая структура называется 2-категорией.

На языке 2-категорий объекты называются 0-клетками, стрелки между ними — 1-клетками, а стрелки между стрелками — 2-клетками.

Очевидным обобщением этой картины является наличие 3-клеток, которые идут между 2-клетками и так далее. В n-категории имеются клетки, доходящие до n-го уровня.

Но почему бы не пойти дальше, вводя бесконечные категории? ∞-категории действительно существуют, и не из-за любопытства, у них есть практическое применение. Например, они используются в алгебраической топологии для описания точек, путей между точками, перемещаемых путями поверхностей, перемещаемых поверхностями объемов, и так далее, до бесконечности.

9.10 Полезные формулы

- Лемма Йонеды для ковариантных функторов:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F) \cong Fa$$

- Лемма Йонеды для контравариантных функторов:

$$[\mathcal{C}^{\mathsf{op}}, \mathbf{Set}](\mathcal{C}(-, a), F) \cong Fa$$

- Следствия леммы Йонеды:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(x, -), \mathcal{C}(y, -)) \cong \mathcal{C}(y, x)$$
$$[\mathcal{C}^{\mathsf{op}}, \mathbf{Set}](\mathcal{C}(-, x), \mathcal{C}(-, y)) \cong \mathcal{C}(x, y)$$

Глава 10

Сопряжения

Скульптор отсекает лишнее, пока не получится скульптура. Математик абстрагируется от второстепенных деталей, пока не появится закономерность.

Мы смогли определить множество конструкций, используя их свойства отображения-внутрь и отображения-вне. Те, в свою очередь, могут быть компактно записаны как изоморфизмы между hom-множествами. Этот шаблон естественных изоморфизмов между hom-множествами называется сопряжением, и, будучи однажды признанным, оно появляется практически везде.

10.1 Каррированное сопряжение

Определение экспоненциала является классическим примером сопряжением, которое связывает отображения-вне и отображения-внутрь. Каждое отображение-вне произведения соответствует единственному отображению-внутрь экспоненциала:

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, b^a)$$

Объект b играет роль фокуса с левой стороны; объект e становится наблюдателем с правой стороны.

Можно заметить два действующих функтора. Оба они параметризуются посредством a. Слева имеется функтор произведения $(- \times a)$, примененный к e. Справа находится функтор экспоненциала $(-)^a$, примененный к b.

Если записать эти функторы как:

$$L_a e = e \times a$$
$$R_a b = b^a$$

то естественный изоморфизм

$$\mathcal{C}(L_a e, b) \cong \mathcal{C}(e, R_a b)$$

называется сопряжением между ними.

В компонентах этот изоморфизм свидетельствует о том, что для заданного отображения $\phi \in \mathcal{C}(L_a e, b)$, существует единственное отображение $\phi^T \in \mathcal{C}(e, R_a b)$, и наоборот. Эти отображения иногда называют *транспонированными* друг другу — терминология из матричной алгебры.

Сокращенное обозначение для сопряжения — $L \dashv R$. Подставив функтор произведения вместо L и экспоненциальный функтор вместо R, можно записать каррированное сопряжение в компактной форме:

$$(- \times a) \dashv (-)^a$$

Экспоненциальный объект b^a иногда называют *внутренним-hom* и записывают как $[a, b]$. Это отличается от *внешнего-hom*, который представляет собой множество $\mathcal{C}(a, b)$. Внешний-hom *не* является объектом в $\mathcal{C}$ (кроме случаев, когда сама категория $\mathcal{C}$ есть **Set**). Используя эту нотацию, каррированное сопряжение тогда может быть записано как:

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, [a, b])$$

Категория, в которой имеет место это сопряжение, называется декартово замкнутой.

Поскольку функции играют центральную роль в каждом языке программирования, декартово замкнутые категории составляют основу всех моделей программирования. Мы интерпретируем экспоненту b^a как функциональный тип $a \to b$.

Здесь e играет роль внешней среды — Γ лямбда-исчисления. Морфизм в $\mathcal{C}(\Gamma \times a, b)$ интерпретируется как выражение типа b в среде Γ, расширенное переменной типа a. Таким образом, функциональный тип $a \to b$ представляет замыкание, которое может захватывать значение типа e из своего окружения.

Кстати, категория (малых) категорий $\mathbf{Cat}$ также является декартово замкнутой, что отражено в этом сопряжении между категориями произведений и категориями функторов, в котором используется одна и та же нотация внутреннего-hom:

$$\mathbf{Cat}(\mathcal{A} \times \mathcal{B}, \mathcal{C}) \cong \mathbf{Cat}(\mathcal{A}, [\mathcal{B}, \mathcal{C}])$$

Здесь, обе стороны представляют собой множества естественных преобразований.

10.2 Сопряжения суммы и произведения

Каррированное сопряжение связывает два эндофунктора, но сопряжение может быть легко обобщено на функторы, которые идут между разными категориями. Сначала рассмотрим несколько примеров.

Диагональный функтор

Типы суммы и произведения были определены с помощью биекций, где одна из сторон была одной стрелкой, а другая — парой стрелок. Пара стрелок может рассматриваться как одна стрелка в категории произведений.

Чтобы исследовать эту идею, нужно определить диагональный функтор Δ, который представляет собой специальное отображение от $\mathcal{C}$ к $\mathcal{C} \times \mathcal{C}$. Он принимает объект x и дублирует его, создавая пару объектов $\langle x, x \rangle$. Также он принимает стрелку f и дублирует ее $\langle f, f \rangle$.

Интересно, что диагональный функтор связан с постоянным функтором, встречавшимся ранее. Постоянный функтор можно рассматривать как функтор двух переменных — он просто игнорирует вторую. Мы видели это в определении на Haskell:

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

Чтобы увидеть эту связь, рассмотрим категорию произведений $\mathcal{C} \times \mathcal{C}$ как категорию функторов $[\mathbf{2}, \mathcal{C}]$, другими словами, экспоненциальный объект $\mathcal{C}^{\mathbf{2}}$ в $\mathbf{Cat}$. Действительно, функтор от $\mathbf{2}$ (категория фигурок с двумя объектами) выбирает пару объектов, которая эквивалентна одному объекту в категории произведений.

Функтор $\mathcal{C} \to [\mathbf{2}, \mathcal{C}]$ может быть декаррирован в $\mathcal{C} \times \mathbf{2} \to \mathcal{C}$. Диагональный функтор игнорирует второй аргумент, исходящий от $\mathbf{2}$: он делает одно и то же, независимо от того, равен ли второй аргумент 1, или 2. Точно так же действует постоянный функтор. Вот почему мы используем один и тот же символ Δ для обоих случаев.

Между прочим, этот аргумент можно легко обобщить на любую индексную категорию, а не только на $\mathbf{2}$.

Сопряжение суммы

Напомним, что сумма определяется своим свойством отображения-вне. Существует взаимно однозначное соответствие между стрелками, исходящими от суммы $a + b$, и парами стрелок, исходящими от a и b по отдельности. В терминах hom-множеств это можно записать так:

$$\mathcal{C}(a + b, x) \cong \mathcal{C}(a, x) \times \mathcal{C}(b, x)$$

где произведение в правой части — это просто декартово произведение множеств, то есть множество пар. Более того, мы видели ранее, что эта биекция естественна по x.

Мы знаем, что пара стрелок — это одна стрелка в категории произведений. Таким образом, можно рассматривать элементы с правой стороны как стрелки в $\mathcal{C} \times \mathcal{C}$, идущие от объекта $\langle a, b \rangle$ к объекту $\langle x, x \rangle$. Последнее можно получить, действуя диагональным функтором Δ на x. Имеем:

$$\mathcal{C}(a + b, x) \cong (\mathcal{C} \times \mathcal{C})(\langle a, b \rangle, \Delta_x)$$

Это биекция между hom-множествами двух разных категорий. Она удовлетворяет условиям естественности, поэтому является естественным изоморфизмом.

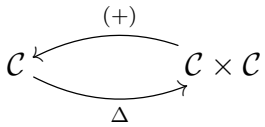
Здесь также можно обнаружить пару функторов. Слева присутствует функтор, который принимает пару объектов $\langle a, b \rangle$ и создает их сумму $a + b$:

$$(+) : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

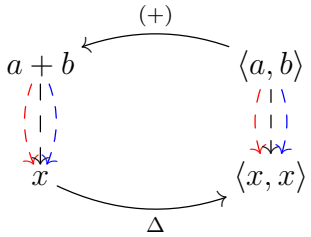
В правой части присутствует диагональный функтор Δ, идущий в противоположном направлении:

$$\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$$

В целом, имеется пара функторов между парой категорий:



и изоморфизм между hom-множествами:



Другими словами, имеется сопряжение:

$$(+) \dashv \Delta$$

Сопряжение произведения

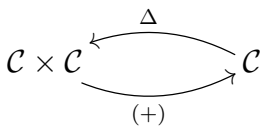
Можно применить те же рассуждения к определению произведения. На этот раз, имеется естественный изоморфизм между парами стрелок и отображение-внутрь произведения.

$$\mathcal{C}(x, a) \times \mathcal{C}(x, b) \cong \mathcal{C}(x, a \times b)$$

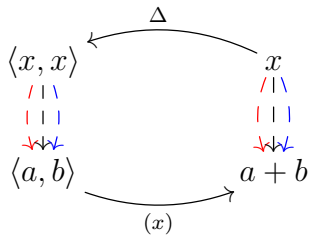
Заменив пары стрелок на стрелки из категории произведений, получим:

$$(\mathcal{C} \times \mathcal{C})(\Delta_x, \langle a, b \rangle) \cong \mathcal{C}(x, a \times b)$$

Это два функтора, идущие во взаимно противоположных направлениях:



а это изоморфизм hom-множеств:



Другими словами, имеется сопряжение:

$$\Delta \dashv (\times)$$

Дистрибутивность

В би-декартово замкнутой категории произведения распределяются по суммам. Мы уже сталкивались с одним вариантом соответствующего доказательства, используя универсальные конструкции. Сопряжения в сочетании с леммой Йонеды предоставляют более мощные инструменты для решения этой задачи.

Мы хотим вывести естественный изоморфизм:

$$(b + c) \times a \cong b \times a + c \times a$$

Вместо того, чтобы доказывать эту эквивалентность напрямую, покажем, что отображения, с обеих сторон, к произвольному объекту x изоморфны:

$$\mathcal{C}((b + c) \times a, x) \cong \mathcal{C}(b \times a + c \times a, x)$$

Левая часть — это отображение произведения, поэтому можно применить к нему каррирующее сопряжение:

$$\mathcal{C}((b + c) \times a, x) \cong \mathcal{C}(b + c, x^a)$$

Это дает отображение суммы, которое по сопряжению суммы изоморфно произведению двух отображений:

$$\mathcal{C}(b + c, x^a) \cong \mathcal{C}(b, x^a) \times \mathcal{C}(c, x^a)$$

Теперь можно применить к обоим компонентам действие, обратное каррирующему сопряжению:

$$\mathcal{C}(b, x^a) \times \mathcal{C}(c, x^a) \cong \mathcal{C}(b \times a, x) \times \mathcal{C}(c \times a, x)$$

Используя обращение сопряжения суммы, приходим к окончательному результату:

$$\mathcal{C}(b \times a, x) \times \mathcal{C}(c \times a, x) \cong \mathcal{C}(b \times a + c \times a, x)$$

Каждый шаг в этом доказательстве был естественным изоморфизмом, поэтому их композиция также является естественным изоморфизмом. Следовательно, по лемме Йонеды, два объекта, образующие левую и правую части дистрибутивного закона, изоморфны.

Гораздо более короткое доказательство этого утверждения следует из свойства левых сопряжений, которое мы вскоре обсудим.

10.3 Сопряжение между функторами

В общем, сопряжение связывает два функтора, идущих в противоположных направлениях между двумя категориями, левый функтор

$$L : \mathcal{D} \to \mathcal{C}$$

и правый функтор:

$$R : \mathcal{C} \to \mathcal{D}$$

Сопряжение $L \dashv R$ определяется как естественный изоморфизм между двумя hom-множествами.

$$\mathcal{C}(Lx, y) \cong \mathcal{D}(x, Ry)$$

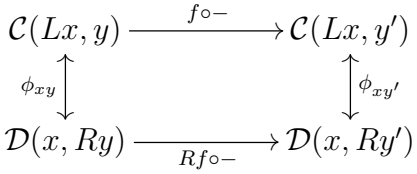
Другими словами, имеется семейство обратимых функций между множествами:

$$\phi_{xy} : \mathcal{C}(Lx, y) \to \mathcal{D}(x, Ry)$$

естественных, как по x, так и по y. Например, естественность по y означает, что, для любой $f : y \to y'$, следующая диаграмма является коммутативной:

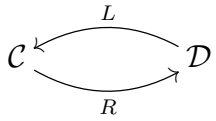
$$\begin{array}{ccc} \mathcal{C}(Lx, y) & \xrightarrow{\mathcal{C}(Lx, f)} & \mathcal{C}(Lx, y') \\ \Big\updownarrow{\scriptstyle \phi_{xy}} & & \Big\updownarrow{\scriptstyle \phi_{xy'}} \\ \mathcal{D}(x, Ry) & \xrightarrow[\mathcal{D}(x, Rf)]{} & \mathcal{D}(x, Ry') \end{array}$$

или, учитывая, что поднятие стрелок hom-функторами — это то же самое, что и пост-композиция:

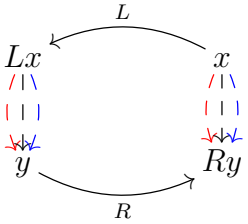


Двунаправленные стрелки можно перемещать в любом направлении (используя ϕ_{xy}^{-1}, при движении вверх), поскольку они являются компонентами изоморфизма.

Наглядно, это два функтора:



и, для любой пары x и y, два изоморфных hom-множества:



Эти hom-множества принадлежат к двум разным категориям, но множества — это просто множества. Говорят, что L является левым сопряженным к R, или что R является правым сопряженным к L.

На Haskell, упрощенная версия этого может быть закодирована как многопараметрический класс типа:

```
class (Functor left, Functor right) =>
                  Adjunction left right where
  ltor :: (left x -> y)  -> (x -> right y)
  rtol :: (x -> right y) -> (left x -> y)
```

Для этого требуется следующая прагма в верхней части программного файла:

```
{- # language MultiParamTypeClasses # -}
```

Следовательно, в бидекартовой категории сумма является левым сопряженным к диагональному функтору; а произведение — это правый сопряженный. Это можно записать очень лаконично:

$$(+) \dashv \Delta \dashv (\times)$$

Упражнение 10.3.1. *Изобразите коммутативный квадрат, свидетельствующий о естественности по x сопряженной функции ϕ_{xy}.*

Упражнение 10.3.2. hom-*множество $\mathcal{C}(Lx, y)$ в левой части формулы сопряжения предполагает, что Lx можно рассматривать как представляющий объект для некоторого функтора* (*ко-предпучка*). *Что это за функтор*? *Подсказка*: *он отображает y к множеству. Что это за множество*?

Упражнение 10.3.3. *И наоборот, представляющий объект a для предпучка P* определяется следующим образом:

$$Px \cong \mathcal{D}(x; a)$$

Что представляет собой предпучок, для которого Ry, в формуле сопряжения, является представляющим объектом?

10.4 Пределы и копределы как сопряжения

Определение предела также включает естественный изоморфизм между hom-множествами:

$$[\mathcal{J}, \mathcal{C}](\Delta_x, D) \cong \mathcal{C}(x, \mathrm{Lim} D)$$

hom-множество слева относится к категории функторов. Его элементами являются конусы или естественные преобразования между постоянным функтором и диаграммным функтором. То, что справа, является hom-множеством в $\mathcal{C}$.

В категории, где существуют все пределы, имеется сопряжение между этими двумя функторами:

$$\Delta_{(-)} : \mathcal{C} \to [\mathcal{J}, \mathcal{C}]$$
$$Lim_{(-)} : [\mathcal{J}, \mathcal{C}] \to \mathcal{C}$$

Двойственно, ко-предел описывается следующим естественным изоморфизмом:

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(\mathrm{Colim} D, x)$$

Оба сопряжения можно выразить, используя одно компактное выражение:

$$Colim \dashv \Delta \dashv Lim$$

В частности, поскольку категория произведений $\mathcal{C} \times \mathcal{C}$ эквивалентна $\mathcal{C}^2$, или категории функторов $[\mathbf{2}, \mathcal{C}]$, то можно переписать произведение и копроизведение как предел и ко-предел:

$$[\mathbf{2}, \mathcal{C}](\Delta_x, \langle a, b \rangle) \cong \mathcal{C}(x, a \times b)$$
$$\mathcal{C}(a + b, x) \cong [\mathbf{2}, \mathcal{C}](\langle a, b \rangle, \Delta_x)$$

где $\langle a, b \rangle$ обозначает диаграмму, являющуюся действием функтора $D : \mathbf{2} \to \mathcal{C}$ на двух объектах из $\mathbf{2}$.

10.5 Единица и коединица сопряжения

Мы сравниваем стрелки на равенство, но предпочитаем использовать изоморфизмы для сравнения объектов.

Однако, обнаруживается проблема, когда это касается функторов. С одной стороны, они являются объектами в категории функторов, поэтому изоморфизмы — это решение; с другой стороны, они — стрелки в **Cat**, так что, может быть, их можно сравнивать на равенство?

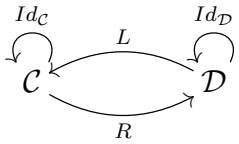
Чтобы прояснить эту ситуацию, мы должны спросить себя, *почему* для стрелок используется равенство. Не потому, что нам нравится равенство, а потому, что основное, что можно делать в множестве, содержащим повторяющиеся элементы, это сравнивать элементы на равенство. Тогда два элемента hom-множества либо равны, либо нет.

Это не относится к **Cat**, которая, как мы знаем, является 2-категорией. Здесь, сами hom-множества имеют структуру категории — категории функторов. В 2-категории имеются стрелки между стрелками, поэтому, в частности, можно определить изоморфизмы между стрелками. В **Cat** это будут естественные изоморфизмы между функторами.

Однако, хотя у нас есть возможность заменить равенства стрелок изоморфизмами, категорные законы в **Cat** по-прежнему выражаются

в виде равенств. Например, композиция функтора F с тождественным функтором *равна* F, и то же самое касается ассоциативности. 2-категория, в которой законы выполняются непременно, называется *строгой*, и $\mathbf{Cat}$ является примером строгой 2-категории.

Но, что касается сравнения категорий, здесь имеется больше возможностей. Категории являются объектами в $\mathbf{Cat}$, поэтому можно определить изоморфизм категорий как пару функторов L и R:



таких, что:

$$L \circ R = Id_{\mathcal{C}}$$
$$Id_{\mathcal{D}} = R \circ L$$

Однако, это определение включает равенство функторов. Что еще хуже, воздействие этих функторов на объекты предполагает равенство объектов:

$$L(Rx) = x$$
$$y = R(Ly)$$

Поэтому правильнее говорить о более слабом понятии *эквивалентности* категорий, где равенства заменены изоморфизмами:

$$L \circ R \cong Id_{\mathcal{C}}$$
$$Id_{\mathcal{D}} \cong R \circ L$$

На объектах эквивалентность категорий означает, что при обходе создается объект, изоморфный, а не равный исходному. В большинстве случаев это именно то, что нам нужно.

Сопряжение также определяется как пара функторов, идущих в противоположных направлениях, поэтому имеет смысл поинтересоваться, каков результат обхода туда и обратно. Изоморфизм, определяющий сопряжение, работает для любой пары объектов x и y

$$\mathcal{C}(Lx, y) \cong \mathcal{D}(x, Ry)$$

так что, в частности, можно заменить y на Lx

$$\mathcal{C}(Lx, Lx) \cong \mathcal{D}(x, R(Lx))$$

Теперь мы можем использовать трюк Йонеды и выбрать тождественную стрелку id_{Lx} слева. Изоморфизм отображает ее в единственную стрелку справа, которую обозначим η_x:

$$\eta_x : x \to R(Lx)$$

Это отображение не только определено для каждого x, но и естественно по x. Естественное преобразование η называется *единицей* сопряжения. Если заметить, что x слева является действием тождественного функтора на x, то можно записать:

$$\eta : Id_{\mathcal{D}} \to R \circ L$$

В качестве примера, оценим единицу сопряжения копроизведения:

$$C(a + b, x) \cong (C \times C)(\langle a, b\rangle, \Delta x)$$

заменив x на $a + b$. Плучаем:

$$\eta_{\langle a,b\rangle} : \langle a, b\rangle \to \Delta(a + b)$$

Это пара стрелок, которые точно соответствуют двум инъекциям $\langle \mathrm{Left}, \mathrm{Right}\rangle$.

Можно проделать аналогичный трюк, заменив x на Ry:

$$\mathcal{C}(L(Ry), y) \cong \mathcal{D}(Ry, Ry)$$

Соответствуя id_{R_y} справа, получаем семейство стрелок слева:

$$\varepsilon_y : L(Ry) \to y$$

которые образуют другое естественное преобразование, называемое *коединицей* сопряжения:

$$\varepsilon : L \circ R \to Id_{\mathcal{C}}$$

Заметим, что если бы эти два естественных преобразования были обратимы, они свидетельствовали бы об эквивалентности категорий. Но такая «полуэквивалентность» еще более интересна в контексте теории категорий.

В качестве примера, оценим ко-единицу сопряжения произведения:

$$(\mathcal{C} \times \mathcal{C})(\Delta x, \langle a, b\rangle) \cong \mathcal{C}(x, a \times b)$$

заменив x на $a \times b$. Получаем:

$$\varepsilon_{\langle a,b\rangle} : \Delta(a \times b) \to \langle a, b\rangle$$

Это пара стрелок, которые точно соответствуют двум проекциям $\langle \mathrm{fst}, \mathrm{snd}\rangle$.

Упражнение 10.5.1. *Выведите ко-единицу сопряжения копроизведения и единицу сопряжения произведения.*

Тождества треугольника

Мы можем использовать пару единица/ко-единица для формулировки эквивалентного определения сопряжения. Для этого, начинаем с пары естественных преобразований:

$$\eta : Id_{\mathcal{D}} \to R \circ L$$
$$\varepsilon : L \circ R \to Id_{\mathcal{C}}$$

и приложим дополнительные *тождества треугольника*.

Такие тождества могут быть получены из стандартного определения сопряжения, если заметить, что η можно использовать для замены тождественного функтора составным функтором $R \circ L$, что фактически позволяет вставлять $R \circ L$ везде, где работает тождественный функтор.

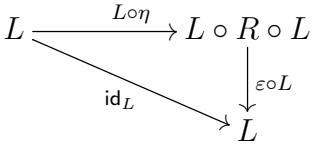
Подобным образом, ε можно использовать для устранения составного функтора $L \circ R$ (т.е. заменой его на тождественность).

Начнем с L:

$$L = L \circ Id_{\mathcal{D}} \xrightarrow{L\circ\eta} L \circ R \circ L \xrightarrow{\varepsilon\circ L} Id_{\mathcal{C}} \circ L = L$$

Здесь мы использовали горизонтальную композицию естественного преобразования, содержащую тождественное преобразование (известное как вискеринг).

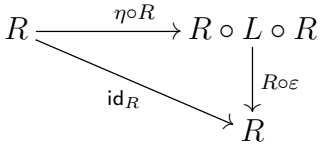
Первое тождество треугольника является условием того, что эта цепочка преобразований приводит к тождественному естественному преобразованию. Наглядно:



Аналогично, мы хотим, чтобы следующая цепочка естественных преобразований также компоновалась к тождеству:

$$R = Id_{\mathcal{D}} \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ Id_{\mathcal{C}} = R$$

или, схематично:



Оказывается, сопряжение можно альтернативно определить с помощью двух естественных преобразований, η и ε, удовлетворяющих тождествам треугольника:

$$(\varepsilon \circ L) \cdot (L \circ \eta) = \mathsf{id}_L$$
$$(R \circ \varepsilon) \cdot (\eta \circ R) = \mathsf{id}_R$$

Отсюда, отображение hom-множеств легко восстанавливается. Например, начнем со стрелки $f : x \to Ry$, которая является элементом $\mathcal{D}(x, Ry)$. Можно поднять ее до

$$Lf : Lx \to L(Ry)$$

Затем можно использовать η, чтобы свернуть составной функтор $L \circ R$ в тождество. Результатом является стрелка $Lx \to y$, которая является элементом из $\mathcal{C}(Lx, y)$.

Определение сопряжения с использованием единицы и ко-единицы является более общим в том смысле, что его можно перевести в настройку произвольной 2-категории.

Упражнение 10.5.2. *Для стрелки $g : Lx \to y$ реализуйте стрелку $x \to Ry$, используя ε, и тот факт, что R является функтором. Подсказка*: *начните с объекта x и подумайте, как можно добраться от него к Ry через один промежуточный объект.*

Единица и коединица каррированного сопряжения

Найдем единицу и ко-единицу каррированного сопряжения:

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, b^a)$$

Если заменить b на $e \times a$, то получим

$$\mathcal{C}(e \times a, e \times a) \cong \mathcal{C}(e, (e \times a)^a)$$

В соответствии с тождественной стрелкой слева, получаем единицу сопряжения справа:

$$\eta : e \to (e \times a)^a$$

Это является каррированной версией конструктора произведения, что на Haskell будет выглядеть так:

```
unit :: e -> (a -> (e, a))
unit  = curry id
```

Ко-единица более интересна. Заменяя e на b^a , получим:

$$\mathcal{C}(b^a \times a, b) \cong \mathcal{C}(b^a, b^a)$$

В соответствии с тождественной стрелкой справа, имеем:

$$\varepsilon : b^a \times a \to b$$

что является стрелкой применения функции.

На Haskell:

```
counit :: (a -> b, a) -> b
counit  = uncurry id
```

Когда сопряжение относится к двум эндофункторам, можно записать его альтернативное определение на Haskell, используя `unit` и `counit`:

```
class (Functor left, Functor right) =>
  Adjunction left right | left  -> right,
                          right -> left where
    unit   :: x -> right (left x)
    counit :: left (right x) -> x
```

Два дополнительных выражения, `left -> right` и `right -> left`, сообщают компилятору, что при использовании экземпляра сопряжения, один функтор может быть получен из другого. Это определение требует следующих расширений для компиляции:

```
{- # language MultiParamTypeClasses # -}
{- # LANGUAGE FunctionalDependencies # -}
```

Два функтора, образующие каррированное сопряжение, могут быть записаны в виде:

```
data L r x = L (x, r)   deriving (Functor, Show)
data R r x = R (r -> x) deriving Functor
```

а экземпляр `Adjunction` для каррирования:

```
instance Adjunction (L r) (R r) where
  unit   x            = R (\r -> L (x, r))
  counit (L (R f, r)) = f r
```

Первое тождество треугольника утверждает, что следующая полиморфная функция:

```
triangle :: L r x -> L r x
triangle  = counit . fmap unit
```

является тождественностью, как и вторая функция:

```
triangle' :: R r x -> R r x
triangle'  = fmap counit . unit
```

Отметим, что эти две функции требуют четкого определения использования функциональных зависимостей. Тождества треугольников не могут быть выражены в Haskell, так что доказывать их должен разработчик сопряжения.

Упражнение 10.5.3. *Протестируйте несколько примеров тождества первого треугольника для каррированного сопряжения. Для примера*:

```
triangle (L (2, 'a'))
```

Упражнение 10.5.4. *Как можно было бы проверить тождество второго треугольника для каррированного сопряжения? Подсказка: результат для* `triangle'` *— это функция, поэтому ее нельзя изобразить, но можно вызвать.*

10.6 Сопряжения с использованием универсальных стрелок

Мы рассмотрели определение сопряжения с использованием изоморфизма hom-множеств, и другое — с помощью пары единица/ко-единица. Оказывается, можно определить сопряжение, используя только один элемент этой пары, если он удовлетворяет некоторому условию универсальности. Чтобы убедиться в этом, создадим новую категорию, объектами которой будут стрелки.

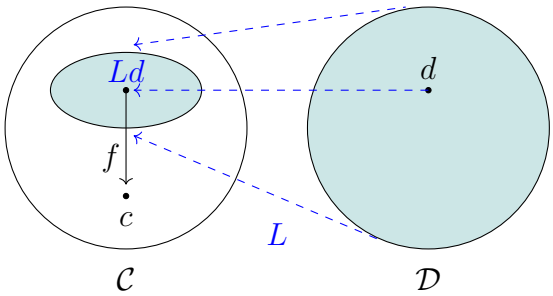
Мы рассматривали пример такой категории — категорию среза $\mathcal{C}/c$, которая собирает все стрелки, сходящиеся к c. Такая категория описывает представление объекта c со всех возможных сторон в $\mathcal{C}$.

Относительная категория

При работе с сопряжением

$$\mathcal{C}(Ld, c) \cong \mathcal{D}(d, Rc)$$

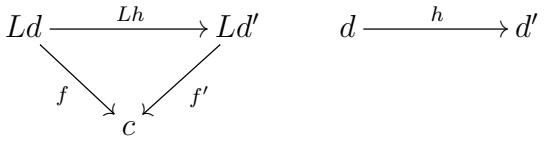
мы наблюдаем за объектом c с более узкой точки зрения, определяемой функтором L. Можно представлять себе L как определение модели категории $\mathcal{D}$ внутри $\mathcal{C}$. Нас интересует взгляд на c с точки зрения этой модели. Стрелки, описывающие такое представление, образуют относительную категорию (или, категорию запятой) L/c.



Объект в *относительной категории* L/c является парой $\langle d, f\rangle$, где d — объект из $\mathcal{D}$, а $f : Ld \to c$ — стрелка в $\mathcal{C}$.

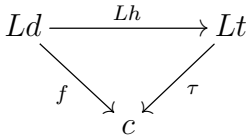
Морфизм от $\langle d, f\rangle$ к $\langle d', f'\rangle$ — это стрелка $h : d \to d'$, которая делает

коммутативной диаграмму слева:



Универсальная стрелка

Универсальная стрелка от L к c определяется как терминальный объект в относительной категории L/c. Раскроем это определение. Терминальный объект — это пара $\langle t, \tau \rangle$ с единственным морфизмом от любого объекта $\langle d, f \rangle$. Такой морфизм представляет собой стрелку $h : d \to t$, удовлетворяющую условию коммутативности:



Другими словами, для любой f, в hom-множестве $\mathcal{C}(Ld, c)$ существует единственный элемент h в hom-множестве $\mathcal{D}(d, t)$, такой, что:

$$f = \tau \circ Lh$$

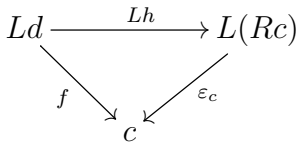
Такое взаимно-однозначное отображение между элементами двух hom-множеств намекает на лежащее в основе сопряжение.

Универсальные стрелки из сопряжений

Сначала убедимся, что если функтор L имеет правый сопряженный R, то для каждого c существует универсальная стрелка от L к c. Действительно, эта стрелка задается парой $\langle Rc, \varepsilon_c \rangle$, где ε — ко-единица сопряжения. Прежде всего, компонент ко-единицы имеет правильную сигнатуру для этого объекта в относительной категории L/c:

$$\varepsilon_c : L(Rc) \to c$$

Надо показать, что $\langle Rc, \varepsilon_c \rangle$ — это терминальный объект в L/c. То есть, для любого объекта $\langle d, f : Ld \to c \rangle$ существует единственная $h : d \to Rc$ такая, что $f = \varepsilon_c \circ Lh$:



Чтобы доказать это, запишем одно из условий естественности для ϕ_{dc}, как функции для d:

$$\phi_{dc} : \mathcal{C}(Ld, c) \to \mathcal{D}(d, Rc)$$

Для любой стрелки $h : d \to d'$ следующая диаграмма должна быть коммутативной:

$$\begin{array}{ccc} \mathcal{C}(Ld', c) & \xrightarrow{-\circ Lh} & \mathcal{C}(Ld, c) \\ \uparrow{\scriptstyle \phi_{d'c}} & & \uparrow{\scriptstyle \phi_{dc}} \\ \mathcal{D}(d', Rc) & \xrightarrow[-\circ h]{} & \mathcal{D}(d, Rc) \end{array}$$

Теперь можно использовать трюк Йонеды, установив d' равным Rc.

$$\begin{array}{ccc} \mathcal{C}(L(Rc), c) & \xrightarrow{-\circ Lh} & \mathcal{C}(Ld, c) \\ \uparrow{\scriptstyle \phi_{Rcc}} & & \uparrow{\scriptstyle \phi_{dc}} \\ \mathcal{D}(Rc, Rc) & \xrightarrow[-\circ h]{} & \mathcal{D}(d, Rc) \end{array}$$

Можно выбрать специальный элемент hom-множества $\mathcal{D}(Rc, Rc)$, а именно тождественную стрелку id_{Rc}, и распространить ее по остальной части диаграммы. Верхний левый угол становится ε_c, нижний правый угол становится h, а верхний правый угол становится сопряженным к h, который обозначен через f:

$$\begin{array}{ccc} \varepsilon_c & \xmapsto{-\circ Lh} & f \\ \uparrow{\scriptstyle \phi_{Rcc}} & & \uparrow{\scriptstyle \phi_{dc}} \\ \mathsf{id}_{Rc} & \xmapsto[-\circ h]{} & h \end{array}$$

Тогда верхняя стрелка дает искомое равенство $f = \varepsilon_c \circ Lh$.

Сопряжение из универсальных стрелок

Обратный результат еще интереснее. Если для каждого c имеется универсальная стрелка от L к c, то есть, терминальный объект $\langle t_c, \varepsilon_c \rangle$ в относительной категории L/c, то можно построить функтор R, который является правым сопряженным к L. Действие этого функтора на объекты задается формулой $Rc = t_c$, а семейство ε_c автоматически естественно в c и это образует ко-единицу сопряжения.

Существует также двойственное утверждение: сопряжение можно построить, исходя из семейства универсальных стрелок η_d, образующих инициальные объекты в относительной категории d/R.

Эти результаты помогут нам доказать теорему Фрейда о сопряженном функторе.

10.7 Свойства сопряжений

Левые сопряженные сохраняют ко-пределы

Ко-пределы были определены как универсальные ко-конусы. Для каждого ко-конуса — это естественное преобразование диаграммы $D : \mathcal{J} \to \mathcal{C}$ к постоянному функторуΔ_x — существует единственный факторизующий морфизм от ко-предела $\mathrm{Colim}D$ к x. Это условие можно записать как взаимно однозначное соответствие между множеством ко-конусов и $\hom$-множеством:

$$[\mathcal{J}, \mathcal{C}](D, \Delta_x) \cong \mathcal{C}(\mathrm{Colim}D, x)$$

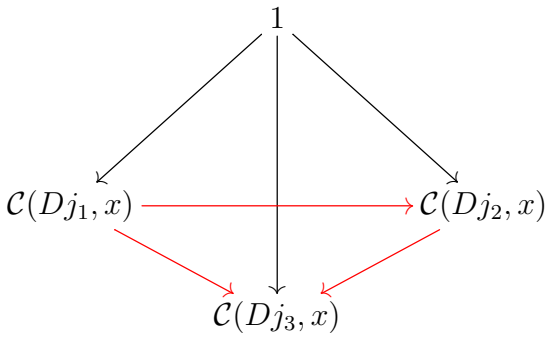
Условие факторизации закодировано в естественности этого изоморфизма.

Оказывается, множество ко-конусов, которое, само по себе, — объект из **Set**, является пределом следующего **Set**-значного функтора $F : \mathcal{J} \to$ **Set**:

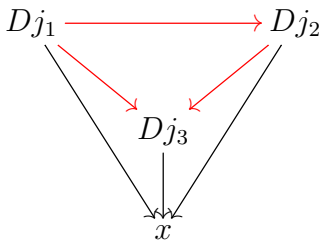
$$Fj = \mathcal{C}(Dj, x)$$

Чтобы показать это, начнем с предела F и закончим множеством ко-конусов. Напомним, что предел **Set**-значного функтора есть множество конусов с вершиной 1 (одноэлементному множеству). В нашем случае, каждый такой конус описывает выбор морфизмов из соответствующего

hom-множества $\mathcal{C}(Dj, x)$:



Каждый из этих морфизмов имеет целью один и тот же объект x, так что они образуют стороны ко-конуса с вершиной x.



Условия коммутации для конуса с вершиной 1 являются одновременно условиями коммутации для этого ко-конуса с вершиной x. А ведь, это именно ко-конусы в множестве $[\mathcal{J}, \mathcal{C}](D, \Delta_x)$.

Поэтому можно заменить исходный набор ко-конусов пределом $\mathcal{C}(\mathcal{D}-, x)$, получая:

$$\mathrm{Lim}\,\mathcal{C}(D-, x) \cong \mathcal{C}(\mathrm{Colim} D, x)$$

Предел (контравариантного) hom-функтора, действующего на диаграмме D, изоморфен hom-функтору, действующему на ко-пределе этой диаграммы. Это обычно сокращается до: hom-функтор сохраняет ко-пределы.

Функтор, сохраняющий ко-пределы, называется ко-непрерывным. Таким образом, контравариантный hom-функтор ко-непрерывен.

Предположим теперь, что имеется сопряжение $L \dashv R$, где $L : \mathcal{C} \to \mathcal{D}$, а R идет в обратном направлении. Надо показать, что левый функтор L сохраняет ко-пределы, то есть:

$$L(\mathrm{Colim} D) \cong \mathrm{Colim}(L \circ D)$$

для любой диаграммы $D : \mathcal{J} \to \mathcal{C}$, для которой существует ко-предел.

Воспользуемся леммой Йонеды, чтобы показать, что отображения-вне с обеих сторон к произвольному x изоморфны:

$$\mathcal{D}(L(\mathrm{Colim}D), x) \cong \mathcal{D}(\mathrm{Colim}(L \circ D), x)$$

Применим указанное сопряжение к левой части, получая:

$$\mathcal{D}(L(\mathrm{Colim}D), x) \cong \mathcal{C}(\mathrm{Colim}D, Rx)$$

Сохранение ко-пределов hom-функтором дает:

$$\cong \mathrm{Lim}\,\mathcal{C}(D-, Rx)$$

Снова используя сопряжение, получаем

$$\cong \mathrm{Lim}\,\mathcal{D}((L \circ D)-, x)$$

А второе применение сохранения ко-пределов приводит к желаемому результату:

$$\cong \mathcal{D}(\mathrm{Colim}(L \circ D), x)$$

Теперь можно использовать этот результат, чтобы переформулировать наше предыдущее доказательство дистрибутивности в декартово замкнутой категории. Воспользуемся тем, что произведение является левым сопряжением экспоненциала. Левые сопряженные сохраняют ко-пределы. Ко-произведение является ко-пределом, поэтому:

$$(b + c) \times a \cong b \times a + c \times a$$

Здесь, левый функтор есть $Lx = x \times a$, а диаграмма D выбирает пару объектов b и c.

Правые сопряженные сохраняют пределы

Используя двойственный аргумент, можно показать, что правые сопряженные сохраняют пределы, то есть:

$$R(\mathrm{Lim}D) \cong \mathrm{Lim}(R \circ D)$$

Начнем с демонстрации того, что (ковариантный) hom-функтор сохраняет пределы.

$$\mathrm{Lim}\,\mathcal{C}(x, D-) \cong \mathcal{C}(x, \mathrm{Lim}D)$$

Это следует из аргументации: множество конусов, определяющее предел, изоморфно пределу **Set**-значного функтора.

$$Fj = \mathcal{C}(x, Dj)$$

Функтор, сохраняющий пределы, называется непрерывным.

Чтобы показать, что, при наличии сопряжения $L \dashv R$, правый функтор $R : \mathcal{D} \to \mathcal{C}$ сохраняет пределы, воспользуемся аргументом Йонеды:

$$\mathcal{C}(x, R(\mathrm{Lim}D)) \cong \mathcal{C}(x, \mathrm{Lim}(R \circ D))$$

Действительно, имеем

$$\mathcal{C}(x, R(\mathrm{Lim}D)) \cong \mathcal{D}(Lx, \mathrm{Lim}D) \cong \mathrm{Lim}\mathcal{D}(Lx, D-) \cong \mathcal{C}(x, \mathrm{Lim}(R \circ D))$$

10.8 Теорема Фрейда о сопряженном функторе

В общем случае функторы допускают потери, т.е. являются необратимыми. В некоторых случаях можно восполнить потерянную информацию, заменой ее «наилучшей догадкой». Если делать это целенаправленно, то можно завершить эту деятельность получением сопряжения. Вопрос в следующем: если задан функтор между двумя категориями, то каковы условия, при которых можно построить его сопряженный.

Ответ на этот вопрос дает теорема Фрейда о сопряженном функторе. Сначала может показаться, что это техническая теорема, включающая очень абстрактную конструкцию, называемую условием множества решений. Позже мы увидим, что это условие напрямую переводится в технику программирования, называемую дефункциализацией.

В дальнейшем мы сосредоточим внимание на построении сопряженного справа функтора $L : \mathcal{D} \to \mathcal{C}$. Двойственное рассуждение может быть использовано для решения обратной задачи нахождения сопряженного слева функтора $R : \mathcal{C} \to \mathcal{D}$.

Первое наблюдение состоит в том, что, поскольку левый функтор в сопряжении сохраняет ко-пределы, мы должны предположить, что наш функтор L сохраняет ко-пределы. Это дает намек на то, что построение правого сопряженного зависит от возможности построения ко-пределов в

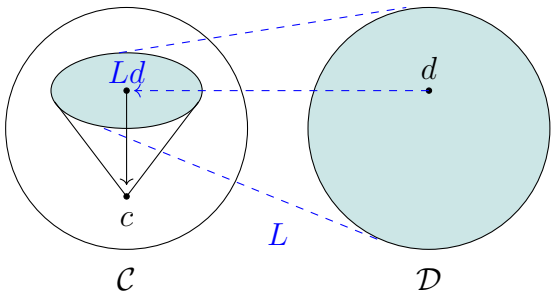
$\mathcal{D}$ и возможности каким-то образом перенести их обратно в $\mathcal{C}$, используя L.

Можно было бы потребовать, чтобы в $\mathcal{D}$ существовали все ко-пределы, большие и малые, но это условие слишком сильное. Даже небольшая категория со всеми ко-пределами автоматически является предпорядком, то есть, в ней не может быть более одного морфизма между любыми двумя объектами.

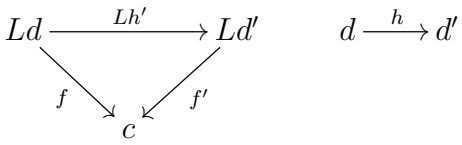
Но давайте, чуть отвлечемся от проблем с размером и посмотрим, как можно построить правый сопряженный функтор L, сохраняющий ко-пределы, чья исходная категория $\mathcal{D}$ является малой и имеет все ко-пределы, большие и малые (таким образом, это предпорядок).

Теорема Фрейда в предпорядке

Самый простой способ определить правый сопряженный к L — построить, для каждого объекта c, универсальную стрелку от L к c. Такая стрелка является терминальным объектом в относительной категории L/c — категории стрелок, сходящихся к объекту c и исходящих от образа L.

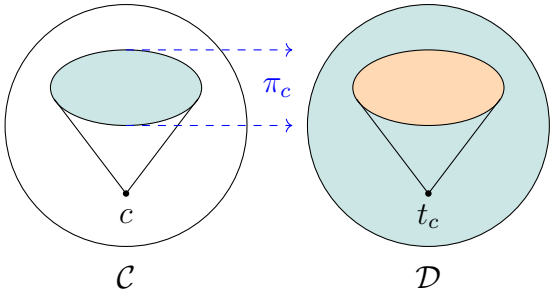


Важным наблюдением является то, что эта относительная категория описывает ко-конус в $\mathcal{C}$. Основание этого ко-конуса формируется теми объектами в образе L, которые имеют беспрепятственный обзор c. Стрелки в основании ко-конуса — это морфизмы в L/c. Это именно те стрелки, которые заставляют стороны ко-конуса коммутировать.



Основание этого ко-конуса можно спроецировать обратно в $\mathcal{D}$. Существует проекция π_c, которая отображает каждую пару (d, f) в L/c обратно к d, забывая, таким образом, о стрелке f. Она также отображает каждый морфизм из L/c к стрелке из $\mathcal{D}$, которая его породила. Таким образом π_c определяет диаграмму в $\mathcal{D}$. Ко-предел этой диаграммы существует, потому что мы предположили, что в $\mathcal{D}$ существуют все ко-пределы. Обозначим этот ко-предел как t_c:

$$t_c = \operatorname{colim} \pi_c$$



Подумаем, можно ли использовать этот t_c для построения терминального объекта в L/c. Нужно найти стрелку, обозначим ее $\varepsilon_c : Lt_c \to c$, такую, чтобы пара $\langle t_c, \varepsilon_c \rangle$ была терминальной в L/c.

Отметим, что L отображает диаграмму, сгенерированную π_c, обратно к основанию ко-конуса, определенного с помощью L/c. Проекция π_c игнорировала лишь стороны этого ко-конуса, оставляя его основание нетронутым.

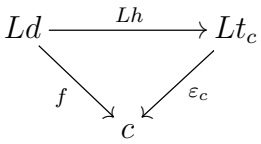
Так что, теперь имеем два ко-конуса в $\mathcal{C}$ с одинаковым основанием: исходный, с вершиной c, и новый, полученный применением L к ко-конусу в $\mathcal{D}$. Поскольку L сохраняет ко-пределы, ко-предел нового ко-конуса есть Lt_c — образ ко-предела t_c:

$$\operatorname{colim} (L \circ \pi_c) = L(\operatorname{colim} \pi_c) = Lt_c$$

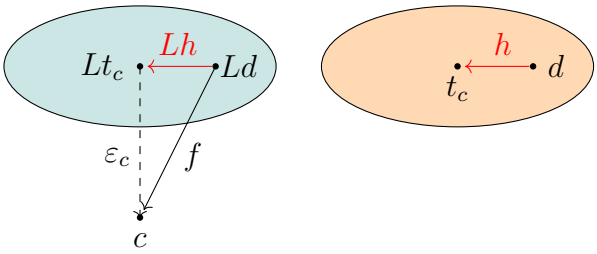
Следуя универсальной конструкции, заключаем, что должен существовать единственный ко-конусный морфизм от ко-предела Lt_c к c. Этот морфизм, который будем обозначать ε_c, делает коммутативными соответствующие треугольники.

Остается показать, что $\langle t_c, \varepsilon_c \rangle$ является терминальной в L/c, то есть для любой $< d, f : Ld \to c >$ существует единственный морфизм h :

$d \to t_c$ относительной категории, который делает следующий треугольник коммутативным:



Отметим, что любой такой d автоматически является частью диаграммы, создаваемой π_c (это результат действия π_c на $\langle d, f \rangle$). Мы знаем, что t_c — предел этой диаграммы. Значит, в ограничивающем ко-конусе должна быть струна от d к t_c. Мы выбираем эту струну в качестве h.



Тогда, условие коммутативности следует из того, что ε_c — морфизм ко-конуса. Такой морфизм является единственным просто потому, что $\mathcal{D}$ — предпорядок.

Это доказывает, что существует универсальная стрелка $\langle t_c, \varepsilon_c \rangle$ для каждого c, поэтому имеем функтор R, определенный на объектах как $Rc = t_c$, который является правым сопряженным к L.

Условие множества решений

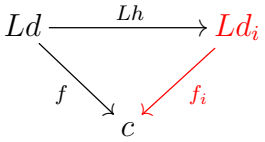
Проблема с предыдущим доказательством заключается в том, что относительные категории, в большинстве практических случаев, велики: их объекты не образуют множества. Но, может быть, можно аппроксимировать относительную категорию, выбрав более меньшее, но репрезентативное множество объектов и стрелок?

Для выбора объектов мы использовали бы отображение от некоторого индексирующего множества I. Определим множество объектов d_i, где $i \in I$. Поскольку мы пытаемся аппроксимировать относительную категорию L/c, то выбираем объекты вместе со стрелками $f_i : Ld_i \to c$.

Соответствующая часть относительной категории была закодирована в морфизме между объектами, удовлетворяющими условию комму-

татирования. Можно было бы попытаться применить это условие только внутри нашего семейства объектов, но этого было бы недостаточно. Надо найти способ исследования всех других объектов относительной категории.

Для этого реинтерпретируем условие коммутатирования согласно рецепту факторизации произвольной $f : Ld \to c$ через некоторую пару $\langle d_i, f_i \rangle$:



Множество решений — это семейство пар $\langle d_i, f_i : Ld_i \to c_i \rangle$, проиндексированное множеством I, которое можно использовать для факторизации любой пары $\langle d, f : Ld \to c_i \rangle$. Это означает, что существует индекс $i \in I$ и стрелка $h : d \to d_i$, факторизующая f:

$$f = f_i \circ Lh$$

Другой способ выразить это свойство состоит в том, чтобы сказать, что существует *слабо терминальное множество* объектов в относительной категории L/c. Слабо терминальное множество обладает тем качеством, что для любого объекта категории существует морфизм хотя бы к одному объекту этого множества.

Ранее мы видели, что наличие терминального объекта в относительной категории L/c, для каждого c, достаточно для определения сопряжения. Оказывается, можно достичь той же цели, используя множество решений.

Предположения теоремы Фрейда о сопряженном функторе утверждают, что имеется сохраняющий ко-пределы функтор $L : D \to C$ от малой ко-полной категории. Оба эти условия относятся к малым диаграммам. Если можно выбрать множество решений $\langle d_i, f_i : Ld_i \to c_i \rangle$, для каждого c, то правый сопряженный R существует. Множества решений для разных c могут быть различными.

Мы уже видели, что в ко-полной категории существования слабо терминального множества достаточно, чтобы определить терминальный объект. В нашем случае это означает, что для любого c можно построить универсальную стрелку от L к c. И этого достаточно для определения всего сопряжения.

Двойственная версия теоремы о сопряженном функторе может быть использована для построения левого сопряженного.

Дефункциализация

Каждый язык программирования позволяет определять функции, но не все языки поддерживают функции высших порядков (функции, принимающие функции в качестве аргументов или возвращающие функции) или анонимные функции (или безымянные функции, также известные как лямбда-выражения). Оказывается, даже в таких языках функции высших порядков могут быть реализованы с помощью процесса, называемого *дефункциализацией*. Этот метод основан на теореме о сопряженном функторе. Более того, дефункциализацию можно использовать всякий раз, когда передача функций нецелесообразна, например, в распределенных системах.

Идея дефункциализации заключается в том, что функциональный тип определяется как правый сопряженный к произведению.

$$\mathcal{C}(e \times a, b) \cong \mathcal{C}(e, b^a)$$

Теорему о сопряженном функторе можно использовать для аппроксимации этого сопряженного.

Вообще, любая конечная программа может иметь только конечное число определений функций. Эти функции (вместе с захватываемой ими средой) образуют множество решений, которое можно использовать для построения типа функции. На практике это делается только для небольшого подмножества функций, которые встречаются в качестве аргументов или возвращаются из других функций.

Типичным примером использования функций высшего порядка является стиль передачи продолжения. Приведем пример функции, которая вычисляет сумму элементов списка, но вместо возврата суммы она вызывает продолжение `k`:

```
sumK            :: [Int] -> (Int -> r) -> r
sumK [] k        = k 0
sumK (i : is) k = sumK is (\s -> k (i + s))
```

Если список пуст, то эта функция вызывает продолжение с нулем. В противном случае она вызывает себя рекурсивно, с двумя аргументами, хвостом списка `is` и новым продолжением:

```
\s -> k (i + s)
```

Это новое продолжение вызывает предыдущее продолжение `k`, передавая ему сумму заголовка списка и его аргумента `s` (который является накопленной суммой).

Обратите внимание, что эта лямбда является замыканием: это функция одной переменной `s`, но она также имеет доступ к `k` и `i` из своего окружения.

Чтобы извлечь окончательную сумму, вызывается определенная рекурсивная функция с тривиальным продолжением, тождественностью:

```
sumList    :: [Int] -> Int
sumList as = sumK as (\i -> i)
```

Анонимные функции удобны, но ничто не мешает использовать именованные функции. Однако, если требуется исключить продолжения, требуется четко указать передачу в окружениях. Например, можно заменить первую лямбду на функцию `more`, но необходимо явно передать ей окружение типа `(Int, Int -> r)`:

```
more          :: (Int, Int -> r) -> Int -> r
more (i, k) s = k (i + s)
```

Другая лямбда, тождественность, использует пустое окружение:

```
done  :: Int -> Int
done i = i
```

Реализация нашего алгоритма с использованием именованных функций:

```
sumK'            :: [Int] -> (Int -> r) -> r
sumK' []        k = k 0
sumK' (i : is) k = sumK' is (more (i, k))

sumList          :: [Int] -> Int
sumList is        = sumK' is done
```

На самом деле, если все, что нас интересует, это вычисление суммы, то можно заменить полиморфный тип `r` на `Int` без каких-либо других изменений.

Эта реализация по-прежнему использует функции высшего порядка. Чтобы устранить их, надо проанализировать, что означает передача функции в качестве аргумента. Такую функцию можно использовать только одним способом: ее можно применить к ее аргументам. Это свойство функционального типа выражается как часть каррирующего сопряжения:

$$\varepsilon : b^a \times a \to b$$

или, на Haskell, как функция высшего порядка:

```
apply :: (a -> b, a) -> b
```

Теперь нас интересует построение ко-единицы из первооснов. Мы видели, что это можно сделать с помощью относительной категории. В нашем случае объектом такой категории для функтора произведения $L_a = (-) \times a$ является пара

$$(e, f : (e \times a) \to b)$$

или, на Haskell:

```
data Comma a b e = Comma e ((e, a) -> b)
```

Морфизмом в этой категории между (e, f) и (e', f') является стрелка $h : e \to e'$, удовлетворяющая условию коммутативности:

$$f' \circ h = f$$

Мы интерпретируем этот морфизм как «сокращающий» окружение. Стрелка f' способна произвести тот же результат типа b, используя потенциально меньшую среду, заданную посредством $h(e)$. Например, e может содержать переменные, которые не имеют отношения к вычислению b из a, и h проецирует их вовне.

На самом деле мы выполнили такую редукцию при определении `more` и `done`. В принципе, можно было бы передать `is` обеим функциям, так как он доступен в точке вызова. Но мы знаем, что им это не нужно.

Формально можно было бы определить функциональный объект $a \to b$ как ко-предел диаграммы, определяемой относительной категорией. Такой ко-предел, по сути, является гигантским ко-произведением всех

сред по модулю идентификаций, заданных морфизмами относительной категории. Эта идентификация выполняет работу по сокращению среды, необходимой для $a \to b$, до абсолютного минимума.

В рассматриваемом примере, интересующие нас продолжения — это функции типа `Int -> Int`. На самом деле, нас не интересует генерация общего типа функции `Int -> Int`; только минимального, который вместил бы две функции, `more` и `done`. Мы можем сделать это, создав очень небольшое множество решений.

В нашем случае, множество решений состоит из пар $(e_i, f_i : L_a e_i \to b)$, таких, что любую пару $(e, f : L_a e \to b)$ можно разложить на множители, по одному от f_i. Точнее, интерес вызывают только два окружения, `(Int, Int -> Int)` для `more` и пустое окружение `()` для `done`.

В принципе, наше множество решений должно допускать факторизацию каждого объекта относительной категории, то есть пары типа:

```
(e, (e, Int) -> Int)
```

но здесь нас интересуют только две конкретные функции. Кроме того, нас не волнует единственность представления, поэтому вместо использования ко-предела (как это делалось для теоремы о сопряженном функторе) просто будем использовать копроизведение всех сред, вызывающих интерес. В итоге, получаем следующий тип данных, представляющий собой сумму двух интересующих нас сред:

```
data Kont = Done | More Int Kont
```

Отметим, что рекурсивно закодирована часть среды `Int -> Int` (та, которая была использована для больших потребностей) как `Kont`.

Если внимательно рассмотреть это определение, то можно обнаружить, что это определение списка `Int` по модулю некоторых переименований. Каждый вызов `More` помещает еще одно целое число в стек `Kont`. Эта интерпретация согласуется с нашей интуицией, согласно которой рекурсивные алгоритмы требуют своего рода стек времени выполнения.

Теперь мы готовы реализовать нашу аппроксимацию к ко-единице сопряжения. Она скомпонована из тел двух функций с учетом того, что рекурсивные вызовы также проходят через `apply`:

```
apply               :: (Kont, Int) -> Int
apply (Done, i)      = i
apply (More i k, s) = apply (k, i + s)
```

Теперь основной алгоритм можно переписать без каких-либо функций высшего порядка или лямбда-выражений:

```
sumK''            :: [Int] -> Kont -> Int
sumK'' [] k        = apply (k, 0)
sumK'' (i : is) k = sumK'' is (More i k)

sumList'' is       = sumK'' is Done
```

Основное преимущество дефункциализации заключается в том, что ее можно использовать в распределенных средах. Аргументы отдаленных функций, если они являются структурами данных, а не функциями, могут быть сериализованы и отправлены по сети. Все, что нужно, это чтобы получатель имел доступ к `apply`.

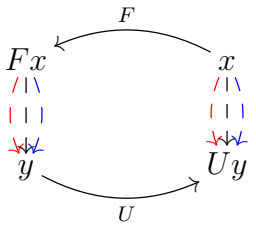
10.9 Свободные/забывающие сопряжения

Два функтора в сопряжении играют разные роли: картина сопряжения несимметрична. Нигде это не проиллюстрировано лучше, чем в случае свободного/забывающего сопряжения.

Забывающий функтор — это функтор, который «забывает» часть структуры своей исходной категории. Это не строгое определение, но в большинстве случаев совершенно очевидно, о какой забывающй структуре идет речь. Очень часто целевой категорией является именно категория множеств, которая считается воплощением бесструктурности. Результат забывающего функтора в этом случае называется «основным» множеством, а сам функтор часто обозначают U.

Точнее, говорят, что функтор забывает структуру, если отображение hom-множеств не является сюръективным, то есть в целевом hom-множестве имеются стрелки, которым нет соответствующих стрелок в исходном hom-множестве. Интуитивно это означает, что стрелки в источнике должны сохранять некоторую структуру, поэтому их меньше; и эта структура отсутствует в цели.

Сопряженный слева к забывающему функтору называется *свободным функтором*.

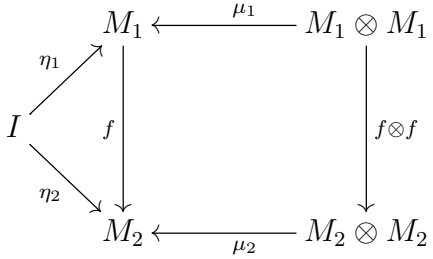


Классический пример свободного/забывающего сопряжения — конструкция свободного моноида.

Категория моноидов

Моноиды в моноидальной категории $\mathcal{C}$ образуют свою собственную категорию $\mathbf{Mon}(\mathcal{C})$. Ее объекты — моноиды, а стрелки — это стрелки из $\mathcal{C}$, сохраняющие моноидальную структуру.

Следующая диаграмма объясняет то, что f является моноидным морфизмом, идущим от моноида (M_1, η_1, μ_1) к моноиду (M_2, η_2, μ_2):



Моноидальный морфизм f должен отображать единицу в единицу, а это означает, что:

$$f \circ \eta_1 = \eta_2$$

и он должен отображать умножение к умножению:

$$f \circ \mu_1 = \mu_2 \circ (f \otimes f)$$

Запомните, что тензорное произведение $\otimes$ функториально, поэтому оно может поднимать пары стрелок, как в случае $f \otimes f$.

В частности, категория **Set** является моноидальной с декартовым произведением и терминальным объектом, обеспечивающими моноидальную структуру.

Моноиды в **Set** — это множества с дополнительной структурой. Они образуют свою собственную категорию $\mathbf{Mon}(\mathbf{Set})$, и существует забывающий функтор U, который просто отображает моноид в множество его элементов. Когда мы говорим, что моноид — это множество, имеется в виду основное множество.

Свободный моноид

Мы хотим построить свободный функтор

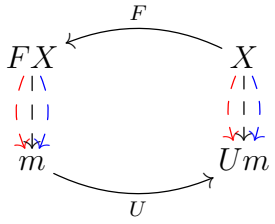
$$F : \mathbf{Set} \to \mathbf{Mon}(\mathbf{Set})$$

сопряженный с забывающим функтором U.

Начнем с произвольного множества X и произвольного моноида m. В правой части сопряжения находится множество функций между двумя множествами, X и Um, а в левой части — множество строго подчиненных ограничениям сохраняющих-структуру моноидных морфизмов от FX к m. Как эти два множества могут быть изоморфны?

В $\mathbf{Mon}(\mathbf{Set})$, моноиды — это просто множества элементов, а моноидный морфизм — это функция между такими множествами, удовлетворяющая дополнительным ограничениям: сохранение единицы и умножения.

С другой стороны, стрелки в **Set** — это просто функции без дополнительных ограничений. Таким образом, в общем случае между моноидами меньше стрелок, чем между их основными множествами.



Отсюда идея: если мы хотим иметь взаимно однозначное соответствие между стрелками, то требуется, чтобы FX было намного больше, чем X. Таким образом, будет намного больше функций от него к m —

так много, что даже после игнорирования тех функций, которые не сохраняют структуру, их все равно будет достаточно, чтобы задействовать каждую функцию $f : X \to Um$.

Мы построим моноид FX, начиная с множества X, и добавляя все больше и больше элементов. Назовем инициальное множество X *образующими* FX. Построим моноидный морфизм $g : FX \to m$, начиная с исходной функции f и расширяя ее на все большее количество элементов.

На образующих $x \in X$, g работает так же, как f:

$$gx = fx$$

Поскольку FX должен быть моноидом, он должен иметь единицу. Мы не можем выбрать один из образующих в качестве единицы, потому что это наложило бы ограничения на часть g, которая уже зафиксирована f — она должна была бы отобразить ее на единицу e' моноида m. Поэтому мы просто добавим дополнительный элемент e к FX и назовем его единицей. Мы определим действие g на нем, сказав, что оно отображается в единицу e' моноида m:

$$ge = e'$$

Также необходимо определить моноидальное умножение в FX. Начнем с произведения двух образующих a и b. Результат умножения не может быть другим образующим, потому что, опять же, это ограничило бы часть g, которая зафиксирована f — произведения должны быть отображены на произведения. Поэтому надо сделать все произведения образующих новыми элементами FX. Опять же, действие g на эти произведения фиксировано:

$$g(a \cdot b) = ga \cdot gb$$

Продолжая эту конструкцию, любое новое умножение производит новый элемент FX, за исключением случаев, когда его можно свести к существующему элементу, применяя законы моноида. Например, новая единица e, умноженная на образующую a, должна быть равна a. Но мы убедились, что e отображается к единице из m, так что произведение $ge \cdot ga$ автоматически равно ga.

Другой способ взглянуть на эту конструкцию — представлять множество X как алфавит. Тогда элементы FX представляют собой строки

символов из этого алфавита. Образующие представляют собой однобуквенные строки, «a», «b» и так далее. Единицей является пустая строка, «». Умножение — это конкатенация строк, поэтому «a», умноженное на «b», — это новая строка «ab». Конкатенация автоматически ассоциативна и унитальна, с пустой строкой в качестве единицы.

Интуиция, стоящая за свободными функторами, заключается в том, что они образуют структуру «свободно», то есть, «без дополнительных ограничений». К тому же, делают они это «лениво»: вместо того, чтобы выполнять операции, они их просто фиксируют. Они создают общие программы для конкретной предметной области, которые могут быть выполнены позже конкретными интерпретаторами.

Свободный моноид «запоминает умножение» позже. Он сохраняет аргументы умножения в строке, но не выполняет умножение. Разрешено только упростить свои записи на основе общих моноидальных законов. Например, ему не нужно хранить команду для умножения на единицу. Он также может «пропускать скобки», согласно ассоциативности.

Упражнение 10.9.1. *Каковы единица и ко-единица сопряжения свободного моноида* $F \dashv U$?

Свободный моноид в программировании

На Haskell моноиды определяются с использованием следующего класса типов:

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

Здесь, `mappend` — каррированная форма отображения от произведения: `(m, m) -> m`. Элемент `mempty` соответствует стрелке от терминального объекта (единицы моноидальной категории), или просто, и элементу `m`.

Свободный моноид, сгенерированный некоторым типом `a`, который служит множеством образующих, представлен типом списка `[a]`. Пустой список служит единицей; а умножение моноидов реализовано как конкатенация списков, традиционно записываемая в инфиксной форме:

```
(++)             :: [a] -> [a] -> [a]
(++) []  ys       = ys
(++) (x : xs) ys = x : xs ++ ys
```

Список является экземпляром `Monoid`:

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

Чтобы показать, что это свободный моноид, мы должны иметь возможность построить моноидный морфизм от списка `a` к произвольному моноиду `m`, при условии, что имеется (неограниченное) отображение от `a` к (основному множеству) `m`. Мы не можем выразить все это на Haskell, но можем определить функцию:

```
foldMap  :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

Эта функция преобразует элементы списка в моноидальные значения, используя `f`, а затем сворачивает их, используя `mappend`, начиная с единицы `mempty`.

Легко видеть, что пустой список отображается к моноидальной единице. Также ясно, что конкатенация двух списков отображается к моноидальному произведению результатов. Так что, действительно, `foldMap` есть моноидный морфизм.

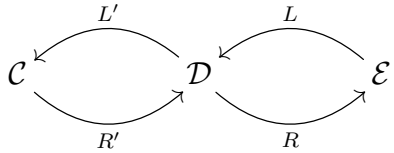
Следуя интуитивному пониманию того, что свободный моноид является программой для умножения, отражающей специфику предметной области, `foldMap` предоставляет *интерпретатор* для этой программы. Он выполняет все умножения, которые были отложены. Заметим, что одну и ту же программу можно интерпретировать по-разному, в зависимости от выбора конкретного моноида и функции `f`.

Мы вернемся к свободным моноидам в виде списков в главе об алгебрах.

Упражнение 10.9.2. *Напишите программу, которая принимает список целых чисел и интерпретирует его различными способами: с использованием аддитивного и с использованием мультипликативного моноида целых чисел.*

10.10 Категория сопряжений

Композицию сопряжений можно определить, воспользовавшись композицией определяющих их функторов. Два сопряжения, $L \dashv R$ и $L' \dashv R'$, компонуемы, если они содержат общую категорию посередине:



Компонуя функторы, мы получаем новое сопряжение $(L' \circ L) \dashv (R \circ R')$.

Действительно, рассмотрим hom-множество:

$$\mathcal{C}(L'(Le), c)$$

Используя сопряжение $L' \dashv R'$, можно транспонировать L' вправо, где оно становится R':

$$\mathcal{D}(Le, R'c)$$

а используя $L \dashv R$, можно аналогичным образом транспонировать L:

$$\mathcal{E}(e, R(R'c))$$

Комбинируя эти морфизмы, получаем сопряжение:

$$\mathcal{C}((L' \circ L)e, c) \cong \mathcal{E}(e, (R \circ R')c)$$

Поскольку композиция функторов ассоциативна, композиция сопряжений также ассоциативна. Легко видеть, что пара тождественных функторов образует тривиальное сопряжений, которое служит тождественностью относительно композиции присоединения. Следовательно, можно определить категорию $\mathbf{Adj}(\mathbf{Cat})$, в которой объекты являются категориями, а стрелки являются сопряжениями (по соглашению, указывая в направлении левого сопряжения).

Сопряжения могут быть определены исключительно в терминах функторов и естественных преобразований, то есть 1-клеток и 2-клеток в 2-категории $\mathbf{Cat}$. В $\mathbf{Cat}$ нет ничего особенного, и, фактически, сопряжения могут быть определены в любой 2-категории. Более того, категория сопряжений сама по себе является 2-категорией.

10.11 Уровни абстракции

Теория категорий занимается структурированием наших знаний. В частности, их можно применить к знанию о самой теории категорий. Следовательно, мы видим смешение уровней абстракции в теории категорий. Структуры, которые встречаются на одном уровне, можно сгруппировать в структуры более высокого уровня, демонстрирующие еще более высокие уровни структуры, и так далее.

В программировании мы привыкли строить иерархии абстракций. Значения группируются в типы, типы в виды. Функции, работающие со значениями, обрабатываются иначе, чем функции, работающие с типами. Мы часто используем другой синтаксис для разделения уровней абстракции. Но в теории категорий дело обстоит не так.

Множество, говоря категорным языком, может быть описано как дискретная категория. Элементы множества являются объектами этой категории и, кроме обязательных тождественных морфизмов, между ними нет стрелок.

Это же множество можно рассматривать как объект в категории **Set**. Стрелки в этой категории — это функции между множествами.

Категория **Set**, в свою очередь, является объектом категории **Cat**. Стрелками в **Cat** являются функторы.

Функторы между любыми двумя категориями $\mathcal{C}$ и $\mathcal{D}$ являются объектами в категории функторов $[\mathcal{C}, \mathcal{D}]$. Стрелки в этой категории — естественные преобразования.

Мы можем определить функторы между категориями функторов, категориями произведений, противоположными категориями и так далее, до бесконечности.

Замыкая круг: hom-множества в каждой категории являются множествами. Можно определить отображения и изоморфизмы между ними, охватывая и несопоставимые категории. Сопряжения также возможны, потому что мы можем сравнивать hom-множества, находящиеся в разных категориях.

Глава 11

Зависимые типы

Мы уже сталкивались с типами, которые зависят от других типов. Они определяются с помощью конструкторов типов с параметрами типа, такими как `Maybe` или `[]`. Большинство языков программирования имеют некоторую поддержку универсальных типов данных — типов данных, параметризованных другими типами данных.

С категорной точки зрения такие типы моделируются как функторы[1].

Естественным обобщением этой идеи является наличие типов, параметризованных значениями. Например, часто выгодно закодировать длину списка в его типе. Список нулевой длины будет иметь тип, отличный от типа списка единичной длины, и т.д.

Очевидно, нельзя изменить длину такого списка, так как это изменит его тип. Это не проблема в функциональном программировании, где все типы данных и так неизменяемы. Когда вы добавляете элемент в список, вы создаете новый список, по крайней мере, концептуально. Со списком, закодированным по длине, этот новый список имеет другой тип, вот и все!

Типы, параметризованные значениями, называются *зависимыми типами*. Существуют такие языки, как Idris или Agda, полностью поддерживающие зависимые типы. В Haskell также можно реализовать зависимые типы, но их поддержка пока не систематизирована (во время написания этого текста).

Причина использования зависимых типов в программировании состо-

[1]Конструктор типа, который не сдержит экземпляра `Functor`, можно рассматривать как функтор от дискретной категории — категории без стрелок, кроме тождественных.

ит в том, чтобы сделать программы доказуемо правильными. Для этого компилятор должен иметь возможность проверить предположения, сделанные программистом.

Haskell с его мощной системой типов может обнаруживать множество ошибок во время компиляции. Например, он не пропустит выражение `a <> b` (инфиксная нотация для `mappend`), если вы не предоставите экземпляр `Monoid` для типа переменных.

Однако в системе типов Haskell нет способа выразить или, тем более, обеспечить выполнение законов единиц и ассоциативности для моноидов. Для этого, экземпляр класса типа `Monoid` должен был бы содержать в себе доказательства равенства (а не фактический код):

```
assoc :: m <> (n <> p) = (m <> n) <> p
lunit :: mempty <> m   = m
runit :: m <> mempty   = m
```

Зависимые типы, и, в частности, типы равенства, прокладывают путь к этой цели.

Материал этой главы является более сложным и не используется в остальной части книги, поэтому вы можете смело пропустить его при первом чтении. Кроме того, чтобы избежать путаницы между расслоениями и функциями, было решено использовать заглавные буквы для объектов в некоторых частях этой главы.

11.1 Зависимые векторы

Начнем со стандартного примера счетного списка или вектора:

```
data Vec n a where
  VNil  :: Vec Z a
  VCons :: a -> Vec n a -> Vect (S n) a
```

Компилятор распознает это определение как зависимо типизированное, если вы включите следующие языковые прагмы:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
```

Первым аргументом конструктора типа является натуральное число `n`. Обратите внимание: это значение, а не тип. Средство проверки типов может определить это по использованию `n` в двух конструкторах данных. Первый создает вектор типа `Vec Z a`, а второй — вектор типа `Vect (S n) a`, где `Z` и `S` определены как конструкторы натуральных чисел:

```
data Nat = Z | S Nat
```

Можно более подробно указать параметры, если воспользоваться прагмой:

```
{-# LANGUAGE KindSignatures #-}
```

и импортировать библиотеку:

```
import Data.Kind
```

Затем можно указать, что `n` — это `Nat`, тогда как `a` — это `Type`:

```
data Vec (n :: Nat) (a :: Type) where
  VNil  :: Vec Z a
  VCons :: a -> Vec n a -> Vec (S n) a
```

Используя одно из этих определений, можно, например, построить вектор (целых чисел) нулевой длины:

```
emptyV  :: Vec Z Int
emptyV   = VNil
```

Он имеет тип, отличный от типа вектора длины один:

```
singleV :: Vec (S Z) Int
singleV  = VCons 42 VNil
```

Теперь можно определить функцию зависимого типа, которая возвращает первый элемент вектора:

```
headV            :: Vec (S n) a -> a
headV (VCons a _) = a
```

Эта функция гарантированно работает исключительно с векторами ненулевой длины. Это векторы, размер которых соответствует `(S n)`, и который не может быть `Z`. Если вы попытаетесь вызвать эту функцию с `emptyV`, компилятор сообщит об ошибке.

Другой пример — функция, которая объединяет два вектора в один. В его сигнатуре типа закодировано требование, чтобы два вектора были одного размера `n` (результат также имеет размер `n`):

```
zipV              :: Vec n a ->
                     Vec n b -> Vec n (a, b)
zipV (VCons a as)
     (VCons b bs) = VCons (a, b) (zipV as bs)
zipV VNil VNil    = VNil
```

Упражнение 11.1.1. *Реализуйте функцию* `tailV`*, которая возвращает хвост вектора ненулевой длины. Попробуйте вызвать ее с помощью* `emptyV`.

11.2 Категорная характеристика зависимых типов

Самый простой способ визуализировать зависимые типы — думать о них как о семействах типов, индексируемых элементами некоторого множества. В случае счетных векторов индексным множеством будет множество натуральных чисел $\mathbb{N}$.

Нулевой тип будет типом единицы `()`, представляющим пустой вектор. Тип, соответствующий `(S Z)`, будет `a`; тогда получим пару `(a, a)`, тройку `(a, a, a)` и т.д., со все более и более высокими степенями `a`.

Если мы хотим говорить обо всем семействе как об одном большом множестве, то можем образовать сумму всех этих типов. Например, сумма всех степеней a — это знакомый тип списка, также известный как свободный моноид:

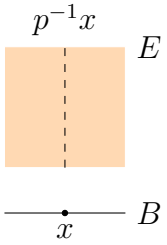
$$List(a) = 1 + a + a \times a + a \times a \times a + \ldots = \sum_{n:\mathbb{N}} a^n$$

Расслоения

Хотя эта точка зрения интуитивно проста для визуализации, она плохо обобщается на теорию категорий, где не рекомендуется смешивать множества с объектами. Поэтому, вместо того, чтобы говорить о вложении членов семейства в сумму, будем рассматриваем отображение, которое идет в обратном направлении.

Опять же, сначала будем проводить визуализацию, используя множества. Имеем одно большое множество E, описывающее *все* семейство, и функцию p, называемую *проекцией*, или *представляемым отображением*, которое идет от E к индексному множеству B (также называемому *базой*).

Эта функция, как правило, отображает несколько элементов в один. Тогда можно говорить об обратном образе конкретного элемента $x \in B$ как о множестве элементов, которые отображаются на него с помощью p. Это множество называется *слоем* и записывается $p^{-1}x$ (хотя, вообще говоря, p необратимо в обычном смысле). Рассматриваемое как совокупность слоев, E часто называют *пространством расслоения* или просто пространством.



Теперь забудем о множествах. *Расслоением* в произвольной категории является пара объектов e, b и стрелка $p : e \to b$.

Так что, на самом деле, это просто стрелка, но все решает контекст. Когда некоторую стрелку мы называем расслоением, то используем интуицию из множеств и представляем ее источник e как набор слоев, со стрелкой p, проецирующей каждый слой к единственной точке в базе b.

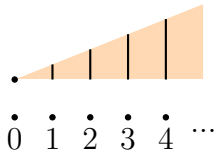
Мы можем пойти еще дальше: поскольку (малые) категории образуют категорию $\mathbf{Cat}$ с функторами в качестве стрелок, то можно определить расслоение категории, используя в качестве базы другую категорию.

Семейства типов как расслоения

Так что, будем моделировать семейства типов расслоениями. Например, семейство счетных векторов можно представить в виде расслоения, базой которого является тип натуральных чисел. Все семейство представляет собой сумму (копроизведение) последовательных степеней (произведений) a:

$$List(a) = a^0 + a^1 + a^2 + \ldots = \sum_{n:\mathbb{N}} a^n$$

с нулевой степенью — инициальным объектом, представляющим вектор нулевого размера.



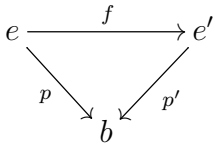
Проекция $p : List(a) \to \mathbb{N}$ — это ивестная функция $length$ (длина списка).

В теории категорий принято описывать сущности в целом, определяя их внутреннюю структуру с помощью, сохраняющих структуру, отображений. Таким же образом дело обстоит и с расслоениями. Если мы зафиксируем объект b базы и рассмотрим все возможные исходные объекты в категории $\mathcal{C}$, и все возможные проекции к b, то получим *категорию среза* $\mathcal{C}/b$. Эта категория представляет все способы, которыми мы можем слоить объекты из C над базой b.

Напомним, что объекты в категории среза — это пары $\langle e, p : e \to b \rangle$, а морфизм между двумя объектами $\langle e, p \rangle$ и $\langle e', p' \rangle$ — это стрелка $f : e \to e'$, переключающая проекции, то есть:

$$p' \circ f = p$$

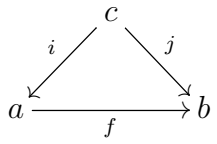
Лучший способ визуализировать это — заметить, что такой морфизм отображает слои p к слоям p'. Это — «сохраняющее слои» отображение между пространствами.



Счетные векторы можно рассматривать как объекты в категории среза $\mathcal{C}/\mathbb{N}$, заданной парами $\langle List(a), length \rangle$.

Категории ко-слоев

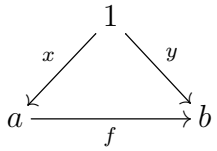
Существует также двойственное понятие категории ко-слоев $c/\mathcal{C}$, также известное как под-категория. Это категория стрелок, исходящих от фиксированного объекта c. Объектами этой категории являются пары $\langle a, i : c \to a \rangle$. Морфизмы в $c/\mathcal{C}$ — это стрелки, делающие соответствующие треугольники коммутативными.



В частности, если категория $\mathcal{C}$ содержит терминальный объект 1, то категория ко-слоев $1/\mathcal{C}$ имеет в качестве объектов глобальные элементы всех объектов из $\mathcal{C}$.

Существует забывающий функтор $U : 1/\mathcal{C} \to \mathcal{C}$, который отображает $\langle a, i \rangle$ к a. Этот функтор определяет расслоение $1/\mathcal{C}$ с базой $\mathcal{C}$. Объекты в слое над некоторым a находятся во взаимно однозначном соответствии с глобальными элементами a.

Морфизмы в $1/\mathcal{C}$, соответствующие стрелкам $f : a \to b$, отображают множество глобальных элементов a к множеству глобальных элементов b.

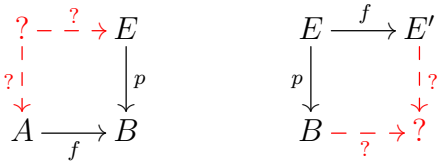


Таким образом, это расслоение оправдывает нашу интуицию о типах как о множествах значений, причем значения представлены глобальными элементами типов.

Обратные образы

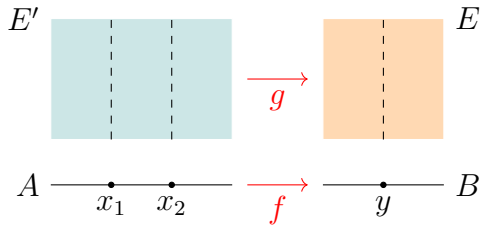
Мы уже видели много примеров коммутативных квадратов. Каждый такой квадрат является графическим представлением уравнения: два пути, каждый из которых является результатом композиции двух морфизмов, между противоположными углами квадрата, равны.

Как и в случае с любым равенством, можем заменить одну или несколько его составляющих неизвестным (неизвестными) и попытаться решить полученное уравнение. Например, можно задать вопрос: существует ли объект вместе с двумя стрелками, который завершал бы коммутирующий квадрат? Если таких объектов много, существует ли универсальный? Если недостающая часть — это верхний левый угол квадрата (источник), мы называем его *обратным образом* (или, расслоенным произведением). Если это правый нижний угол (цель), мы называем его *амальгамой* (или, расслоенным ко-произведением).



Начнем с конкретного расслоения $p : E \to B$. Что произойдет, если изменить базу с B на некоторое A, связанных отображением $f : A \to B$? Можно ли «вытянуть слои обратно» вдоль f?

Опять же, сначала обратимся к множествам. Представьте, что вы выбираете слой в E над некоторой точкой $y \in B$, которая является образом f. Если бы f была обратимой, то существовал бы элемент $x = f^{-1}y$. Мы разместили бы над ним наш слой. Однако, в общем случае f необратима. Это означает, что может быть больше элементов A, которые отображаются к нашему y. На рисунке ниже вы видите два таких элемента, x_1 и x_2. Мы просто продублировали слой над y и поместили его поверх всех элементов, которые отображаются к y. Таким образом, к каждой точке в A будет «прикреплен» слой. Сумма всех этих слоев образует новое пространство E'.



Таким образом, построено новое расслоение с базой A. Его проекция $p' : E' \to A$ отображает каждую точку данного слоя в точку, к которой этот слой был прикреплен. Существует также очевидное отображение $g : E' \to E$, переводящее слои в соответствующие им слои.

По построению, это новое расслоение $\langle E', p' \rangle$ удовлетворяет условию:

$$p \circ g = f \circ p'$$

которое можно изобразить в виде коммутативного квадрата:

$$\begin{array}{ccc} E' & \xrightarrow{g} & E \\ {\scriptstyle p'}\downarrow & & \downarrow{\scriptstyle p} \\ A & \xrightarrow[f]{} & B \end{array}$$

В **Set** можно явно построить E' как *подмножество* декартова произведения $A \times E$ с $p' = \pi_1$ и $g = \pi_2$ (две декартовы проекции). Элементом E' является пара $\langle a, e \rangle$ такая, что:

$$f(a) = p(e)$$

Приведенный коммутативный квадрат является отправной точкой для категорного обобщения. Однако, даже в **Set** существует много различных расслоений над A, делающих эту диаграмму коммутативной. Мы должны выбрать универсальное. Такой универсальной конструкцией является *обратный образ*, или *расслоенное произведение*.

В теории категорий, обратный образ $p : e \to b$ вдоль $f : a \to b$ представляет собой объект e' вместе с двумя стрелками $p' : e' \to a$ и $g : e' \to e$, превращающими следующую диаграмму в коммутативную:

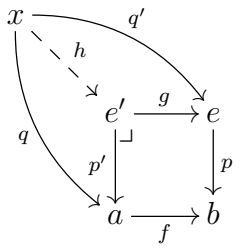
$$\begin{array}{ccc} e' & \xrightarrow{g} & e \\ {\scriptstyle p'}\downarrow & & \downarrow{\scriptstyle p} \\ a & \xrightarrow[f]{} & b \end{array}$$

и это удовлетворяет универсальному условию.

Это универсальное условие гласит, что для любого другого объекта-кандидата x с двумя стрелками $q' : x \to e$ и $q : x \to a$ такими, что $p \circ q' = f \circ q$ (что делает внешний «квадрат» на следующей диаграмме коммутативным), существует единственная стрелка $h : x \to e'$, то есть имеет место

$$q = p' \circ h$$
$$q' = g \circ h$$

что делает два треугольника коммутативными:



Символ ⌟ в верхнем углу квадрата используется (в соответствующей ориентации) для обозначения обратных образов.

Если посмотреть на обратный образ через призму множеств и расслоений, то e будет пространством над b, и мы строим новое пространство e' из слоев, взятых из e. Как мы разместим эти слои над a, определяется (прообразом) f. Эта процедура делает e' пространством как над a, так и над b, последнее с проекцией $p \circ g = f \circ p'$.

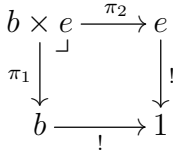
Вершина x на этом изображении — какое-то другое пространство над a с проекцией q. Это одновременно и пространство над b с проекцией $f \circ q = p \circ q'$. Единственное h отображает слои из x, заданные q^{-1}, к слоям e', заданных p'^{-1}.

Все отображения на этой диаграмме работают со соями. Некоторые из них перестраивают слои по новым основаниям — это деятельность обратного образа. Другие отображения модифицируют отдельные слои — так действует отображение $h : x \to e'$.

Если думать о пучках как о контейнерах слоев, перестановки слоев соответствуют естественным преобразованиям, а модификации слоев — действию `fmap`.

Тогда, универсальное условие заключается в том, что q' можно разложить (факторизовать) на модификацию слоев h, за которым следует перестановка слоев g.

Стоит отметить, что выбор терминального объекта или одноэлементного множества в качестве цели обратного образа автоматически дает определение декартова произведения:



В качестве альтернативы можно думать об этом, как о размещении стольких копий e, сколько элементов содержится в b. Мы будем использовать эту аналогию, когда будем говорить о зависимой сумме и зависимом произведении.

Заметим также, что одиночный слой можно извлечь из расслоения, протянув его обратно к терминальному объекту. В этом случае отображение $x : 1 \to b$ выбирает элемент базы, а обратный образ по нему извлекает единственный слой φ:

$$\begin{array}{ccc} \varphi F & \xrightarrow{g} & e \\ {\scriptstyle !}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ 1 & \xrightarrow[x]{} & b \end{array}$$

Стрелка g вводит этот слой обратно в e. Изменяя x, можно выбирать разные слои из e.

Упражнение 11.2.1. *Покажите, что обратный образ с терминальным объектом в качестве цели является произведением.*

Упражнение 11.2.2. *Покажите, что обратный образ можно определить как предел диаграммы из категории фигурок с тремя объектами*:

$$a \to b \leftarrow c$$

Упражнение 11.2.3. *Покажите, что обратный образ в $\mathcal{C}$ с целью b является произведением в категории среза $\mathcal{C}/b$. Подсказка*: *определите две проекции как морфизмы в категории среза. Используйте универсальность обратного образа, чтобы показать универсальность произведения.*

Функтор замены базы

В качестве модели для программирования мы использовали декартово замкнутую категорию. Для моделирования зависимых типов нужно наложить дополнительное условие: мы требуем, чтобы категория была *локально декартово замкнутой*. Это категория, в которой все категории срезов являются декартово замкнутыми.

В частности, у таких категорий есть все обратные образы, поэтому всегда можно изменить базу любого расслоения. Изменение базы вызывает отображение между категориями срезов, которое является функториальным.

Для двух категорий срезов $\mathcal{C}/b$ и $\mathcal{C}/a$, и стрелки между базами $f : b \to a$, функтор замены базы $f^* : \mathcal{C}/a \to \mathcal{C}/b$ отображает расслоение $\langle e, p\rangle$ к расслоению $f^* \langle e, p\rangle = \langle f^*e, f^*p\rangle$, что задается обратным образом:

$$\begin{array}{ccc} f^*e & \xrightarrow{g} & e \\ {\scriptstyle f^*p}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ b & \xrightarrow[f]{} & a \end{array}$$

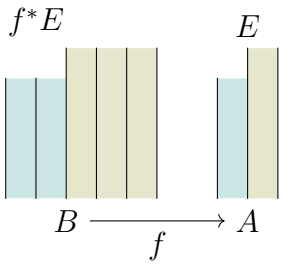
Обратите внимание, что функтор f^* идет в направлении, противоположном стрелке f.

Для визуализации функтора изменения базы, рассмотрим, как он работает на множествах.

$$\begin{array}{ccc} f^*e & \xrightarrow{g} & E \\ {\scriptstyle f^*p}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ B & \xrightarrow[f]{} & A \end{array}$$

Интуитивно, расслоение p разбивает множество E на слои над каждой точкой A.

Можно думать об f как о другом расслоении, которое аналогично разлагает B. Назовем эти слои в B «заплатками». Например, если A — просто двухэлементное множество, то расслоение, заданное f разбивает B на две заплатки. Обратный образ принимает слой из E и размещает его по всей заплатке в B. Результирующее множество f^*E выглядит как лоскутное одеяло, где каждая заплатка «раскрашена» клонами одного слоя из E.



Поскольку имеется функция от B к A, которая может отображать множество элементов в один, расслоение над B имеет более мелкую структуру, чем более грубое расслоение над A. Самый простой способ с наименьшими усилиями превратить расслоение E над A в расслоение над B, состоит в том, чтобы распределить существующие слои по заплаткам, определяемым (инверсией) f. В этом суть универсальной конструкции обратного образа.

В частности, если A — одноэлементное множество (терминальный объект), то имеется только один слой (целиком E), а пространство f^*E — это декартово произведение $B \times E$. Такое пространство называется *тривиальным пространством*.

Нетривиальное пространство не является произведением, но его можно *локально* разложить на произведения. Точно так же, как B является суммой заплаток, f^*E является суммой произведений этих заплаток и соответствующих слоев E.

Вы также можете думать об A как об *атласе*, в котором перечислены все заплатки в *базе* B. Представьте, что A — это множество стран, а B — множество городов. Отображение f ставит в соответствие страну каждому городу.

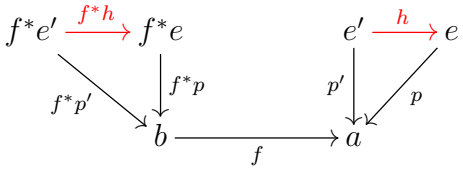
Развивая этот пример, пусть E будет множеством языков, используемых в стране. Если предположить, что в каждом городе говорят на языке данной страны, функтор замены базы «пересаживает» языки страны в каждый из ее городов.

Между прочим, эта идея, использования локальных заплаток и атласа, восходит к дифференциальной геометрии и общей теории относительности, где часто склеивают локальные системы координат для описания топологически нетривиальных расслоений, таких как ленты Мёбиуса или бутылки Клейна.

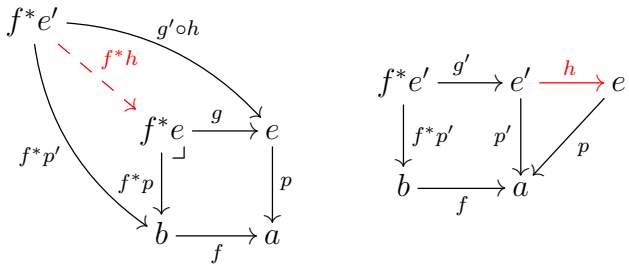
Как мы вскоре увидим, в локально декартовой замкнутой категории функтор замены базы имеет как левое, так и правое сопряжение. Левое сопряженное называется зависимой суммой, а правое сопряженное называется зависимым произведением или зависимой функцией.

Упражнение 11.2.4. *Определите действие функтора замены базы на морфизмах в C/a, то есть, по заданному морфизму h, постройте его*

аналог, f^*h,



Подсказка: *используйте универсальность обратного образа и условие коммутативности*: $g' \circ h \circ p = f^*p' \circ f$.



11.3 Зависимая сумма

В теории типов, зависимая сумма, $\sum_{x:B} T(x)$, определяется как тип пар, в которых *тип* второго компонента зависит от *значения* первого компонента.

Концептуально, тип-сумма определяется с помощью его свойства отображения-вне. Отображение-вне суммы представляет собой пару отображений, как показано в сопряжении:

$$\mathcal{C}(F_1 + F_2, F) \cong (\mathcal{C} \times \mathcal{C})(\langle F_1, F_2 \rangle, \Delta F)$$

Здесь, имеется пара стрелок $(F_1 \to F, F_2 \to F)$, которые определяют отображение-вне суммы $S = F_1 + F_2$. В **Set** сумма представляет собой размеченное объединение. Зависимая сумма — это сумма, помеченная элементами другого множества.

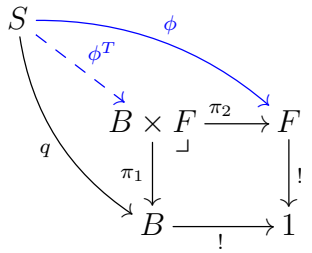
Так, счетный векторный тип можно рассматривать как зависимую сумму, помеченную натуральными числами. Элементом этого типа является натуральное число `n` (значение) в паре с элементом n-кортежного типа `(a, a, ... a)`. Вот несколько счетных векторов целых чисел, записанных в этом представлении:

```
(0, ())
(1, 42)
(2, (64, 7))
(5, (8, 21, 14, -1, 0))
```

В более общем смысле, правило введения зависимой суммы предполагает наличие семейства типов $T(x)$, индексируемых элементами базового типа B. Тогда элемент $\sum_{x:B} T(x)$ строится из пары элементов $x : B$ и $y : T(x)$.

Категорно, зависимая сумма моделируется как левый сопряженный функтор замены базы.

Чтобы убедиться в этом, сначала вернемся к определению пары, которая является элементом произведения. Мы уже замечали, что произведение может быть записано как обратный образ от одноэлементного множества — терминального объекта. Универсальная конструкция для произведения/обратного образа (нотация предвосхищает цель этой конструкции) имеет вид:



Мы также знаем, что произведение можно определить с помощью сопряжения. Можно заметить это сопряжение на приведенной диаграмме: для каждой пары стрелок $\langle \phi, q \rangle$ существует единственная стрелка ϕ^T , которая превращает треугольники в коммутативные.

Заметим, что если мы оставляем q фиксированным, то получаем взаимно однозначное соответствие между стрелками ϕ и ϕ^T. Это и будет интересующее нас сопряжение.

Теперь можно надеть фибрационные очки и заметить, что $\langle S, q \rangle$ и $\langle B \times F, \pi_1 \rangle$ — это два расслоения над одной и той же базой B. Коммутативный треугольник делает ϕ^T морфизмом в категории среза $\mathcal{C}/B$, или послойным отображением. Другими словами, ϕ^T является элементом hom-множества:

$$(\mathcal{C}/B)\left(\left\langle \begin{array}{c} S \\ q \end{array} \right\rangle, \left\langle \begin{array}{c} B \times F \\ \pi_1 \end{array} \right\rangle\right)$$

Поскольку ϕ является элементом hom-множества $\mathcal{C}(S, F)$, мы можем переписать взаимно однозначное соответствие между ϕ^T и ϕ как изоморфизм hom-множеств:

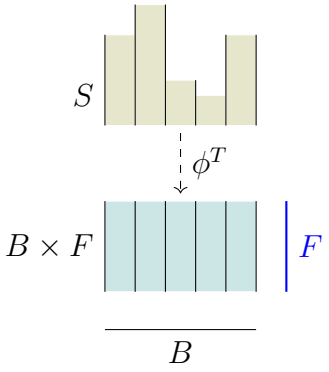
$$\mathcal{C}(S,F) \cong (\mathcal{C}/B)\left(\left\langle \begin{array}{c} S \\ q \end{array} \right\rangle, \left\langle \begin{array}{c} B \times F \\ \pi_1 \end{array} \right\rangle\right) \cong \mathcal{C}(S,F)$$

Фактически, это сопряжение, в котором левый функтор является забывающим функтором $U : \mathcal{C}/B \to \mathcal{C}$, который отображает $\langle S, q\rangle$ к S, подобным образом забывая расслоение.

Если внимательно присмотреться к этому сопряжению, то можно увидеть очертания определения S как категорной суммы (копроизведения).

Во-первых, справа имеется отображение-вне S. Можно интерпретировать S как сумму слоев, которые определяются расслоением $\langle S, q\rangle$.

Во-вторых, напомним, что расслоение $\langle B \times F, \pi_1\rangle$ можно рассматривать как порождающее множество копий F, расположенных над точками в B. Это обобщение диагонального функтора Δ, дублирующего F — здесь производятся «B копий» F. Левая часть сопряжения — это просто совокупность стрелок, каждая из которых отображает отдельный слой S к целевому слою F.



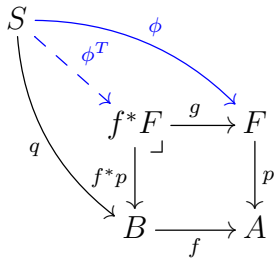
Применяя эту идею к примеру со счетными векторами, φ^T обозначает бесконечно много функций, по одной на каждое натуральное число. На практике эти функции определяются с помощью рекурсии. Например, отображение-вне вектора целых чисел можно определить следующим образом:

```
sumV             :: Vec n Int -> Int
sumV VNil         = 0
```

```
sumV (VCons n v) = n + sumV v
```

Добавление атласа

Мы можем обобщить нашу диаграмму, заменив терминальный объект произвольной базой A (*атлас*). Вместо одного слоя, теперь иисполь-зуем расслоение $\langle F, p\rangle$, и получаем квадрат обратного образа, который определяет функтор замены базы f^*:



Можно предположить, что расслоение над B более тонкое, поскольку f может отображать несколько точек в одну. Представьте, например, функцию `even :: Nat -> Bool`, которая создает две группы чисел, четных и нечетных. На этой диаграмме, f определяет более грубую «передискретизацию» исходной S.

Универсальность обратного образа приводит к следующему изоморфизму hom-множеств:

$$(\mathcal{C}/B)\left(\left\langle \begin{array}{c} S \\ q \end{array} \right\rangle, f^* \left\langle \begin{array}{c} F \\ p \end{array} \right\rangle\right) \cong (\mathcal{C}/A)\left(\left\langle \begin{array}{c} S \\ f \circ q \end{array} \right\rangle, \left\langle \begin{array}{c} F \\ p \end{array} \right\rangle\right)$$

Здесь, ϕ^T — элемент левой части, а ϕ — соответствующий элемент правой части.

Мы интерпретируем этот изоморфизм как сопряжение функтора замены базы f^*, слева, и функтора зависимой суммы, справа.

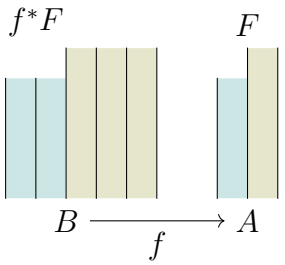
$$(\mathcal{C}/B)\left(\left\langle \begin{array}{c} S \\ q \end{array} \right\rangle, f^* \left\langle \begin{array}{c} F \\ p \end{array} \right\rangle\right) \cong (\mathcal{C}/A)\left(\sum_f \left\langle \begin{array}{c} S \\ q \end{array} \right\rangle, \left\langle \begin{array}{c} F \\ p \end{array} \right\rangle\right)$$

Таким образом, зависимая сумма определяется по формуле:

$$\sum_f \left\langle \begin{array}{c} S \\ q \end{array} \right\rangle = \left\langle \begin{array}{c} S \\ f \circ q \end{array} \right\rangle$$

Это говорит о том, что если S расслаивается над B с использованием q, и существует отображение f от B к A, то S автоматически расслаивается (более грубо) над A, причем проекция представляет собой композицию $f \circ q$.

Мы уже видели, что, в **Set**, f определяет заплатки внутри B. Слои F переносятся на эти заплатки, формируя f^*F. Локально, то есть, внутри каждой заплатки, f^*F выглядит как декартово произведение.



Само S расслаивается двумя способами: грубо — над A, используя $f \circ q$, и мелко — над B, используя q.

Экзистенциальная квантификация

В интерпретации «высказывания как типы», семействам типов соответствуют семейства высказываний. Зависимый тип суммы $\sum_{(x:B)} T(x)$ соответствует высказыванию: существует x, для которого $T(x)$ истинно:

$$\exists_{x:B} T(x)$$

Действительно, терм типа $\sum_{(x:B)} T(x)$ — это пара элементов $x : B$ и $y : T(x)$, что означает: $T(x)$ обитаемо для некоторого x.

11.4 Зависимое произведение

В теории типов, зависимое произведение или зависимая функция, $\prod_{(x:B)} T(x)$, определяется как функция, *тип* возвращаемого значения которой зависит от значения ее аргумента. Она является функцией, потому что ее можно вычислить. Зависимая функция $f : \prod_{(x:B)} T(x)$ применяется к аргументу $x : B$ для получения значения $f(x) : T(x)$.

Зависимое произведение на Haskell

Простой пример зависимого произведения — функция, которая создает вектор заданного размера и заполняет его копиями заданного значения[2]:

```
replicateV          :: a -> SNat n -> Vec n a
replicateV _ SZ      = VNil
replicateV x (SS n) = VCons x (replicateV x n)
```

В этом случае число, которое является аргументом для `replicateV`, передается как одноэлементное натуральное значение:

```
data SNat n where
  SZ :: SNat Z
  SS :: SNat n -> SNat (S n)
```

(отметим, что `replicateV` является функцией двух аргументов, поэтому ее можно считать либо зависимой функцией пары, либо обычной функцией, возвращающей зависимую функцию).

Зависимое произведение множеств

Прежде чем описывать категорную модель зависимых функций, полезно рассмотреть, как они работают с множествами. Зависимая функция выбирает один элемент из каждого множества $T(x)$.

Можно визуализировать совокупность этого выбора как гигантский кортеж — элемент декартова произведения. Например, в тривиальном случае B, являющимся двухэлементным множеством $\{1, 2\}$, зависимый функциональный тип — это просто декартово произведение $T(1) \times T(2)$. В общем случае, получается один компонент кортежа на каждое значение x. Это гигантский кортеж, проиндексированный элементами B. В этом и состоит смысл обозначения произведения $\prod_{(x:B)} T(x)$.

В нашем примере, `replicateV` подбирает конкретный счетный вектор для каждого значения `n`. Счетные векторы эквивалентны кортежам, поэтому для `n`, равного нулю, `replicateV` возвращает пустой кортеж

[2]На момент написания книги поддержка зависимых типов в Haskell была ограничена, поэтому реализация зависимых функций требует использования одноэлементных типов.

`()`; для `n = 1` возвращается одно значение `x`; для `n`, равного двум, дублируется `x`, возвращая `(x,x)`; и т.д.

Функция `replicateV`, вычисляемая при некотором `x :: a`, эквивалентна бесконечному кортежу кортежей:

$$((), x, (x, x), (x, x, x), \ldots)$$

который является конкретным элементом типа:

$$((), a, (a, a), (a, a, a), \ldots)$$

Детальнее о зависимом произведении

Чтобы построить категорную модель зависимых функций, нужно изменить нашу точку зрения с семейства типов на расслоение. Начнем с пространства E/B, расслоенного проекцией $p : E \to B$. Зависимая функция называется *сечением* этого пространства.

Если визуализировать пространство как совокупность слоев, торчащих из базы B, то сечение подобно стрижке: оно подрезает каждый слой, чтобы получить соответствующее значение. В физике такие сечения называются полями — с пространством-временем в качестве базы.

Точно так же, как мы говорили о функциональном объекте, представляющим множество функций, можем говорить об объекте $S(E)$, представляющем множество сечений заданного пространства E.

Так же, как было определено применение функции в качестве отображения-вне произведения:

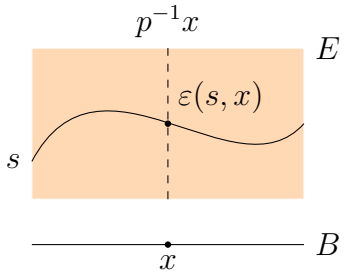
$$\varepsilon_{BC} : C^B \times B \to C$$

можно определить применение зависимой функции как отображение:

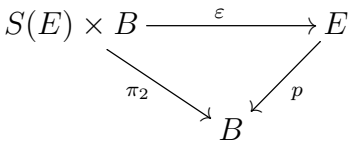
$$\varepsilon : S(E) \times B \to E$$

Это можно визуализировать как выбор сечения s в $S(E)$ и элемента x базы B, и создание значения в пространстве E (в физике это соответствовало бы измерению поля в определенной точке пространстве-времени).

Но на этот раз, нужно добиться того, чтобы это значение было в подходящем слое. Если мы спроецируем результат применения ε к (s, x), то значение x должно восстановиться.

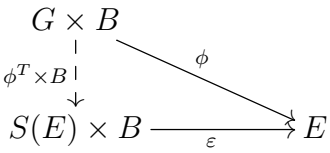


Другими словами, следующая диаграмма должна быть коммутативной:



Это делает ε морфизмом в категории среза $\mathcal{C}/B$.

И точно так же, как экспоненциальный объект был универсальным, универсальным является и объект сечений. Условие универсальности имеет ту же форму: для любого другого объекта G со стрелкой $\phi : G \times B \to E$ существует единственная стрелка $\phi^T : G \to S(E)$, которая делает следующую диаграмму коммутативной:



Разница в том, что, и ε, и ϕ , являются морфизмами в категории среза $\mathcal{C}/B$.

Взаимно однозначное соответствие между ε и ϕ^T образует сопряжение:

$$(\mathcal{C}/B)\left(\left\langle \begin{array}{c} G \times B \\ \pi_2 \end{array} \right\rangle, \left\langle \begin{array}{c} F \\ p \end{array} \right\rangle\right) \cong \mathcal{C}(G, S(E))$$

которое можно использовать как определение объекта сечений $S(E)$. Коединицей этого сопряжения является применение зависимой функции. Она получается заменой G на $S(E)$ и выбором тождественного морфизма

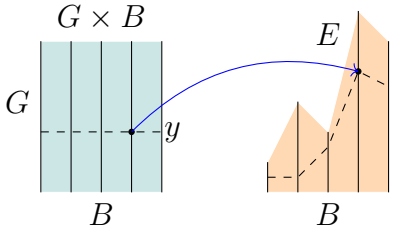
справа. Таким образом, ко-единица является элементом hom-множества:

$$(C/B)\left(\left\langle \begin{array}{c} S(E)\times B \\ \pi_2 \end{array} \right\rangle, \left\langle \begin{array}{c} E \\ p \end{array} \right\rangle\right)$$

Сравним приведенное выше сопряжение с каррирующим сопряжением, определяющим функциональный объект E^B:

$$\mathcal{C}(G\times B, E) \cong \mathcal{C}(G, EB)$$

Теперь напомним, что в **Set** мы интерпретируем произведение $G\times B$ как размещение копий G, в качестве идентичных слоев над каждым элементом B. Таким образом, один элемент левой части нашего сопряжения представляет собой семейство функций, по одной на каждый слой. Любой заданный $y \in G$ разрезает горизонтальный срез посредством $G\times B$. Это пары (y, b), для всех $b \in B$. Наше семейство функций отображает этот срез к соответствующим слоям E, создавая, таким образом, сечение E.



Указанное сопряжение означает, что это семейство отображений однозначно определяет функцию от G к $S(E)$. Таким образом, каждый элемент $y \in G$ отображается к другому элементу s из $S(E)$. Поэтому, элементы из $S(E)$ находятся во взаимно однозначном соответствии с сечениями из E .

Это всё теоретико-множественные интуиции. Можно обобщить их, заметив сначала, что правая часть сопряжения может быть просто выражена как hom-множество в категории среза $\mathcal{C}/1$ над терминальным объектом.

Действительно, существует взаимно однозначное соответствие между объектами X в $\mathcal{C}$ и объектами $\langle X, !\rangle$ в $\mathcal{C}/1$ (здесь, ! — это единственная стрелка к терминальному объекту). Стрелки в $\mathcal{C}/1$ — это стрелки из $\mathcal{C}$ без дополнительных ограничений. Таким образом, имеем:

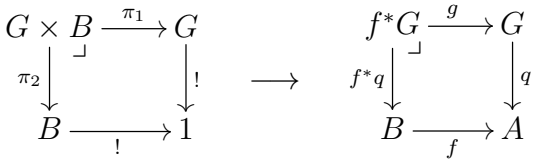
$$(\mathcal{C}/B)\left(\left\langle \begin{array}{c} G\times B \\ \pi_2 \end{array} \right\rangle, \left\langle \begin{array}{c} F \\ p \end{array} \right\rangle\right) \cong (\mathcal{C}/1)\left(\left\langle \begin{array}{c} G \\ ! \end{array} \right\rangle, \left\langle \begin{array}{c} S(E) \\ ! \end{array} \right\rangle\right)$$

Добавление атласа

Следующий шаг — «размывание фокуса» — замена терминального объекта более общей базой A, служащей атласом.

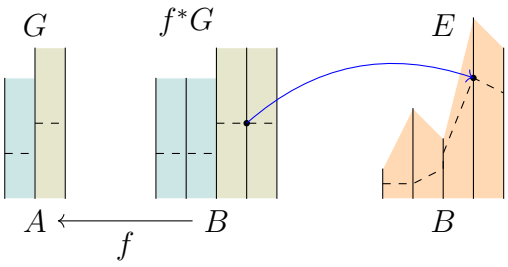
Правая часть сопряжения становится hom-множеством в категории среза $\mathcal{C}/A$. Сам G грубо расслаивается с использованием некоторой $q : G \to A$.

Напомним, что $G \times B$ можно понимать как обратный образ вдоль отображения $! : B \to 1$, или изменение базы с 1 на B. Если требуется заменить 1 на A, то надо заменить произведение $G \times B$ на более общий обратный образ q. Такая замена базы параметризуется новым морфизмом $f : B \to A$.



В результате, вместо пространства слоев G над B, получаем обратный образ f^*G, который заселен группами слоев из расслоения $q : G \to A$. Таким образом, A служит атласом, который подсчитывает все заплатки, заселенные единообразными слоями.

Представим, к примеру, что A — это двухэлементное множество. Расслоение q расщепит G на два слоя. Они будут служить общими слоями. Эти слои теперь перемещаются на два лоскутка в B, формируя f^*G. Перемещением управляет f^{-1}.

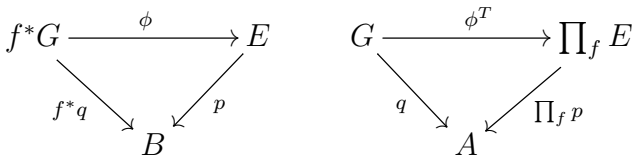


Таким образом, сопряжение, определяющее тип зависимой функции, есть:

$$(\mathcal{C}/B)\left(f^*\left\langle \begin{smallmatrix} G \\ q \end{smallmatrix} \right\rangle, \left\langle \begin{smallmatrix} E \\ p \end{smallmatrix} \right\rangle\right) \cong (\mathcal{C}/A)\left(\left\langle \begin{smallmatrix} G \\ q \end{smallmatrix} \right\rangle, \prod_f \left\langle \begin{smallmatrix} S(E) \\ ! \end{smallmatrix} \right\rangle\right)$$

Это есть обобщение сопряжения, которое мы использовали для определения объекта сечений $S(E)$. Оно определяет новый объект $\prod_f E$, который является перестройкой объекта сечений.

Сопряжение представляет собой отображение между морфизмами в соответствующих категориях срезов:

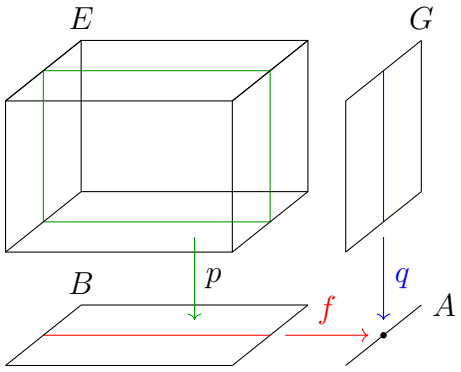


Чтобы получить некоторое представление об этом сопряжении, рассмотрим, как оно работает на множествах.

- Правая часть работает в «крупнозернистом» расслоении над атласом A. Это семейство функций, по одной функции на заплатку. Для каждой заплатки получаем функцию от «толстого слоя» G (изображено ниже синим цветом) к «толстому слою» $\prod_f E$ (не показано).

- Левая часть работает в более «тонком» расслоении над B. Эти слои сгруппированы в небольшие пространства над заплатками. Как только мы выбираем заплатку (изображенную ниже красным), то получаем семейство функций от этой заплатки к соответствующей заплатке в E (изображенной зеленым) — — сечение малого пространства в E. Таким образом, заплатка-за-заплаткой, мы получаем малые сечения E.

Это сопряжение означает, что элементы «толстого» слоя $\prod_f E$ соответствуют малым сечениям E над одним и тем же лоскутком.



Следующие упражнения проливают свет на роль морфизма f. Это можно рассматривать как локализацию сечений E путем ограничения их «окрестностями», определяемыми f^{-1}.

Упражнение 11.4.1. *Рассмотрите, что происходит, когда A представляет собой двухэлементное множество $\{0,1\}$, а f отображает B целиком в один элемент, скажем, 1. Как бы вы определили эту функцию в правой части сопряжения? Что она должна сделать со слоем над 0?*

Упражнение 11.4.2. *Пусть G — одноэлементное множество 1, а $x : 1 \to A$ является расслоением, которое выбирает элемент в A.* Используя сопряжение, покажите, что:

- f^*1 имеет два типа слоев: *синглетоны над элементами $f^{-1}(x)$* и пустые множества;
- *отображение $\phi : f^*1 \to E$ эквивалентно выбору элементов, по одному из каждого слоя E над элементами $f^{-1}(x)$. Другими словами, это частичное сечение E над подмножеством $f^{-1}(x)$ из B*;
- *слой из $\prod_f E$ над заданным x является таким частичным сечением*;
- *что происходит, когда A также является одноэлементным множеством*?

Универсальная квантификация

Логическая интерпретация зависимого произведения $\prod_{(x:B)} T(x)$ является универсально квантифицированным высказыванием. Элемент из $\prod_{(x:B)} T(x)$ — это сечение — доказательство возможности выбора элемента из каждого члена семейства $T(x)$. Это означает, что ни один из них не является пустым. Другими словами, это доказательство высказывания:

$$\forall_{(x:B)} T(x)$$

11.5 Равенство

Наш первый опыт в математике связан с равенством. Мы узнаем, что

$$1 + 1 = 2$$

и потом не задумываемся об этом.

Но что значит, что 1+1 равно 2? Два — это число, но один плюс один — это выражение, так что это не одно и то же. Прежде чем объявить эти две сущности равными, необходимо провести некоторую ментальную обработку.

Сравните это с утверждением 0=0, в котором обе стороны равенства являются *одним и тем же*.

Если мы хотим определить равенство, то должны, по крайней мере, убедиться, что все равно самому себе. Это свойство называется *рефлексивностью*.

Напомним определение натуральных чисел:

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

Вот как можно определить равенство для натуральных чисел:

```
equal             :: Nat -> Nat -> Bool
equal Z Z          = True
equal (S m) (S n) = equal m n
equal _ _          = False
```

Здесь рекурсивно удаляются все `S` в каждом числе, пока одно из них не достигнет `Z`. Если, одновременно с этим, и другое число достигает `Z`, мы объявляем числа, с которых началось сравнение, равными, в противном случае они не равны.

Эквациональные рассуждения

Отметим, что при определении равенства в Haskell мы уже использовали знак равенства. Например, знак равенства в:

```
equal Z Z          = True
```

означает, что везде, где встречается выражение `equal Z Z`, его можно заменить на `True`, и наоборот.

Это принцип замены равных равными, который является основой эквационального рассуждения в Haskell. Мы не можем записать доказательства равенства непосредственно на Haskell, но можем использовать рассуждения по уравнениям, чтобы рассуждать о программах на Haskell. Это одно из основных преимуществ чистого функционального программирования. Такие замены нельзя выполнять в императивных языках из-за побочных эффектов.

Если надо доказать, что $1+1$ равно 2, то сначала необходимо определить сложение. Определение может быть рекурсивным как по первому, так и по второму аргументу. Здесь это производится на втором аргументе:

```
add         :: Nat -> Nat -> Nat
add n Z      = n
add n (S m) = S (add n m)
```

Запишем $1+1$ как:

```
add (S Z) (S Z)
```

Теперь можно использовать определение `add`, чтобы упростить это выражение. Попытка сопоставить первое предложение не приводит к успеху, потому что `S Z` — это не то же самое, что `Z`. Но второе предложение сопоставимо. В нем `n` — произвольное число, поэтому можно подставить вместо него `S Z`, и получить:

```
add (S Z) (S Z) = S (add (S Z) Z)
```

В этом выражении мы можем выполнить другую замену равенства, используя первое предложение определения `add` (опять же, с заменой `n` на `S Z`):

```
add (S Z) Z = (S Z)
```

Мы приходим к:

```
add (S Z) (S Z) = S (S Z)
```

Можно ясно видеть, что правая часть представляет собой кодировку 2. Но не показано, что наше определение равенства является рефлексивным, поэтому в принципе непонятно, будет ли

```
eq (S (S Z)) (S (S Z))
```

давать `True`. Приходится снова использовать пошаговые рассуждения об уравнениях:

```
equal (S (S Z) (S (S Z)) =
{- second clause of the definition of equal -}
equal (S Z) (S Z)        =
{- second clause of the definition of equal -}
equal Z Z                =
{- first clause of the definition of equal -}
True
```

Можно использовать такого рода рассуждения для доказательства утверждений о конкретных числах, но мы сталкиваемся с проблемами при рассуждениях об общих числах — например, при демонстрации того, что что-то верно для всех `n`. Используя наше определение сложения, можно легко показать, что `add n Z` — это то же самое, что и `n`. Но мы не можем доказать, что `add Z n` — это то же самое, что и `n`. Последнее доказательство требует применения индукции.

В итоге мы различаем два вида равенства. Один из них доказывается с помощью подстановок или правил перезаписи и называется *дефиниционным равенством*. Вы можете думать об этом как о макрорасширении или встроенном расширении в языках программирования. Он также включает в себя β-редукции: выполнение применения функции путем замены формальных параметров фактическими аргументами, например:

```
(\x -> x + x) 2 =
{- beta reduction -}
2 + 2
```

Второй, более интересный вид равенства, называется *пропозициональным равенством*, но для него могут потребоваться фактические доказательства.

Сопоставление равенства и изоморфизма

Мы уже говорили, что категорные теоретики предпочитают изоморфизм равенству — по крайней мере, когда дело касается объектов. Верно, что в пределах категории невозможно провести различие между изоморфными объектами. Однако, в общем случае равенство сильнее изоморфизма. Это проблема, потому что очень удобно иметь возможность подставлять равные равными, но не всегда ясно, что можно заменить изоморфное изоморфным.

Математики разбирались с этой проблемой, в основном пытаясь изменить определение изоморфизма, но настоящий прорыв произошел, когда они решили одновременно ослабить определение равенства. Это привело к развитию гомотопической теории типов (HoTT).

Грубо говоря, в теории типов, а именно в теории зависимых типов Мартина-Лёфа, равенство представляется как тип, и чтобы доказать равенство, нужно построить элемент этого типа — в духе интерпретации Карри-Ховарда.

Более того, в HoTT сами доказательства можно сравнивать на равенство, и так до бесконечности. Вы можете представить себе это, рассматривая доказательства равенства не как точки, а как некие абстрактные пути, которые можно трансформировать друг в друга; отсюда и язык гомотопий.

В этой установке вместо изоморфизма, предполагающего строгое равенство стрелок:

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

определяется *эквивалентность*, в которой эти равенства рассматриваются как типы.

Основная идея HoTT заключается в том, что можно навязать *аксиому унивалентности*, которая, грубо говоря, утверждает, что равенства эквивалентны эквивалентностям, или символически:

$$(A = B) \cong (A \cong B)$$

Заметим, что это аксиома, а не теорема. Мы можем либо принять это, либо нет, а теория по-прежнему будет действительна (по крайней мере, так хочется думать).

Типы равенств

Предположим, требуется сравнить два терма на равенство. Первое требование состоит в том, чтобы оба терма были одного типа. Нельзя сравнивать яблоки с апельсинами. Пусть вас не смущают некоторые языки программирования, допускающие сравнение непохожих терминов: в каждом таком случае происходит неявное преобразование, и окончательное равенство всегда достигается между значениями одного типа.

Для каждой пары значений в принципе существует отдельный тип доказательств равенства. Есть тип для $0 = 0$, есть тип для $1 = 1$ и есть тип для $1 = 0$; последний, надо надеяться, необитаемый.

Таким образом, *тип равенства*, также известный как тип тождественности, является зависимым типом: он зависит от двух сравниваемых значений. Обычно, он записывается как Id_A, где A — тип обоих значений, или, с использованием инфиксной нотации, $x =_A y$.

Например, тип равенства двух нулей записывается как $Id_{\mathbb{N}}(0, 0)$ или:

$$0 =_{\mathbb{N}} 0$$

Заметим, это не утверждение и не терм. Это *тип*, такой же как, например, `Int` или `Bool`. Вы можете определить значение этого типа, если для него имеется правило введения.

Правило введения

Правило введения для типа равенства является зависимой функцией:

$$refl_A : \prod_{(x:A)} Id_A(x, x)$$

которое можно интерпретировать в духе высказываний как типов, как доказательство утверждения:

$$\forall_{(x:A)} x = x$$

Это известная рефлексивность: она показывает, что для всех x типа A, x равен самому себе. Вы можете применить эту функцию к некоторому конкретному значению x типа A, и она создаст новое значение типа $Id_A(x, x)$.

Теперь можно доказать, что $0 = 0$. Мы можем выполнить $refl_{\mathbb{N}}(0)$, чтобы получить значение типа $0 =_{\mathbb{N}} 0$. Это значение является доказательством того, что тип обитаем, и, следовательно, соответствует истинному высказыванию.

Для равенства, это единственное правило введения. Поэтому можно подумать, что все доказательства равенства сводятся к «они равны, потому что они одинаковы». Удивительно, но это не так.

β-редукция и η-конверсия

В теории типов существует такое взаимодействие правил введения и исключения, которое по существу делает их обратными друг другу.

Рассмотрим определение произведения. Оно введено предоставлением двух значений, $x : A$ и $y : B$, и получением значения $p : A \times B$. При необходимости, два исходных значения можно исключить, используя проекции. Но как узнать, те ли это значения, которые были использовали для его построения? Это то, что надо постулировать . Такое правило называется правилом вычисления или правилом β-редукции.

И наоборот, если дано значение $p : A \times B$, то можно извлечь два компонента с помощью проекций, а затем использовать правило введения, чтобы перекомпоновать их. Но откуда следует, что получится то же самое p? Это также требует постулирования, и иногда называется условием уникальности или правилом η-конверсии.

В категорной модели теории типов эти два правила следуют из универсальной конструкции.

Тип равенства также имеет правило исключения, которое мы вскоре обсудим, но не накладывая условия уникальности. Это означает, что возможно, имеются некоторые доказательства равенства, которые не были получены с помощью $refl$.

Именно ослабление понятия равенства делает HoTT интересным для математиков.

Принцип индукции для натуральных чисел

Прежде чем сформулировать правило исключения для равенства, полезно сначала обсудить более простое правило исключения для натуральных чисел. Мы уже рассматривали такое правило, описывающее при-

митивную рекурсию. Это позволило определить рекурсивные функции, указав значение $init$ и функцию $step$.

Используя зависимые типы, можно обобщить это правило, чтобы определить правило зависимого исключения, эквивалентное принципу математической индукции.

Принцип индукции можно описать как средство для доказательства сразу целых семейств высказываний, индексированных натуральными числами. Например, утверждение, что `add Z n` равно `n`, на самом деле является бесконечным числом утверждений, по одному на каждое значение `n`.

В принципе, можно написать программу, которая тщательно проверяла бы это утверждение для очень большого числа случаев, но мы никогда не были бы уверены, верно ли оно в целом. Есть некоторые гипотезы о натуральных числах, которые были проверены подобным образом, с помощью компьютеров, но, очевидно, они никогда не могут исчерпать бесконечное множество случаев.

Упрощая, можно разделить все математические теоремы на две группы: те, которые понятно формулируются, и те, формулировка которых сложна. Их можно далее подразделить на те, доказательства которых просты, и те, которые трудно или невозможно доказать. Например, знаменитая Великая теорема Ферма имеет чрезвычайно простую формулировку, но для ее доказательства потребовался чрезвычайно сложный математический аппарат.

Здесь нас интересуют теоремы о натуральных числах, которые имеют простую формулировку и их легко доказать. Предположим, что нам известно, как генерировать семейство высказываний или, что то же самое, зависимый тип $T(n)$, где n — натуральное число.

Также предположим, что имеется значение:

$$init : T(Z)$$

или, что то же самое, доказательство нулевого высказывания, и зависимая функция:

$$step : \prod_{(n:\mathbb{N})}(T(n) \to T(Sn))$$

Эта функция интерпретируется как порождающая доказательство $(n{+}1)$-го высказывания из доказательства n-го высказывания.

Правило зависимого исключения для натуральных чисел постулирует, что при таких $init$ и $step$, существует зависимая функция:

$$f : \prod_{(n:\mathbb{N})} T(n)$$

Эта функция интерпретируется как доказательство истинности $T(n)$ для всех n.

Более того, эта функция применительно к нулю воспроизводит $init$:

$$f(Z) = init$$

а применительно к преемнику n, согласуется с выполнением $step$:

$$f(Sn) = (step(n))(f(n))$$

(здесь, $step(n)$ создает функцию, которая затем применяется к значению $f(n)$). Это два *правила вычисления* натуральных чисел.

Заметим, что принцип индукции не является теоремой о натуральных числах. Это часть *определения* типа натуральных чисел.

Не все зависимые отображения натуральных чисел можно декомпозировать на $init$ и $step$, так же как не все теоремы о натуральных числах можно доказать индуктивно. Для натуральных чисел не существует правила η-конверсии.

Правило исключения равенства

Правило исключения для типа равенства в чем-то аналогично принципу индукции для натуральных чисел. Там мы использовали $init$, чтобы зафиксировать базу начала пути, и $step$, чтобы двигаться дальше. Правило исключения для равенства требует более мощной начальной фиксации, но не имеет $step$. На самом деле, нет хорошей аналогии того, как это работает, кроме как через слепую веру.

Идея состоит в том, что мы хотим построить отображение-*вне* типа равенства. Но поскольку тип равенства сам по себе является семейством типов с двумя параметрами, отображение должно быть зависимой функцией. Целью этой функции является другое семейство типов:

$$T(x, y, p)$$

которое зависит от пары сравниваемых значений $x, y : A$, и доказательства равенства $p : Id(x, y)$.

Функция, которую мы пытаемся построить, это:

$$f : \prod_{(x,y:A)} \prod_{(p:Id(x,y))} T(x, y, p)$$

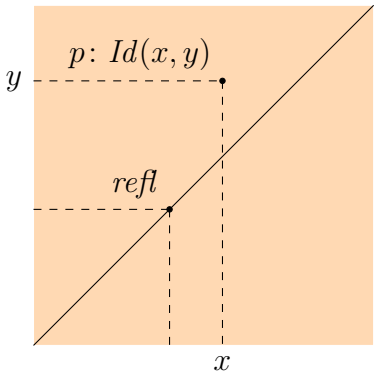
Удобно думать об этом как о доказательстве того, что для всех точек x и y, и для каждого доказательства их равенства, высказывание $T(x, y, p)$ истинно. Заметим, что потенциально имеются разные утверждения для *каждого доказательства* равенства двух точек.

Наименьшее, что можно потребовать от $T(x, y, p)$, — это чтобы оно было истинным, когда x и y буквально одинаковы, и доказательством равенства является очевидное $refl$. Это требование может быть выражено в виде зависимой функции:

$$t : \prod_{(x:A)} T(x, x, refl(x))$$

Заметим, что мы даже не рассматриваем доказательства вида $x = x$, кроме тех, которые задаются рефлексивностью. Существуют ли такие доказательства? Мы не знаем и нам это все равно.

Итак, и это наша основа, отправная точка путешествия, которое должно привести к определению нашего f для всех пар точек и всех доказательств равенства. Интуиция такова, что мы определяем f как функцию на плоскости (x, y), с третьим измерением, заданным p. Для этого нам дано то, что определяет диагональ (x, x), с p, ограниченным значением $refl$:



Думается, что нам нужно что-то большее, какой-то *шаг*, который переместил бы нас из одной точки в другую. Но, в отличие от натуральных чисел, здесь нет *следующей* точки или *следующего* доказательства равенства, к которому можно было бы перейти. В нашем распоряжении есть только функция t и ничего более.

Поэтому мы постулируем, что для семейства типов $T(x, y, p)$ и функции $t : \prod_{(x:A)} T(x, x, refl(x))$ существует функция

$$f : \prod_{(x,y:A)} \prod_{(p:Id(x,y))} T(x, y, p)$$

такая, что (имеется правило вычисления):

$$f(x, x, refl(x)) = t(x)$$

Заметим, что равенство в этом правиле вычисления — это *дефинициальное равенство* (т.е., определяемое равенство), а не тип.

Устранение равенства свидетельствует о том, что всегда возможно распространить функцию t, определенную на диагонали, на все трехмерное пространство. Это очень сильный постулат. Один из способов понять его — утверждать, что в рамках теории типов, сформулированной с использованием языка правил введения и исключения, а также правил манипулирования ими, *невозможно* определить семейство типов $T(x, y, p)$, что *не* удовлетворяло бы правилу устранения равенства.

Самая близкая аналогия, которую мы встречали до сих пор, — результат параметризации, который утверждает, что в Haskell все полиморфные функции между эндофункторами автоматически являются естественными преобразованиями. Другим примером, на этот раз из исчисления, является то, что любая аналитическая функция, определенная на действительной оси, имеет уникальное продолжение на всю комплексную плоскость.

Использование зависимых типов стирает грань между программированием и математикой. Существует целый спектр языков, начиная с Haskell, только осваивающих зависимые типы, но при этом прочно зарекомендовавших себя в коммерческом использовании, вплоть до средств доказательства теорем, которые помогают математикам формализовать математические доказательства.

Глава 12

Алгебры

Суть алгебры — формальное манипулирование выражениями. Но что такое выражение и как им можно манипулировать?

Первое, что нужно сказать об алгебраических выражениях, таких как $2(x + y)$ или $ax^2 + bx + c$, — это то, что их бесконечно много. Существует конечное число правил их создания, но эти правила можно использовать в бесконечном количестве комбинаций. Это говорит о том, что правила используются *рекурсивно*.

В программировании выражения практически синонимичны деревьям (их разбору). Рассмотрим простой пример арифметического выражения:

```
data Expr = Val Int
          | Plus Expr Expr
```

Рецепт постройки деревьев: мы начинаем с маленьких деревьев, используя конструктор `Val`; затем мы располагаем эти саженцы в узлах и т.д.

```
e2 = Val 2
e3 = Val 3
e5 = Plus e2 e3
e7 = Plus e5 e2
```

Такие рекурсивные определения отлично работают в языке программирования. Проблема в том, что для каждой новой рекурсивной структуры данных потребуется собственная библиотека функций, которые с ней работают.

В терминах теории типов мы смогли определить рекурсивные типы, такие как натуральные числа или списки, предоставив в каждом случае особые правила введения и исключения. Нам нужно что-то более общее, процедура генерации произвольных рекурсивных типов из более простых подключаемых компонентов.

Когда речь заходит о рекурсивных структурах данных, возникают две ортогональные проблемы. Одной из них является механизм рекурсии. Другой — это подключаемые компоненты, которые будут использоваться рекурсией.

Известно, как работать с рекурсией: мы предполагаем, что знаем, как строить маленькие деревья. Затем используется рекурсивный шаг, чтобы разместить эти деревья в узлы, делая деревья большего размера.

Теория категорий подсказывает, как формализовать это неточное описание.

12.1 Алгебры из эндофункторов

Идея размещения меньших деревьев в узлах требует формализации того, что значит иметь структуру данных с дырами — «контейнер для сущностей». Именно для этого и нужны функторы. Поскольку мы хотим использовать эти функторы рекурсивно, они должны быть *эндо*-функторами.

Например, эндофунктор из предыдущего примера будет определяться следующей структурой данных, где `x` отмечает точки:

```
data ExprF x = ValF Int
             | PlusF x x
```

Информация обо всех возможных формах выражений абстрагируется в один функтор.

Другой важной частью информации является рецепт вычисления выражений, что можно закодировать с помощью того же эндофунктора.

Думая рекурсивно, предположим, что мы знаем, как вычислить все поддеревья большего выражения. Оставшийся шаг — подключить эти результаты к узлу верхнего уровня и вычислить его.

Например, предположим, что все `x` в функторе были заменены целыми числами — результатами подсчета поддеревьев. Совершенно очевидно, что нужно сделать на последнем шаге. Если вершиной дерева

является лист `ValF` (что означает: поддеревья для подсчета отсутствуют), то просто возвращается хранящееся в нем целое число. Если это узел `PlusF`, то в него добавляются два целых числа. Этот рецепт может быть закодирован так:

```
eval            :: ExprF Int -> Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

Мы сделали несколько, казалось бы, очевидных предположений, основанных на опыте. Например, поскольку узел называется `PlusF`, логично предположить, что должны быть сложены два числа. Но умножение или вычитание будут работать одинаково хорошо.

Поскольку лист `ValF` содержит целое число, мы предположили, что выражение должно вычисляться как целое число. Но имеется столь же правдоподобный вычислитель, который красиво печатает выражение, преобразовывая его в строку, и использует конкатенацию вместо сложения:

```
pretty            :: ExprF String -> String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

Фактически, существует бесконечно много вычислителей. Любой выбор целевого типа и любой выбор вычислителя должны быть в равной степени действующими. Это приводит к следующему определению:

Алгебра для эндофунктора F — это пара (c, α). Объект c называется *носителем* алгебры, а вычислитель $\alpha : Fc \to c$ называется *структурным отображением*.

На Haskell для заданного функтора `f` определяем:

```
type Algebra f c = f c -> c
```

Заметим, что вычислитель *не* является полиморфной функцией. Это конкретный выбор функции для конкретного типа `c`. Может быть много вариантов выбора типов носителей и может быть много разных вычислителей для данного типа. Все они определяют отдельные алгебры.

Выше мы определили две алгебры для `ExprF`. Следующая алгебра имеет `Int` в качестве носителя:

```
eval            :: Algebra ExprF Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

12.2 Категория алгебр

Алгебры для заданного эндофунктора F образуют категорию. Стрелка в этой категории — это алгебраический морфизм, который представляет собой стрелку, сохраняющую структуру, между ее объектами-носителями.

Сохранение структуры в этом случае означает, что стрелка должна коммутировать с двумя структурными отображениями. Здесь начинает действовать функториальность: чтобы переключиться с одного структурного отображения на другое, должна существовать возможность поднять стрелку, которая проходит между их носителями.

Для заданного эндофунктора F, алгебраический *морфизм* между двумя алгебрами (a, α) и (b, β) представляет собой стрелку $f : a \to b$, которая делает коммутативной диаграмму:

$$\begin{array}{ccc} Fa & \xrightarrow{Ff} & Fb \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ a & \xrightarrow[f]{} & b \end{array}$$

Другими словами, должно выполняться следующее равенство:

$$f \circ \alpha = \beta \circ Ff$$

Композиция двух алгебраических морфизмов снова является алгебраическим морфизмом, что можно увидеть, соединив вместе две такие диаграммы (функторными отображениями композиции к композиции). Тождественная стрелка также является алгебраическим морфизмом, потому что

$$\mathsf{id}_a \circ \alpha = \alpha \circ F(\mathsf{id}_a)$$

(функтор отображает тождество в тождество).

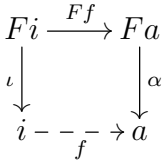
Коммутативное условие в определении алгебраического морфизма является очень ограничительным. Рассмотрим, например, функцию, которая отображает целое число в строку. В Haskell имеется функция `show` (фактически, метод класса `Show`), которая это делает. Это *не* алгебраический морфизм от `eval` к `pretty`.

Упражнение 12.2.1. *Покажите, что* `show` *не является алгебраическим морфизмом. Подсказка*: *рассмотрите, что происходит с узлом* `PlusF`.

Инициальная алгебра

Инициальный объект в категории алгебр, для заданного функтора F, называется *инициальной алгеброй* и, как мы увидим, играет очень важную роль.

По определению, инициальная алгебра (i, ι) имеет единственный алгебраический морфизм f от нее в любую другую алгебру (a, α). Схематически:



Этот единственный морфизм называется *катаморфизмом* алгебры (a, α).

Упражнение 12.2.2. *Определим две алгебры для следующего функтора*:

```
data FloatF x = Num Float | Op x x
```

Первая алгебраа:

```
addAlg          :: Algebra FloatF Float
addAlg (Num x)   = log x
addAlg (Op x y) = x + y
```

Вторя алгебра:

```
mulAlg          :: Algebra FloatF Float
mulAlg (Num x)   = x
mulAlg (Op x y) = x * y
```

Приведите убедительный аргумент того, что `log` (*логарифм*) — *это алгебраический морфизм между двумя определенными* (`Float` — *это встроенный тип числа с плавающей запятой*).

12.3 Лемма Ламбека и неподвижные точки

Лемма Ламбека утверждает, что структурное отображение ι инициальной алгебры является изоморфизмом.

Причиной этого является самоподобие алгебр. Можно поднять любую алгебру (a, α), используя F, и результат $(Fa, F\alpha)$ также будет алгеброй со структурным отображением $F\alpha : F(Fa) \to Fa$.

В частности, если поднять инициальную алгебру (i, ι), то получим новую алгебру с носителем Fi и структурным отображением $F\iota : F(Fi) \to Fi$. Отсюда следует, что должен существовать единственный алгебраический морфизм от инициальной алгебры в нее же:

$$\begin{array}{ccc} Fi & \xrightarrow{Fh} & F(Fi) \\ {\scriptstyle\iota}\downarrow & & \downarrow{\scriptstyle F\iota} \\ i & \xrightarrow[h]{} & Fi \end{array}$$

Этот h является обратным к ι. Чтобы убедиться в этом, рассмотрим композицию $\iota \circ h$ — стрелку в нижней части диаграммы

$$\begin{array}{ccccc} Fi & \xrightarrow{Fh} & F(Fi) & \xrightarrow{F\iota} & Fi \\ {\scriptstyle\iota}\downarrow & & \downarrow{\scriptstyle F\iota} & & \downarrow{\scriptstyle\iota} \\ i & \xrightarrow[h]{} & Fi & \xrightarrow[\iota]{} & i \end{array}$$

Это склеивание исходной диаграммы с тривиально коммутативной диаграммой. Следовательно, весь прямоугольник коммутативен. Можно интерпретировать это как $\iota \circ h$, являющийся алгебраическим морфизмом от (i, ι) к себе. Но такой алгебраический морфизм уже имеется — тождественность. Итак, в силу единственности отображения-вне инициальной алгебры, имеет место:

$$\iota \circ h = \mathsf{id}_i$$

Зная это, можно вернуться теперь к предыдущей диаграмме, из которой следует:

$$h \circ \iota = F\iota \circ Fh$$

Поскольку F — функтор, он отображает композицию в композицию, а тождество — в тождество. Поэтому правая часть есть:

$$F(\iota \circ h) = F(\mathsf{id}_i) = \mathsf{id}_{Fi}$$

Таким образом, показано, что h является обратным к ι, а это означает, что ι является изоморфизмом. Другими словами:

$$Fi \cong i$$

Мы интерпретируем это выражение как утверждение, что i является неподвижной точкой F (с точностью до изоморфизма). Действие F на i «не меняет его».

Может быть много неподвижных точек, но одна из них является *наименьшей неподвижной точкой*, потому что от нее существует алгебраический морфизм к любой другой неподвижной точке. Наименьшая неподвижная точка эндофунктора F обозначается μF, поэтому имеем:

$$i = \mu F$$

Неподвижная точка на Haskell

Рассмотрим, как определение неподвижной точки работает с нашим исходным примером (с эндофунктором):

```
data ExprF x = ValF Int | PlusF x x
```

Его неподвижная точка представляет собой структуру данных, определяемую тем свойством, что `ExprF`, воздействуя на нее, воспроизводит ее. Если обозначим эту неподвижную точку как `Expr`, уравнение с неподвижной точкой примет вид (в псевдо-Haskell):

```
Expr = ExprF Expr
```

Раскрывая `ExprF`, получаем:

```
Expr = ValF Int | PlusF Expr Expr
```

Сравните это с рекурсивным определением (фактический Haskell):

```
data Expr = ValF Int | Plus Expr Expr
```

Мы получили рекурсивную структуру данных как решение уравнения с неподвижной точкой.

На Haskell можно определить структуру данных с неподвижной точкой для любого функтора (или даже просто конструктора типа). Как мы увидим позже, это не всегда дает нам носитель инициальной алгебры. Это работает только для тех функторов, у которых есть компонент наподобие «лист».

Назовем `Fix f` неподвижной точкой функтора `f`. Символически, уравнение с неподвижной точкой можно записать как:

$$f(\mathrm{Fix}f) \cong \mathrm{Fix}f$$

или, в коде,

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

Конструктор данных `In` — это в точности структурное отображение инициальной алгебры, носителем которой является `Fix f`. Его инверсия:

```
out       :: Fix f -> f (Fix f)
out (In x) = x
```

Стандартная библиотека Haskell содержит более идиоматическое определение:

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Для создания термов типа `Fix f` часто используются «умные конструкторы». Например, с функтором `ExprF` мы бы определили:

```
val       :: Int -> Fix ExprF
val n      = In (ValF n)

plus      :: Fix ExprF -> Fix ExprF -> Fix ExprF
plus e1 e2 = In (PlusF e1 e2)
```

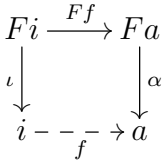
и использовали его для создания деревьев выражений, подобных этому:

```
e9 :: Fix ExprF
e9  = plus (plus (val 2) (val 3)) (val 4)
```

12.4 Катаморфизмы

Наша цель, как программистов, состоит в том, чтобы иметь возможность выполнять вычисления над рекурсивной структурой данных, чтобы «свернуть» ее. Теперь мы располагаем всеми необходимыми компонентами.

Структура данных определяется как неподвижная точка функтора. Алгебра для этого функтора определяет операцию, которую необходимо выполнить. Мы видели, как объединены неподвижная точка и алгебра:



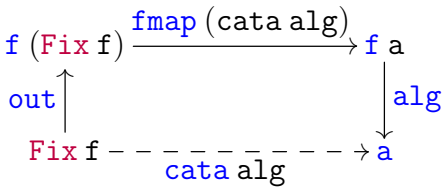
определяющее *катаморфизм* f для алгебры (a, α).

Последняя деталь информации — это лемма Ламбека, которая раскрывает, что ι может быть инвертирована, потому что это изоморфизм. Это означает, что можно представить эту диаграмму как:

$$f = \alpha \circ Ff \circ \iota^{-1}$$

интерпретируя это как рекурсивное определение f.

Изобразим эту же диаграмму, используя нотацию Haskell. Катаморфизм зависит от алгебры, поэтому, для алгебры с носителем `a` и вычислителем `alg`, имеем катаморфизм `cata alg`.



Просто, следуя по стрелкам, получаем следующее рекурсивное определение:

```
cata    :: Functor f => Algebra f a ->
                                Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

Вот что здесь происходит: мы применяем это определение к некоторому `Fix f`. Каждый `Fix f` получается применением `In` к заполненному функторами `Fix f`:

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

Функция `out` «обнажает» конструктор данных `In`:

```
out        :: Fix f -> f (Fix f)
out (In x) = x
```

Теперь, «вычисление» всех функторов из `Fix f` происходит путем применения `fmap` к `cata alg`. Это рекурсивное применение. Идея состоит в том, что деревья внутри функтора (по «объему») «меньше» исходного дерева, поэтому рекурсия в конечном итоге завершается, когда достигает листьев.

После этого шага остается функтор, полный значений, и к нему применяется вычислитель `alg`, для получения окончательного результата.

Мощь этого подхода в том, что вся рекурсия инкапсулирована в один тип данных и одну библиотечную функцию: имеем определение `Fix` и катаморфизм `cata`. Клиент библиотеки должен предоставить *нерекурсивные* части: функтор и алгебру. С этим справиться уже гораздо проще. Таким образом, мы разложили сложную проблему на более простые составляющие.

Примеры

Можно сразу же применить эту конструкцию к нашим предыдущим примерам. Проверьте, что:

```
cata eval e9
```

сводится к 9, а

```
cata pretty e9
```

имеет в качестве результата: `"2 + 3 + 4"`.

Иногда требуется отобразить дерево на нескольких строках с отступом. Это требует передачи счетчика глубины к рекурсивным вызовам. Есть прием, который использует функциональный тип в качестве носителя:

```
pretty'                :: Algebra ExprF
                          (Int -> String)
pretty' (ValF n) i     = indent i ++ show n
pretty' (PlusF f g) i = f (i + 1) ++ "\n" ++
                          indent i ++ "+" ++
                          "\n" ++ g (i + 1)
```

Вспомогательная функция `indent` повторяет символ пробела:

```
indent n = replicate (n * 2) ' '
```

Результат:

```
cata pretty' e9 0
```

на выводе имеет примерно такой вид:

```
    2
  +
    3
+
  4
```

Определим алгебры для других знакомых функторов. Неподвижная точка функтора `Maybe`:

```
data Maybe x = Nothing | Just x
```

после некоторого переименования, эквивалентна типу натуральных чисел

```
data Nat = Z | S Nat
```

Алгебра для этого функтора включает выбранный носитель `a` и вычислитель:

```
alg :: Maybe a -> a
```

Отображение-вне `Maybe` определяется двумя моментами: значением, соответствующим `Nothing`, и функцией `a -> a`, соответствующей `Just`. В нашем обсуждении типа натуральных чисел они названы `init` и `step`. Теперь понятно, что правило исключения для `Nat` является катаморфизмом для этой алгебры.

Списки как инициальные алгебры

Тип списка эквивалентен неподвижной точке следующего функтора, который параметризуется типом содержимого списка `a`:

```
data ListF a x = NilF | ConsF a x
```

Алгебра для этого функтора — это отображение

```
alg             :: ListF a c -> c
alg NilF         = init
alg (ConsF a c) = step (a, c)
```

которое определяется значением `init` и функцией `step`:

```
init :: c
step :: (a, c) -> c
```

Катаморфизмом такой алгебры является рекурсор списка:

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
```

где `(List a)` можно отождествить с неподвижной точкой `Fix (ListF a)`.

Мы уже рассматривали рекурсивную функцию, которая инвертирует список. Это было реализовано путем добавления элементов в конец списка, что очень неэффективно. Эту функцию легко переписать с помощью катаморфизма, хотя проблема остается.

С другой стороны, добавление элементов в начало намного более эффективно. Так что, улучшенный алгоритм должен сводиться к проходу по списку, накапливанием элементов в очереди, «первым пришел — первым обслужен», а затем извлечением их один за другим и добавлением в новый список.

Режим очереди можно реализовать с помощью композиции замыканий: каждое замыкание — это функция, которая запоминает свое окружение. Вот алгебра, носителем которой является функциональный тип:

```
revAlg             :: Algebra (ListF a)
                              ([a] -> [a])
revAlg NilF         = id
revAlg (ConsF a f) = \as -> f (a : as)
```

На каждом шаге эта алгебра создает новую функцию. Эта функция затем будет применять предыдущую функцию `f` к списку. Этот список есть результат добавления текущего элемента `a` к функциональному аргументу `as`. Результирующее замыкание запоминает текущий элемент `a` и предыдущую функцию `f`.

Катаморфизм для этой алгебры накапливает очередь таких замыканий. Чтобы инвертировать список, мы применяем результат катаморфизма для этой алгебры к пустому списку:

```
reverse   :: Fix (ListF a) -> [a]
reverse as = (cata revAlg as) []
```

Этот трюк лежит в основе функции левой свертки, `foldl`. При ее использовании следует соблюдать осторожность из-за опасности переполнения стека.

Списки настолько распространены, что их выделители (так называемые «свертки») включены в стандартную библиотеку. Но существует бесконечно много возможных рекурсивных структур данных, каждая из которых генерируется своим функтором, и мы можем использовать один и тот же катаморфизм для всех них.

Стоит отметить, что построение списка работает в любой моноидальной категории с копроизведениями. Можно заменить функтор списка более общим:

$$Fx = I + a \otimes x$$

где I — единичный объект, а $\otimes$ — тензорное произведение. Решение уравнения с неподвижной точкой:

$$L_a \cong I + a \otimes L_a$$

формально можно записать в виде ряда:

$$L_a = I + a + a \otimes a + a \otimes a \otimes a + \dots$$

Мы интерпретируем это как определение списка, который может быть пустым I, синглетоном a, двухэлементным списком $a \otimes a$ и т.д.

Кстати, если призадуматься, то это решение можно получить, выполнив последовательность формальных преобразований:

$$\begin{aligned} L_a &\cong I + a \otimes L_a \\ L_a - a \otimes L_a &\cong I \\ (I - a) \oplus L_a &\cong I \\ L_a &\cong I/(I - a) \\ L_a &\cong I + a + a \otimes a + a \otimes a \otimes a + \ldots \end{aligned}$$

где на последнем шаге используется формула суммы этого геометрического ряда. По общему признанию, промежуточные шаги не имеют смысла, поскольку на объектах не определены вычитание или деление, но конечный результат имеет смысл, в чем мы убедимся далее, и его можно сделать точным, рассматривая копредел цепочки объектов.

12.5 Инициальная алгебра из универсальности

Другой способ взглянуть на инициальную алгебру, по крайней мере, в **Set**, состоит в том, чтобы рассматривать ее как набор катаморфизмов, которые в целом намекают на существование лежащего в их основе объекта. Вместо того, чтобы рассматривать μF как множество деревьев, можно рассматривать его как множество функций от алгебр к их носителям.

В каком-то смысле, это еще одно проявление леммы Йонеды: каждая структура данных может быть описана либо отображениями-внутрь, либо отображениями-вне. Отображения-внутрь, в этом случае, являются конструкторами рекурсивной структуры данных. Отображения-вне — это все катаморфизмы, которые могут быть применены к этой структуре.

Прежде всего, сделаем явным полиморфизм в определении `cata`:

```
cata    :: Functor f => forall a.
                Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

а затем устраним аргументы. Получим:

```
cata'       :: Functor f => Fix f -> forall a.
                                Algebra f a -> a
cata' (In x) = \alg -> alg
                       (fmap (flip cata' alg) x)
```

Функция `flip` меняет порядок аргументов на обратный:

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

Это дает отображение от `Fix f` к множеству полиморфных функций.

И наоборот, если задана полиморфная функция типа:

```
forall a. Algebra f a -> a
```

можно реконструировать `Fix f`:

```
uncata     :: Functor f =>
              (forall a. Algebra f a -> a)
                                    -> Fix f
uncata alga = alga In
```

Фактически, эти две функции, `cata'` и `uncata`, являются обратными друг другу, устанавливая изоморфизм между `Fix f` и типом полиморфных функций:

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

Теперь можно заменить `Mu f` везде, где было использовано `Fix f`.

Свёртывать `Mu f` просто, так как `Mu` заключает в себе множество катаморфизмов:

```
cataMu           :: Algebra f a -> (Mu f -> a)
cataMu alg (Mu h) = h alg
```

Термы типа `Mu f`, например, для списков, можно построить с помощью рекурсии:

```
fromList   :: forall a. [a] -> Mu (ListF a)
fromList as = Mu h
  where h    :: forall x. Algebra (ListF a)
                                  x -> x
        h alg = go as
          where
            go []       = alg NilF
            go (n: ns) = alg (ConsF n (go ns))
```

Чтобы компиляция этого кода прошла успешно, необходимо использовать языковую прагму:

```
{-# language ScopedTypeVariables #-}
```

которая помещает переменную типа `a` в область видимости условия `where`.

Упражнение 12.5.1. *Создайте тест, который принимает список целых чисел, преобразует его к форме* `Mu` *и вычисляет сумму, используя* `cataMu`.

12.6 Инициальная алгебра как копредел

В общем случае, нет никакой гарантии, что инициальный объект в категории алгебр существует. Но если он существует, то лемма Ламбека утверждает, что это есть неподвижная точка эндофунктора для этих алгебр. Конструкция этой неподвижной точки немного загадочна, поскольку включает в себя завязывание рекурсивного узла.

Грубо говоря, неподвижная точка достигается после бесконечного применения функтора. Тогда еще одно его применение ничего не изменит. Бесконечность плюс единица — по-прежнему, бесконечность. Эта идея может быть уточнена, если осуществлять ее шаг за шагом. Для упрощения, рассмотрим алгебры в категории множеств, обладающей всеми ее замечательными свойствами.

В примерах было продемонстрировано, что создание экземпляров рекурсивных структур данных всегда начинается с листьев. Листья — это части в определении функтора, которые не зависят от параметра типа: `NilF` списка, `ValF` дерева, `Nothing` для `Maybe`, и т.д.

Мы можем выделить их, если применим наш функтор F к инициальному объекту — пустому множеству 0. Поскольку в пустом множестве нет элементов, экземпляры типа $F0$ — только листья.

Действительно, единственный обитатель типа `Maybe Void` построен из `Nothing`, а единственными обитателями типа `ExprF Void` являются `ValF n`, где `n` — значение типа `Int`.

Другими словами, $F0$ — это «тип листьев» для функтора F. Листья — это деревья глубины один. Для функтора `Maybe` имеется только один — тип листьев для этого функтора — синглетон:

```
m1 :: Maybe Void
m1  = Nothing
```

На второй итерации мы применяем F к листьям из предыдущего шага и получаем деревья глубины не более двух. Их тип — $F(F0)$.

Например, это все термы типа `Maybe(Maybe Void)`:

```
m2, m2' :: Maybe (Maybe Void)
m2       = Nothing
m2'      = Just Nothing
```

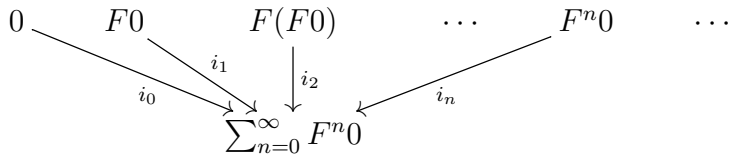
Можно продолжать этот процесс, добавляя все более глубокие деревья на каждом шаге. На n-ой итерации тип F^n0 (n-кратное применение F к инициальному объекту) описывает все деревья глубины до n. Однако, для каждого n, остается бесконечно много неохваченных деревьев глубины больше n.

Если бы мы знали, как определить $F^\infty 0$, то это охватило бы все возможные деревья. Следующее лучшее, что можно было бы попробовать, — это подсчитать все эти частичные деревья и сконструировать тип бесконечной суммы. Точно так же, как мы определили суммы двух типов, можно определить суммы многих типов, в том числе, когда их бесконечно много.

Бесконечная сумма (копроизведение):

$$\sum_{n=0}^{\infty} F^n 0$$

похожа на конечную сумму, за исключением того, что у нее бесконечно много конструкторов i_n:



Она обладает тем же универсальным свойством отображения-вне, как и сумма двух типов, только с бесконечным числом случаев (очевидно, это невозможно выразить на Haskell).

Чтобы построить дерево глубины n, нужно сначала выбрать его из F^n0 и использовать n-й конструктор i_n, чтобы ввести это дерево в сумму.

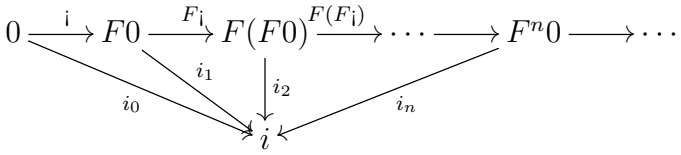
Остается только одна проблема: такое же дерево может быть построено с использованием любого из F^m0, где $m > n$.

Действительно, мы видели, как лист `Nothing` появляется в `Maybe Void` и `Maybe(Maybe Void)`. На самом деле, это проявляется в любой ненулевой степени `Maybe`, действующей на `Void`.

Точно так же, `Just Nothing` появляется при всех степенях, начиная со второй. `Just(Just(Nothing))` появляется во всех степенях, начиная с третьей, и т.д.

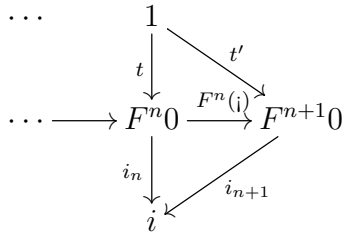
Но есть способ избавиться от всех этих копий, если сумму заменить некоторым копределом. Вместо диаграммы, состоящей из дискретных объектов, можно построить цепочку. Обозначим эту цепочку Γ, а ее копредел i:

$$i = Colim\,\Gamma$$



Это почти то же самое, что и сумма, но с дополнительными стрелками в основании ко-конуса. Эти стрелки являются кумулятивными поднятиями единственной стрелки ¡, которая идет от инициального объекта к $F0$ (в Haskell для этого используется `absurd`). Эффект этих стрелок состоит в том, чтобы сжать множество бесконечного числа копий одного и того же дерева до одного представителя.

Чтобы убедиться в этом, рассмотрим, например, дерево глубины n. Сначала его можно найти, как элемент, в $F^n 0$, , то есть как стрелку $t : 1 \to F^n 0$. Она вводится в копредел i как компоновка $i_n \circ t$.



Такая же форма дерева встречается и в $F^{n+1} 0$, как компоновка $t' = F^n(¡) \circ t$. Она вводится в копредел как компоновка $i_{n+1} \circ t' = i_{n+1} \circ F^n(¡) \circ t$.

На этот раз, однако, имеется коммутативный треугольник — грань ко-конуса:

$$i_{n+1} \circ F^n(¡) = i_n$$

который означает, что:

$$i_{n+1} \circ t' = i_{n+1} \circ F^n(¡) \circ t = i_n \circ t$$

Итак, две копии дерева были идентифицированы в копределе. Можно убедиться, что эта процедура удаляет все дубликаты.

Непосредственно можно доказать, что $i = Colim\,\Gamma$ — инициальная алгебра. Однако, требуется сделать одно предположение: функтор F должен сохранять копредел. Копредел $F\Gamma$ должен быть равен F_i.

$$Colim(F\Gamma) \cong Fi$$

К счастью, это предположение выполняется в **Set**.

Приведем набросок доказательства: сначала построим стрелку $i \to Fi$, а затем стрелку в обратном направлении (пропускаем доказательство того, что они обратны друг другу).

Начнем с универсальности копредела. Если можно построить ко-конус от цепочки Γ к $Colim\,(F\Gamma)$, то в силу универсальности должна существовать стрелка от i к $Colim\,(F\Gamma)$. А последний, по нашему предположению, что F сохраняет ко-пределы, изоморфен Fi. Итак, имеем отображение $i \to Fi$.

Чтобы построить этот ко-конус, заметим, что $Colim\,(F\Gamma)$, по определению, является вершиной ко-конуса $F\Gamma$.

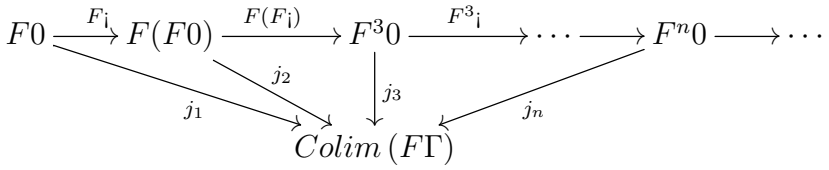
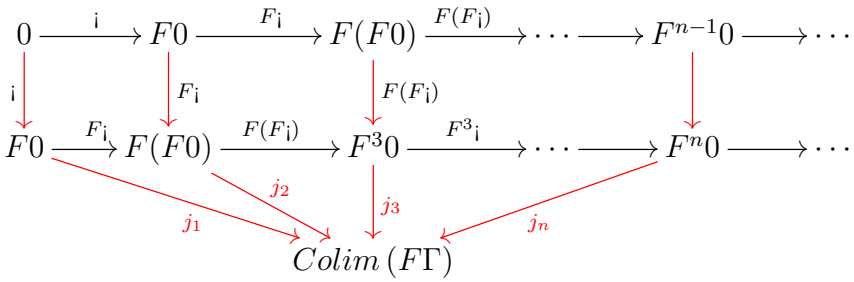


Диаграмма $F\Gamma$ такая же, как Γ, за исключением того, что в ней отсутствует голый инициальный объект в начале цепочки.

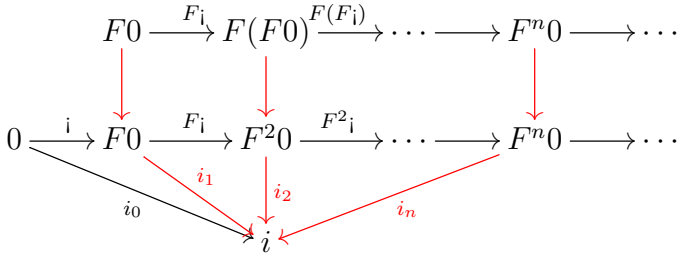
Спицы искомого ко-конуса отмечены красным цветом:



Поскольку $i = Colim\,\Gamma$ есть вершина универсального ко-конуса, основанного на Γ, то должно существовать единственное отображение-вне от него к $Colim\,(F\Gamma)$, которое, как мы сказали, равно Fi:

$$i \to Fi$$

Далее, заметим, что цепочка $F\Gamma$ является подцепью Γ, поэтому ее можно встроить в нее. Это означает, что можно построить ко-конус от $F\Gamma$ к вершине i, пройдя через (подцепь) Γ.



Из универсальности $Colim\,(F\Gamma)$ следует, что существует отображение-вне

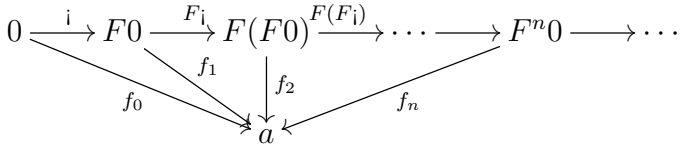
$$Colim\,(F\Gamma) \to i$$

и, таким образом, имеется отображение в другом направлении:

$$Fi \to i$$

Это показывает, что i является носителем алгебры. На самом деле можно показать, что эти два отображения обратны друг другу, как и следовало ожидать из леммы Ламбека.

Чтобы показать, что это действительно инициальная алгебра, надо построить отображение-вне от нее к произвольной алгебре $(a, \alpha : Fa \to a)$. Опять же, можно использовать универсальность, пока можно построить ко-конус от Γ к a.



Нулевая спица этого ко-конуса идет от 0 к a, так что, это просто $f_0 = ¡$. Первая спица, $F_0 \to a$, есть $f_1 = \alpha \circ Ff_0$, поскольку $Ff_0 : F0 \to Fa$ и $\alpha : Fa \to a$. Третья спица, $F(F0) \to a$ — это $f_2 = \alpha \circ Ff_1$, и т.д.

Тогда единственным отображением от i к a является наш катаморфизм. Еще немного произведя диаграммный поиск, можно показать, что это действительно алгебраический морфизм.

Заметим еще раз, что эта конструкция работает только в том случае, если можно «запустить» процесс, создавая листья функтора. Если, с другой стороны, $F0 \cong 0$, то листья отсутствуют, и все дальнейшие итерации будут просто воспроизводить 0.

Глава 13

Коалгебры

Коалгебры — это просто алгебры из противоположной категории. Конец главы!

Но, может быть, и нет. Как мы уже видели, категория, с которой мы работаем, не симметрична по отношению к двойственности. В частности, если сравнивать терминальные и инициальные объекты, то их свойства не симметричны. Наш инициальный объект не имеет входящих стрелок, тогда как терминальный, кроме уникальных входящих стрелок, имеет множество исходящих стрелок.

Поскольку инициальные алгебры были построены, начиная с инициального объекта, мы могли бы ожидать, что терминальные коалгебры — их двойственные, порожденные из терминального объекта — будут не просто их зеркальными отражениями, а будут добавлять свои собственные характерные особенности.

Мы видели, что основное применение алгебр заключалось в обработке рекурсивных структур данных, в их свертывании. Двойственно, основное применение коалгебр заключается в создании или развертывании рекурсивных древовидных структур данных. Развертка осуществляется с помощью анаморфизма.

Мы используем катаморфизмы, чтобы «рубить» деревья, а анаморфизмы — чтобы их выращивать.

Мы не можем получать информацию из ничего, поэтому, как правило, и катаморфизм, и анаморфизм, уменьшают количество информации, содержащейся на их входе.

После суммирования содержимого списка целых чисел невозможно восстановить исходный список.

Точно так же, если вы «выращиваете» рекурсивную структуру данных, используя анаморфизм, начальное значение должно содержать всю информацию, которая обеспечивает создание дерева. Вы не получаете новую информацию, но преимущество в том, что информация теперь хранится в более удобном для дальнейшей обработки виде.

13.1 Коалгебры из эндофункторов

Коалгебра для эндофунктора F — это пара, состоящая из носителя a и структурного отображения — стрелки $a \to Fa$.

На Haskell, определяем:

```
type Coalgebra f a = a -> f a
```

Носитель можно понимать как тип источника, из которого мы будем выращивать структуру данных, будь то список или дерево.

Например, функтор, который можно использовать для создания бинарного дерева с целыми числами, хранящимися в узлах, имеет вид:

```
data TreeF x = LeafF | NodeF Int x x
  deriving (Show, Functor)
```

Нам даже не нужно определять для него экземпляр `Functor` — предложение `deriving` сигнализирует компилятору сгенерировать канонический экземпляр (вместе с экземпляром `Show`, чтобы разрешить преобразование в `String`, если потребуется его отобразить).

Коалгебра — это функция, которая принимает начальное значение типа носителя и производит функтор, полный новых начальных значений. Затем эти значения можно использовать для рекурсивного создания поддеревьев.

К примеру, коалгебра для функтора `TreeF`, которая принимает список целых чисел в качестве начального значения:

```
split          :: Coalgebra TreeF [Int]
split []        = LeafF
split (n : ns) = NodeF n left right
  where
    (left, right) = partition (<= n) ns
```

Если начальное значение пусто, порождается лист; в противном случае создается новый узел. Этот узел хранит заголовок списка и заполняет узел двумя новыми начальными значениями. Библиотечная функция `partition` разбивает список с помощью определяемого пользователем предиката, здесь `(<= n)`, меньше либо равно `n`. Результатом является пара списков: первый удовлетворяет предикату, а второй — нет.

Вы можете убедиться, что рекурсивное применение этой коалгебры создает бинарное отсортированное дерево. Позже мы воспользуемся этой коалгеброй для реализации сортировки.

13.2 Категория коалгебр

По аналогии с морфизмами алгебр можно определить коалгебраические морфизмы, как стрелки, удовлетворяющие условию коммутативности, между носителями.

Для двух коалгебр (a, α) и (b, β), стрелка $f : a \to b$ является морфизмом коалгебры, если следующая диаграмма коммутативна:

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ Fa & \xrightarrow[Ff]{} & Fb \end{array}$$

Интерпретация заключается в том, что не имеет значения, сначала мы отображаем носители, а затем применяем коалгебру β, или сначала применяем коалгебру α, а затем применяем стрелку к ее содержимому, используя подъем Ff.

Коалгебраические морфизмы могут быть скомпонованы, а тождественная стрелка автоматически является морфизмом коалгебры. Понятно, что коалгебры, как и алгебры, образуют категорию.

Однако, на этот раз нас интересует терминальный объект в этой категории — *терминальная коалгебра*. Если терминальная коалгебра (t, τ) существует, то она удовлетворяет двойственной лемме Ламбека.

Упражнение 13.2.1. *Лемма Ламбека*: *покажите, что структурное отображение τ терминальной коалгебры (t, τ) является изоморфизмом. Подсказка*: *доказательство двойственно доказательству для инициальной алгебры.*

Вследствие леммы Ламбека носителем терминальной алгебры является неподвижная точка рассматриваемого эндофунктора

$$Ft \cong t$$

причем τ и τ^{-1} служат свидетельствами этого изоморфизма.

Отсюда также следует, что (t, τ^{-1}) — алгебра; точно так же, как (i, ι^{-1}) является коалгеброй, в предположении, что (i, ι) является инициальной алгеброй.

Мы уже видели, что носитель инициальной алгебры является неподвижной точкой. В принципе, у одного и того же эндофунктора может быть много неподвижных точек. Инициальная алгебра — это наименьшая неподвижная точка, а терминальная коалгебра — наибольшая неподвижная точка.

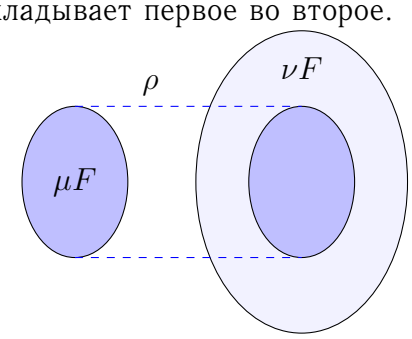
Наибольшая неподвижная точка эндофунктора F обозначается νF, поэтому имеем:

$$t = \nu F$$

Также понятно, что должен существовать единственный алгебраический морфизм (катаморфизм) от инициальной алгебры к терминальной коалгебре. Это связано с тем, что терминальная коалгебра является алгеброй.

Точно так же, существует единственный морфизм коалгебр от инициальной алгебры (которая также является коалгеброй) к терминальной коалгебре. На самом деле, можно показать, что в обоих случаях это один и тот же основной морфизм $\rho : \mu F \to \nu F$.

В категории множеств, множество носителей инициальной алгебры является подмножеством множества носителей терминальной коалгебры, причем функция ρ вкладывает первое во второе.



Далее мы увидим, что в Haskell эта ситуация более тонкая, из-за ленивых вычислений. Но, по крайней мере, для функторов, имеющих ли-

стовую компоненту (то есть, их действие на инициальный объект нетривиально), тип неподвижной точки в Haskell работает как носитель, как для инициальной алгебры, так и для терминальной коалгебры.

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

Упражнение 13.2.2. *Покажите, что для тождественного функтора в* **Set**, *каждый объект является неподвижной точкой, пустое множество является наименьшей неподвижной точкой, а одноэлементное множество является наибольшей неподвижной точкой. Подсказка: от наименьшей неподвижной точки стрелки должны вести ко всем остальным неподвижным точкам, а наибольшая неподвижная точка должны быть "мишенью" для стрелок, исходящих от всех остальных неподвижных точек.*

Упражнение 13.2.3. *Покажите, что пустое множество является носителем инициальной алгебры для тождественного функтора в* **Set**. *Двойственно, покажите, что одноэлементное множество является терминальной коалгеброй этого функтора. Подсказка*: *покажите, что единственные стрелки действительно являются* (ко-) *алгебраическими морфизмами.*

13.3 Анаморфизмы

Терминальная коалгебра (t, τ) определяется своим универсальным свойством: существует единственный коалгебраический морфизм h от любой коалгебры (a, α) к (t, τ). Этот морфизм называется *анаморфизмом*. Будучи морфизмом коалгебры, он делает следующую диаграмму коммутативной:

$$\begin{array}{ccc} a & \overset{h}{\dashrightarrow} & t \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\tau} \\ Fa & \xrightarrow[Fh]{} & Ft \end{array}$$

Как и в случае с алгебрами, можно использовать лемму Ламбека, чтобы «решить» для `h` уравнение:

$$h = \tau^{-1} \circ Fh \circ \alpha$$

Поскольку терминальная коалгебра (точно так же, как и инициальная алгебра) является неподвижной точкой функтора, приведенную выше рекурсивную формулу можно напрямую перевести на Haskell как:

```
ana     :: Functor f => Coalgebra f a ->
                                  a -> Fix f
ana coa = In . fmap (ana coa) . coa
```

Интерпретация этой формулы: если задано начальное значение типа `a`, то сначала подействуем на него коалгеброй `coa`. Это дает функтор начальных значений. Мы расширяем эти начальные значения, рекурсивно применяя анаморфизм с помощью `fmap`. Затем мы применяем конструктор `In` для получения окончательного результата.

В качестве примера, мы можем применить анаморфизм к коалгебре `split`, которая была определена ранее: `ana split` принимает список целых чисел и создает отсортированное дерево.

Затем можем использовать катаморфизм, чтобы свернуть это дерево в отсортированный список. Определим алгебру:

```
toList                :: Algebra TreeF [Int]
toList LeafF           = []
toList (NodeF n ns ms) = ns ++ [n] ++ ms
```

которая соединяет левый список с одноэлементной основой и правым списком. Чтобы отсортировать список, мы комбинируем анаморфизм с катаморфизмом:

```
qsort = cata toList . ana split
```

Это дает (очень неэффективную) реализацию быстрой сортировки. Мы вернемся к ней в следующем разделе.

Бесконечные структуры данных

При изучении алгебр мы полагались на структуры данных, которые имели листовые компоненты — эндофункторы, которые при воздействии на инициальный объект дают результат, отличный от инициального объекта. При построении рекурсивных структур данных нам нужно было с чего-то начинать, а это означало, что сначала нужно создать листья.

С коалгебрами можно отказаться от этого требования. Нам больше не нужно создавать рекурсивные структуры данных «вручную» — имеются анаморфизмы, которые сделают это самостоятельно. Эндофунктор, у которого нет листьев, вполне приемлем: его коалгебры будут генерировать бесконечные структуры данных.

Бесконечные структуры данных могут быть представлены на Haskell, благодаря механизму ленивости. Вычисляются только те части бесконечной структуры данных, которые явно требуются; работа с остальными погружено в анабиоз.

Чтобы реализовать бесконечные структуры данных в строгих языках, нужно прибегнуть к представлению значений в виде функций — то, что Haskell делает «за кулисами» (эти функции называются *преобразователями*).

Рассмотрим простой пример: бесконечный поток значений. Чтобы сгенерировать его, сначала определим функтор, очень похожий на тот, который был использован для генерации списков, за исключением того, что в нем отсутствует листовой компонент (конструктор пустого списка). Можно распознать в нем функтор-произведение с фиксированным первым компонентом — полезной нагрузкой потока:

```
data StreamF a x = StreamF a x
  deriving Functor
```

Бесконечный поток является неподвижной точкой этого функтора.

```
type Stream a = Fix (StreamF a)
```

Вот простая коалгебра, в которой в качестве начального значения используется одно целое число `n`:

```
step  :: Coalgebra (StreamF Int) Int
step n = StreamF n (n + 1)
```

Она сохраняет начальное значение в качестве полезной нагрузки и задает следующий порождающий поток с `n + 1`.

Анаморфизм для этой коалгебры с начальным нулем порождает поток всех натуральных чисел.

```
allNats :: StreamF Int
allNats  = ana step 0
```

В неленивом языке этот анаморфизм будет работать вечно, но в Haskell он выполняется мгновенно. Возрастающая цена оплачивается только тогда, когда мы хотим получить некоторые данные, например, с помощью таких методов доступа:

```
head                    :: Stream a -> a
head (In (StreamF a _)) = a

tail                    :: Stream a -> Stream a
tail (In (StreamF _ s)) = s
```

13.4 Гиломорфизмы

Тип выхода анаморфизма — неподвижная точка функтора, который имеет тот же тип, что и вход катаморфизма. На Haskell они оба описываются одним и тем же типом данных `Fix f`. Поэтому их можно компоновать, как это было сделано при реализации быстрой сортировки. На самом деле, можно объединить коалгебру с алгеброй в одну рекурсивную функцию, называемую *гиломорфизмом*:

```
hylo         :: Functor f => Algebra f b ->
                             Coalgebra f a -> a -> b
hylo alg coa = alg . fmap (hylo alg coa) . coa
```

Можно переписать быструю сортировку, используя гиломорфизм:

```
qsort = hylo toList split
```

Заметим, что в определении гиломорфизма нет следов неподвижной точки. Концептуально, коалгебра используется для построения (развертывания) рекурсивной структуры данных из начального значения, а алгебра используется для ее свертывания в значение типа `b`. Но, из-за лености Haskell, промежуточная структура данных может не полностью материализоваться в памяти. Это особенно важно при работе с очень большими промежуточными деревьями. Оцениваются только ветки, которые в данный момент проходятся, и, как только они будут обработаны, они передадутся сборщику мусора.

Гиломорфизмы в Haskell — удобная замена рекурсивным алгоритмам поиска с возвратом, которые очень сложно корректно реализовать

на императивных языках. Мы используем тот факт, что проектирование структуры данных проще, чем отслеживание сложного потока управления и отслеживание нашего места в рекурсивном алгоритме.

Таким образом, структуры данных можно использовать для визуализации сложных потоков управления.

Несоответствие импеданса

Мы видели, что в категории множеств инициальные алгебры не обязательно совпадают с терминальными коалгебрами. Например, тождественный функтор имеет пустое множество в качестве носителя инициальной алгебры и одноэлементное множество в качестве носителя своей терминальной коалгебры.

Имеются и другие функторы, не имеющие листовых компонентов, например функтор потока. Инициальной алгеброй для такого функтора также является пустое множество.

В **Set**, инициальная алгебра является подмножеством терминальной коалгебры, и гиломорфизмы могут быть определены только для этого подмножества. Это означает, что можно использовать гиломорфизм только в том случае, если анаморфизм для конкретной коалгебры приводит в это подмножество. В этом случае, поскольку вложение инициальных алгебр в терминальные коалгебры инъективно, можно найти соответствующий элемент в инициальной алгебре и применить к нему катаморфизм.

Однако, в Haskell имеется один тип `Fix f`, комбинирующий, как инициальную алгебру, так и терминальную коалгебру. Именно здесь упрощенная интерпретация типов Haskell как множеств значений не работает. Рассмотрим простую алгебру потока:

```
add                 :: Algebra (StreamF Int) Int
add (StreamF n sum) = n + sum
```

Ничто не мешает использовать гиломорфизм для вычисления суммы всех натуральных чисел:

```
sumAllNats :: Int
sumAllNats  = hylo add step 1
```

Это прекрасно сформированная программа на Haskell, которая проходит проверку типов. Итак, какое значение она производит, когда мы ее запускаем? (Подсказка: это не $-1/12$) Ответ: мы не знаем, потому что эта программа никогда не завершится. Она упирается в бесконечную рекурсию и, в конечном итоге, истощает ресурсы компьютера.

Это аспект реальных вычислений, которые не могут моделировать простые функции между множествами. Некоторые компьютерные функции могут никогда не завершиться.

Рекурсивные функции формально описываются *теорией областей* как пределы частично определенных функций. Если функция не определена для конкретного значения аргумента, говорят, что она возвращает нижнее значение $\bot$. Если мы включим их как специальные элементы каждого типа (они тогда называются *поднятыми* типами), то можем сказать, что функция `sumAllNats` возвращает нижнее значение типа `Int`. В общем, катаморфизмы для бесконечных типов не завершаются, поэтому можно рассматривать их как возвращающиеся нижние значения.

Однако, следует отметить, что включение нижних значений усложняет категорную интерпретацию Haskell. В частности, многие универсальные конструкции, основанные на единственности отображений, больше не работают так, как было объявлено.

Суть в том, что код на Haskell следует рассматривать как иллюстрацию категорных концепций, а не как источник строгих доказательств.

13.5 Терминальная коалгебра из универсальности

Определение анаморфизма можно рассматривать как выражение универсального свойства терминальной коалгебры, с явно выраженной универсальной количественной характеристикой:

```
ana     :: Functor f => forall a.
             Coalgebra f a -> (a -> Fix f)
ana coa = In . fmap (ana coa) . coa
```

Оно информирует, что для любой коалгебры существует отображение от ее носителя к носителю терминальной коалгебры `Fix f`. Из леммы

Ламбека известно, что это отображение на самом деле является коалгебраическим морфизмом.

Приведем это определение к виду:

```
ana          :: Functor f => forall a.
                             (a -> f a, a) -> Fix f
ana (coa, x) = In (fmap (curry ana coa) (coa x))
```

Мы можем использовать эту формулу в качестве альтернативы определения носителя для терминальной коалгебры. Можно заменить `Fix f` определяемым нами типом — обозначим его `Nu f`. Сигнатура типа:

```
forall a. (a -> f a, a) -> Nu f
```

говорит, что можно построить элемент `Nu f` из пары `(a -> f a, a)`. Он выглядит так же, как конструктор данных, за исключением того, что он полиморфен в `a`.

Типы данных с полиморфным конструктором называются *экзистенциальными типами*. В псевдокоде (не на Haskell) мы бы определили `Nu f` как:

```
data Nu f = Nu (exists a. (Coalgebra f a, a))
```

Сравните это с определением наименьшей неподвижной точки алгебры:

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

Чтобы построить элемент экзистенциального типа, имеется возможность выбрать наиболее удобный тип — тип, для которого есть данные, требуемые конструктором.

Например, можно построить терм типа `Nu (StreamF Int)`, выбрав `Int` в качестве удобного типа и предоставив пару:

```
nuArgs :: (Int -> StreamF Int Int, Int)
nuArgs  = (\n  -> StreamF n (n+1) , 0)
```

Клиенты экзистенциального типа данных понятия не имеют, какой тип использовался при его построении. Все, что они знают, это то, что такой тип *существует* (exists) — отсюда и название. Если они хотят

использовать экзистенциальный тип, они должны делать это так, чтобы не зависеть от выбора, сделанного при его построении. На практике это означает, что экзистенциальный тип должен содержать в себе как производителя, так и потребителя скрытого значения.

В нашем примере это действительно так: производитель — это просто значение типа `a`, а потребитель — функция `a -> f a`.

Наивно: все, что клиенты могли сделать с этой парой, не зная, что такое тип `a`, — это применить функцию к значению. Но если `f` — функтор, они смогут делать гораздо больше. Они могут повторить процесс, применив поднятую функцию к содержимому `f a`, и т.д. В итоге, они получают всю информацию, содержащуюся в бесконечном потоке.

Haskell поддерживает несколько способов определения экзистенциальных типов данных. Мы можем использовать не-каррированную версию анаморфизма непосредственно в качестве конструктора данных:

```
data Nu f where
     Nu :: forall a f. (a -> f a, a) -> Nu f
```

Отметим, что на Haskell, если мы явно квантифицируем один тип, все другие переменные типа также должны быть квантифицированы: здесь, это конструктор типа `f` (однако, `Nu f` не является экзистенциальным в `f`, так как это явный параметр).

Мы также можем вообще не использовать квантификацию:

```
data Nu f where
     Nu :: (a -> f a, a) -> Nu f
```

Это связано с тем, что переменные типа, которые не являются аргументами конструктора типа, автоматически обрабатываются как экзистенциальные.

Также можно использовать более традиционную форму:

```
data Nu f = forall a. Nu (a -> f a, a)
```

(это требует квантификации `a`).

На момент написания этого текста было предложение ввести в Haskell ключевое слово `exists`, которое заставит работать это определение

```
data Nu f = Nu (exists a. (a -> f a, a))
```

(далее мы увидим, что экзистенциальные типы данных соответствуют ко-концам из теории категорий).

Конструктор `Nu f` — это буквально (некаррированный) анаморфизм:

```
anaNu      :: Coalgebra f a -> a -> Nu f
anaNu coa a = Nu (coa, a)
```

Если дан поток в виде `Nu (Stream a)`, то можно получить доступ к его элементу, используя функции доступа. Следующая функция извлекает первый элемент:

```
head                :: Nu (StreamF a) -> a
head (Nu (unf, s))  =
  let (StreamF a _) = unf s
  in a
```

а эта — продвигает поток:

```
tail                 :: Nu (StreamF a) -
> Nu (StreamF a)
tail (Nu (unf, s))   =
  let (StreamF _ s') = unf s
  in Nu (unf, s')
```

Вы можете протестировать их на бесконечном потоке целых чисел:

```
allNats = Nu nuArgs
```

13.6 Терминальная коалгебра как предел

В теории категорий мы не сторонимся бесконечностей — мы придаем им смысл.

На первый взгляд идея о том, что можно построить терминальную коалгебру, применяя функтор F бесконечно много раз к некоторому объекту, скажем, терминальному объекту 1, не имеет смысла. Но идея очень убедительна: применить F еще раз — все равно, что прибавить единицу к бесконечности — это все еще бесконечность. Итак, наивно, это неподвижная точка F:

$$F(F^{\infty}1) \cong F^{\infty+1}1 \cong F^{\infty}1$$

Чтобы превратить это свободное рассуждение в строгое доказательство, надо «укротить» бесконечность, а значит, необходимо определить некую предельную процедуру.

В качестве примера рассмотрим функтор произведения:

$$F_a x = a \times x$$

Его терминальная коалгебра представляет собой бесконечный поток. Аппроксимируем это, начав с терминального объекта 1. Следующий шаг:

$$F_a 1 = a \times 1 \cong a$$

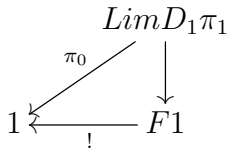
который можно было бы представить, является потоком длины один. Можно продолжить:

$$F_a(F_a 1) = a \times (a \times 1) \cong a \times a$$

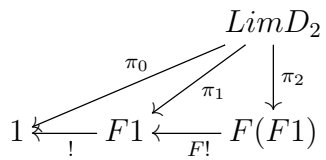
— потоком длины два и т.д.

Это выглядит многообещающе, но нужен один объект, который объединил бы все эти приближения. Нам нужен способ «приклеить» следующее приближение к предыдущему.

Напомним из одного предыдущего упражнения о пределе диаграммы «шагающая стрелка». Этот предел имеет те же элементы, что и исходный объект на диаграмме. В частности, рассмотрим предел на следующей диаграмме:

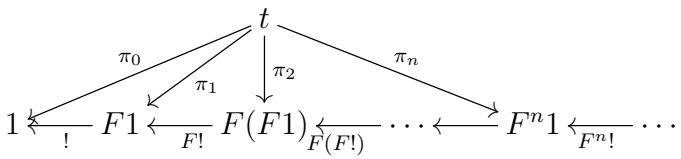


(! — единственный морфизм, нацеленный на терминальный объект 1). Этот предел имеет те же элементы, что и $F1$. Точно так же, предел:



имеет те же элементы, что и $F(F1)$.

Можно продолжать расширять эту диаграмму до бесконечности. Предел бесконечной цепи является носителем неподвижной точки терминальной коалгебры.



Доказательство этого факта можно получить из аналогичного доказательства для инициальных алгебр, обращением стрелок.

Глава 14

Монады

Что общего у автомобильного колеса, глиняного горшка и деревянного дома? Все они полезны из-за пустоты внутри.

Лао-цзы говорит: «Ценность происходит из того, что есть, а польза — от того, чего нет».

Что общего у функтора `Maybe`, функтора списка и функтора `Reader`? Все они пусты внутри.

Когда монады объясняются в контексте программирования, трудно увидеть общий шаблон, когда вы фокусируетесь на функторах. Чтобы понять монады, вы должны заглянуть внутрь функторов и к месту соединения между функциями.

14.1 Программирование с побочными эффектами

До сих пор речь шла о программировании в терминах вычислений, которые моделировались в основном функциями между множествами (за исключением не-завершения). В программировании такие функции называются *тотальными* и *чистыми*.

Тотальная функция определяется для всех значений ее аргументов.

Чистая функция реализуется исключительно с точки зрения ее аргументов и, в случае замыканий, захваченных значений — у нее нет доступа к внешнему миру, а тем более возможности изменять его.

Однако большинству реальных программ приходится взаимодействовать с внешним миром: они считывают и записывают файлы, обраба-

тывают сетевые пакеты, запрашивают у пользователей данные и т.д. Большинство языков программирования решают эту проблему, допуская побочные эффекты. Побочный эффект — это все то, что нарушает полноту или чистоту функции.

К сожалению, побочные эффекты, принятые в императивных языках, чрезвычайно усложняют рассуждения о программах. При составлении эффективных вычислений необходимо тщательно обдумывать состав эффектов в каждом конкретном случае. Еще более усложняет ситуацию то, что большинство эффектов скрыто внутри реализации функции и всех функций, которые она вызывает рекурсивно.

Решение, принятое чисто функциональными языками, такими как Haskell, состоит в том, чтобы закодировать побочные эффекты в возвращаемом типе чистой функции. Удивительно, но это возможно для всех соответствующих эффектов.

Идея состоит в том, что, вместо вычисления типа `a -> b` с побочными эффектами, используется функция `a -> f b`, где конструктор типа `f` кодирует соответствующий эффект. В этой ситуации на `f` не накладывается никаких условий. Это даже не обязательно должен быть `Functor`, не говоря уже о монаде. Детали будут обсуждаться позже, когда мы поговорим о композиции эффектов.

Ниже приведена характеристика общих эффектов и их чисто функциональных версий.

Частичность

В императивных языках, частичность функции часто кодируется с помощью исключений. Когда функция вызывается с «недопустимым» значением аргумента, она генерирует исключение. В некоторых языках тип исключения кодируется в сигнатуре функции с помощью специального синтаксиса.

На Haskell, частичное вычисление может быть реализовано функцией, возвращающей результат внутри функтора `Maybe`. Такая функция при вызове с «неправильным» аргументом возвращает `Nothing`, иначе оборачивает результат в конструктор `Just`.

Если мы хотим закодировать больше информации о типе сбоя, то можем использовать функтор `Either`, с `Left`, традиционно передающим данные об ошибке (часто, просто `String`), и `Right`, инкапсулирующим действительный ответ, если таковой имеется.

Вызывающий функцию с `Maybe`-значением не может просто игнорировать исключительное условие. Чтобы извлечь значение, он должны сопоставить результат с шаблоном и решить, что делать с `Nothing`. Это контрастирует с «необладанием» `Maybe` некоторых императивных языков, где условие ошибки кодируется с помощью нулевого указателя.

Протоколирование

Иногда вычисление должно регистрировать некоторые значения в некоторой внешней структуре данных. Ведение журнала или *протоколирование* — это побочный эффект, который особенно опасен в параллельных программах, когда несколько потоков могут одновременно пытаться получить доступ к одному и тому же журналу.

Простое решение состоит в том, чтобы функция возвращала вычисленное значение в паре с протоколируемым элементом. Другими словами, протоколирование вычисления типа `a -> b` можно заменить чистой функцией:

```
a -> Writer w b
```

где функтор `Writer` — это тонкая инкапсуляция произведения:

```
newtype Writer w a = Writer (a, w)
```

где `w` — тип журнала.

Затем вызывающая сторона этой функции отвечает за извлечение запротоколированного значения. Это распространенный прием: заставить функцию предоставлять все данные, а вызывающая сторона должна обрабатывать эффекты.

Внешняя среда

Для некоторых вычислений требуется доступ только для чтения к некоторым внешним данным, хранящимся в среде. Среда, предназначенная только для чтения, вместо того, чтобы быть тайно доступной для вычислений, может быть просто передана функции в качестве дополнительного аргумента. Если имеется вычисление `a -> b`, которому требуется доступ к некоторому окружению `e`, мы заменяем его функцией `(a, e)`

`-> b`. Сначала кажется, что это не соответствует шаблону кодирования побочных эффектов в возвращаемом типе. Однако такую функцию всегда можно привести к форме:

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

Возвращаемый тип этой функции может быть закодирован в функторе считывания, который сам параметризуется типом среды `e`:

```
newtype Reader e a = Reader (e -> a)
```

Это пример отсроченного побочного эффекта. Функция:

```
a -> Reader e a
```

не хочет иметь дело с эффектами, поэтому делегирует ответственность вызывающей стороне. Это можно представить себе как создание сценария (скрипта), который будет выполнен позже. Функция `runReader` играет роль очень простого интерпретатора такого сценария:

```
runReader             :: Reader e a -> e -> a
runReader (Reader h) e = h e
```

Состояние

Наиболее распространенный побочный эффект связан с доступом и возможным изменением некоторого общего состояния. К сожалению, совместно используемое состояние является печально известным источником ошибок параллелизма. Это серьезная проблема в объектно-ориентированных языках, где объекты с состоянием могут быть прозрачно разделены между многими клиентами. В Java такие объекты могут быть снабжены отдельными взаимными исключениями (mutexes) за счет ухудшения производительности и риска взаимоблокировок.

В функциональном программировании операции с состоянием делаются явными: состояние передается в качестве дополнительного аргумента и возвращается измененное состояние в паре со значением. Мы заменяем вычисление с сохранением состояния `a -> b` на

```
(a, s) -> (b, s)
```

где `s` — тип состояния. Как и прежде, можно каррировать такую функцию, приводя ее к форме

```
a -> (s -> (b, s))
```

Этот возвращаемый тип может быть инкапсулирован в следующем функторе:

```
newtype State s a = State (s -> (a, s))
```

Предполагается, что вызывающая сторона такой функции извлекает результат и измененное состояние, предоставляя начальное состояние и вызывая вспомогательную функцию, интерпретатор `runState`:

```
runState             :: State s a -> s -> (a, s)
runState (State h) s = h s
```

Обратите внимание, что, по модулю распаковки конструктора, `runState` является полноценным применением функции.

Недетерминизм

Представьте себе проведение квантового эксперимента, в котором измеряется спин электрона. Половину времени он будет вращаться вверх, а другую половину — вниз. Результат является недетерминированным. Один из способов описать это — использовать многомировую интерпретацию: когда мы проводим эксперимент, Вселенная разделяется на две вселенные, по одной для каждого результата.

Что означает, что функция недетерминирована? — при каждом вызове она будет возвращать разные результаты (с одними и теми же входными значениями). Мы можем смоделировать это поведение, используя многомировую интерпретацию: мы позволяем функции возвращать *все возможные результаты* одновременно. На практике мы довольствуемся (возможно, бесконечным) списком результатов.

Заменим недетерминированное вычисление `a -> b` чистой функцией, возвращающей функтор, полный результатов — здесь, это функтор списка:

```
a -> [b]
```

Опять же, вызывающая сторона должна решать, что делать с этими результатами.

Ввод/Вывод

Это самый сложный побочный эффект, поскольку он содержит взаимодействие с внешним миром. Очевидно, что мы не можем смоделировать весь мир внутри компьютерной программы. Таким образом, чтобы программа оставалась чистой, взаимодействие должно происходить вне ее. Трюк заключается в том, чтобы позволить программе генерировать сценарий. Затем этот сценарий передается среде выполнения для выполнения. Среда выполнения — это эффективная виртуальная машина, на которой выполняется программа.

Сам этот сценарий находится внутри непрозрачного предопределенного функтора `IO`. Значения, скрытые в этом функторе, недоступны для программы: функции `runIO` не существует. Вместо этого значение `IO`, созданное программой, выполняется, по крайней мере концептуально, после завершения программы.

На самом деле, вследствие "ленивости" Haskell, выполнение операций ввода/вывода чередуется с выполнением остальной части программы. Чистые функции, которые составляют большую часть вашей программы, выполняются по запросу, который определяется выполнением сценария `IO`. Если бы не ввод/вывод, все было бы устроено намного проще.

Объект `IO`, создаваемый программой на Haskell, называется `main`, а его сигнатура типа есть

```
main :: IO ()
```

Это функтор `IO`, содержащий единицу, что означает: нет никакого полезного значения, кроме сценария ввода/вывода.

Вскоре мы поговорим о том, как создаются действия `IO`.

Продолжение

Мы видели, что вследствие леммы Йонеды, можно заменить значение типа `a` функцией, которая принимает обработчик для этого значения. Этот обработчик называется *продолжением*. Вызов обработчика считается побочным эффектом вычислений. В термина чистых функций это кодируется как

```
a -> Cont r b
```

где `Cont r` есть функтор:

```
newtype Cont r a = Cont ((a -> r) -> r)
```

Вызывающий следующую функцию обязан обеспечить продолжение, функцию `k :: a -> r`, и получить результат:

```
runCont            :: Cont r a -> (a -> r) -> r
runCont (Cont f) k = f k
```

Это экземпляр `Functor` для `Cont r`:

```
instance Functor (Cont r) where
  -- f :: a -> b
  -- k :: b -> r
  fmap f c = Cont (\k -> runCont c (k . f))
```

Заметим, что это ковариантный функтор, потому что тип `a` находится в двояко негативной позиции.

В декартовой замкнутой категории продолжения порождаются эндофункторами:

$$K_r a = r^{r^a}$$

14.2 Композиционные эффекты

Теперь, когда понятно, как сделать один большое действие, используя функцию, которая производит как значение, так и побочный эффект, следующая проблема состоит в том, чтобы выяснить, как разложить это действие на более мелкие шаги. Или, наоборот, как объединить эти шаги в один большой.

Способ компоновки эффективных вычислений в императивных языках состоит в том, чтобы использовать обычную композицию функций для значений и позволять побочным эффектам объединяться вынужденно.

Когда мы представляем эффективные вычисления в виде чистых функций, мы сталкиваемся с проблемой компоновки двух функций вида

```
g :: a -> f b
h :: b -> f c
```

Во всех интересующих нас случаях конструктор типа `f` оказывается `Functor`, поэтому в дальнейшем мы будем предполагать, что это так.

Наивным подходом была бы распаковка результата первой функции, передача значения следующей функции, затем компоновка эффектов обеих функций и объединение с результатом второй функции. Это не всегда возможно, даже для случаев, которые рассматривались до сих пор, не говоря уже о конструкторе произвольного типа.

Ради аргумента, поучительно рассмотреть, как это можно сделать для функтора `Maybe`. Если первая функция возвращает `Just`, то компонуем ее с шаблоном, чтобы извлечь содержимое и вызвать с ним следующую функцию.

Но если первая функция возвращает `Nothing`, значение для вызова второй функции отсутствует. Тогда надо действовать «в обход» и вернуть `Nothing` напрямую. Таким образом, композиция возможна, но это означает изменение потока управления путем пропуска второго вызова из-за побочного эффекта первого вызова.

Для одних функторов возможна композиция побочных эффектов, для других нет. Как же можно охарактеризовать эти «хорошие» функторы?

Чтобы функтор кодировал скомпонованные побочные эффекты, необходимо, по крайней мере, иметь возможность реализовать следующую полиморфную функцию высшего порядка:

```
composeWithEffects :: Functor f =>
       (b -> f c) -> (a -> f b) -> (a -> f c)
```

Это очень похоже на обычную композицию функций:

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

поэтому естественно задаться вопросом, существует ли категория, в которой первая определяет композицию стрелок. Рассмотрим, что еще нужно для построения такой категории.

Объекты в этой новой категории — это те же типы Haskell, что и ранее. Но стрелка от $a \nrightarrow b$ реализуется как функция Haskell:

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

Затем, функцию `composeWithEffects` можно использовать для реализации композиции таких стрелок.

Чтобы иметь категорию, требуется, чтобы эта композиция была ассоциативной. Также нужна тождественная стрелка для каждого объекта `a`. Это стрелка от $a \nrightarrow a$, поэтому она соответствует функции Haskell:

```
idWithEffects :: a -> f a
```

По отношению к `composeWithEffects`, она должна вести себя как тождественность.

Мы только что определили монаду! После некоторого переименования и реорганизации можно записать ее как класс типов:

```
class Functor m => Monad m where
  (<=<)  :: (b -> m c) -> (a -> m b)
                       -> (a -> m c)
  return :: a  -> m a
```

Инфиксный оператор `<=<` заменяет функцию `composeWithEffects`. Функция возврата — это тождественная стрелка в нашей новой категории (это не определение монады, которое находится в библиотеке `Prelude` из Haskell, но, как вскоре мы увидим, эквивалентно ему).

В качестве упражнения, определим экземпляр `Monad` для `Maybe`. Оператор «рыба», `<=<`, компонует две функции:

```
f :: a -> Maybe b
g :: b -> Maybe c
```

в одну функцию типа `a -> Maybe c`. Единица этой композиции, `return`, заключает значение в конструктор Just.

```
instance Monad Maybe where
  g <=< f = \a -> case f a of
                    Nothing -> Nothing
                    Just b  -> g b
  return  = Just
```

Можно легко убедиться, что категорные законы выполняются. В частности, `return <=< g` совпадает с `g`, а `f <=< return` совпадает с `f`. Доказательство ассоциативности также довольно простое: если какая-либо из функций возвращает `Nothing`, результатом будет `Nothing`; в противном случае — это просто ассоциативная композиция функций.

Категория, которая только что была определена, называется *категорией Клейсли* для монады `m`. Функции `a -> m b` называются *стрелками Клейсли*. Они компонуются с помощью `<=<`, а тождественной стрелкой является `return`.

Все функторы из предыдущего раздела являются экземплярами `Monad`. Если рассматривать их как конструкторы типов или даже как функторы, между ними трудно увидеть какое-либо сходство. Но их объединяет то, что их можно использовать для реализации *компонуемых стрелок* Клейсли.

Как сказал бы Лао-Цзы: «Композиция — это то, что происходит *между* сущностями. Сосредотачивая внимание на сущностях, часто упускается из виду находящееся в промежутках».

14.3 Альтернативные определения

Преимущество определения монады с помощью стрелок Клейсли состоит в том, что законы монады — это просто ассоциативность и законы единиц категории. Имеются еще два эквивалентных определения монады, одно из которых предпочитают математики, а другое — программисты.

Сначала заметим, что при реализации оператора `<=<` в качестве аргументов задаются две функции. Единственное, для чего функция полезна, — для применения к аргументу. Когда мы применяем первую функцию `f :: a -> m b`, то получаем значение типа `m b`. На этом мы бы застряли, если бы не тот факт, что `m` — функтор. Функториальность позволяет применить вторую функцию `g :: b -> m c` к `m b`. Действительно, подъем `g`, согласно `m`, имеет вид:

```
m b -> m (m c)
```

Это почти тот результат, который мы ищем, если бы только можно было сгладить `m(m c)` до `m c`. Это действие обозначается `join`. Другими словами, если дано:

```
join :: m (m a) -> m a
```

можно реализовать `<=<`:

```
g <=< f = \a -> join (fmap g (f a))
```

или, используя бесточечную нотацию:

```
g <=< f = join . fmap g . f
```

И наоборот, `join` может быть реализовано через `<=<`:

```
join = id <=< id
```

Это может быть не сразу очевидно, пока не будет осознано, что правый `id` применяется к `m (m a)`, а левый — к `m a`. Мы интерпретируем функцию Haskell:

```
m (m a) -> m (m a)
```

как стрелку в категории Клейсли $m(ma) \twoheadrightarrow ma$. Точно так же, функция:

```
m a -> m a
```

реализует стрелку Клейсли $ma \twoheadrightarrow a$. Их Клейсли-композиция дает стрелку Клейсли $m(ma) \twoheadrightarrow a$, или функцию Haskell:

```
m (m a) -> m a
```

Это приводит к эквивалентному определению монады в терминах `join` и `return`:

```
class Functor m => Monad m where
  join   :: m (m a) -> m a
  return :: a       -> m a
```

Это все еще не то определение, которое содержится в стандартной версии Haskell `Prelude`. Поскольку оператор `<=<` является обобщением оператора точки, его использование эквивалентно бесточечному программированию. Это позволяет компоновать стрелки без указания промежуточных значений. Хотя некоторые считают бесточечные программы более элегантными, большинству программистов непривычно следовать этому стилю.

Но на самом деле композиция функций выполняется в два этапа: мы применяем первую функцию, а затем применяем вторую функцию к результату. Явное вычленение именованного промежуточного результата часто помогает понять, что происходит.

Чтобы сделать то же самое со стрелками Клейсли, надо знать, как применить вторую стрелку Клейсли к именованному монадическому значению — результату первой стрелки Клейсли. Функция, которая это делает, называется *связыванием* и записывается как инфиксный оператор:

```
(>>=) :: m a -> (a -> m b) -> m b
```

Очевидно, что можно реализовать композицию Клейсли в обозначении связывания:

```
g <=< f = \a -> (f a) >>= g
```

И наоборот, связывание может быть реализовано использованием стрелки Клейсли:

```
ma >>= k = (k <=< id) ma
```

Это приводит к следующему определению:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a   -> m a
```

Это почти то же определение, что и в `Prelude`, за исключением дополнительного ограничения. Это ограничение утверждает тот факт, что каждый экземпляр `Monad` также является экземпляром `Applicative`. Отложим обсуждение, так называемых, аппликативов до раздела о моноидальных функторах.

Также, можно реализовать `join` с помощью связывания:

```
join     :: (Monad m) => m (m a) -> m a
join mma = mma >>= id
```

Haskell-функция `id` идет от `m a` к `m a` или, как стрелка Клейсли, $ma \nrightarrow a$.

Интересно, что монада, определенная с помощью связывания, автоматически становится функтором. Функция подъема для нее обозначается `liftM`:

```
liftM     :: Monad m => (a -> b) -> (m a -> m b)
liftM f ma = ma >>= (return . f)
```

14.4 Примеры монад

Теперь мы готовы определить экземпляры монад для функторов, которые были использованы для побочных эффектов. Это позволит компоновать побочные эффекты.

Частичность

Мы уже сталкивались с версией монады `Maybe`, реализованной с помощью композиции Клейсли. Вот более знакомая реализация, с использованием связывания:

```
instance Monad Maybe where
  Nothing  >>= k = Nothing
  (Just a) >>= k = k a
  return         = Just
```

Протоколирование

Для композиции функций, создающих журналы, нужен способ совмещения отдельных записей журнала. Вот почему монада

```
newtype Writer w a = Writer (a, w)
```

требует, чтобы тип журнала был экземпляром `Monoid`. Это позволяет пополнять журналы, а также создавать пустой журнал.

```
instance Monoid w => Monad (Writer w) where
  (Writer (a, w)) >>= k
           = let (Writer (b, w')) = k a
             in Writer (b, mappend w w')
  return a = Writer (a, mempty)
```

Предложение `let` используется для введения локальных привязок. Здесь результатом применения `k` является сопоставление с образцом, а локальные переменные `b` и `w'` инициализируются. Конструкция `let ... in ...` — это выражение, значение которого определяется содержанием предложения `in`.

Внешняя среда

Монада чтения `Reader` — это тонкая инкапсуляция функции от внешней среды к возвращаемому типу:

```
newtype Reader e a = Reader (e -> a)
```

Экземпляр `Monad`:

```
instance Monad (Reader e) where
  ma >>= k = Reader (\e ->
               let a = runReader ma e
               in runReader (k a) e)
  return a  = Reader (\e -> a)
```

Реализация связывания для монады чтения создает функцию, которая принимает среду в качестве аргумента. Эта среда используется дважды: сначала для запуска `ma` для получения значения `a`, а затем — для вычисления значения, созданного `k a`.

Реализация `return` игнорирует среду.

Упражнение 14.4.1. *Определите* `Functor` *и экземпляр* `Monad` *для следующего типа данных*:

```
newtype E e a = E (e -> Maybe a)
```

Подсказка: *вы можете использовать функцию*:

```
runE         :: E e a -> e -> Maybe a
runE (E f) e = f e
```

Состояние

Как и монада чтения, монада состояния является функциональным типом:

```
newtype State s a = State (s -> (a, s))
```

Его связывание аналогично, за исключением того, что результат `k`, воздействующий на `a`, теперь выполняется с измененным состоянием `s'`.

```
instance Monad (State s) where
  st >>= k = State (\s ->
                    let (a, s') = runState st s
                    in  runState (k a) s')
  return a  = State (\s -> (a, s))
```

Применение связывания к тождественности дает определение `join`:

```
join    :: State s (State s a) -> State s a
join mma = State (\s ->
                  let (ma, s') = runState mma s
                  in runState ma s')
```

Заметим, что мы, по сути, передаем результат первого `runState` второму `runState`, за исключением того, что нам нужно отменить каррирование второго, чтобы он мог принять пару:

```
join mma = State (\s -> (uncurry runState)
                        (runState mma s))
```

В таком виде его легко преобразовать в бесточечную нотацию:

```
join mma = State (uncurry runState . runState mma)
```

Имеются две основные стрелки Клейсли (первая, концептуально, исходит от терминального объекта `()`), с помощью которых можно построить произвольное вычисление с сохранением состояния. Первая извлекает текущее состояние:

```
get  :: State s s
get   = C (\s -> (s, s))
```

а вторая изменяет его:

```
set  :: s -> State s ()
set s = State (\_ -> ((), s))
```

Многие монады поставляются со своими собственными библиотеками предопределенных основных стрелок Клейсли.

Недетерминизм

Для монады списка, рассмотрим, как можно было бы реализовать `join`. Надо превратить список списков в один список. Это можно сделать, объединив все внутренние списки с помощью библиотечной функции `concat`. Отсюда можно вывести реализацию связывания.

```
instance Monad [] where
  as >>= k = concat (fmap k as)
  return a = [a]
```

Здесь, `return` создает одноэлементный список.

То, что в императивных языках делается с помощью вложенных циклов, в Haskell можно сделать, используя монаду списка. Можно воспринимать `as`, в привязке, как объединение результатов выполнения внутреннего цикла, а `k` — как код, который выполняется во внешнем цикле.

Во многих отношениях список в Haskell больше похож на то, что в императивных языках называется *итератором* или *генератором*. Из-за ленивости элементы списка редко размещаются в памяти все сразу, так что, вы можете концептуализировать список Haskell как указатель на голову и рецепт для продвижения к хвосту. Также можно представлять список как сопрограмму, которая по запросу создает элементы последовательности.

Продолжение

Реализация привязки для монады продолжения

```
newtype Cont r a = Cont ((a -> r) -> r)
```

требует некоторого обратного мышления из-за присущей инверсии управления — принципа «не звоните нам, мы сами вам позвоним».

Результатом привязки является тип `Cont r b`. Для его построения понадобится функция, принимающая в качестве аргумента `k :: b -> r`:

```
ma >>= fk = Cont (\k -> ...)
```

Мы должны собрать такую функцию, имея в своем распоряжении два ингредиента:

```
ma :: Cont r a
fk :: a -> Cont r b
```

Мы хотели бы запустить `ma`, а для этого нужно продолжение, которое принимало бы `a`.

```
runCont ma (\a -> ...)
```

Когда имеется `a`, то можно выполнить `fk`. Результат имеет тип `Cont r b`, поэтому можно запустить его с нашим продолжением `k :: b -> r`.

```
runCont (fk a) k
```

В совокупности, этот сложный процесс приводит к следующей реализации:

```
instance Monad (Cont r) where
  ma >>= fk  = Cont (\k -> runCont ma
                           (\a -> runCont (fk a) k))
  return a   = Cont (\k -> k a)
```

Как упоминалось ранее, компоновка продолжений — занятие не для слабонервных. Однако реализовать его приходится только один раз — в определении монады продолжения. С этого момента нотация `do` ... сделает все остальное относительно простым.

Ввод/Вывод

Реализация монады `IO` встроена в язык. Основные примитивы ввода/вывода доступны через библиотеку. Они доступны, либо в виде стрелок Клейсли, либо объектов `IO` (концептуально, стрелки Клейсли от терминального объекта `()`).

Например, следующий объект содержит команду для чтения строки из стандартного ввода:

```
getLine :: IO String
```

Нет возможности извлечь из него строку, так как ее еще нет; но программа может обработать его через дальнейшую серию стрелок Клейсли.

Монада `IO` — величайший откладывальщик: композиция ее стрелок Клейсли накапливает задачу за задачей, которые позже выполняются исполняющей средой Haskell.

Чтобы вывести строку, за которой следует новая строка, можно использовать следующую стрелку Клейсли:

```
putStrLn :: String -> IO ()
```

Комбинируя их, можно создать простой объект `main`:

```
main :: IO ()
main  = getLine >>= putStrLn
```

который выводит введенную строку.

14.5 do-нотация

Стоит повторить, что единственное назначение монад в программировании — позволить разложить одну большую стрелку Клейсли на несколько меньших.

Это можно сделать либо напрямую, в бесточечном стиле, используя композицию Клейсли `<=<`, либо через именование промежуточных значений и их привязки к стрелкам Клейсли с помощью `>>=`.

Некоторые стрелки Клейсли определены в библиотеках, другие можно реализовать самостоятельно и использовать повторно, чтобы гарантировать внештатную реализацию, но на практике большинство реализовано как однократные встраиваемые лямбда-выражения.

Вот простой пример:

```
main :: IO ()
main  =
  getLine >>= \s1 ->
    getLine >>= \s2 ->
      putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

в котором используется специальная стрелка Клейсли типа `String -> IO ()`, определяемая лямбда-выражением:

```
\s1 ->
  getLine >>= \s2 ->
    putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

Тело этого лямбда-выражения далее декомпозируется с помощью другой специальной стрелки Клейсли:

```
\s2 -> putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

Такие конструкции настолько распространены, что существует специальный синтаксис, называемый do-нотацией, который позволяет избежать многих шаблонов. Приведенный выше код, например, можно записать так:

```
main = do
  s1 <- getLine
  s2 <- getLine
  putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

Компилятор автоматически преобразует его в серию вложенных лямбда-выражений. Фрагмент `s1 <- getLine` обычно читается как: «`s1` *получает* результат `getLine`».

Вот еще один пример: функция, использующая монаду списка для генерации всех возможных пар элементов, взятых из двух списков,

```
pairs      :: [a] -> [b] -> [(a, b)]
pairs as bs = do
                a <- as
                b <- bs
                return (a, b)
```

Отметим, что последняя строка в блоке do должна выдавать монадическое значение — здесь это достигается с помощью `return`.

Большинству императивных языков не хватает мощности абстракции для общего определения монады, и вместо этого они пытаются жестко закодировать некоторые из наиболее распространенных монад. Например, они реализуют исключения, как альтернативу монаде `Either`, или параллельные задачи, как альтернативу монаде продолжения. Некоторые, например C++, вводят сопрограммы, которые имитируют нотацию Haskell do.

Упражнение 14.5.1. *Реализуйте следующую функцию, которая работает для любой монады*:

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

Подсказка: *используйте* do*-нотацию, чтобы извлечь функцию и аргумент. Используйте* `return` *для возврата результата.*

Упражнение 14.5.2. *Перепишите функцию* `pairs`, *используя операторы привязки и лямбда-выражения.*

14.6 Стиль передачи продолжения

Как упоминалось ранее, нотация `do` предоставляет синтаксический сахар, который делает работу с продолжениями более естественной. Одним из наиболее важных применений продолжений является преобразование программ для использования *стиля передачи продолжений*, или CPS (от Continuation Passing Style). Использование CPS распространено в конструкции компилятора. Еще одно очень важное применение CPS — преобразование рекурсии в итерацию.

Общая проблема с глубоко рекурсивными программами заключается в том, что они могут повредить стек времени выполнения. Вызов функции обычно начинается с помещения аргументов функции, локальных переменных и адреса возврата в стек. Таким образом, глубоко вложенные рекурсивные вызовы могут быстро исчерпать стек среды выполнения (обычно фиксированного размера), что приведет к ошибке времени выполнения. Это главная причина, по которой императивные языки предпочитают циклы рекурсии и почему большинство программистов узнают о циклах до того, как изучают рекурсию. Однако даже в императивных языках, когда речь идет об обходе рекурсивной структуры данных, такой как связанные списки или деревья, рекурсивные алгоритмы более естественны.

Однако проблема с использованием циклов заключается в том, что они требуют мутации. Обычно имеется какой-то счетчик или указатель, который модифицируется и проверяется при каждой итерации цикла. Вот почему чисто функциональные языки, избегающие мутаций, должны использовать рекурсию вместо циклов. Но поскольку циклы более эффективны и не потребляют стек времени выполнения, компилятор пытается скрыть рекурсивные вызовы циклов. В Haskell все хвостовые рекурсивные функции превращаются в циклы.

Хвостовая рекурсия и CPS

Хвостовая рекурсия означает, что рекурсивный вызов происходит в самом конце функции. Функция не выполняет никаких дополнительных операций над результатом хвостового вызова. Например, следующий код не является хвостовой рекурсией, поскольку содержит добавление `i` к результату рекурсивного вызова:

```
sum1          :: [Int] -> Int
sum1 []        = 0
sum1 (i : is) = i + sum1 is
```

Напротив, следующая реализация является хвостовой рекурсией, потому что результат рекурсивного вызова `go` возвращается без дальнейших изменений:

```
sum2 = go 0
  where go n []       = n
        go n (i : is) = go (n + i) is
```

Компилятор может легко превратить последнее в цикл. Вместо рекурсивного вызова он перезапишет значение первого аргумента `n` на `n + i`, перезапишет указатель на начало списка указателем на его конец, а затем перейдет к началу функции.

Заметим, однако, что это не означает, что компилятор Haskell не сможет разумно оптимизировать первую реализацию. Это просто означает, что вторая реализация, являющаяся хвостовой рекурсией, *гарантированно* превратится в цикл.

На самом деле, всегда можно превратить рекурсию в хвостовую рекурсию, выполнив преобразование CPS. Это связано с тем, что продолжение инкапсулирует *оставшуюся часть вычислений*, поэтому это всегда последний вызов функции.

Чтобы увидеть, как это работает на практике, рассмотрим простой обход дерева. Определим дерево, которое хранит строки, как в узлах, так и в листьях:

```
data Tree = Leaf String | Node Tree String Tree
```

Чтобы объединить эти строки, используем обход, который сначала рекурсивно переходит в левое поддерево, а затем в правое поддерево:

```
show                  :: Tree -> String
show (Node lft s rgt) =
                 let ls = show lft
                     rs = show rgt
                 in ls ++ s ++ rs
```

Это определенно не является хвостовой рекурсивной функцией, и не очевидно, как превратить ее в таковую. Однако, можно почти механически переписать его, используя монаду продолжения:

```
showk                  :: Tree -> Cont r String
showk (Leaf s)          = return s
showk (Node lft s rgt) = do
                           ls <- showk lft
                           rs <- showk rgt
                           return (ls ++ s ++ rs)
```

Мы можем запустить результат с тривиальным продолжением `id`:

```
show  :: Tree -> String
show t = runCont (showk t) id
```

Эта реализация автоматически является хвостовой рекурсией. Можно ясно увидеть это, убрав нотацию `do`:

```
showk                :: Tree -> (String -> r) -> r
showk (Leaf s) k      = k s
showk (Node lft s rgt) k
                      =
                        showk lft (\ls ->
                          showk rgt (\rs ->
                            k (ls ++ s ++ rs)))
```

Проанализируем этот код. Функция вызывает себя, передавая левое поддерево `lft` и следующее продолжение:

```
(\ls ->
     showk rgt (\rs ->
       k (ls ++ s ++ rs)))
```

Эта лямбда, в свою очередь, вызывает `showk` с правым поддеревом `rgt` и другим продолжением:

```
\rs -> k (ls ++ s ++ rs)
```

Эта самая внутренняя лямбда имеет доступ ко всем трем строкам: левой, средней и правой. Она объединяет их и вызывает самое внешнее продолжение `k` с этим результатом.

В каждом случае рекурсивный вызов `showk` является последним вызовом, и его результат возвращается немедленно. Тип результата — общий тип `r`, который сам по себе гарантирует, что мы не можем выполнять над ним какие-либо операции. Когда, наконец, запускается результат от `showk`, ему передается идентификатор (экземпляр типа `String`):

```
show  :: Tree -> String
show t = runCont (showk t) id
```

Использование именованных функций

Но предположим, что наш язык программирования не поддерживает анонимные функции. Можно ли заменить лямбды именованными функциями? Мы делали это ранее, когда обсуждали теорему о сопряженном функторе. Заметим, что лямбда-выражения, сгенерированные монадой-продолжением, являются замыканиями — они захватывают некоторые значения из своего окружения. Если желательно использовать именованные функции, то придется явно передать среду.

Заменим первую лямбду на вызов функции с именем `next` и передадим ей необходимое окружение в виде кортежа из трех значений `(s, rgt, k)`:

```
showk                :: Tree -> (String -> r) -> r
showk (Leaf s) k      = k s
showk (Node lft s rgt) k
                      = showk lft (next (s, rgt, k))
```

Три значения — это строка из текущего узла дерева, правого поддерева и внешнего продолжения.

Функция `next` осуществляет рекурсивный вызов `showk`, передавая ей правое поддерево и продолжение `conc`:

```
next                :: (String, Tree, String -> r)
                                    -> String -> r
next (s, rgt, k) ls = showk rgt (conc (ls, s, k))
```

Опять же, `conc` явно фиксирует окружение, содержащее две строки и внешнее продолжение. Она выполняет конкатенацию и вызывает внешнее продолжение с конкатенированным результатом:

```
conc               :: (String, String, String -> r)
                                        -> String -> r
conc (ls, s, k) rs = k (ls ++ s ++ rs)
```

Наконец, определим тривиальное продолжение:

```
done  :: String -> String
done s = s
```

которое используется для извлечения конечного результата:

```
show t = showk t done
```

Дефункциализация

Стиль передачи продолжения требует использования функций высшего порядка. Если это является проблемой, например, при реализации распределенных систем, то всегда можно использовать теорему о сопряженном функторе, чтобы дефункциализировать программу.

Первый шаг — создание суммы всех соответствующих сред, включая пустую, которую была использована в `done`:

```
data Kont = Done
          | Next String Tree Kont
          | Conc String String Kont
```

Заметим, что эту структуру данных можно переинтерпретировать как список или стек. Его можно рассматривать как список элементов следующего тип-суммы:

```
data Sum = Next' String Tree | Conc' String String
```

Этот список является нашей версией стека времени выполнения, необходимым для реализации рекурсивного алгоритма.

Поскольку нас интересует только создание строки в качестве конечного результата, будем аппроксимировать тип функции `String -> String`. Это — аппроксимация ко-единицы сопряжения, которое его определяет (см. теорему о сопряженном функторе):

```
apply                    :: (Kont, String)
                                        -> String
apply (Done, s)           = s
apply (Next s rgt k, ls) = showk rgt (Conc ls s k)
apply (Conc ls s k, rs)   = apply (k, ls ++ s ++ rs)
```

Функция `showk` теперь может быть реализована без обращения к функциям высшего порядка:

```
showk                    :: Tree -> Kont -> String
showk (Leaf s) k          = apply (k, s)
showk (Node lft s rgt) k = showk lft (Next s rgt k)
```

To extract the result, we call it with `Done`:

```
showTree t = showk t Done
```

14.7 Монады с категорной точки зрения

В теории категорий монады впервые возникли при изучении алгебр. В частности, мы можем использовать оператор связывания для реализации очень важной операции подстановки.

Подстановка

Рассмотрим простой тип выражения, параметризованный типом `x`, который можно использовать для именования наших переменных:

```
data Ex x = Val Int
          | Var x
          | Plus (Ex x) (Ex x)
  deriving (Functor, Show)
```

Мы можем, например, построить выражение $(2 + a) + b$:

```
ex :: Ex Char
ex  = Plus (Plus (Val 2) (Var 'a')) (Var 'b')
```

Также, можно реализовать экземпляр `Monad` для `Ex`:

```
instance Monad Ex where
  Val  n >>= k     = Val n
  Var  x >>= k     = k x
  Plus e1 e2 >>= k =
    let x = e1 >>= k
        y = e2 >>= k
    in (Plus x y)

  return x = Var x
```

Теперь предположим, что необходимо произвести подстановку, заменив переменную a на $x_1 + 2$, а b — на x_2 (для простоты не будем заботиться о других буквах алфавита). Эта подстановка представлена стрелкой Клейсли `sub`:

```
sub    :: Char -> Ex String
sub 'a' = Plus (Var "x1") (Val 2)
sub 'b' = Var "x2"
```

Заметьте, мы даже можем изменить тип имен переменных с `Char` на `String`.

Когда мы связываем эту стрелку Клейсли с `ex`

```
ex' :: Ex String
ex'  = ex >>= sub
```

то получаем, как и ожидалось, дерево, соответствующее

$$(2 + (x_1 + 2)) + x_2$$

Монада как моноид

Проанализируем определение монады, использующей `join`:

```
class Functor m => Monad m where
  join   :: m (m a) -> m a
  return :: a       -> m a
```

Имеем эндофунктор `m` и две полиморфные функции.

В теории категорий, функтор, определяющий монаду, традиционно обозначается через T (возможно, потому, что монады изначально назывались «тройками»). Две полиморфные функции становятся естественными преобразованиями. Первое, соответствующее соединению, отображает «квадрат» T (композицию T с самим собой) к T:

$$\mu : T \circ T \to T$$

(конечно, таким образом возводить в квадрат можно только эндофункторы).

Второе преобразование, соответствующее `return`, отображает тождественный функтор к T:

$$\eta : Id \to T$$

Сравните это с более ранним определением моноида в моноидальной категории:

$$\mu : m \otimes m \to m$$
$$\eta : I \to m$$

Сходство поразительное. Вот почему естественное преобразование μ часто называют *монадическим умножением*. Но в какой категории композицию функторов можно считать тензорным произведением?

Введем категорию эндофункторов. Объектами этой категории являются эндофункторы, а стрелками — естественные преобразования.

Но в этой категории больше структуры. Известно, что любые два эндофунктора могут быть скомпонованы. Как можно интерпретировать эту композицию, если мы хотим рассматривать эндофункторы как объекты? Операция, которая берет два объекта и создает третий объект, похожа на тензорное произведение. В конце концов, единственное условие, которое мы налагаем на тензорное произведение, состоит в том, что оно было функториально по обоим аргументам. То есть, для пары стрелок:

$$\alpha : T \to T'$$
$$\beta : S \to S'$$

можно поднять его к отображению тензорного произведения:

$$\alpha \otimes \beta : T \otimes S \to T' \otimes S'$$

В категории эндофункторов стрелки являются естественными преобразованиями, поэтому, если заменить $\otimes$ на $\circ$, поднятие является отображением:

$$\alpha \circ \beta : T \circ T' \to S \circ S'$$

Но это всего лишь горизонтальная композиция естественных преобразований (понятно, почему она обозначена кружком).

Единичным объектом в этой моноидальной категории является тождественный эндофунктор, а единичные законы точно выполняются, т.е.

$$\mathrm{Id} \circ T = T = T \circ \mathrm{Id}$$

Нам не требуются никакие юниторы. Нам также не нужны ассоциаторы, поскольку функторная композиция автоматически ассоциативна.

Моноидальная категория, в которой единица и ассоциаторы являются тождественными морфизмами, называется *строгой* моноидальной категорией.

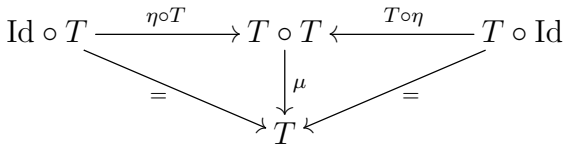
Заметим, однако, что композиция не симметрична, так что это не симметричная моноидальная категория.

Итак, сказанное позволяет сформулировать, что монада — это моноид в моноидальной категории эндофункторов.

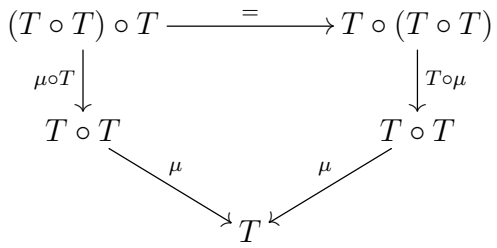
Монада (T, η, μ) состоит из объекта в категории эндофункторов, что означает эндофунктор T, и двух стрелок, означающих естественные преобразования:

$$\eta : \mathrm{Id} \to T$$
$$\mu : T \circ T \to T$$

Чтобы это был моноид, эти стрелки должны удовлетворять законам моноида. Законы единицы (с заменой униторов на строгие равенства) задаются диаграммой:



а закон ассоциативности — диаграммой:



Здесь использована нотация вискеринга для горизонтальной композиции $\mu \circ T$ и $T \circ \mu$.

Это законы монад в терминах μ и η. Их можно напрямую перевести в законы для `join` и `return`. Они также эквивалентны законам категории Клейсли, построенным из стрелок $a \to Tb$.

14.8 Свободные монады

Монада — это эндофунктор $T : [\mathcal{C}, \mathcal{C}]$, снабженный дополнительной структурой, заданной двумя естественными преобразованиями. Этот вид определения можно уточнить, определив забывающий функтор, который игнорирует дополнительную структуру. В нашем случае, для монады (T, η, μ) можно было бы оставили только T. Но чтобы определить такой функтор, сначала нужно определить категорию монад.

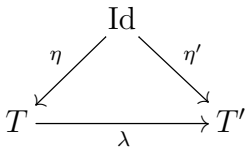
Категория монад

Объекты категории монад $\mathbf{Mon}(\mathcal{C})$ являются монадами (T, η, μ). Можно определить стрелку между двумя монадами (T, η, μ) и (T', η', μ') как естественное преобразование между двумя эндофункторами:

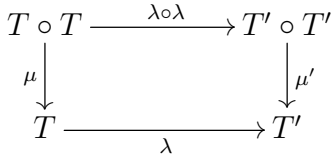
$$\lambda : T \to T'$$

Однако, поскольку монады являются эндофункторами *со структурой*, мы хотим, чтобы эти естественные преобразования сохраняли структуру. Сохранение единицы означает, что следующая диаграмма должна быть

коммутативной:



Сохранение умножения означает коммутативность диаграммы:



Другой способ понять $\mathbf{Mon}(\mathcal{C})$ состоит в том, что это категория моноидов в $[\mathcal{C},\mathcal{C}]$.

Свободная монада

Теперь, когда в нашем распоряжении имеется категория монад, можно определить забывающий функтор:

$$U : \mathbf{Mon}(\mathcal{C}) \to [\mathcal{C},\mathcal{C}]$$

который отображает каждую тройку (T,η,μ) к T, а каждый монадный морфизм к, лежащему в основе, естественному преобразованию.

Мы хотели бы, чтобы свободная монада порождалась левым сопряженным к этому забывающему функтору. Проблема в том, что этот левый сопряженный не всегда существует. Как обычно, это связано с проблемами размера: монады склонны нарушать устоявшийся порядок. Суть в том, что свободные монады существуют для некоторых, но не для всех эндофункторов. Следовательно, мы не можем определить свободную монаду через сопряжение. К счастью, в большинстве интересующих нас случаях свободную монаду можно определить как неподвижную точку алгебры.

Такая построение аналогично тому, как мы определили свободный моноид в качестве инициальной алгебры для функтора списка:

```
data ListF a x = NilF | ConsF a x
```

или более формально:

$$F_a x = I + a \otimes x$$

Однако, на этот раз, моноидальной категорией, в которой монада определяется как моноид, является категория эндофункторов $([\mathcal{C}, \mathcal{C}], \mathrm{Id}, \circ)$. Свободный моноид в этой категории является инициальной алгеброй для функтора «список», который отображает функторы к функторам:

$$\Phi_F G = \mathrm{Id} + F \circ G$$

Здесь, копроизведение двух функторов определяется поточечно; на объектах:

$$(F + G)a = Fa + Ga$$

и на стрелках:

$$(F + G)f = Ff + Gf$$

(можно образовать копроизведение двух морфизмов, используя функториальность копроизведения. Мы предполагаем, что $\mathcal{C}$ является декартовой, то есть все копроизведения существуют).

Инициальная алгебра — это (наименьшая) неподвижная точка этого оператора, или решение тождества:

$$L_F \cong \mathrm{Id} + F \circ L_F$$

Эта формула устанавливает естественный изоморфизм между двумя функторами. В частности, справа-налево, отображение-вне суммы эквивалентно паре естественных преобразований:

$$\mathrm{Id} \to L_F$$
$$F \circ L_F \to L_F$$

При переводе на Haskell, компоненты этих преобразований становятся двумя конструкторами. Мы получаем следующий тип данных, параметризованный функтором `f`:

```
data FreeMonad f a where
  Pure :: a                  -> FreeMonad f a
  Free :: f (FreeMonad f a) -> FreeMonad f a
```

Если рассматривать этот функтор `f` как контейнер значений, конструктор `Free` принимает контейнер, заполненный `(FreeMonad f a)`, и скрывает его. Таким образом, произвольное значение типа `FreeMonad f a` является деревом, в котором каждый узел является функтором ветвей, а каждый лист содержит значение типа `a`.

Поскольку это определение рекурсивно, экземпляр `Functor` также является рекурсивным:

```
instance Functor f => Functor (FreeMonad f) where
  fmap g (Pure a)   = Pure (g a)
  fmap g (Free ffa) = Free (fmap (fmap g) ffa)
```

Внешний `fmap` использует экземпляр `Functor` для `f`, а внутренний `(fmap g)` рекурсивно проходит по ветвям.

Монадическая единица `eta` — это всего лишь тонкая инкапсуляция тождественного функтора:

```
eta  :: a -> FreeMonad f a
eta a = Pure a
```

Монадическое умножение, или `join`, определяется рекурсивно:

```
mu            :: Functor f => FreeMonad f
                  (FreeMonad f a) -> FreeMonad f a
mu (Pure fa)  = fa
mu (Free ffa) = Free (fmap mu ffa)
```

Следовательно, экземпляр `Monad` для `FreeMonad f` есть:

```
instance Functor f => Monad (FreeMonad f) where
  return a = eta a
  m >>= k  = mu (fmap k m)
```

Мы также можем определить привязку напрямую:

```
(Pure a)   >>= k = k a
(Free ffa) >>= k = Free (fmap (>>= k) ffa)
```

Свободная монада накапливает монадические действия в древовидной структуре, не привязываясь к любой определенной стратегии оценки. Это дерево можно «интерпретировать» с помощью алгебры. На этот раз, это алгебра из категории эндофункторов, поэтому ее носителем является эндофунктор G, а структурное отображение α представляет собой естественное преобразование $\Phi_F G \to G$:

$$\alpha : Id + F \circ G \to G$$

Это естественное преобразование, являющееся отображением-вне суммы, эквивалентно паре естественных преобразований

$$\lambda : Id \to G$$
$$\rho : F \circ G \to G$$

Можно перевести это на Haskell как пару полиморфных функций:

```
type MAlg f g a = (a -> g a, f (g a) -> g a)
```

Поскольку свободная монада является инициальной алгеброй, существует единственное отображение — катаморфизм — от нее к любой другой алгебре. Напомним, как мы определили катаморфизм регулярной алгебры:

```
cata    :: Functor f => Algebra f a ->
                                Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

Выходная часть разворачивает содержимое неподвижной точки. Здесь это можно сделать путем сопоставления с образцом двух конструкторов свободной монады. Если это лист, к нему применяется λ. Если это узел, то его содержимое обрабатывается рекурсивно и ρ применяется к результату:

```
mcata         :: Functor f => MAlg f g a ->
                              FreeMonad f a -> g a
mcata (l, r) (Pure a)
              = l a
mcata (l, r) (Free ffa)
              = r (fmap (mcata (l, r)) ffa)
```

Многие древовидные монады на самом деле являются свободными монадами для простых функторов.

Упражнение 14.8.1. (*Непустое*) *розовое дерево определяется как*:

```
data Rose a = Leaf a | Rose [Rose a]
  deriving Functor
```

Реализуйте преобразование между `Rose a` *и* `FreeMonad [] a`, *туда и обратно*.

Упражнение 14.8.2. *Реализуйте преобразования между бинарным деревом и* `FreeMonad Bin a`, *где*:

```
data Bin a = Bin a a
```

Упражнение 14.8.3. *Найдите функтор, чья свободная монада эквивалентна монаде списка* `[a]`.

Пример стекового калькулятора

В качестве примера рассмотрим стековый калькулятор, реализованный в виде встроенного предметно-ориентированного языка EDSL. Мы будем использовать свободную монаду для накопления простых команд, написанных на этом языке.

Команды определяются функтором `StackF`. Считайте параметр `k` продолжением.

```
data StackF k = Push Int k
              | Top (Int -> k)
              | Pop k
              | Add k
  deriving Functor
```

Например, предполагается, что `Push` помещает целое число в стек, а затем вызывает продолжение `k`.

Свободную монаду для этого функтора можно представить как дерево, в котором большинство ветвей имеют только один дочерний элемент, образуя, таким образом, списки. Исключением является узел `Top`, у которого много потомков, по одному на каждое значение `Int`.

Свободная монада для этого функтора:

```
type FreeStack = FreeMonad StackF
```

Для создания предметно-ориентированных программ определим несколько вспомогательных функций. Нужна общая функция, которая поднимает функтор значений до свободной монады:

```
liftF   :: (Functor f) => f r -> FreeMonad f r
liftF fr = Free (fmap (Pure) fr)
```

Также понадобится ряд «умных конструкторов», которые являются стрелками Клейсли для нашей свободной монады:

```
push  :: Int -> FreeStack ()
push n = liftF (Push n ())

pop   :: FreeStack ()
pop    = liftF (Pop ())

top   :: FreeStack Int
top    = liftF (Top id)

add   :: FreeStack ()
add    = liftF (Add ())
```

Поскольку свободная монада — это, прежде всего, монада, можно удобно комбинировать стрелки Клейсли, используя `do`-нотацию. К примеру, вот маленькая программа, складывающая два числа и возвращающая их сумму:

```
calc :: FreeStack Int
calc  = do
          push 3
          push 4
          add
          x <- top
          pop
          return x
```

Чтобы выполнить эту программу, нужно определить алгебру, носителем которой является эндофунктор. Поскольку мы хотим реализовать калькулятор на основе стека, то будем использовать версию этого функтора

состояния. Это состояние представляет собой стек — список целых чисел. Этот функтор состояния определен как функциональный тип, здесь, это функция, принимающая список и возвращающая новый список вместе с параметром типа `k`:

```
data StackAction k = St ([Int] -> ([Int], k))
  deriving Functor
```

Для запуска этого действия применим эту функцию к стеку:

```
runAction             :: StackAction k ->
                            [Int] -> ([Int], k)
runAction (St akt) ns = akt ns
```

Определим алгебру как пару полиморфных функций, соответствующих двум конструкторам свободной монады, `Pure` и `Free`:

```
runAlg :: MAlg StackF StackAction a
runAlg  = (stop, go)
```

Первая функция завершает выполнение программы и возвращает значение:

```
stop  :: a -> StackAction a
stop a = St (\xs -> (xs, a))
```

Второй функциональный шаблон соответствует типу команды. Каждая команда имеет продолжение. Это продолжение должно выполняться с (возможно, измененным) стеком. Каждая команда изменяет стек по-своему:

```
go            :: StackF (StackAction k) ->
                                StackAction k
go (Pop k)     = St (\ns -> runAction k
                                   (tail ns))
go (Top ik)    = St (\ns -> runAction
                              (ik (head ns)) ns)
go (Push n k) = St (\ns -> runAction k
                                    (n : ns))
go (Add k)     = St (\ns -> runAction k
                     ((head ns +
                       head (tail ns)) :
                          tail (tail ns)))
```

Например, `Pop` отбрасывает вершину стека. `Top` выбирает целое число из вершины стека и использует его для выбора выполняемой ветви, применяя функцию `ik` к целому числу. `Add` добавляет два числа в верх стека, сдвигая результат.

Заметим, что определенная нами алгебра не использует рекурсию. Отделение рекурсии от действий — одно из преимуществ подхода, использующего свободные монады. Вместо этого, рекурсия закодирована в катаморфизме.

Функция, которую можно использовать для запуска нашей программы:

```
run      :: FreeMonad StackF k -> ([Int], k)
run prog = runAction (mcata runAlg prog) []
```

Очевидно, что использование частичных функций `head` и `tail` делает интерпретатор ненадежным. Плохо сформированная программа вызовет ошибку времени выполнения. Более надежная реализация будет использовать алгебру, которая допускает распространение ошибок.

Но, еще одно преимущество использования свободных монад состоит в том, что одну и ту же программу можно интерпретировать с использованием разных алгебр.

Упражнение 14.8.4. *Реализуйте «простенький принтер», выводящий на экран программу, построенную с помощью нашей свободной монады. Подсказка*: *реализуйте алгебру, использующую функтор* `Const` в качестве носителя:

```
showAlg :: MAlg StackF (Const String) a
```

14.9 Моноидальные функторы

Мы уже рассмотрели несколько примеров моноидальных категорий. Такие категории снабжены какой-либо бинарной операцией, например, декартовым произведением, суммой, композицией (в категории эндофункторов) и т.д. Они также имеют специальный объект, служащий единицей по отношению к этой бинарной операции. Законы единицы и ассоциативности выполняются, либо непременно (в строгих моноидальных категориях), либо с точностью до изоморфизма.

Каждый раз, когда мы имеем дело с более, чем одним экземпляром какой-либо структуры, то можем задать себе вопрос: существует ли целая категория таких структур?

В рассматриваемом случае: образуют ли моноидальные категории свою собственную категорию? Чтобы это было так, нужно сначала определить стрелки между моноидальными категориями.

Моноидальный функтор F от моноидальной категории $(\mathcal{C}, \otimes, i)$ к другой моноидальной категории $(\mathcal{D}, \otimes, j)$ отображает тензорное произведение в тензорное произведение, а и единицу в единицу, — все с точностью до изоморфизма:

$$Fa \otimes Fb \cong F(a \otimes b)$$
$$j \cong Fi$$

Здесь, слева, тензорное произведение и единица — в целевой категории, а справа — их аналоги в исходной категории.

Если рассматриваемые моноидальные категории не являются строгими, т.е. законы единицы и ассоциативности выполняются только с точностью до изоморфизма, то существуют дополнительные условия когерентности, обеспечивающие отображение униторов в униторы, а ассоциаторов в ассоциаторы.

Категория моноидальных категорий с моноидальными функторами в качестве стрелок обозначается $\mathbf{MonCat}$. Фактически, это 2-категория, поскольку можно определить сохраняющие структуру естественные преобразования между моноидальными функторами.

Слабые моноидальные функторы

Одним из преимуществ моноидальных категорий является то, что они позволяют определять моноиды. Просто убедиться, что моноидальные функторы отображают моноиды в моноиды. Оказывается, что для выполнения этого, не требуется вся мощь моноидальных функторов. Давайте рассмотрим, каковы минимальные требования к функтору для отображения моноидов в моноиды.

Начнем с моноида (m, μ, η) в моноидальной категории $(\mathcal{C}, \otimes, i)$. Рассмотрим функтор F, отображающий m к Fm. Мы хотим, чтобы Fm был моноидом в целевой моноидальной категории $(\mathcal{D}, \otimes, j)$. Для этого нужно

найти два отображения:

$$\eta' : j \to Fm$$
$$\mu' : Fm \otimes Fm \to Fm$$

удовлетворяющие моноидальным законам.

Поскольку m — моноид, в нашем распоряжении имеются поднятия исходных отображений:

$$F\eta : Fi \to Fm$$
$$F\mu : F(m \otimes m) \to Fm$$

Чего не хватает для реализации η' и μ', так это двух дополнительных стрелок:

$$j \to Fi$$
$$Fm \otimes Fm \to F(m \otimes m)$$

Моноидальный функтор предоставил бы такие стрелки. Однако для того, что мы пытаемся сделать, не требуется обратимость этих стрелок.

Слабый моноидальный функтор — это функтор, снабженный морфизмом ϕ_i и естественным преобразованием ϕ_{ab}:

$$\phi_i : j \to Fi$$
$$\phi_{ab} : Fa \otimes Fb \to F(a \otimes b)$$

удовлетворяющие соответствующим условиям унитальности и ассоциативности.

Такой функтор отображает моноид (m, μ, η) к моноиду (Fm, μ', η') с:

$$\eta' = F\eta \circ \phi_i$$
$$\mu' = F\mu \circ \phi_{ab}$$

Простейшим примером слабого моноидального функтора является эндофунктор, который сохраняет обычное декартово произведение. Его можно определить в Haskell как класс типов:

```
class Monoidal f where
  unit  :: f ()
  (>*<) :: f a -> f b -> f (a, b)
```

В соответствии с ϕ_{ab} имеется инфиксный оператор, который, согласно соглашениям Haskell, записывается в каррированной форме.

Упражнение 14.9.1. *Реализуйте экземпляр* `Monoidal` *для функтора списка.*

Функториальная прочность

Существует еще один способ, которым функтор может взаимодействовать со скрытой моноидальной структурой, когда мы программируем. Считается само собой разумеющимся, что функции имеют доступ к окружению. Такие функции называются замыканиями.

Пример функции, которая получает переменную `a` из среды и спаривает ее со своим аргументом:

```
\x -> (a, x)
```

Это определение, само по себе, не имеет смысла, но он появляется, когда окружение содержит переменную `a`, например,

```
pairWith  :: Int -> (String -> (Int, String))
pairWith a = \x  -> (a, x)
```

Функция, возвращаемая при вызове `pairWith 5` «закрывает» 5 из своего окружения.

Теперь рассмотрим следующую модификацию, которая возвращает одноэлементный список, содержащий замыкание:

```
pairWith'  :: Int -> [String -> (Int, String)]
pairWith' a = [\x -> (a, x)]
```

Как программист, вы были бы очень удивлены, если бы это не сработало. Но то, что здесь делается, весьма нетривиально: мы протаскиваем окружение под функтором списка. Согласно нашей модели лямбда-исчисления, замыкание — это морфизм произведения окружения и аргумента функции, здесь `(Int, String)`.

Свойство, которое позволяет переносить окружение под функтором, называется *функториальной прочностью* или *тензориальной прочностью* и может быть реализовано на Haskell как:

```
strength         :: Functor f => (e, f a) ->
                                 f (e, a)
strength (e, as) = fmap (e, ) as
```

Обозначение `(e, )` называется *сечением кортежа* и эквивалентно частичному применению конструктора пары: `(,) e`.

В теории категорий, прочность для эндофунктора F определяется как естественное преобразование, так что давайте перенесем тензорное произведение в функтор:

$$\sigma : a \otimes F(b) \to F(a \otimes b)$$

Имеются некоторые дополнительные условия, которые гарантируют, что это хорошо работает с унуторами и ассоциатором рассматриваемой моноидальной категории.

Тот факт, что мы смогли реализовать `strength` для любого функтора, означает, что в Haskell каждый функтор является прочным. По этой причине не нужно беспокоиться о доступе к окружению изнутри функтора.

Что еще более важно, каждая монада в Haskell прочная в силу того, что она является функтором. Вот почему каждая монада автоматически является `Monoidal`.

```
instance Monad m => Monoidal m where
  unit       = return ()
  ma >*< mb = do
                a <- ma
                b <- mb
                return (a, b)
```

(предупреждение: чтобы скомпилировать этот код, потребуется включить несколько расширений компилятора.) Если вы избавитесь от синтаксического сахара в этом коде, чтобы использовать монадическую привязку и лямбда-выражения, то заметите, что для окончательного `return` требуется доступ как к `a`, так и к `b`, которые определены во внешних средах. Это было бы невозможно без прочной монады.

Однако, в теории категорий не каждый эндофунктор моноидальной категории является прочным. Причина, по которой это работает в Haskell, заключается в том, что категория, с которой мы работаем, является декартово замкнутой. На данный момент, магическое заклинание

состоит в том, что такая категория является само-обогащенной, поэтому каждый эндофунктор канонически обогащается. Мы вернемся к этому, когда будем говорить о расширенных категориях. При Haskell-прочности это сводится к тому, что `fmap` можно всегда частично применить к конструктору пары `(a, )`.

Аппликативные функторы

В программировании идея аппликативных функторов возникла из следующего вопроса: функтор позволяет поднять функцию одной переменной, а как можно поднять функцию двух или более переменных?

По аналогии с `fmap`, хотелось бы иметь функцию:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Функция двух аргументов — здесь, в ее каррированной форме, — это функция одного аргумента, возвращающая функцию. Итак, предполагая, что `f` — функтор, можно обработать с помощью `fmap`, первый аргумент `liftA2`, который имеет тип:

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

над вторым аргументом, `(f a)`, получая:

```
f (b -> c)
```

Проблема в том, что непонятно, как применить `f (b -> c)` к оставшемуся аргументу `(f b)`.

Существует класс функторов `Applicative`, которые позволяют это делать. Оказывается, если известно, как поднимать функцию с двумя аргументами, то можно поднимать функции с любым количеством аргументов, кроме нуля. Функция без аргументов — это просто значение, поэтому его поднятие означает реализацию функции:

```
pure :: a -> f a
```

Вот определение на Haskell:

```
class Functor f => Applicative f where
  pure  :: a          -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Это применение функтора, заполненного функциями, к функтору, заполненному аргументами, определяется как инфиксный оператор, который принято обозначать `<*>` (оператор «звездочка»).

Существует также инфиксная версия `fmap`:

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

которую можно использовать в краткой реализации `liftA2`:

```
liftA2        :: Applicative f =>
                 (a -> b -> c) -> f a ->
                                  f b -> f c
liftA2 g as bs = g <$> as <*> bs
```

Оба оператора привязываются к левому краю, что делает этот синтаксис имитирующим применение обычной функции.

Аппликативный функтор также должен удовлетворять ряду законов:

```
pure id <*> v = v                    --
тождественность
pure f <*> pure x
              = pure (f x)           -- гомоморфизм
u <*> pure y  = pure ($ y) <*> u     -- обмен
pure (.) <*> u <*> v <*> w
              = u <*> (v <*> w)      -- композиция
```

Упражнение 14.9.2. *Реализуйте* `liftA3`, *функцию, которая поднимает функцию с тремя аргументами с помощью аппликативного функтора.*

Замкнутые функторы

Если внимательно посмотреть на определение оператора звездочки

```
(<*>) :: f (a -> b) -> (f a -> f b)
```

то можно увидеть в нем отображение функционального объекта к функциональному объекту.

Это становится яснее, если рассмотреть функтор между двумя категориями, обе из которых замкнуты. Можете начать с функционального объекта b^a в исходной категории и применить к нему функтор F:

$$F(b^a)$$

В качестве альтернативы, можно отобразить два объекта a и b и построить между ними функциональный объект в целевой категории:

$$(Fb)^{Fa}$$

Если потребовать, чтобы два пути были изоморфны, то получим определение строгого *замкнутого функтора*. Но, как и в случае с моноидальными функторами, больший интерес вызывает нестрогая версия, снабженная односторонним естественным преобразованием:

$$F(b^a) \to (Fb)^{Fa}$$

Если F является эндофунктором, это напрямую переводится в определение оператора звездочки.

Полное определение нестрогого замкнутого функтора включает отображение моноидальной единицы и некоторые условия когерентности.

В замкнутой декартовой категории экспоненциал связан с декартовым произведением посредством каррирующего сопряжения. Поэтому неудивительно, что в такой категории слабые моноидальные и слабые замкнутые эндофункторы совпадают.

Можно легко выразить это на Haskell:

```
instance (Functor f, Monoidal f) =>
                      Applicative f where
  pure a    = fmap (const a) unit
  fs <*> as = fmap apply (fs >*< as)
```

где `const` — функция, игнорирующая второй аргумент:

```
const    :: a -> b -> a
const a b = a
```

а `apply` — это некаррированное применение функции:

```
apply       :: (a -> b, a) -> b
apply (f, a) = f a
```

Наоборот, имеем:

```
instance Applicative f => Monoidal f where
  unit      = pure ()
  as >*< bs = (,) <$> as <*> bs
```

В последнем случае, мы использовали конструктор пары `(,)` как функцию с двумя аргументами.

Монады и аппликативы

Поскольку в декартово замкнутой категории каждая монада является слабой моноидальной, она автоматически является аппликативом. Можно показать это напрямую, реализовав `ap`, который имеет сигнатуру того же типа, что и оператор звездочки:

```
ap      :: (Monad m) => m (a -> b) -> m a -> m b
ap fs as = do
             f <- fs
             a <- as
             return (f a)
```

Эта связь выражается в определении `Monad` с помощью `Applicative` в качестве суперкласса:

```
class Applicative m => Monad m where
  (>>=)  :: forall a b. m a -> (a -> m b) -> m b
  return :: a -> m a
  return  = pure
```

Обратите внимание на реализацию `return`, по умолчанию, как `pure`.

Обратное не является верным: не всякий `Applicative` является `Monad`. Стандартным контр-примером является экземпляр `Applicative` для функтора списка, который использует архивирование:

```
instance Applicative [] where
  pure      = repeat
  fs <*> as = zipWith apply fs as
```

Конечно, функтор списка также является монадой, поэтому на его основе существует еще один экземпляр `Applicative`. Его оператор звездочки применяет каждую функцию к каждому аргументу.

В программировании, монада является более мощной, чем аппликатив. Это потому, что монадический код позволяет исследовать содержимое монадического значения и осуществлять переход в зависимости от него. Это верно даже для монады `IO`, которая, в противном случае, не предоставляет средств для извлечения значения. В этом примере мы переходим к содержимому объекта `IO`:

```
main :: IO ()
main  = do
          s <- getLine
          if s == "yes"
          then putStrLn "Thank you!"
          else putStrLn "Next time."
```

Конечно, проверка значения откладывается до тех пор, пока интерпретатор времени выполнения `IO` не получит этот код.

Аппликативная композиция с использованием оператора звездочки не позволяет одной части вычислений проверять результат другой. Это ограничение можно превратить в преимущество. Отсутствие зависимостей позволяет проводить вычисления параллельно. Параллельные библиотеки Haskell широко используют аппликативное программирование.

С другой стороны, монады позволяют использовать очень удобный синтаксис `do`, который, возможно, более удобочитаем, чем аппликативный синтаксис. К счастью, существует расширение языка `ApplicativeDo`, которое указывает компилятору выборочно использовать аппликативные конструкции при интерпретации блоков `do`, при отсутствии зависимостей.

Упражнение 14.9.3. *Проверьте законы* `Applicative` *для экземпляра архивирования функтора списка.*

Глава 15

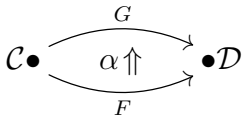
Монады из сопряжений

15.1 Струнные диаграммы

Прямую на плоскости можно представлять, либо как разделение плоскости, либо как соединение двух частей плоскости.

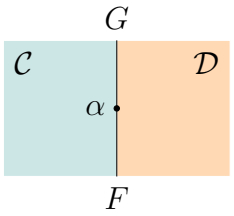
Точку на линии можно представлять, либо как разделяющую две ее части, либо как их соединение.

В традиционном диаграммном представлении, зачастую, категории представляются точками, функторы — стрелками, а естественные преобразования — двойными стрелкамию.



Но та же идея может быть представлена изображением: категорий — в виде областей плоскости, функторов — в виде линий между областями и естественных преобразований — в виде точек, соединяющих линии.

Суть в том, что функтор всегда находится между парой категорий, поэтому его можно провести как границу между ними. Естественное преобразование всегда проходит между парой функторов, поэтому его можно изобразить в виде точки, соединяющей два отрезка прямой:



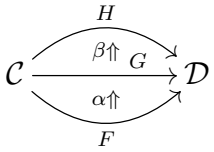
Это пример, так называемой, *струнной диаграммы*. Интерпретация таких диаграмм происходит снизу вверх, слева направо (как в системе координат (x, y)).

Внизу этой диаграммы показан функтор F, идущий от $\mathcal{C}$ к $\mathcal{D}$. В верхней части диаграммы изображен функтор G, который проходит между теми же двумя категориями. Переход происходит в середине, где естественное преобразование α отображает F к G.

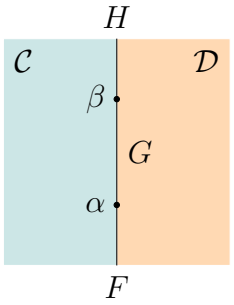
На Haskell эта диаграмма интерпретируется как полиморфная функция между двумя эндофункторами:

```
alpha :: forall x. F x -> G x
```

Пока что, не похоже, что мы много выигрываем, используя это новое визуальное представление. Но давайте применим его к чему-то более интересному: вертикальной композиции естественных преобразований:



На соответствующей струнной диаграмме показаны две категории и три функтора между ними, соединенные двумя естественными преобразованиями:



Как видите, можно восстановить исходную диаграмму из диаграммы струн, просканировав ее снизу вверх.

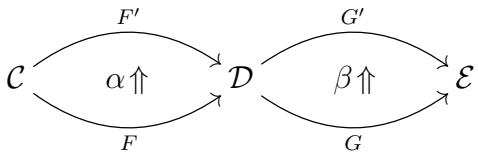
Опять же, в Haskell мы будем иметь дело с тремя эндофункторами и вертикальной композицией `beta` после `alpha`:

```
alpha :: forall x. F x -> G x
beta  :: forall x. G x -> H x
```

реализуется с помощью обычной функциональной композиции:

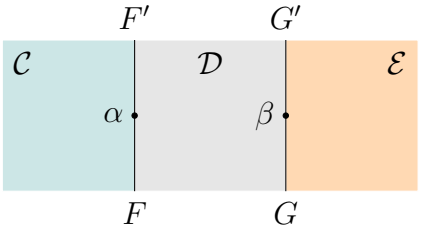
```
beta_alpha :: forall x. F x -> H x
beta_alpha  = beta . alpha
```

Продолжим с горизонтальной композицией естественных преобразований:



На этот раз присутствуют три категории, поэтому на струнной диаграмме должно быть три области.

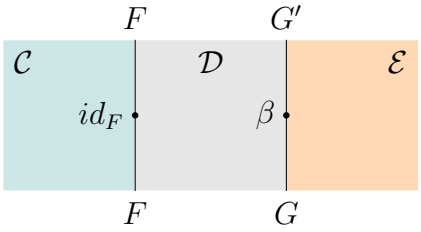
Нижняя часть струнной диаграммы соответствует композиции функторов $G \circ F$ (именно в таком порядке). Верх соответствует $G' \circ F'$. Одно естественное преобразование, α, соединяет F с F'; другое, β, — G с G'.



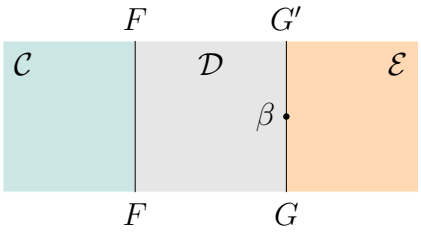
Параллельные вертикальные линии в этой новой системе соответствуют функторной композиции.

Можно представить себе горизонтальную композицию естественных преобразований как проходящую вдоль воображаемой горизонтальной линии в середине диаграммы. Но что, если на изображении одна из точек оказалась немного выше другой? Как оказывается, точное расположение точек не имеет значения из-за закона обмена.

Но сначала, проиллюстрируем вискеринг: горизонтальную композицию, в которой одним из естественных преобразований является тождество. Можно изобразить это так:



Но, на самом деле, тождественность можно вставить в любую точку вертикальной линии, так что даже не нужно ее изображать. Следующая диаграмма представляет вискеринг $\beta \circ F$.



На Haskell, где `beta` — это полиморфная функция:

```
beta :: forall x. G x -> G' x
```

последнюю диаграмму можно представить как:

```
beta_f :: forall x. G (F x) -> G' (F x)
beta_f  = beta
```

с пониманием того, что средство проверки типов создает полиморфную функцию `beta` корректного типа.

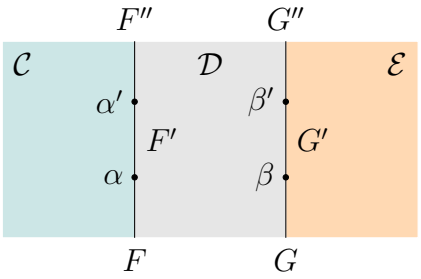
Точно так же можно легко представить себе диаграмму для $G \circ \alpha$ и ее реализацию на Haskell:

```
g_alpha :: forall x. G (F x) -> G (F' x)
beta_f   = fmap alpha
```

с

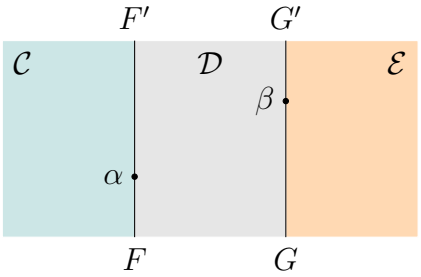
```
alpha :: forall x. F x -> F' x
```

Струнная диаграмма, соответствующая закону обмена, имеет вид:



Эта диаграмма намеренно неоднозначна. Должны ли мы сначала делать вертикальную композицию естественных преобразований, а затем горизонтальную? Или сначала надо скомпоновать $\beta \circ \alpha$ и $\beta' \circ \alpha'$ по горизонтали, а затем скомпоновать результаты по вертикали? Закон взаимообмена говорит, что это не имеет значения: результат будет один и тот же.

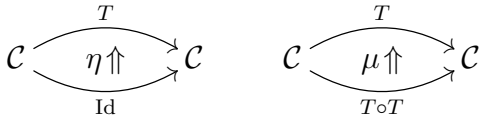
Теперь попробуем заменить пару естественных преобразований на этой диаграмме тождественностями. Если заменить α' и β', то получим горизонтальную композицию $\beta \circ \alpha$. Если же заменить α' и β тождественными естественными преобразованиями, и переименовать β' в β, то получится диаграмму, в которой α сдвинута вниз по отношению к β, и т.д.



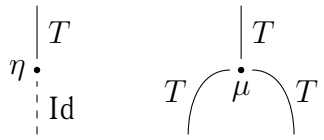
Закон обмена выражает равнозначность всех этих диаграмм. Мы вольны нанизывать естественные преобразования на линии, как бусинки на нитку.

Струнные диаграммы для монад

Монада определяется как эндофунктор, снабженный двумя естественными преобразованиями, как показано на следующих диаграммах:



Поскольку в данном случае мы имеем дело только с одной категорией, при переводе этих диаграмм в струнные диаграммы можно избавиться от именования (и цветных оттенков) категорий и просто изображать только линии и точки (с их именами).

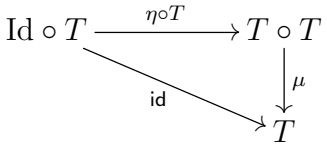


На первой диаграмме принято опускать пунктирную линию, соответствующую тождественному функтору. Точку η можно использовать для свободной вставки T-линии в диаграмму. Две линии T можно соединить точкой μ.

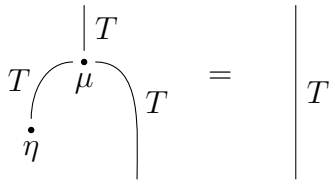
Струнные диаграммы особенно полезны для выражения монадных законов. Например, существует левый закон тождественности:

$$\mu \circ (\eta \circ T) = id$$

который можно представить в виде коммутативной диаграммы:

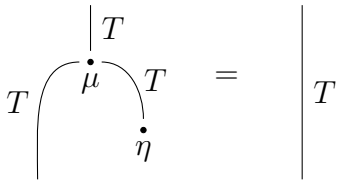


Соответствующие струнные диаграммы представляют равенство двух путей посредством следующей диаграммы:

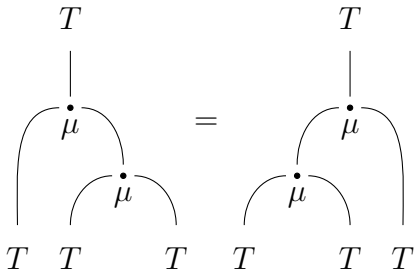


Можно представлять это равенство как результат натягивания верхней и нижней струн, в результате чего придаток η сливается с прямой линией.

Существует симметричный правый закон тождественности:



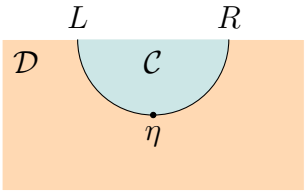
Наконец, следующая диаграмма выражает закон ассоциативности в терминах струнных диаграмм:



Струнные диаграммы для сопряжений

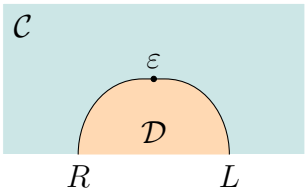
Как обсуждалось ранее, сопряжение — это отношение между парой функторов, $L : \mathcal{D} \to \mathcal{C}$ и $R : \mathcal{C} \to \mathcal{D}$. Его можно определить парой естественных преобразований, единицей η и ко-единицей ε, удовлетворяющих треугольным тождествам.

Единицу сопряжения можно проиллюстрировать диаграммой в виде «чашки»:



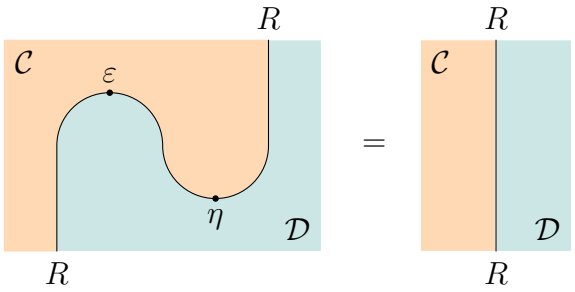
Тождественный функтор в нижней части этой диаграммы опущен. Точка η превращает тождественный функтор под ней в композицию $R \circ L$ над ней.

Точно так же, ко-единица может быть визуализирована как струнная диаграмма в форме «шапки» с неявным тождественным функтором наверху:



Треугольные тождества можно легко выразить с помощью струнных диаграмм. Они также имеют интуитивный смысл, так как можно представить, как нужно тянуть за нить с обеих сторон, чтобы выпрямить кривую.

Например, это первое треугольное тождество, иногда называемое *зигзагообразной* тождественностью:



Чтение левой диаграммы снизу вверх приводит к ряду отображений:

$$Id_{\mathcal{D}} \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ Id_{\mathcal{C}}$$

Это должно быть равно правой части, которую можно интерпретировать как (не изображенное) тождественное естественное преобразование на R.

В случае, когда R является эндофунктором, можно перевести первую диаграмму непосредственно на Haskell. Вискеринг единицы сопряжения η с помощью R приводит к тому, что полиморфная функция `unit` реализуется в `R x`. Вискеринг ε приводит к подъему `counit` функтором R. Вертикальная композиция преобразуется в функциональную композицию:

```
triangle :: forall x. R x -> R x
triangle  = fmap counit . unit
```

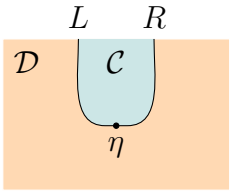
Упражнение 15.1.1. *Изобразите струнные диаграммы для второго тождества треугольника и переведите первое тождество на Haskell.*

15.2 Монады из сопряжений

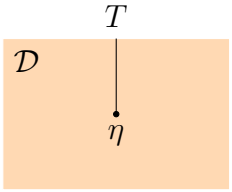
Вы могли заметить, что один и тот же символ η используется для единицы сопряжения и для единицы монады. Это не совпадение.

Сразу может показаться, что мы сравниваем яблоки с апельсинами: сопряжение определяется двумя функторами между двумя категориями, а монада определяется одним эндофунктором, работающим с одной категорией. Однако, композиция двух функторов, идущих в противоположных направлениях, является эндофунктором, а единица сопряжения отображает единичный эндофунктор к эндофунктору $R \circ L$.

Сравните эту диаграмму:

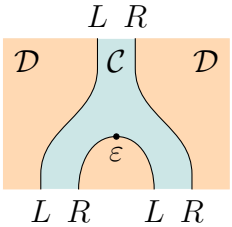


с той, которая определяет монадическую единицу:



(«механически», вторая диаграмма получается из первой, если на первой, «удерживая» точку, синхронно потянуть вверх L и R, которые соединятся в одну линию, T; *прим.перев.*).

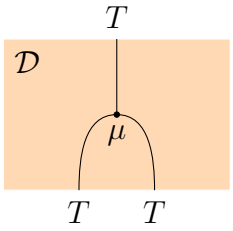
Оказывается, что, для любого сопряжения $L \dashv R$, эндофунктор $T = R \circ L$ — это монада с умножением μ, определяемая следующей диаграммой:



Считывая эту диаграмму снизу вверх, получаем следующее преобразование (представьте, что диаграмма разрезается горизонтально по точке):

$$R \circ L \circ R \circ L \xrightarrow{R \circ \varepsilon \circ L} R \circ L$$

Сравните это с определением монадического μ:



Получаем определение μ для монады $R \circ L$ как двойного вискеринга ε:

$$\mu = R \circ \varepsilon \circ L$$

Преобразование на Haskell строковой диаграммы, определяющей μ в терминах ε, всегда возможно. Монадическое умножение, или `join`, принимает вид:

```
join :: forall x. T (T x) -> T x
join  = fmap counit
```

где `fmap` соответствует поднятию эндофунктором T, определенным как композиция $R \circ L$. Заметим, что $\mathcal{D}$, в данном случае, является категорией типов и функций Haskell, но $\mathcal{C}$ может быть внешней категорией.

Для полноты картины, мы можем использовать струнные диаграммы для вывода монадических законов, используя тождества треугольников. Хитрость заключается в том, чтобы заменить все строки в монадических законах парами параллельных строк, а затем переставить их в соответствии с правилами.

Подводя итог, каждое сопряжение $L \dashv R$ с единицей η и ко-единицей ε определяет монаду $(R \circ L, \eta, R \circ \varepsilon \circ L)$.

Позже мы увидим, что, двойственно, другая композиция, $L \circ R$, определяет ко-монаду.

Упражнение 15.2.1. *Изобразите струнные диаграммы, чтобы проиллюстрировать монадические законы (единицы и ассоциативности) для монады, полученной из сопряжения.*

15.3 Примеры монад из сопряжений

Мы рассмотрим несколько примеров сопряжений, которые генерируют некоторые из монад, которые используются в программировании, и подробнее остановимся на этих примерах позже, когда будем говорить о преобразователях монад.

В большинстве примеров задействованы функторы, исходящие от категории типов и функций Haskell, даже несмотря на то, что путь туда и обратно, порождающий монаду, оказывается эндофунктором. Вот почему часто невозможно выразить такие сопряжения в Haskell.

Как будто для того, чтобы дополнительно все усложнить, существует много требований, связанных с явным именованием конструкторов данных, что необходимо для вывода типов. Иногда это может скрывать простоту основных формул.

Свободный моноид и монада списка

Монада списка генерируется сопряжением свободного моноида, с которым мы сталкивались ранее. Единица этого сопряжения, $\eta_X : X \to U(FX)$, вводит элементы множества X в качестве образующих свободного моноида FX, после чего U извлекает основное множество.

На Haskell, мы представляем свободный моноид в виде типа списка, а его образующие — одноэлементными списками. Единица η_X отображает элементы X к таким синглетонам:

```
return x = [x]
```

Чтобы реализовать ко-единицу, $\varepsilon_M : F(UM) \to M$, мы используем моноид M, забываем о его умножении и используем его множество элементов в качестве образующих для нового свободного моноида. Компонент ко-единицы является тогда моноидным морфизмом от этого свободного моноида обратно к M. Оказывается, что этот моноидный морфизм является частным случаем катаморфизма.

Прежде всего, напомним реализацию общего спискового катаморфизма на Haskell:

```
foldMap  :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

Мы можем интерпретировать это как взятие функции от `a` к основному множеству моноида `m` и создание моноидного морфизма от свободного моноида, порожденного `a` (это есть список этих `a`), к `m`.

Коединица является моноидным морфизмом `[m] -> m`. Мы получаем его, применяя `foldMap` к тождественности. Результатом является `foldMap id`, или, в терминах `foldr`:

```
epsilon = foldr mappend mempty
```

Это моноидный морфизм, поскольку он отображает пустой список к моноидальной единице, а конкатенацию — к моноидальному произведению.

Монадное умножение задается вискерингом ко-единицы:

$$\mu = U \circ \varepsilon \circ F$$

Можно легко убедиться, что вискеринг слева здесь ничего не значит, так как это просто поднятие моноидного морфизма забывающим функтором. Он сохраняет функцию, забывая при этом о своем особом свойстве сохранять структуру.

Правый вискеринг при F интереснее. Он означает, что компонент μ_X соответствует компоненте ε при FX, который является свободным моноидом, сгенерированном из множества X. Этот свободный моноид определяется следующим образом:

```
mempty  = []
mappend = (++)
```

что задает определение μ или `join`:

```
join = foldr (++) []
```

Как и ожидалось, это то же самое, что и `concat`. В монаде списка умножение является конкатенацией.

Каррирование сопряжения и монада состояния

Монада состояния генерируется каррированием сопряжения, которое мы использовали для определения экспоненциального объекта. Левый функтор определяется произведением с некоторым фиксированным объектом s:

$$L_s a = a \times s$$

Его можно реализовать как тип Haskell:

```
newtype L s a = L (a, s)
```

Правый функтор — это возведение в степень, параметризованное тем же объектом s:

$$R_s c = c^s$$

На Haskell, это тонко инкапсулированный функциональный тип:

```
newtype R s c = R (s -> c)
```

Монада задается композицией этих двух функторов. На объектах:

$$(R_s \circ L_s)a = (a \times s)^s$$

На Haskell мы бы записали это так:

```
newtype St s a = St (R s (L s a))
```

Если расширить это определение, то в нем легко распознать функтор `State`:

```
newtype State s a = State (s -> (a, s))
```

Единицей сопряжения $L_s \dashv R_s$ является отображение:

$$\eta_a : a \to (a \times s)^s$$

который может быть реализован в Haskell как:

```
unit  :: a -> R s (L s a)
unit a = R (\s -> L (a, s))
```

Можно распознать в нем слегка завуалированную версию `return` для монады состояния:

```
return  :: a -> State s a
return a = State (\s -> (a, s))
```

Компонент ко-единицы этого сопряжения при c есть:

$$\varepsilon_c : c^s \times s \to c$$

Это может быть реализовано на Haskell как:

```
counit                :: L s (R s a) -> a
counit (L ((R f), s)) = f s
```

что, после удаления конструкторов данных, эквивалентно `apply`, или некаррированной версии `runState`.

Monad multiplication μ is given by the whiskering of ε from both sides:

$$\mu = R_s \circ \varepsilon \circ L_s$$

Вот как это переводится на Haskell:

```
mu :: R s (L s (R s (L s a))) -> R s (L s a)
mu  = fmap counit
```

Здесь, вискеринг справа не делает ничего, кроме выбора компонента естественного преобразования. Это делается автоматически механизмом вывода типов Haskell.

Вискеринг слева выполняется путем поднятия компонента естественного преобразования. Опять же, вывод типов выбирает правильную реализацию `fmap` (здесь это эквивалентно пред-композиции).

Сравните это с реализацией `join`:

```
join    :: State s (State s a) -> State s a
join mma = State (fmap (uncurry runState)
                                (runState mma))
```

Обратите внимание на двойное использование `runState`:

```
runState            :: State s a -> s -> (a, s)
runState (State h) s = h s
```

Когда он некаррирован, его сигнатура типа становится такой:

```
uncurry runState :: (State s a , s) -> (a, s)
```

что эквивалентно `counit`.

При частичном применении, `runState` просто удаляет конструктор данных, раскрывая базовый функциональный тип:

```
runState st :: s -> (a, s)
```

M-множества и монада Writer

Записывающая монада:

```
newtype Writer m a = Writer (a, m)
```

параметризуется моноидом `m`. Этот моноид используется для накопления записей журнала. Сопряжение, которое мы собираемся использовать, включает категорию M-множеств для этого моноида.

M-множество — это множество S, на котором определено действие моноида M. Такое действие является отображением:

$$a : M \times S \to S$$

Мы часто используем каррированную версию действия с элементом моноида в позиции нижнего индекса. Таким образом, a_m становится функцией $S \to S$.

Это отображение должно удовлетворять некоторым ограничениям. Действие моноидальной единицы 1 не должно изменять множество, поэтому оно должно быть тождественной функцией:

$$a_1 = \mathsf{id}_S$$

а два последовательных действия должны сочетаться с действием их моноидального произведения:

$$a_{m_1} \circ a_{m_2} = a_{m_1 \cdot m_2}$$

Этот выбор порядка умножения определяет то, что названо *левым действием* (правое действие меняет местами два моноидальных элемента в правой части).

M-множества образуют категорию **MSet**. Объекты представляют собой пары $(S, a : M \times S \to S)$, а стрелки представляют собой *эквивариантные отображения*, то есть функции между множествами, сохраняющие действия.

Функция $f : S \to R$ является *эквивариантным* отображением от (S, a) к (R, b), если следующая диаграмма коммутативна для каждого $m \in M$:

$$\begin{array}{ccc} S & \xrightarrow{f} & R \\ {\scriptstyle a_m}\downarrow & & \downarrow{\scriptstyle b_m} \\ S & \xrightarrow[f]{} & R \end{array}$$

Другими словами, не имеет значения, выполним ли мы сначала действие a_m, а затем отобразим множество; или сначала отобразим множество, а затем выполним соответствующее действие b_m.

Существует забывающий функтор U от **MSet** к **Set**, который назначает множеству S пару (S, a), таким образом забывая о действии.

Ему соответствует свободный функтор F. Его действие на множество S порождает M-множество. Это есть множество, являющееся декартовым произведением S и M, где M рассматривается как множество элементов (иными словами, результат действия забывающего функтора на моноиде). Элементом этого M-множества является пара $(x \in S, m \in M)$, а свободное действие определяется следующим образом:

$$\phi_n : (x, m) \mapsto (x, n \cdot m)$$

оставляя элемент x без изменений и умножая только m-компоненту.

Чтобы показать, что F является левым сопряженным к U, мы должны построить следующий естественный изоморфизм, для любого множества S и любого M-множества Q:

$$\mathbf{MSet}(FS, Q) \cong \mathbf{Set}(S, UQ)$$

Если представить Q парой (R, b), то элемент правой стороны сопряжения есть функция $g : S \to R$.

Ключевой момент здесь заключается в том, чтобы заметить, что эквивариантное отображение f слева полностью определяется своим действием на элементы формы $(x, 1) \in FS$, что, в свою очередь, полностью определяется $g : x \mapsto p$.

Действительно, из условия эквивалентности следует, что:

$$\begin{array}{ccc} (x,1) & \xrightarrow{\quad f \quad} & r \\ {\scriptstyle \phi_m}\downarrow & & \downarrow{\scriptstyle b_m} \\ (x, m \cdot 1) & \xrightarrow[\quad f \quad]{} & r' \end{array}$$

или

$$f(x, m) = b_m(f(x, 1))$$

Таким образом, каждая функция $g : S \to R$ однозначно определяет эквивариантное отображение $f : FS \to Q$:

$$f(x, m) = b_m(gx)$$

Единица этого сопряжения $\eta_S : S \to U(FS)$ отображает элемент x к паре $(x, 1)$. Сравните это с определением `return` для записывающей монады:

```
return a = Writer (a, mempty)
```

Ко-единица задается эквивариантным отображением:

$$\varepsilon_Q : F(UQ) \to Q$$

Левая часть представляет собой M-множество, построенное путем использования основного множества Q и его произведения с основным множеством M. Исходное действие Q забывается и заменяется свободным действием. Очевидный выбор для ко-единицы:

$$\varepsilon_Q : (x, m) \mapsto a_m x$$

где x — элемент (основного множества) Q, а a — действие, определенное в Q.

Монадное умножение μ задается вискерингом ко-единицы.

$$\mu = U \circ \varepsilon \circ F$$

Это означает замену Q в определении ε_Q на свободное M-множество, действие которого является свободным действием. Другими словами, мы заменяем x на (x, m), а a_n — на ϕ_n (вискеринг с U ничего не меняет).

$$\mu_S : ((x, m), n) \mapsto \phi_n(x, m) = (x, n \cdot m)$$

Сравните это с определением `join` для записывающей монады:

```
join  :: Monoid m => Writer m (Writer m a) ->
                                    Writer m a
join (Writer ( Writer (x, m), n))
       = Writer (x, mappend n m)
```

Точечные объекты и монада Maybe

Точечные объекты — это объекты с обозначенным элементом. Поскольку выбор элемента осуществляется с помощью стрелки от терминального объекта, категория точечных объектов определяется с помощью пар $(a, p : 1 \to a)$, где a — объект в $\mathcal{C}$.

Морфизм между этими парами — это стрелки в $\mathcal{C}$, которые сохраняют точки. Таким образом, морфизм от $(a, p : 1 \to a)$ к $(b, q : 1 \to b)$ — это стрелка $f : a \to b$ такая, что $q = f \circ p$. Эта категория также называется *ко-слойной категорией* и обозначается $1/\mathcal{C}$.

Существует очевидный забывающий функтор $U : 1/\mathcal{C} \to \mathcal{C}$, который забывает о точке. Его левый сопряженный — это свободный функтор F, отображающий объект a к паре $(1 + a, \mathrm{Left})$. Другими словами, F свободно добавляет точку к объекту.

Упражнение 15.3.1. *Покажите, что $U \circ F$ является монадой* `Maybe`.

Аналогичным образом строится монада `Either`, при замене 1 фиксированным объектом e.

Монада продолжения

Монада продолжения определяется в терминах пары контравариантных функторов в категории множеств. Нам не нужно модифицировать определение сопряжения для работы с контравариантными функторами. Достаточно выбрать противоположную категорию для одной из конечных точек.

Определим левый функтор как:

$$L_Z : \mathbf{Set}^{\mathsf{op}} \to \mathbf{Set}$$

Он отображает множество X к $\hom$-множеству в $\mathbf{Set}$:

$$L_Z X = \mathbf{Set}(X, Z)$$

Этот функтор параметризуется другим множеством Z. Правый функтор определяется практически по той же формуле:

$$L_Z : \mathbf{Set} \to \mathbf{Set}^{\mathsf{op}}$$
$$L_Z X = \mathbf{Set}^{\mathsf{op}}(Z, X) = \mathbf{Set}(X, Z)$$

Композиция $R \circ L$ может быть записана на языке Haskell как `((x -> r) -> r)`, что совпадает с (ковариантным) эндофунктором, определяющим монаду продолжения.

15.4 Преобразователи монад

Предположим, что требуется объединить несколько эффектов, скажем, состояния с возможностью отказа (аварийного завершения). Один из вариантов — определить свою собственную монаду с нуля. Для этого определим функтор:

```
newtype MaybeState s a = MS (s -> Maybe (a, s))
  deriving Functor
```

и функцию для извлечения результата (или сообщения об ошибке):

```
runMaybeState          :: MaybeState s a -> s ->
                                      Maybe (a, s)
runMaybeState (MS h) s = h s
```

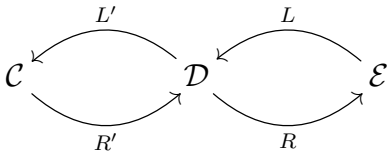
Определим экземпляр монады:

```
instance Monad (MaybeState s) where
  return a = MS (\s -> Just (a, s))
  ms >>= k = MS (\s ->
                 case runMaybeState ms s of
                 Nothing      -> Nothing
                 Just (a, s') -> runMaybeState
                                      (k a) s')
```

и, если приложить достаточно усилий, можно удостовериться, что она удовлетворяет законам монад.

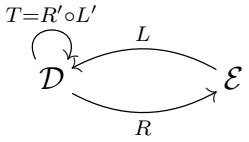
Общего рецепта комбинирования монад не существует. В этом смысле монады не компонуемы. Однако, как мы знаем, компонуемы сопряжения. Мы также знаем, как получить монады из сопряжений, и, как скоро увидим, таким способом можно получить любую монаду. Итак, если мы сможем скомпоновать сопряжения, то монады, которые они порождают, будут автоматически компоноваться.

Рассмотрим два компонуемых сопряжения:



На этой диаграмме изображены три монады: «внутренняя» монада $R' \circ L'$, «внешняя» монада $R \circ L$, а также составная — $R \circ R' \circ L' \circ L$.

Если обозначить $T = R' \circ L'$, то $R \circ T \circ L$ будет составной монадой, называемой *преобразователем монад*, потому что она преобразует монаду T в новую монаду.



В нашем примере, можно рассматривать `Maybe` как внутреннюю монаду:

$$Ta = 1 + a$$

Она преобразуется с помощью внешнего сопряжения $L_s \dashv R_s$, порождающего монаду состояния:

$$L_s a = a \times s$$
$$R_s c = c^s$$

Результатом является:

$$(R_s \circ T \circ L_s)a = (1 + a \times s)^s$$

или, на Haskell:

```
s -> Maybe (a, s)
```

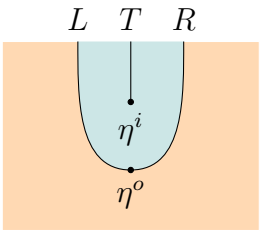
что соответствует определению нашей монады `MaybeState`.

В общем случае, внутренняя монада T определяется своей единицей η^i и умножением μ^i (верхний индекс i означает «внутренний»). «Внешнее» сопряжение определяется единицей η^o и ко-единицей ε^o.

Единицей составной монады является естественное преобразование:

$$\eta : Id \to R \circ T \circ L$$

заданное струнной диаграммой:



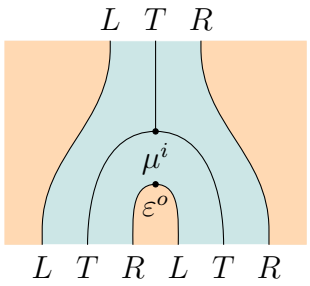
Это вертикальная композиция вискерной внутренней единицы $R \circ \eta^i \circ L$ и внешней единицы η^o. В компонентах:

$$\eta_a = R(\eta^i_{La}) \circ \eta^o_a$$

Умножение составной монады является естественным преобразованием:

$$\mu : R \circ T \circ L \circ R \circ T \circ L \to R \circ T \circ L$$

заданное струнной диаграммой:



Это вертикальная композиция много-вискерной внешней ко-единицы:

$$R \circ T \circ \varepsilon^o \circ T \circ L$$

за которой следует вискерное внутреннее умножение $R \circ \mu^i \circ L$. В компонентах:

$$\mu_c = R\left(\mu^i_{Lc}\right) \circ (R \circ T)\left(\varepsilon^o_{(T \circ L)c}\right)$$

Преобразователь монады состояния

Раскроем эти уравнения для случая преобразователя монады состояния. Монада состояния порождается каррирующим дополнением. Левый функтор L_s — это функтор произведения `(a,s)`, а правый функтор R_s — экспоненциал, также известный как считывающий функтор `(s -> a)`.

Как мы видели ранее, внешняя ко-единица ε^o_a является применением функции:

```
counit        :: (s -> a, s) -> a
counit (f, x) = f x
```

а единица η^o_a является конструктором каррированных пар:

```
unit  :: a -> s -> (a, s)
unit x = \s -> (x, s)
```

Мы оставим внутреннюю монаду (T, η^i, μ^i) произвольной. На Haskell будем обозначать эту тройку как `m`, `return` и `join`.

Составная монада, которая получается применением преобразователя для монады состояния к монаде T, представляет собой композицию $R \circ T \circ L$ или, на Haskell:

```
newtype StateT s m a   = StateT (s -> m (a, s))

runStateT             :: StateT s m
                           a -> s -> m (a, s)
runStateT (StateT h) s = h s
```

Единицей монадного преобразователя является вертикальная композиция η^o и $R \circ \eta^i \circ L$. В компонентах:

$$\eta_a = R(\eta^i_{La}) \circ \eta^o_a$$

В этой формуле есть подвижные части, поэтому проанализируем ее шаг за шагом. Начнем справа: имеется a-компонента единицы сопряжения? которая является стрелкой от a к $R(La)$. Она переведена в функцию на Haskell. Если применить эту функцию к переменной `x :: a`, то можно передать результат следующей функции в цепочке, которой является $R(\eta^i_{La})$. На Haskell, это функция `unit`.

```
unit :: a -> s -> (a, s)
```

Вычислим эту функцию при некотором `x :: a`. Результатом является другая функция `s -> (a, s)`. Мы передаем эту функцию в качестве аргумента к $R(\eta^i_{La})$.

$R(\eta^i_{La})$ — это компонент `return` внутренней монады, взятой в La. Здесь, La — это тип `(a,s)`. Таким образом, создается экземпляр полиморфной функции `return :: a -> m a` как функции `(a,s) -> m (a,s)` (система вывода типов делает это автоматически).

Далее, этот компонент `return` поднимается с помощью R. Здесь, R — это экспоненциал $(-)^s$, поэтому он поднимает функцию с помощью пост-композиции. Это будет пост-композиция `return` к любой переданной ему функции. В нашем случае это функция, которая была создана с

помощью `unit`. Заметьте, что типы совпадают: пост-компоновка `(a,s) -> m (a,s)` производится после `s -> (a, s)`.

Можно записать результат этой композиции как:

```
return x = StateT (return . \s -> (x, s))
```

или, встраивая композицию функций:

```
return x = StateT (\s -> return (x, s))
```

Мы вставили конструктор данных `StateT`, чтобы сделать проверку типов успешной. Это `return` составной монады в терминах `return` внутренней монады.

Те же рассуждения можно применить к формуле для компоненты композиции μ при некотором a:

$$\mu_a = R\left(\mu^i_{La}\right) \circ (R \circ T)\left(\varepsilon^o_{(T\circ L)a}\right)$$

Внутреннее μ^i есть `join` монады `m`. Применение R превращает его в пост-композицию.

Внешнее ε^o — это применение функции, взятое в $T(La)$, или `m (a,s)`. Это функция типа:

```
(s -> m (a, s), s) -> m (a, s)
```

которую, вставив соответствующие конструкторы данных, можно записать как `uncurry runStateT`:

```
uncurry runStateT :: (StateT s m a, s) ->
                                        m (a, s)
```

Применение $(R\circ T)$ поднимает эту компоненту ε с помощью композиции функторов R и T. Первая реализуется как пост-композиция, а вторая является `fmap` монады `m`.

Собирая все это вместе, получим бесточечную формулу для `join` преобразователя монады состояния:

```
join    :: StateT s m (StateT s m a) ->
                                    StateT s m a
join mma = StateT join . fmap
           (uncurry runStateT) . runStateT mma)
```

Здесь, частично примененный `runStateT mma` отделяет конструктор данных от аргумента `mma`:

```
runStateT mma :: s -> m (a, x)
```

Наш предыдущий пример с `MaybeState` теперь можно переписать с помощью монадного преобразователя:

```
type MaybeState s a = StateT s Maybe a
```

Исходную монаду `State` можно восстановить, применив преобразователь монады `StateT` к тождественному функтору, экземпляр `Monad` которого определен в библиотеке (обратите внимание, что последняя переменная типа `a` в этом определении опущена):

```
type State s = StateT s Identity
```

Другие преобразователи монад следуют той же схеме. Они определены в библиотеке Monad Transformer Library, MTL.

15.5 Алгебры монад

Каждое сопряжение порождает монаду, и до сих пор мы могли определять сопряжения для всех интересующих нас монад. Но всякая ли монада порождена сопряжением? Ответ — да, и обычно существует множество сопряжений — фактически целая категория сопряжений — для каждой монады.

Поиск сопряжений для монады аналогичен факторизации. Мы хотим представить функтор как композицию двух других функторов, $T = R \circ L$. Задача усложняется тем, что эта факторизация также требует нахождения соответствующей промежуточной категории. Будем искать такие категории, изучая алгебры для монад.

Монада определяется эндофунктором, для которого, как известно, можно определить алгебры. Математики часто рассматривают монады как инструменты для создания выражений, а алгебры — как инструменты для вычисления этих выражений. Однако выражения, порожденные монадами, накладывают на эти алгебры некоторые условия совместимости.

Например, можно заметить, что монадическая единица $\eta_a : a \to Ta$ имеет сигнатуру типа, которая выглядит как обратное структурное отображение алгебры $\alpha : Ta \to$ a. Конечно, η — естественное преобразование, определенное для каждого типа, тогда как алгебра имеет фиксированный тип носителя. Однако, разумно ожидать, что одно может свести на нет действие другого.

Рассмотрим ранее приведенный пример монады выражения `Ex`. Алгебра для этой монады — это выбор типа носителя, пусть, скажем, `Char`, и стрелки:

```
alg :: Ex Char -> Char
```

Поскольку `Ex` является монадой, она определяет единицу, или `return`, представляющую собой полиморфную функцию, которую можно использовать для создания простых выражений из значений. Единицей для `Ex` является:

```
return x = Var x
```

Можно конкретизировать единицу для произвольного типа, в частности для несущего типа нашей алгебры. Имеет смысл потребовать, чтобы *вычисление* `Var c`, где `c` — символ, должно вернуть то же самое `c`. Другими словами, ожидается, что:

```
alg . return = id
```

Это условие сразу устранит множество алгебр, таких как:

```
alg (Var c) = 'a' -- не совместимо с монадой Ex
```

Второе условие, которое хотелось бы наложить, заключается в том, что алгебра, совместимая с монадой, допускает подстановку. Монада позволяет выравнивать вложенные выражения с помощью `join`. Алгебра позволяет вычислять такие выражения.

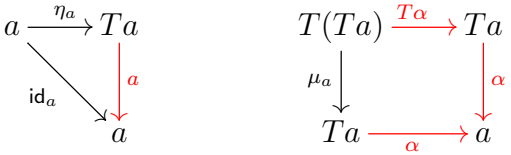
Есть два способа сделать это: можно применить алгебру к выровненному выражению или же можно сначала применить ее к внутреннему выражению (используя `fmap`), а затем вычислить полученное выражение.

```
alg (join mma) = alg (fmap alg mma)
```

где `mma` является вложенным типом `Ex (Ex Char)`.

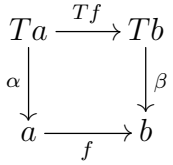
В теории категорий эти два условия определяют *алгебру монад*.

Говорят, что $(a, \alpha : Ta \to a)$ является *алгеброй монады*, для монады (T, μ, η) , если следующие диаграммы являются коммутативными:



Эти законы иногда называют законом единицы и законом умножения для монадных алгебр.

Поскольку алгебры монад — это просто специальные виды алгебр, они образуют подкатегорию алгебр. Напомним, что алгебраические морфизмы — это стрелки, удовлетворяющие следующему условию:



В свете этого определения можно переинтерпретировать вторую диаграмму монадной алгебры как утверждение, что структурное отображение α монадной алгебры (нижняя стрелка) также является алгебраическим морфизмом от (Ta, μ_a) к (a, α). Это пригодится в дальнейшем.

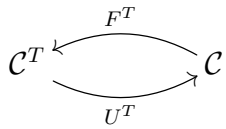
Категория Эйленберга-Мура

Категория монадных алгебр для данной монады T на $\mathcal{C}$ называется категорией Эйленберга-Мура и обозначается $\mathcal{C}^T$. Оказывается, это хороший выбор для промежуточной категории, позволяющей факторизовать монаду T, как композицию пары сопряженных функторов.

Процесс идет следующим образом: мы определяем пару функторов, показываем, что они образуют сопряжение, а затем показываем, что монада, порожденная этим сопряжением, является исходной монадой.

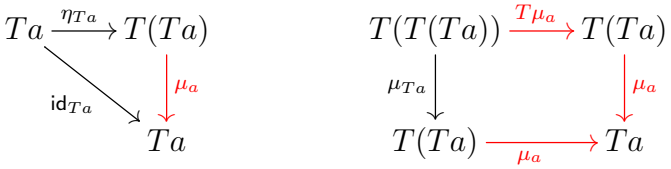
Прежде всего, существует очевидный забывающий функтор, который обозначим U^T, от $\mathcal{C}^T$ к $\mathcal{C}$. Он отображает алгебру (a, α) к ее носителю a, и рассматривает алгебраические морфизмы как регулярные морфизмы между носителями.

Что более интересно, существует свободный функтор F^T, являющийся левым сопряженным к U^T.



На объектах, F^T отображает объект a из $\mathcal{C}$ к *монадной алгебре*, объекту в $\mathcal{C}^T$. В качестве носителя этой алгебры, выберем не a, а Ta. Для структурного отображения, которое является отображением $T(Ta) \to Ta$, выбираем компонент монадного умножения $\mu_a : T(Ta) \to Ta$.

Легко проверить, что эта алгебра, (Ta, μ_a), действительно является монадной алгеброй — необходимые условия компоновки следуют из монадных законов. Действительно, подставляя алгебру (Ta, μ_a) в диаграммы монад-алгебр, получаем (с алгебраической частью, выделенной красным):



Первая диаграмма — это всего лишь левый монадический закон единицы в компонентах. Стрелка η_{Ta} соответствует вискерингу $\eta \circ T$. Вторая диаграмма представляет собой ассоциативность μ с двумя вискерингами $\mu \circ T$ и $T \circ \mu$, выраженными в компонентах.

Чтобы доказать наличие сопряжения, определим два естественных преобразования, которые будут служить единицей и ко-единицей этого сопряжения.

В качестве единицы сопряжения выберем монадическую единицу η из T. Они обе имеют одинаковую сигнатуру — в компонентах, $\eta_a : a \to U^T(F^T\mathsf{a})$.

Ко-единица является естественным преобразованием:

$$\varepsilon : F^T \circ U^T \to Id$$

Компонента ε при (a, α) представляет собой алгебраический морфизм от свободной алгебры, порожденной a, то есть, (Ta, μ_a), обратно к (a, α).

Как мы видели ранее, α само по себе является таким морфизмом. Поэтому можно выбрать $\varepsilon(a, \alpha) = \alpha$.

Тождества треугольника для этих определений η и ε следуют из законов единицы для монады и алгебры монад.

Как и для всех сопряжений, композиция $U^T \circ F^T$ является монадой. Покажем, что это та же самая монада, с которой мы начали.

Действительно, на объектах, композиция $U^T(F^T, a)$ она сначала отображает a к свободной монадной алгебре (Ta, μ), а затем забывает структурное отображение. Конечным результатом является отображение a к Ta, что и было сделано в исходной монаде.

На стрелках, она поднимает стрелку $f : a \to b$ с помощью T. Тот факт, что стрелка Tf является алгебраическим морфизмом от (Ta, μ_a) к (Tb, μ_b), следует из естественности μ:

$$\begin{array}{ccc} T(Ta) & \xrightarrow{T(Tf)} & T(Tb) \\ {\scriptstyle \mu_a}\downarrow & & \downarrow{\scriptstyle \mu_b} \\ Ta & \xrightarrow[Tf]{} & Tb \end{array}$$

Наконец, надо показать, что единица и ко-единица монады $U^T \circ F^T$ совпадают с единицей и ко-единицей исходной монады.

Единицы одинаковы по построению.

Монадное умножение $U^T \circ F^T$ задается вискерингом $U^T \circ \varepsilon \circ F^T$ единицы сопряжения. В компонентах это означает создание экземпляра ε при (Ta, μ_a), что дает μ_a (действие U^T на стрелках тривиально). Это действительно является умножением исходной монады.

Таким образом, показано, что для любой монады T можно определить категорию Эйленберга-Мура и пару сопряженных функторов, факторизующих эту монаду.

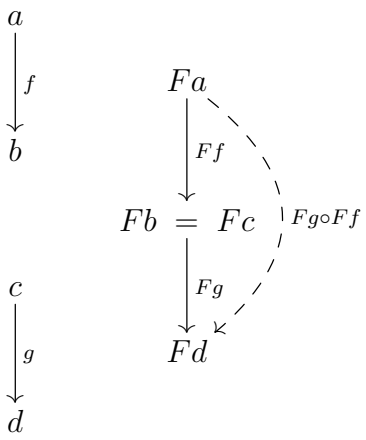
Категория Клейсли

Внутри каждой категории Эйленберга-Мура имеется меньшая *категория Клейсли*, пытающаяся выбраться из нее. Эта меньшая категория является образом свободного функтора, который мы построили в предыдущем разделе.

Несмотря на внешнее представление, образ функтора не обязательно определяет подкатегорию. Конечно, он сопоставляет тождества с тожде-

ствами и композицию с композицией. Проблема может возникнуть, если две стрелки, которые не являлись компонуемыми в исходной категории, станут таковыми в целевой категории. Это может произойти, если цель первой стрелки была сопоставлена с тем же объектом, что и источник второй стрелки.

В приведенном ниже примере, Ff и Fg компонуемы, но их композиция $Fg \circ Ff$ от образа первоначальной категории может отсутствовать.

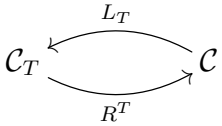


Однако, свободный функтор F^T отображает различные объекты в различные свободные алгебры, поэтому его образ действительно является подкатегорией of $\mathcal{C}^T$.

Мы уже встречались с категорией Клейсли. Существует множество способов построения одной и той же категории, и самый простой из них — описать категорию Клейсли в терминах стрелок Клейсли.

Категория Клейсли для монады (T, η, μ) обозначается $\mathcal{C}_T$. Ее объекты такие же, как объекты в $\mathcal{C}$, но стрелка в $\mathcal{C}_T$ от a к b представляется стрелкой в $\mathcal{C}$, идущей от a к Tb. Вы можете распознать ее как стрелку Клейсли `a -> m b` , которая была определена ранее. Поскольку T — монада, эти стрелки Клейсли можно компоновать с помощью оператора «рыба», `<=<`.

Чтобы установить сопряжение:



определим левый функтор $L_T : \mathcal{C} \to \mathcal{C}_T$ как тождественность на объектах. Еще нужно определить, что он делает со стрелками. Он должен отображать обычную стрелку $f : a \to b$ к стрелке Клейсли от a к b. Эта стрелка Клейсли $a \twoheadrightarrow b$ представлена стрелкой $a \to Tb$ в $\mathcal{C}$. Такая стрелка всегда существует, как составная, $\eta_b \circ f$:

$$L_T f : a \xrightarrow{f} b \xrightarrow{\eta_b} Tb$$

Правый функтор $R_T : \mathcal{C}_T \to \mathcal{C}$ определяется на объектах как отображение, переводящее a из категории Клейсли к объекту Ta из $\mathcal{C}$. Для стрелки Клейсли $a \twoheadrightarrow b$, которая представлена стрелкой $g : a \to Tb$, R_T сопоставит ее со стрелкой $R_T a \to R_T b$, то есть со стрелкой $Ta \to Tb$ из $\mathcal{C}$. Мы принимаем эту стрелку, как составную, $\mu_b \circ Tg$:

$$Ta \xrightarrow{Tg} T(Tb) \xrightarrow{\mu_b} Tb$$

Чтобы установить сопряжение, покажем изоморфизм hom-множеств:

$$\mathcal{C}_T(L_T a, b) \cong C(a, R_T b)$$

Элементом левой стороны является стрелка Клейсли $a \twoheadrightarrow b$, которая представлена $f : a \to Tb$. Можно обнаружить эту же стрелку справа, поскольку $R_T b$ есть Tb. Таким образом, изоморфизм существует между стрелками Клейсли в $\mathcal{C}^T$ и стрелками в $\mathcal{C}$, которые их представляют.

Композиция $R_T \circ L_T$ равна T и, действительно, можно показать, что это сопряжение порождает исходную монаду.

В общем, может быть много сопряжений, порождающих одну и ту же монаду. Сами сопряжения образуют 2-категорию, поэтому можно сравнивать сопряжения, используя морфизмы сопряжений (1-клетки в 2-категории). Получается, что сопряжение Клейсли является инициальным объектом среди всех сопряжений, порождающих данную монаду. Двойственным образом, сопряжение Эйленберга-Мура является терминальным.

Глава 16

Ко-монады

Если бы это было легко произносимо, то побочные эффекты, вероятно, назвали бы «нтекстом», потому что двойственным побочным эффектом является «ко-нтекст».

Точно так же, как мы использовали стрелки Клейсли для работы с побочными эффектами, мы используем стрелки ко-Клейсли для работы с контекстами.

Начнем со знакомого примера внешней среды как контекста. Ранее мы построили из него считывающую монаду, каррируя стрелку:

```
(a, e) -> b
```

На этот раз, однако, мы будем рассматривать ее как стрелку со-Клейсли, которая является стрелкой от «контекстуализированного» аргумента.

Как и в случае с монадами, нас интересует возможность компоновать такие стрелки. Это относительно легко для средо-каррированных стрелок:

```
composeWithEnv    :: ((b, e) -> c) ->
                         ((a, e) -> b) ->
                           ((a, e) -> c)
composeWithEnv g f = \(a, e) -> g (f (a, e), e)
```

Также просто реализовать стрелку, которая служит тождественностью по отношению к этой композиции:

```
idWithEnv       :: (a, e) -> a
idWithEnv (a, e) = a
```

Это показывает, что существует категория, в которой стрелки ко-Клейсли служат морфизмами.

Пример 16.0.1. *Покажите, что композиция стрелок Клейсли, использующая* `composeWithEnv`, *является ассоциативной.*

16.1 Ко-монады в программировании

Функтор `w` (считайте его стилизованным перевернутым `m`) является ко-монадой, если он поддерживает композицию ко-Клейсли стрелок:

```
class Functor w => Comonad w where
  (=<=)   :: (w b -> c) -> (w a -> b) ->
                                  (w a -> c)
  extract :: w a -> a
```

Здесь, композиция записывается в виде инфиксного оператора; а единица композиции обозначена `extract`, поскольку она извлекает значение из контекста.

Попробуем это на уже рассмотренном примере. В качестве первого компонента пары удобно передавать окружение (т.е. среду). Ко-монада задается функтором, который является частичным применением конструктора пар `((,) e)`.

```
instance Comonad ((,) e) where
  g =<= f = \ea -> g (fst ea, f ea)
  extract = snd
```

Как и в случае с монадами, композицию ко-Клейсли можно использовать в бесточечном стиле программирования. Но мы также можем использовать двойное `join`, обозначаемое `duplicate`:

```
duplicate :: w a -> w (w a)
```

или двойную привязку, обозначаемую `extend`:

```
extend :: (w a -> b) -> w a -> w b
```

Вот как можно реализовать композицию ко-Клейсли в терминах `duplicate` и `fmap`:

```
g =<= f = g . fmap f . duplicate
```

Упражнение 16.1.1. *Реализуйте* `duplicate` *в терминах* `extend`*, и наоборот.*

Ко-монада Stream

Интересные примеры ко-монад имеют дело с большими, иногда бесконечными, контекстами. Вот бесконечный поток:

```
data Stream a = Cons a (Stream a)
  deriving Functor
```

Если мы рассматриваем такой поток как значение типа `a` в контексте бесконечного хвоста, то можем предоставить для него экземпляр `Comonad`:

```
instance Comonad Stream where
  extract (Cons a as)   = a
  duplicate (Cons a as) = Cons (Cons a as)
                               (duplicate as)
```

Здесь, `extract` возвращает начало потока, а `duplicate` превращает поток в поток потоков, в котором каждый последующий поток является хвостом предыдущего.

Интуиция такова, что `duplicate` закладывает основу для итерации, но делает это в очень общем виде. Голова каждого из подпотоков может быть интерпретирована как возможная «текущую позицию» в исходном потоке.

Было бы легко выполнить вычисление, которое проводится над головными элементами этих потоков. Но мощь ко-монады не в этом. Она позволяет выполнять вычисления, требующие произвольного упреждения. Для такого вычисления требуется доступ не только к головам последовательных подпотоков, но и к их хвостам.

А это то, что делает `extend`: она применяет заданную стрелку ко-Клейсли ко всем потокам, сгенерированным с помощью `duplicate`:

```
extend f (Cons a as) = Cons (f (Cons a as))
                            (extend f as)
```

Пример стрелки ко-Клейсли, которая усредняет первые пять элементов потока:

```
avg :: Stream Double -> Double
avg  = (/5). sum . stmTake 5
```

Она использует вспомогательную функцию, которая извлекает первые `n` элементов:

```
stmTake               :: Int -> Stream a -> [a]
stmTake 0 _            = []
stmTake n (Cons a as) = Double a : stmTake
                                     (n - 1) as
```

Мы можем запустить `avg` для всего потока, используя `extend`, чтобы сгладить локальные флуктуации. Инженеры-электрики могут распознать в этом простой фильтр нижних частот с `extend`, реализующим свертку. Это производит скользящее среднее исходного потока.

```
smooth :: Stream Double -> Stream Double
smooth  = extend avg
```

Ко-монады полезны для структурирования вычислений в пространственно или во времени расширенных структурах данных. Такие вычисления достаточно локальны, чтобы определить «текущее местоположение», но требуют сбора информации из соседних местоположений. Обработка сигналов или обработка изображений являются хорошими примерами. То же самое относится и к моделированию, в рамках которого дифференциальные уравнения должны решаться итеративно внутри объемов, например, в областях: моделирование климата, космологические модели или ядерные реакции. Игра «Жизнь» Конвея также является хорошим полигоном для ко-монадических методов.

Иногда удобно выполнять расчет на непрерывных потоках данных, откладывая выборку до самого последнего шага. Вот пример сигнала, который является функцией времени (представленный использованием `Double`)

```
data Signal a = Sig (Double -> a) Double
```

Первый компонент представляет собой непрерывный поток значений `a`, реализованный как функция времени. Второй компонент — это текущее время.

Это экземпляр `Comonad` для непрерывного потока:

```
instance Comonad Signal where
  extract (Sig f x)   = f x
  duplicate (Sig f x) = Sig (\y ->
                             Sig f (x - y)) x
  extend g (Sig f x)  = Sig (\y -> g
                             (Sig f (x - y))) x
```

Здесь, `extend` сворачивает фильтр

```
g :: Signal a -> a
```

по всему потоку.

Упражнение 16.1.2. *Реализуйте экземпляр* `Comonad` *для двунаправленного потока*:

```
data BiStream a = BStr [a] [a]
```

Предположим, что оба списка бесконечны. Подсказка: *считайте первый список прошлым (в обратном порядке), начало второго списка — настоящим, а его конец — будущим.*

Упражнение 16.1.3. *Реализуйте фильтр нижних частот для* `BiStream` *из предыдущего упражнения, который усредняет три значения: текущее, одно из ближайшего прошлого и одно из ближайшего будущего. Для инженеров-электриков: реализовать фильтр Гаусса.*

16.2 Ко-монады в категорном смысле

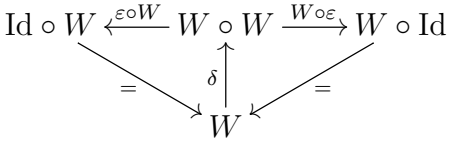
Можно получить определение ко-монады, обратив стрелки в определении монады. Наш `duplicate` соответствует обратному `join`, а `extract` — обратному `return`.

Таким образом, комонада — это эндофунктор W, снабженный двумя естественными преобразованиями:

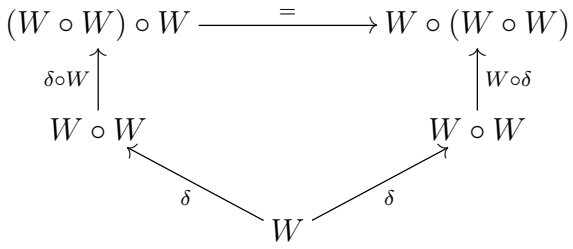
$$\delta : W \to W \circ W$$
$$\varepsilon : W \to Id$$

Эти преобразования (соответствующие `duplicate` и `extract`, соответственно) должны удовлетворять тем же тождествам, что и монада, за исключением перевернутых стрелок.

Это законы ко-единицы:



а это закон ассоциативности:



Ко-моноиды

Мы видели, как монадические законы следовали из моноидных законов. Мы можем ожидать, что законы ко-монады должны следовать из двойственной версии моноида.

Действительно, *ко-моноид* w — это объект в моноидальной категории $(\mathcal{C}, \otimes, I)$, снабженный двумя морфизмами, называемыми ко-умножением и ко-единицей:

$$\delta : w \to w \otimes w$$
$$\varepsilon : w \to I$$

Можно заменить тензорное произведение на эндофункторную композицию, а единичный объект — на тождественный функтор, чтобы получить определение ко-монады, как ко-моноида в категории эндофункторов.

На Haskell можно определить класс типов `Comonoid` для декартова произведения:

```
class Comonoid w where
  split   :: w -> (w, w)
  destroy :: w -> ()
```

О ко-моноидах говорят меньше, чем о моноидах, главным образом потому, что они воспринимаются как должное. В декартовой категории каждый объект можно превратить в ко-моноид: просто используя диагональное отображение $\Delta_a : a \to a \times a$ для коумножения и единственную стрелку к терминальному объекту для ко-единицы.

В программировании это то, что мы делаем, не задумываясь. Коумножение означает возможность дублировать значение, а ко-единица означает возможность отказаться от значения.

На Haskell можно легко реализовать экземпляр `Comonoid` для любого типа:

```
instance Comonoid  w where
  split w   = (w, w)
  destroy w = ()
```

На самом деле, мы не задумываемся дважды о том, чтобы использовать аргумент функции дважды или вообще не использовать его. Но, если мы хотим быть точными, функции вроде:

```
f x = x + x
g y = 42
```

можно записать как:

```
f x = let (x1, x2) = split x
      in x1 + x1
g y = let ()       = destroy y
      in 42
```

Однако случаются ситуации, когда дублирование или отбрасывание переменной нежелательно. Это тот случай, когда аргументом является внешний ресурс, такой как дескриптор файла, сетевой порт или участок памяти, выделенный в куче. Предполагается, что такие ресурсы имеют четко определенное время жизни между выделением и освобождением. Отслеживание времени жизни объектов, которые можно легко продублировать или отбросить, очень сложно и является печально известным источником ошибок программирования.

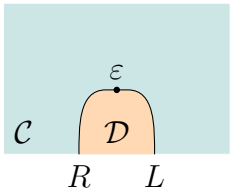
Модель программирования, основанная на декартовой категории, всегда будет вызывать эту проблему. Решение состоит в том, чтобы вместо

этого использовать моноидальную (замкнутую) категорию, которая не поддерживает дублирование или уничтожение объектов. Такая категория является естественной моделью для линейных типов. Элементы линейных типов используются в языке Rust и на момент написания этой статьи проходят испытания в Haskell. В C++ имеются конструкции, имитирующие линейность, такие как `unique_ptr` и семантика перемещения.

16.3 Ко-монады из сопряжений

Мы видели, что сопряжение $L \dashv R$ между двумя функторами $L : \mathcal{D} \to \mathcal{C}$ и $R : \mathcal{C} \to \mathcal{D}$ порождает монаду $R \circ L : \mathcal{D} \to \mathcal{D}$. Другая композиция, $L \circ R$, , являющаяся эндофунктором в $\mathcal{C}$, оказывается ко-монадой.

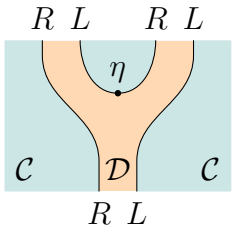
Ко-единица сопряжения служит ко-единицей ко-монады. Это может быть проиллюстрировано следующей струнной диаграммой:



Ко-умножение задается вискерингом η:

$$\delta = L \circ \eta \circ R$$

как показано на диаграмме:



По-прежнему, законы ко-монады могут быть выведены из тождеств треугольника.

Ко-монада ко-состояния

Мы видели, что монада состояния может быть сгенерирована из каррин-га сопряжения между произведением и экспоненциалом. Функтор слева был определен как произведение с некоторым фиксированным объектом s:

$$L_s a = a \times s$$

а функтор справа был возведением в степень, параметризованным тем же объектом s:

$$R_s c = c^s$$

Композиция $L_s \circ R_s$ порождает комонаду, называемую *ко-монадой ко-состояния* или *ко-монадой сохранения*.

В переводе на Haskell функтор справа назначает `c` функцию типа `s -> c`, а функтор слева образует пару `c` с `s`. Результатом композиции является эндофунктор:

```
data Store s c = St (s -> c) s
```

или, используя нотацию GADT:

```
data Store s c where
  St :: (s -> c) -> s -> Store s c
```

Экземпляр функтора пост-компонует эту функцию к первой компоненте `Store`:

```
instance Functor (Store s) where
  fmap g (St f s) = St (g . f) s
```

Ко-единица этого сопряжения, которая становится ко-монодической функцией `extract`, является применением функции:

```
extract          :: Store s c -> c
extract (St f s) = f s
```

Единицей этого сопряжения является естественное преобразование $\eta : Id \to R_s \circ L_s$. Мы использовали его в качестве `return` монады состояния. Его компонент при `c`:

```
unit  :: c -> (s -> (c, s))
unit c = \s -> (c, s)
```

Чтобы получить `duplicate`, нужно вискерингговать его между двумя функторами:

$$\delta = L_s \circ \eta \circ R_s$$

Вискеринг справа означает взятие компонента η при объекте $R_s c$, а вискеринг слева означает поднятие этого компонента с помощью L_s. Поскольку перевод вискеринга на Haskell — сложный процесс, проанализируем его более подробно.

Для простоты, зафиксируем тип `s`, скажем, на `Int`. Инкапсулируем левый функтор в новый тип:

```
newtype Pair c = P (c, Int)
  deriving Functor
```

и оставим правый функтор синонимом типа:

```
type Fun c = Int -> c
```

Единица сопряжения может быть записана как естественное преобразование с использованием явного `forall`:

```
eta  :: forall c. c -> Fun (Pair c)
eta c = \s -> P (c, s)
```

Теперь можно реализовать ко-умножение как вискеринг `eta`. Вискеринг справа закодирован в сигнатуре типа с помощью компоненты `eta` при `Fun c`. Вискеринг слева получен путем подъема `eta` с использованием `fmap`, определенного для функтора `Pair`. Мы используем языковую прагму `TypeApplications`, чтобы указать, какой `fmap` следует использовать:

```
delta :: forall c. Pair (Fun c) -> Pair
                           (Fun (Pair (Fun c)))
delta  = fmap @Pair eta
```

Это можно переписать более явно как:

```
delta (P (f, s)) = P (\s' -> P (f, s'), s)
```

Таким образом, экземпляр `Comonad` можно записать так:

```
instance Comonad (Store s) where
  extract (St f s)   = f s
  duplicate (St f s) = St (St f) s
```

Ко-монада сохранения — полезная концепция программирования. Чтобы понять ее, снова рассмотрим случай, когда `s` есть `Int`.

Мы интерпретируем первый компонент `Store Int c`, функцию `f :: Int -> c`, как метод доступа к воображаемому бесконечному потоку значений, по одному для каждого целого числа.

Второй компонент можно интерпретировать как текущий индекс. Действительно, `extract` использует этот индекс для извлечения текущего значения.

При такой интерпретации, `duplicate` создает бесконечный поток потоков, каждый из которых сдвинут на разное смещение, а `extend` выполняет свертку этого потока. Конечно, ленивость спасает положение: будут выбираться только те значения, которые мы явно затребуем.

Заметим также, что наш предыдущий пример с ко-монадой `Signal` воспроизводится посредством `Store Double`.

Упражнение 16.3.1. *Клеточный автомат можно реализовать с помощью ко-монады сохранения. Стрелка ко-Клейсли, описывающая правило 110*:

```
step          :: Store Int Cell -> Cell
step (St f n) =
  case (f (n-1), f n, f (n+1)) of
       (L, L, L) -> D
       (L, D, D) -> D
       (D, D, D) -> D
       _         -> L
```

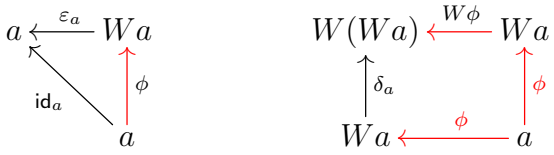
Клетка может быть либо живой, либо мертвой:

```
data Cell = L | D
  deriving Show
```

Проследите несколько поколений этого автомата. Подсказка: используйте функцию `iterate` *из Prelude.*

Ко-монадные коалгебры

Двойственно алгебрам монад, имеются коалгебры ко-монад. По ко-монаде (W, ε, δ), можно построить коалгебру, состоящую из объекта-носителя a и стрелки $\phi : a \to Wa$. Чтобы эта коалгебра хорошо компоновалась с ко-монадой, требуется, чтобы можно было бы извлечь значение, введенное с помощью ϕ, и чтобы поднятие ϕ было эквивалентно дублированию при воздействии на результат ϕ:



Как и в случае с монадными алгебрами, ко-монадные коалгебры образуют категорию. Для ко-монады (W, ε, δ) в $\mathcal{C}$, ее ко-монадные коалгебры образуют категорию, называемую категорией Эйленберга-Мура (иногда с префиксом ко-) $\mathcal{C}^W$.

Существует ко-Клейсли подкатегория $\mathcal{C}^W$, обозначаемая $\mathcal{C}_W$.

По ко-монаде W, можно построить сопряжение, используя либо $\mathcal{C}^W$, либо $\mathcal{C}_W$, воспроизводящие ко-монаду W. Эта конструкция полностью аналогична конструкции для монад.

Линзы

Особый интерес представляет коалгебра для ко-монады `Store`. Сначала произведем переименование. Обозначим носитель `s`, а состояние — `a`.

```
data Store a s = St (a -> s) a
```

Коалгебра задается функцией:

```
phi :: s -> Store a s
```

что эквивалентно паре функций:

```
set :: s -> a -> s
get :: s -> a
```

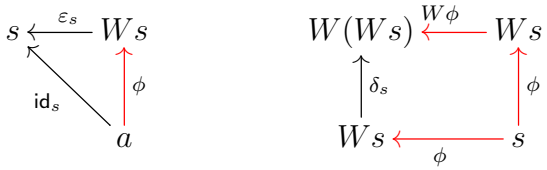
Такая пара называется линзой: `s` называется источником, а `a` — фокусом.

С этой интерпретацией, `get` позволяет извлечь фокус, а `set` заменяет фокус новым значением, для создания нового `s`.

Линзы были впервые введены для описания поиска и модификации данных в записях базы данных. Затем они нашли приложение в работе со структурами данных. Линза объективирует идею доступа для чтения/записи к части более крупного объекта. Например, линза может фокусироваться на одном из компонентов пары или на конкретном компоненте записи. Мы обсудим линзы, и оптику в целом, в следующей главе.

Применим законы коалгебры ко-монады к линзе. Для простоты опустим конструкторы данных; получим следующие упрощенные определения:

```
phi     s      = (set s, get s)
epsilon (f, a) = f a
delta   (f, a) = (\x -> (f, x), a)
```



Первый закон гласит, что применение результата `set` к результату `get` приводит к тождеству:

```
set s (get s)  = s
```

Это называется законом установки/получения линзы. Ничего не должно измениться при замене фокуса на тот же фокус.

Второй закон требует применения `fmap phi` к результату `phi`:

```
fmap phi (set s, get s) = (phi . set s, get s)
```

Это должно быть равно применению `delta`:

```
delta (set s, get s) = (\x -> (set s, x), get s)
```

Сравнивая их, получаем:

```
phi . set s = \x -> (set s, x)
```

Применим это к некоторому `a`:

```
phi (set s a) = (set s, a)
```

Использование определения `phi` дает:

```
(set (set s a), get (set s a)) = (set s, a)
```

Имеем два равенства. Первые компоненты — это функции, поэтому мы применяем их к некоторому `a'` и получаем закон set/set линз:

```
set (set s a) a' = set s a'
```

Установка фокуса на `a`, а затем перезапись его на `a'`, аналогична установке фокуса непосредственно на `a'`.

Вторые компоненты дают закон get/set:

```
get (set s a) = a
```

После того, как фокус установлен на `a`, результатом `get` будет `a`.

Линзы, удовлетворяющие этим законам, называются *законными линзами*. Они являются коалгебрами ко-монад для ко-монады сохранения.

Глава 17

Концы и ко-концы

17.1 Профункторы

В разреженной атмосфере теории категорий мы сталкиваемся с шаблонами, настолько далекими от своего происхождения, что их трудно визуализировать. Не помогает и то, что чем более абстрактным становится шаблон, тем более непохожими становятся его конкретные примеры.

Стрелку от a к b относительно легко визуализировать. Для этого имеется очень знакомая модель: функция, которая потребляет элементы a и производит элементы b. А $\hom$-множество представляет собой набор таких стрелок.

Функтор — это стрелка между категориями. Он потребляет объекты и стрелки из одной категории и производит объекты и стрелки из другой. Можно думать об этом как о рецепте создания таких объектов (и стрелок) из материалов, предоставленных исходной категорией. В частности, мы часто думаем об эндофункторе как о контейнере со строительными материалами.

Профунктор отображает пару объектов $\langle a, b\rangle$ к множеству $P\,\langle a, b\rangle$, а пару стрелок:

$$\langle f : s \to a, g : b \to t\rangle$$

к функции:

$$P\,\langle f, g\rangle : P\,\langle a, b\rangle \to P\,\langle s, t\rangle$$

Профунктор — это абстракция, сочетающая в себе элементы многих других абстракций. Поскольку это функтор $\mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$, то можно думать о нем как о построении множества из пары объектов и функции

от пары стрелок (одна из них идет в противоположном направлении). Однако это не помогает нашему воображению.

К счастью, имеется хорошая модель профунктора: функтор. Множество стрелок между двумя объектами ведет себя как профунктор, когда вы изменяете объекты. Также имеет смысл существование различия между изменением источника и цели hom-множества.

Следовательно, можно думать о произвольном профункторе как об обобщении этого hom-функтора. Профунктор обеспечивает дополнительные мостики между объектами поверх уже существующих hom-множеств.

Однако, существует одно большое различие между элементом hom-множества $\mathcal{C}(a, b)$ и элементом множества $P\,\langle a, b\rangle$. Элементами hom-множеств являются стрелки, а стрелки могут быть скомпонованы. Не совсем понятно, как компоновать профункторы.

Конечно, поднятие стрелок профунктором можно рассматривать как обобщающую композицию — только не между профункторами, а между hom-множествами и профункторами. Например, можно «пред-компоновать» $P\,\langle a, b\rangle$ со стрелкой $f : s \to a$ для получения $P\,\langle s, b\rangle$:

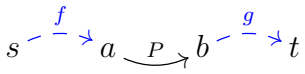
$$P\,\langle f, \mathsf{id}_b\rangle : P\,\langle a, b\rangle \to P\,\langle s, b\rangle$$

Точно так же можно «пост-компоновать» $P\,\langle a, b\rangle$ со стрелкой $g : b \to t$:

$$P\,\langle \mathsf{id}_a, g\rangle : P\,\langle a, b\rangle \to P\,\langle a, t\rangle$$

Такая неоднородная композиция принимает скомпонованную пару, состоящую из стрелки и элемента профунктора, и производит элемент профунктора.

Подобным образом, можно расширить профунктор с обеих сторон, подняв пару стрелок:



Коллажи

Нет причин ограничивать профунктор одной категорией. Можно легко определить профунктор между двумя категориями как функтор $P : \mathcal{C}^{\mathsf{op}} \times \mathcal{D} \to \mathbf{Set}$. Такой профунктор можно использовать для «склеивания» двух категорий, генерируя отсутствующие hom-множества от объектов в $\mathcal{C}$ к объектам в $\mathcal{D}$.

Коллаж (или ко-граф) двух категорий $\mathcal{C}$ и $\mathcal{D}$ — это категория, объекты которой являются объектами из обеих категорий (несвязное объединение). А hom-множество между двумя объектами x и y является: либо hom-множеством в $\mathcal{C}$, если оба объекта содержатся в $\mathcal{C}$; hom-множеством в $\mathcal{D}$, если оба находятся в $\mathcal{D}$; или множеством $P\langle x, y\rangle$, если x находится в $\mathcal{C}$, а y находится в $\mathcal{D}$. Иначе, hom-множество пусто.

Композиция морфизмов является обычной композицией, за исключением случаев, когда один из морфизмов является элементом of $P\langle x, y\rangle$. В этом случае мы поднимаем морфизм, который пытаемся пред- или пост-скомпоновать.

Понятно, что коллаж является категорией. Новые морфизмы, которые идут между двумя сторонами коллажа, иногда называют гетероморфизмами. Они могут идти только от $\mathcal{C}$ к $\mathcal{D}$, но не наоборот.

С этой точки зрения, профунктор $\mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$, на самом деле, следует называть эндо-профунктором. Он определяет коллаж $\mathcal{C}$ самого с собой.

Упражнение 17.1.1. *Покажите, что существует функтор от коллажа двух категорий к категории «шагающая стрелка» в виде контурограммы, имеющей два объекта, одну стрелку между ними и две тождественные стрелки.*

Упражнение 17.1.2. *Покажите, что если существует функтор от $\mathcal{C}$ к категории шагающей стрелки, то $\mathcal{C}$* можно расщепить на коллаж из двух категорий.

Профункторы как отношения

При детальном рассмотрении профунктор выглядит как hom-функтор, а элементы множества $P\langle a, b\rangle$ — как отдельные стрелки. Но когда мы отдаляем такое изображение, то можем рассматривать профунктор как отношение между объектами. Это не обычные отношения; они являются отношениями, *относящимися-к-доказательству*.

Чтобы лучше понять эту концепцию, рассмотрим обычный функтор $F : \mathcal{C} \to \mathbf{Set}$ (другими словами, ко-предпучок). Один из способов его интерпретации состоит в том, что он определяет подмножество объектов $\mathcal{C}$, а именно тех объектов, которые отображаются к непустым множествам. Затем, каждый элемент Fa рассматривается как доказательство того,

что a является членом этого подмножества. Если, с другой стороны, Fa — пустое множество, то a не входит в это подмножество.

Эту же интерпретацию можно применить к профункторам. Если множество $P\langle a, b\rangle$ пусто, говорят, что b не связан с a. Если оно не пусто, то говорят, что каждый элемент множества $P\langle a, b\rangle$ представляет собой доказательство того, что b связано с a. Тогда можно трактовать профунктор как отношение, относящееся-к-доказательству.

Заметим, что мы не делаем никаких предположений об этом отношении. Оно не обязательно должно быть рефлексивным, так как $P\langle a, a\rangle$ может быть пустым (на самом деле $P\langle a, a\rangle$ имеет смысл только для эндо-профункторов). Оно также не обязано быть симметричным.

Поскольку $\hom$-функтор является примером (эндо-) профунктора, такая интерпретация позволяет рассматривать $\hom$-функтор в новом свете: как встроенное, относящееся-к-доказательству, отношение между объектами в категории. Если между двумя объектами имеется стрелка, они являются связанными. Обратите внимание, что это отношение рефлексивно, поскольку $\mathcal{C}(a, a)$ никогда не бывает пустым: по крайней мере, оно содержит тождественный морфизм.

Более того, как мы видели ранее, $\hom$-функторы взаимодействуют с профункторами. Если a связано с b через P, а $\hom$-множества $\mathcal{C}(s, a)$ и $\mathcal{D}(b, t)$ непусты, то s автоматически связано с t через P. Таким образом, профункторы являются относящимися-к-доказательству отношениями, совместимыми со структурой категорий, в которых они действуют.

Мы знаем, как скомпоновать профунктор с $\hom$-функторами, но как скомпоновать два профунктора? Ключ к разгадке кроется в композиции отношений.

Предположим, требуется зарядить мобильный телефон, но зарядное устройство отсутствует. Для того, чтобы подключить ваш телефон к зарядному устройству, достаточно, чтобы у вас был друг, у которого есть зарядное устройство. Подойдет любой друг. Вы совмещаете отношение наличия друга с отношением человека, имеющего зарядное устройство, чтобы получить отношение возможности зарядить свой телефон. Доказательством того, что можно зарядить свой телефон, является пара доказательств: одно — доказательство дружбы и другое — наличие зарядного устройства.

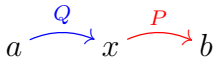
В общем, говорят, что два объекта связаны составным отношением, если между ними существует объект, связанный с обоими.

Профункторная композиция на Haskell

Композицию отношений можно преобразовать в профункторную композицию на Haskell. Сначала напомним определение профунктора:

```
class Profunctor p where
  dimap :: (s -> a) -> (b -> t) -> (p a b ->
                                    p s t)
```

Ключом к пониманию профункторной композиции является то, что она требует *существования* еще одного объекта между компонуемыми объектами. Для того, чтобы объект b был связан с объектом a через компоновку $P \diamond Q$, должен существовать объект x, который перекрывает промежуток между ними:



Это можно закодировать на Haskell, используя экзистенциальный тип. Для двух `p` и `q`, их композиция представляет собой новый профунктор `Procompose p q`:

```
data Procompose p q a b where
  Procompose :: q a x -> p x b -> Procompose
                                    p q a b
```

Мы используем `GADT`, чтобы выразить экзистенциальную природу объекта `x`. Два аргумента конструктора данных можно рассматривать как пару доказательств: одно доказывает, что `x` связано с `a`, а другое — что `b` связано с `x`. Эта пара представляет собой доказательство того, что `b` связано с `a`.

Экзистенциальный тип можно рассматривать как обобщение тип-суммы. Суммирование производится по всем возможным типам `x`. Точно так же, как конечная сумма может быть построена путем использования одной из альтернатив (напомним про два конструктора `Either`), экзистенциальный тип может быть построен путем выбора одного конкретного типа для `x` и внедрения его в определение для `Procompose`.

Так же, как для отображения из тип-суммы требуется пара функций, по одной на каждую альтернативу; отображение-вне экзистенциального типа требует семейства функций, по одной на каждый тип. Отображение-вне от `Procompose`, например, задается полиморфной функцией:

```
mapOut            :: Procompose p q a b ->
                       (forall x. q a x ->
                            p x b -> c) -> c
mapOut (Procompose qax pxb) f
                   = (f qax pxb)
```

Композиция профункторов снова является профунктором, как видно из этого примера:

```
instance (Profunctor p, Profunctor q)  =>
                    Profunctor (Procompose p q)
  where dimap l r (Procompose qax pxb) =
                    Procompose (dimap l id qax)
                               (dimap id r pxb)
```

Это просто говорит о том, что можно расширить составной профунктор, расширением первого профунктора слева, а второго — справа.

То, что это определение профункторной композиции работает в Haskell, объясняется параметричностью. Язык программирования ограничивает типы профункторов таким образом, чтобы это работало. Однако, в общем случае, простое суммирование промежуточных объектов привело бы к пересчету, поэтому в теории категорий это необходимо компенсировать.

17.2 Ко-концы

Пересмотр в наивном определении профункторной композиции происходит, когда два кандидата на объект в середине связаны морфизмом:

$$a \overset{Q}{\curvearrowright} x - \overset{f}{-} \to y \overset{P}{\curvearrowright} b$$

Можно, либо расширить Q справа, подняв $Q\,\langle \mathsf{id}, f\rangle$, и использовать y в качестве серединного объекта; или можно расширить P слева, подняв $P\,\langle f, \mathsf{id}\rangle$, и используя x в качестве посредника.

Чтобы избежать двойного подсчете, необходимо изменить наше определение тип-суммы применительно к профункторам. Полученная конструкция будет называться ко-концом.

Во-первых, переформулируем задачу. Мы пытаемся произвести суммирование по всем объектам x в произведении:

$$P\,\langle a, x\rangle \times Q\,\langle x, b\rangle$$

Двойной подсчет происходит потому, что можно открыть промежуток между двумя профункторами, пока существует морфизм, который можно вписать между ними. Итак, мы действительно рассматриваем более общее произведение:

$$P\,\langle a, x\rangle \times Q\,\langle y, b\rangle$$

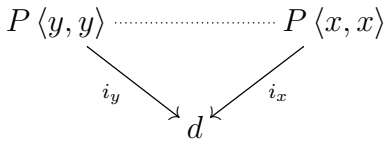
Важное наблюдение состоит в том, что если зафиксировать граничные точки, a и b, то это произведение будет профунктором в $\langle y, x\rangle$. В этом легко убедиться после небольшой перестановки (с точностью до изоморфизма):

$$Q\,\langle y, b\rangle \times P\,\langle a, x\rangle$$

Нас интересует сумма диагональных частей этого профунктора, то есть когда x равен y.

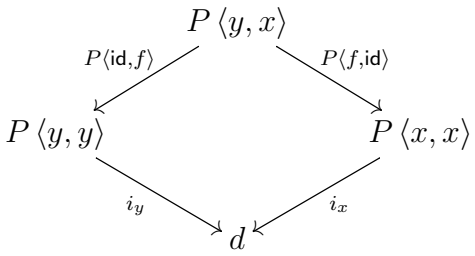
Итак, давайте посмотрим, как мы будем определять сумму всех диагональных элементов общего профунктора P. Фактически, эта конструкция работает для любого функтора $P : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$, а не только для **Set**-значных профункторов.

Сумма диагональных объектов определяется инъекциями, в данном случае по одному на каждый объект в $\mathcal{C}$. Здесь показаны только два из них, а пунктирная линия представляет все остальные:



Если бы мы определяли сумму, то сделали бы ее универсальным объектом, оснащенным такими инъекциями. Но поскольку мы имеем дело с функторами двух переменных, то хотим идентифицировать инъекции, которые связаны «расширением» некоторого общего предка (здесь, $P\,\langle y, x\rangle$). Мы хотим, чтобы следующая диаграмма была коммутативной

всякий раз, когда существует соединительный морфизм $f : x \to y$:



Эта диаграмма называется *ко-клином*, а ее коммутирующее условие — условием ко-клина. Для каждого $f : x \to y$, мы требуем, чтобы выполнялось:

$$i_x \circ P\,\langle f, \mathsf{id}_y\rangle = i_y \circ P\,\langle \mathsf{id}_x, f\rangle$$

Универсальный ко-клин называется *ко-концом*.

Поскольку ко-конец обобщает сумму на потенциально бесконечную область, ее записывают с использованием знака интеграла и верхней «переменной интегрирования»:

$$\int^{x:\mathcal{C}} P\,\langle x, x\rangle$$

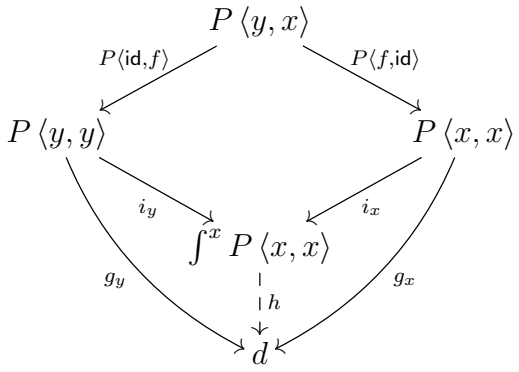
Универсальность означает, что всякий раз, когда существует объект d в $\mathcal{D}$, снабженный семейством стрелок $g_x : P\,\langle x, x\rangle \to d$, удовлетворяющих условию ко-клина, существует единственное отображение-вне от ко-конца:

$$h : \int^{x:\mathcal{C}} P\,\langle x, x\rangle \to d$$

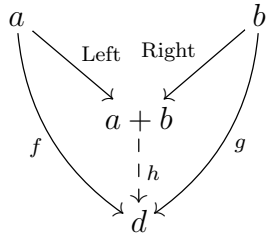
которое факторизует каждый g_x через инъекцию i_x:

$$g_x = h \circ i_x$$

Диаграммно, имеем:



Сравните это с определением суммы двух объектов:



Так же, как сумма была определена как универсальный ко-пролет, ко-конец определяется как универсальный ко-клин.

В частности, если надо построить ко-конец **Set**-значного профунктора, то надо начать с суммы (размеченного объединения) всех множеств $P\langle x,x\rangle$. Затем надо определить все элементы этой суммы, которые удовлетворяют условию ко-клина. Далее требуется отождествить элемент $a \in P\langle x,x\rangle$ с элементом $b \in P\langle y,y\rangle$ всякий раз, когда имеется элемент $c \in P\langle y,x\rangle$ и морфизм $f : x \to y$, такие что:

$$P\langle \mathsf{id}, f\rangle\,(c) = b$$

и

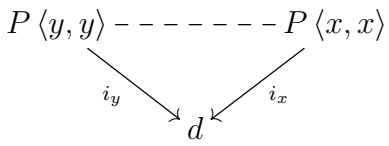
$$P\langle f, \mathsf{id}\rangle\,(c) = a$$

Заметим, что в дискретной категории (которая представляет собой просто множество объектов без стрелок между ними) условие ко-клина тривиально (нет других f, кроме тождественностей), поэтому ко-конец — это просто прямая сумма (копроизведение) диагональных объектов $P\langle x,x\rangle$.

Сверх-естественные преобразования

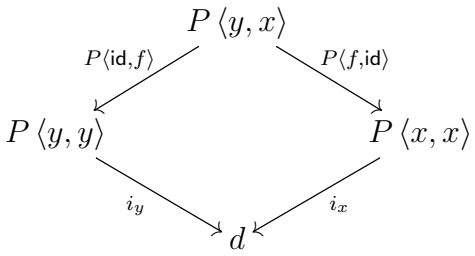
Семейство стрелок в целевой категории, параметризованное объектами исходной категории, часто может быть объединено в одно естественное преобразование между двумя функторами.

Инъекции в определении ко-клина образуют семейство функций, параметризованных объектами, но они не вписываются в определение естественного преобразования.



Проблема в том, что функтор $P : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$ контравариантен по первому аргументу и ковариантен по второму; поэтому его диагональная часть, которая на объектах определяется как $x \mapsto P\langle x,x\rangle$, не является ни тем, ни другим.

Ближайшим аналогом естественности, имеющимся в нашем распоряжении, является условие ко-клина:



В самом деле, как и в случае с квадратом естественности, оно предполагает взаимодействие между поднятием морфизма $f : x \to y$ (здесь, двумя разными способами) и компонентами преобразования i.

Конечно, стандартное условие естественности имеет дело с парами функторов. Здесь целью преобразования является фиксированный объект d. Но мы всегда можем переинтерпретировать его как вывод постоянного функтора $\Delta_d : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{D}$.

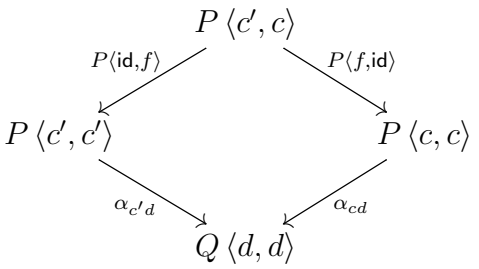
Условие ко-клина можно интерпретировать как частный случай более общего сверх-естественного преобразования, которое представляет собой семейство стрелок:

$$\alpha_{cd} : P\langle c,c\rangle \to Q\langle d,d\rangle$$

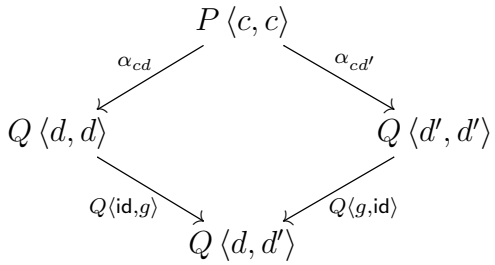
между двумя функторами в виде:

$$P : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{E}$$
$$Q : \mathcal{D}^{\mathsf{op}} \times \mathcal{D} \to \mathcal{E}$$

Сверх-естественность в c означает, что следующая диаграмма коммутативна для любого морфизма $f : c \to c'$:



Сверх-естественность в d означает, что следующая диаграмма коммутативна для любого морфизма $g : d \to d'$:



С учетом этого определения, мы получаем условие ко-клина как сверх-естественности преобразования между профунктором P и постоянным профунктором Δ_d.

Теперь можно переформулировать определение ко-конца как пару (e, i), где e — это объект, снабженный сверх-естественным преобразованием $i : P \to \Delta_e$, которое является универсальным среди таких пар.

Универсальность означает, что для любого объекта d, оснащенного сверх-естественным преобразованием $\alpha : P \to \Delta_d$, существует единственный морфизм $h : e \to d$, который факторизует все компоненты α через компоненты i:

$$\alpha_x = h \circ i_x$$

Этот объект e называется ко-концом и записывается как:

$$e = \int^{x} P\,\langle x, x\rangle$$

Профункторная композиция, использующая ко-концы

Имея определение ко-конца, можно формально определить композицию двух профункторов:

$$(P \diamond Q)\,\langle a, b\rangle = \int^{x:\mathcal{C}} Q\,\langle a, x\rangle \times P\,\langle x, b\rangle$$

Сравните это с:

```
data Procompose p q a b where
  Procompose :: q a x -> p x b -> Procompose
                                    p q a b
```

Причина, по которой в Haskell не нужно беспокоиться об условии ко-клина, аналогична причине, по которой все параметрически полиморфные функции автоматически удовлетворяют условию естественности. Ко-конец определяется с помощью семейства инъекций; в Haskell все эти инъекции определяются одной полиморфной функцией:

```
data Coend p where
  Coend :: p x x -> Coend p
```

Ко-концы вводят новый уровень абстракции в работе с профункторами. Вычисления с использованием ко-концов обычно используют их свойство отображения-вне. Чтобы определить отображение-вне ко-конца к некоторому объекту d:

$$\int^{x} P\,\langle x, x\rangle \to d$$

достаточно определить семейство функций от диагональных элементов функтора к d:

$$g_x : P\,\langle x, x\rangle \to d$$

удовлетворяющих условию ко-клина. Из этого трюка можно извлечь большую пользу, особенно в сочетании с леммой Йонеды. Далее мы увидим примеры этого.

Упражнение 17.2.1. *Определите экземпляр* `Profunctor` *для пары профункторов*:

```
newtype ProPair q p a b x y = ProPair
                                (q a y, p x b)
```

Подсказка: *зафиксируйте первые четыре параметра*:

```
instance (Profunctor p, Profunctor q) =>
                  Profunctor (ProPair q p a b)
```

Упражнение 17.2.2. *Профункторная композиция может быть выражена с помощью ко-конца*:

```
newtype CoEndCompose p q a b = CoEndCompose
                      (Coend (ProPair q p a b))
```

Определите экземпляр `Profunctor` *для* `CoEndCompose`.

Копределы как ко-концы

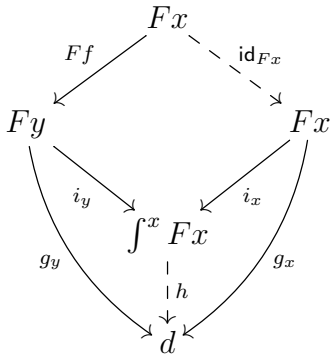
Функция двух переменных, которая игнорирует один из своих аргументов, эквивалентна функции одной переменной. Точно так же профунктор, который игнорирует один из своих аргументов, эквивалентен функтору. И наоборот, по заданному функтору F можно построить профунктор:

$$P\,\langle x, y\rangle = Fy$$

Точно так же, его действие на пару стрелок игнорирует одну из стрелок:

$$P\,\langle f, g\rangle = Fg$$

Для любой $f : x \to y$, наше определение ко-конца для такого профунктора сводится к следующей диаграмме:



После сокращения тождественных стрелок исходный ко-клин становится ко-конусом, а универсальное условие превращается в определение ко-предела. Это оправдывает использование обозначения ко-конца для ко-пределов:

$$\int^x Fx = \operatorname{colim} F$$

Функтор F определяет диаграмму в целевой категории. Исходной категорией, в этом случае, является приведенный шаблон.

Можно получить полезную интуицию, если рассмотреть дискретную категорию, в которой профунктор — это (возможно, бесконечная) матрица, а ко-конец — это сумма (копроизведение) ее диагональных элементов. Постоянный вдоль одной оси профунктор соответствует матрице, строки которой одинаковы (каждая из которых задана «вектором» Fx). Сумма диагональных элементов такой матрицы равна сумме всех компонент вектора Fx.

$$\begin{pmatrix} Fa & Fb & Fc & \dots \\ Fa & Fb & Fc & \dots \\ Fa & Fb & Fc & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

В недискретной категории эта сумма обобщается до ко-предела.

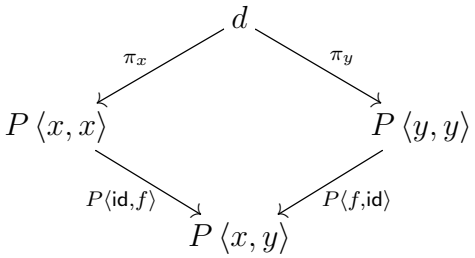
17.3 Концы

Как ко-конец обобщает сумму диагональных элементов профунктора, так и его двойственный, *конец*, обобщает произведение. Произведение определяется своими проекциями, тем же свойством обладает и конец.

Обобщение пролета, который был использован в определении произведения, представляет собой объект d с семейством проекций, по одной на каждый объект x:

$$\pi_x : d \to P\,\langle x, x\rangle$$

Двойственным ко-клину является клин:



Для каждой стрелки $f : x \to y$ требуется, чтобы:

$$P\left\langle f, \mathsf{id}_y \right\rangle \circ \pi_y = P\left\langle \mathsf{id}_x, f \right\rangle \circ \pi_x$$

Конец является универсальным клином. Для него также используется знак интеграла, здесь, с «переменной интегрирования» внизу.

$$\int_{x:\mathcal{C}} P\left\langle x, x \right\rangle$$

Известно, что классические интегралы, основанные на умножении, а не на сложении, редко используются в исчислении. Это связано с тем, что используя логарифмирование, можно заменить умножение сложением. В теории категорий нет подобного механизма, поэтому концы и ко-концы одинаково важны.

Подводя итог, конец — это объект, оснащенный семейством морфизмов (проекций):

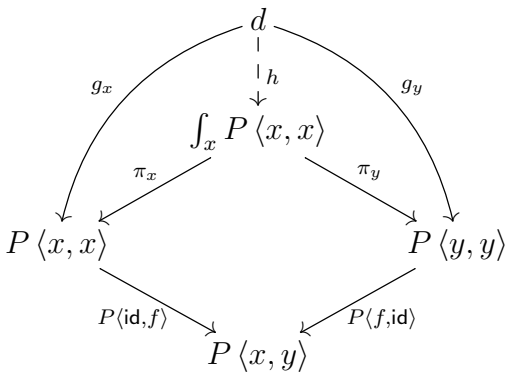
$$\pi_a : \left(\int_x P\left\langle x, x \right\rangle \right) \to P\left\langle a, a \right\rangle$$

удовлетворяющих условию клина.

Он универсален среди подобных объектов; то есть для любого другого объекта d, снабженного семейством стрелок g_x, удовлетворяющих условию клина, существует единственный морфизм h, который факторизует семейство g_x через семейство π_x:

$$g_x = \pi_x \circ h$$

Графически, имеет место:



Эквивалентно, можно сказать, что конец — это пара (e, π), состоящая из объекта e и сверх-естественного преобразования $\pi : \Delta_d \to e$, универсального среди таких пар. Условие клина оказывается частным случаем условия сверх-естественности.

Если бы нужно было сконструировать конец **Set**-значного профунктора, то начинать надо было бы с произведения всех $P\langle x, x\rangle$, для всех объектов в категории, а затем отсечь кортежи, которые не удовлетворяют условию клина.

В частности, представьте, что, вместо d, используется одноэлементное множество 1. Семейство g_x будет выбирать по одному элементу из каждого множества $P\langle x, x\rangle$. В результате получится кортеж огромного размера. Вы бы отсеяли большинство этих кортежей, оставив только те, которые удовлетворяют условию клина.

Опять же, в Haskell, из-за параметричности, условие клина выполняется автоматически, и определение конца для профунктора `p` упрощается до:

```
type End p = forall x. p x x
```

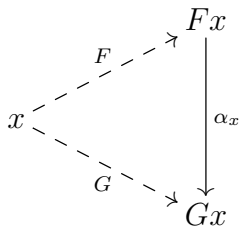
Реализация `End` на Haskell никак не связана с тем фактом, что он двойственен `Coend`, а связано с тем, что, на момент написания этого раздела, Haskell не имел встроенного синтаксиса для экзистенциальных типов. Но, если бы это было так, `Coend` был бы реализован следующим образом:

```
type Coend p = exists x. p x x
```

Экзистенциальная/универсальная двойственность между `Coend` и `End` означает, что сконструировать `Coend` легко — все, что нужно для этого, это выбрать один тип `x`, для которого имеется значение типа `p x x`. С другой стороны, чтобы построить `End`, необходимо предоставить целое семейство значений `p x x`, по одному для каждого типа `x`. Другими словами, нужна полиморфная формула, параметризованная `x`. Определение полиморфной функции является каноническим примером такой формулы.

Естественные преобразования как концы

Самое интересное применение конца — это краткое определение естественных преобразований. Рассмотрим два функтора, F и G, между двумя категориями, $\mathcal{B}$ и $\mathcal{C}$. Естественным преобразованием между ними является семейство стрелок α_x в $\mathcal{C}$. Можно представлять это как выбор одного элемента α_x из каждого hom-множества $\mathcal{C}(F_x, G_x)$.



Мы знаем, что отображение $\langle a, b\rangle \to \mathcal{C}(a, b)$ определяет профунктор. Оказывается, для любой пары функторов, отображение $\langle a, b\rangle \to \mathcal{C}(Fa, Gb)$ также ведет себя как профунктор. Его действие на пару стрелок $\langle f, g\rangle$ представляет собой комбинацию пред- и пост-композиции поднятых стрелок:

$$(Gg) \circ - \circ (Ff)$$

Действительно, элемент множества $\mathcal{C}(Fa, Gb)$ — это стрелка $h : Fa \to Gb$. Мы пытаемся поднять пару стрелок $f : s \to a$ и $g : b \to t$. Это можно сделать парой стрелок в $\mathcal{C}$: первая — $Ff : Fs \to Fa$, а вторая — $Gg : Gb \to Gt$. Композиция $Gg \circ h \circ Ff$ дает желаемый результат $Fs \to Gt$, который является элементом $\mathcal{C}(Fs, Gt)$.

$$Fs \xrightarrow{Ff} Fa \xrightarrow{h} Gb \xrightarrow{Gg} Gt$$

Диагональные части этого профунктора являются хорошими кандидатами на компоненты естественного преобразования. Собственно, конец:

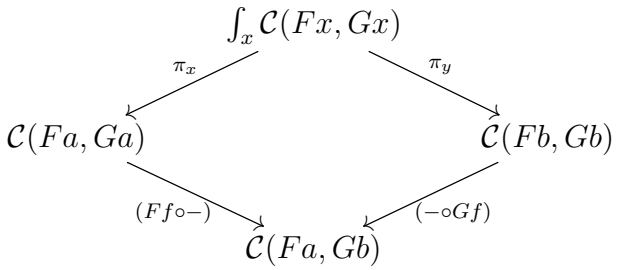
$$\int_{x:\mathcal{B}} \mathcal{C}(Fx, Gx)$$

определяет множество естественных преобразований от F к G.

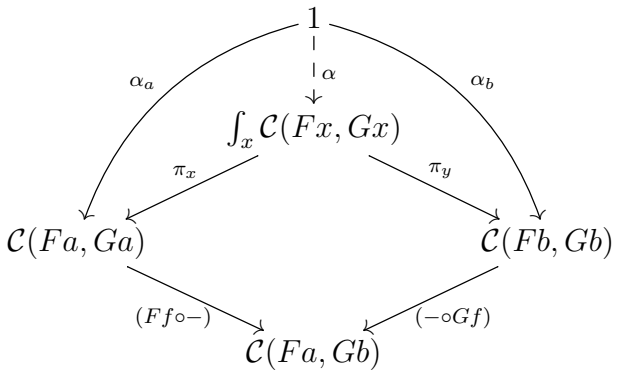
На Haskell это согласуется с нашим предыдущим определением:

```
type Natural f g = forall x. f x -> g x
```

Однако, в теории категорий необходимо проверить условие клина. Подключив наш профунктор, получим:



Можно сосредоточиться на одном элементе множества $\int_x \mathcal{C}(Fx, Gx)$, задав универсальное условие для одноэлементного множества:



Он выбирает компонент α_a из hom-множества $\mathcal{C}(Fa, Ga)$ и компонент α_b из $\mathcal{C}(Fb, Gb)$. Тогда условие клина сводится к следующему:

$$Ff \circ \alpha_a = \alpha_b \circ Gf$$

для любой $f : a \to b$. Это как раз и есть условие естественности. Таким образом, элемент α этого конца действительно является естественным преобразованием.

Таким образом, множество естественных преобразований или hom-множество в категории функторов задается концом:

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_{x:\mathcal{B}} \mathcal{C}(Fx, Gx)$$

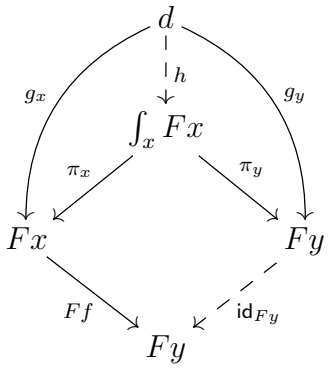
Как уже обсуждалось ранее, для создания `End` необходимо предоставить ему целое семейство значений, параметризованных типами. Здесь, эти значения являются компонентами полиморфной функции.

Пределы как концы

Точно так же, как копределы были выражены выше как ко-концы, можно выразить пределы как концы. Как и ранее, определим профунктор, который игнорирует свой первый аргумент:

$$P\,\langle x, y\rangle = Fy$$
$$P\,\langle f, g\rangle = Fg$$

Универсальное условие, определяющее конец, становится определением универсального конуса:



Таким образом, для пределов можно использовать обозначение конца:

$$\int_x Fx = \lim F$$

17.4 Непрерывность hom-функтора

В теории категорий функтор называется непрерывным, если он сохраняет пределы (и ко-непрерывным, если он сохраняет ко-пределы). Это означает, что если имеется диаграмма в исходной категории, то не имеет значения, используется ли сначала функтор для отображения диаграммы, а затем берете предел; или берется предел в исходной категории и используется функтор для отображения этого предела.

$\hom$-функтор является примером функтора, непрерывного по второму аргументу. Поскольку произведение является простейшим примером предела, это означает, в частности, что:

$$\mathcal{C}(x, a \times b) \cong \mathcal{C}(x, a) \times \mathcal{C}(x, b)$$

В левой части, $\hom$-функтор применяется к произведению (предел пролёта). Правая часть отображает диаграмму, здесь, просто пара объектов, и производит произведение (предел) в целевой категории. Целевой категорией для $\hom$-функтора является $\mathbf{Set}$, так что это просто декартово произведение. Две стороны изоморфны по универсальному свойству произведения: отображение в произведение определяется парой отображений в два объекта.

Непрерывность $\hom$-функтора в первом аргументе обратная: он отображает копределы в пределы. Опять же, простейшим примером копредела является сумма, поэтому имеем:

$$\mathcal{C}(a + b, x) \cong \mathcal{C}(a, x) \times \mathcal{C}(b, x)$$

Это следует из универсальности суммы: отображение-вне суммы определяется парой отображений-вне двух объектов. Можно показать, что конец может быть выражен как предел, а ко-конец как ко-предел. Следовательно, в силу непрерывности $\hom$-функтора, всегда можно вынести знак интеграла из $\hom$-множества. По аналогии с произведением, имеем формулу отображения-внутрь для конца:

$$\mathcal{D}\left(d, \int_a P\left\langle a, a\right\rangle\right) \cong \int_a \mathcal{D}\left(d, P\left\langle a, a\right\rangle\right)$$

По аналогии с суммой имеем формулу отображения-вне для ко-конца:

$$\mathcal{D}\left(\int^a P\left\langle a, a\right\rangle, d\right) \cong \int_a \mathcal{D}\left(P\left\langle a, a\right\rangle, d\right)$$

Заметим, что в обоих случаях правая часть является концом.

17.5 Правило Фубини

Правило Фубини в исчислении устанавливает условия, при которых можно изменить порядок интегрирования в двойных интегралах. Оказывается, аналогичным образом можно поменять местами двойные концы и ко-концы. Правило Фубини для концов работает для функторов вида $P : \mathcal{C} \times \mathcal{C}^{\mathsf{op}} \times \mathcal{D} \times \mathcal{D}^{\mathsf{op}} \to \mathcal{E}$. Следующие выражения, если они существуют, изоморфны:

$$\int_{c:\mathcal{C}} \int_{d:\mathcal{D}} P\,\langle c,c\rangle\,\langle d,d\rangle \cong \int_{d:\mathcal{D}} \int_{c:\mathcal{C}} P\,\langle c,c\rangle\,\langle d,d\rangle \cong \int_{\langle c,d\rangle:\mathcal{C}\times\mathcal{D}} P\,\langle c,c\rangle\,\langle d,d\rangle$$

В последнем конце, функтор P переинтерпретируется как $P : (\mathcal{C} \times \mathcal{D})^{\mathsf{op}} \times (\mathcal{C} \times \mathcal{D}) \to \mathcal{E}$.

Аналогичное правило работает и для ко-концов.

17.6 Лемма ниндзя Йонеды

Выражая множество естественных преобразований как конец, можем перезаписать лемму Йонеды. Оригинальная формулировка выглядит следующим образом:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(a,-), F) \cong Fa$$

Здесь, F — (ковариантный) функтор от $\mathcal{C}$ к $\mathbf{Set}$ (ко-предпучок), как и $\hom$-functor $\mathcal{C}(a,-)$. Выражая множество естественных преобразований в виде конца, получим:

$$\int_{x:\mathcal{C}} \mathbf{Set}(\mathcal{C}(a,x), Fx) \cong Fa$$

Аналогично, имеет место лемма Йонеды для контравариантного функтора (предпучка) G:

$$\int_{x:\mathcal{C}} \mathbf{Set}(\mathcal{C}(x,a), Gx) \cong Ga$$

Эти версии леммы Йонеды, выраженные в терминах концов, часто, полушутя, называют леммами ниндзя-Йонеды. Тот факт, что «переменная интегрирования» является явной, несколько упрощает их использование в сложных формулах.

Существует также двойственное множество лемм ниндзя-ко-Йонеды, в которых используются ко-концы. Для ковариантного функтора имеем:

$$\int^{x:\mathcal{C}} \mathcal{C}(x,a) \times Fx \cong Fa$$

а для контравариантного функтора:

$$\int^{x:\mathcal{C}} \mathcal{C}(a,x) \times Gx \cong Ga$$

Физики могут заметить сходство этих формул с интегралами, включающими дельта-функцию Дирака (фактически, распределение). Вот почему профункторы иногда называют распределителями, следуя изречению, что «распределители относятся к функторам так же, как распределения относятся к функциям». Инженеры могут заметить сходство $\hom$-функтора с импульсной функцией.

Эта интуиция часто выражается в том, что можно выполнить «интегрирование по x» в этой формуле, что приводит к замене x на a в подынтегральном выражении Gx.

Если $\mathcal{C}$ — дискретная категория, ко-конец сводится к сумме (произведению), а $\hom$-функтор сводится к единичной матрице (дельта Кронекера). Лемма ко-Йонеды принимает вид:

$$\sum_j \delta_i^j v_j = v_i$$

Фактически, многое из линейной алгебры напрямую переводится в теорию **Set**-значных функторов. Часто можно рассматривать такие функторы как векторы в векторном пространстве, в котором $\hom$-функторы образуют базис. Профункторы становятся матрицами, а ко-концы могут использоваться для умножения таких матриц, вычисления их следов или умножения векторов на матрицы.

Еще одно название профункторов, особенно используемое в Австралии, — «бимодули». Это связано с тем, что поднятие морфизмов профунктором чем-то похоже на левые и правые действия на множествах.

Доказательство леммы Ко-Йонеды весьма поучительно, так как использует несколько общих приемов. Самое главное, мы полагаемся на следствие из леммы Йонеды, которое выражает то, что если все отображения-вне от объектов к произвольному объекту изоморфны, то и сами эти два объекта изоморфны. Поэтому начнем с такого отображения-вне к произвольному множеству S:

$$\mathbf{Set}\left(\int^{x:\mathcal{C}} \mathcal{C}(x,a) \times Fx, S\right)$$

Используя ко-непрерывность hom-функтора, можно вывести знак интеграла на внешний уровень, заменяя ко-конец на конец:

$$\int_{x:\mathcal{C}} \mathbf{Set}(\mathcal{C}(x,a) \times Fx, S)$$

Поскольку категория множеств является декартово замкнутой, можно каррировать произведение:

$$\int_{x:\mathcal{C}} \mathbf{Set}(\mathcal{C}(x,a), S^{Fx})$$

Теперь можно использовать лемму Йонеды для «интегрирования по x». Результатом является S^{Fa}. Наконец, в **Set**, экспоненциальный объект изоморфен hom-множеству:

$$S^{Fa} \cong \mathbf{Set}(Fa, S)$$

Поскольку S было произвольным, можно заключить, что:

$$\int^{x:\mathcal{C}} \mathcal{C}(x,a) \times Fx \cong Fa$$

Упражнение 17.6.1. *Докажите контравариантную версию леммы ко-Йонеды.*

Лемма Йонеды на Haskell

Мы уже приводили лемму Йонеды, реализованную на Haskell. Теперь можно переписать ее в терминах конца. Начнем с определения профунктора, который будет идти под концом. Его конструктор типов принимает функтор `f`, тип `a` и генерирует профунктор, контравариантный по `x` и ковариантный по `y`:

```
data Yo f a x y = Yo ((a -> x) -> f y)
```

Лемма Йонеды устанавливает изоморфизм между концом над этим профунктором и типом, полученным действием функтора `f` на `a`. Об этом изоморфизме свидетельствует пара функций:

```
yoneda       :: Functor f => End (Yo f a) -> f a
yoneda (Yo g) = g id

yoneda_1     :: Functor f => f a -> End (Yo f a)
yoneda_1 fa   = Yo (\h -> fmap h fa)
```

Точно так же, лемма ко-Йонеды использует ко-конец над следующим профунктором:

```
data CoY f a x y = CoY (x -> a) (f y)
```

Изоморфизм подтверждается парой функций. Первая означает, что если имеется функция `x -> a` и функтор `x`, то можно заполнить этот функтор значениями `a` с помощью `fmap`:

```
coyoneda       :: Functor f => Coend (CoY f a)
                                    -> f a
coyoneda (Coend (CoY g fa))
                = fmap g fa
```

Это можно сделать, ничего не зная об экзистенциальном типе `x`.

Вторая функция означает, что если имеется функтор, полный `a`, то можно создать ко-конец, введя его (вместе с тождественной функцией) в экзистенциальный тип:

```
coyoneda_1     :: Functor f => f a -> Coend
                                      (CoY f a)
coyoneda_1 fa   = Coend (CoY id fa)
```

17.7 D-свертка

Инженеры-электрики знакомы с идеей свертки. Можно свернуть два потока, сдвинув один из них и просуммировав его произведение с другим:

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy$$

Эту формулу можно почти дословно перевести в теорию категорий. Можно начать с замены интеграла ко-концом. Проблема в том, что нам неизвестно, как вычитать объекты. Однако мы знаем, как их добавить в декартову категорию.

Заметим, что сумма аргументов двух функций равна x. Можно было бы усилить это условие, введя дельта-функцию Дирака или «импульсную функцию», $\delta(a+b-x)$. В теории категорий используется hom-функтор, чтобы сделать то же самое. Таким образом, можно определить свертку двух **Set**-значных функторов:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a+b,x) \times Fa \times Gb$$

Неформально, если бы мы могли определить вычитание как правое сопряжение к ко-произведению, то записали бы:

$$\int^{a,b} \mathcal{C}(a+b,x) \times Fa \times Gb \cong$$

$$\int^{a,b} \mathcal{C}(a,b-x) \times Fa \times Gb \cong \int^{b} F(b-x) \times Gb$$

В ко-произведении нет ничего особенного, поэтому в общем случае D-*свертка* (Day convolution) может быть определена для любой моноидальной категории с тензорным произведением:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b,x) \times Fa \times Gb$$

Фактически, D-свертка для моноидальной категории $(\mathcal{C}, \otimes, I)$ наделяет категорию ко-предпучков $[\mathcal{C}, \mathbf{Set}]$ моноидальной структурой. Легко проверить, что D-свертка ассоциативна (с точностью до изоморфизма), и что $\mathcal{C}(I,-)$ служит единичным объектом. Например, имеем:

$$(C(I,-) \star G)x = \int^{a,b} \mathcal{C}(a \otimes b,x) \times \mathcal{C}(I,a) \times Gb \cong \int^{b} \mathcal{C}(I \otimes b,x) \times Gb \cong Gx$$

Таким образом, единицей D-свертки является функтор Йонеды, взятый в моноидальной единице, что соответствует анаграмматическому лозунгу: «ONE of DAY is a YONEDA of ONE».

Если тензорное произведение симметрично, то и соответствующая D-свертка симметрична (с точностью до изоморфизма).

В частном случае декартово замкнутой категории можно использовать каррированное сопряжение для упрощения формулы:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \times b, x) \times Fa \times Gb \cong \int^{a,b} \mathcal{C}(a, x^b) \times Fa \times Gb \cong \int^{b} F(x^b) \times Gb$$

На Haskell, базирующаяся на произведении D-свертка может быть определена с использованием экзистенциального типа:

```
data Day f g x where
  Day :: ((a, b) -> x) -> f a -> g b -> Day
                                        f g x
```

Если мы представляем функторы как контейнеры значений, D-свертка подсказывает, как объединить два разных контейнера в один, имея функцию, которая объединяет два разных значения в одно.

Упражнение 17.7.1. *Определите экземпляр* `Functor` *для* `Day`.

Упражнение 17.7.2. *Реализуйте ассоциатор для* `Day`.

```
assoc :: Day f (Day g h) x -> Day (Day f g) h x
```

Аппликативные функции как моноиды

Мы знакомы с определением аппликативных функторов как нестрогих моноидальных функторов. Оказывается, как и монады, аппликативные функторы также могут быть определены как моноиды.

Напомним, что моноид — это объект в моноидальной категории. Интересующая нас категория — это ко-предпучковая категория $[\mathcal{C}, \mathbf{Set}]$.

Если $\mathcal{C}$ декартова, то ко-предпучковая категория моноидальна относительно D-свертки с единичным объектом $\mathcal{C}(I, -)$. Моноидом в этой категории является функтор F, снабженный двумя естественными преобразованиями, служащими единицей и умножением:

$$\eta : \mathcal{C}(I, -) \to F$$
$$\mu : F \star F \to F$$

В частности, в декартово замкнутой категории, где единицей является терминальный объект, $\mathcal{C}(1, a)$ изоморфно a, а компонент единицы при a есть:

$$\eta_a : a \to Fa$$

Можно распознать эту функцию как `pure` в определении `Applicative`.

```
pure :: a -> f a
```

Рассмотрим множество естественных преобразований, из которых берется μ. Оформим это множество как конец:

$$\mu \in \int_x \mathbf{Set}(F \star F)x, Fx$$

Подставив определение D-свертки, получим:

$$\int_x \mathbf{Set}\left(\int^{a,b} \mathcal{C}(a \times b, x) \times Fa \times Fb, Fx\right)$$

Можно вынести ко-конец, используя ко-непрерывность hom-функтора:

$$\int_{x,a,b} \mathbf{Set}(\mathcal{C}(a \times b, x) \times Fa \times Fb, Fx)$$

Затем, мы можем использовать каррированное сопряжение в **Set**, получая:

$$\int_{x,a,b} \mathbf{Set}(\mathcal{C}(a \times b, x), \mathbf{Set}(Fa \times Fb, Fx))$$

Наконец, применим лемму Йонеды для выполнения интегрирования по x:

$$\int_{a,b} \mathbf{Set}(Fa \times Fb, F(a \times b))$$

Результатом является множество естественных преобразований, из которого выбирается вторая часть нестрогого моноидального функтора:

```
(>*<) :: f a -> f b -> f (a, b)
```

Свободные аппликативы

Мы узнали, что аппликативные функторы — это моноиды в моноидальной категории:

$$([\mathcal{C}, \mathbf{Set}], \mathcal{C}(I, -), \star)$$

Вполне естественно узнать, что представляет собой свободный моноид в этой категории.

Как и в случае со свободными монадами, создадим свободный аппликатив в качестве инициальной алгебры или наименьшей неподвижной точки функтора списка. Напомним, что функтор списка был определен как:

$$\Phi_a x = 1 + a \otimes x$$

В нашем случае получается:

$$\Phi_F G = \mathcal{C}(I, -) + F \star G$$

Его неподвижная точка задается рекуррентной формулой:

$$A_F \cong \mathcal{C}(I, -) + F \star A_F$$

При переводе этого на Haskell наблюдаем, что функции от единицы `() -> a` изоморфны элементам из `a`.

В соответствии с дополнениями в определении A_F, получаем два конструктора:

```
data FreeA f x where
  DoneA :: x -> FreeA f x
  MoreA :: ((a, b) -> x) -> f a ->
                      FreeA f b -> FreeA f x
```

Вставляем определение D-свертки:

```
data Day f g x where
  Day   :: ((a, b) -> x) -> f a -> g b ->
                                    Day f g x
```

Самый простой способ показать, что `FreeA f` является аппликативным функтором, — пройтись по `Monoidal`:

```
class Monoidal f where
  unit  :: f ()
  (>*<) :: f a -> f b -> f (a, b)
```

Поскольку `FreeA f` является обобщением списка, экземпляр `Monoidal` для свободного аппликатива обобщает идею объединения списков. Мы выполняем сопоставление с образцом в первом списке, что приводит к двум случаям.

В первом случае вместо пустого списка имеем `DoneA x`. Добавление его перед вторым аргументом не меняет длину списка, но изменяет тип хранящихся в нем значений. В парах каждый из них — с `x`:

```
(DoneA x) >*< fry = fmap (x,) fry
```

Второй случай — это «список», у которого голова `fa` является функтором, заполненный значениями `a`, а хвост `frb` имеет тип `FreeA f b`. Обе части «склеиваются» с помощью функции `abx :: (a, b) -> x`.

```
(MoreA abx fa frb) >*< fry = MoreA (reassoc abx)
                                   fa (frb >*< fry)
```

Чтобы получить результат, мы объединяем два хвоста, используя рекурсивный вызов `>*<`, и добавляем к нему `fa`. Чтобы приклеить эту голову к новому хвосту, мы должны предоставить функцию, которая повторно связывает пары:

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

Таким образом, имеется полный экземпляр:

```
instance Functor f => Monoidal (FreeA f) where
  unit     = DoneA ()
  (DoneA x) >*< fry
           = fmap (x,) fry
  (MoreA abx fa frb) >*< fry
           = MoreA (reassoc abx) fa (frb >*< fry)
```

Имея экземпляр `Monoidal`, легко создать экземпляр `Applicative`:

```
instance Functor f => Applicative (FreeA f) where
  pure a    = DoneA a
  ff <*> fx = fmap app (ff >*< fx)

app         :: (a -> b, a) -> b
app (f, a)  = f a
```

Упражнение 17.7.3. *Определите экземпляр* `Functor` *для свободного аппликатива.*

17.8 Бикатегория профункторов

Поскольку мы знаем, как компоновать профункторы с помощью ко-концов, возникает вопрос: существует ли категория, в которой они служат морфизмами? Ответ — да, если мы немного ослабим правила. Проблема в том, что категорные законы для профункторной композиции не выполняются «абсолютно», а только с точностью до изоморфизма.

Например, можно попытаться показать ассоциативность профункторной композиции. Начнем с:

$$((P \diamond Q) \diamond R)\,\langle s, t\rangle = \int^b \left(\int^a P\,\langle s, a\rangle \times Q\,\langle a, b\rangle \right) \times R\,\langle b, t\rangle$$

и, после небольших преобразований, получим:

$$((P \diamond Q) \diamond R)\,\langle s, t\rangle = \int^a P\,\langle s, a\rangle \times \left(\int^b Q\,\langle a, b\rangle \times R\,\langle b, t\rangle \right)$$

Мы использовали ассоциативность произведения и тот факт, что можно поменять порядок ко-концов, используя теорему Фубини. Но оба верны

только с точностью до изоморфизма. Мы не получили «однозначную» ассоциативность.

Тождественный профунктор оказывается hom-функтором, который символически может быть записан как $\mathcal{C}(-,=)$, с заполнителями для обоих аргументов. Например:

$$(\mathcal{C}(-,=) \diamond P)\,\langle s,t\rangle = \int^{a} \mathcal{C}(s,a) \times P\,\langle a,t\rangle \cong P\,\langle s,t\rangle$$

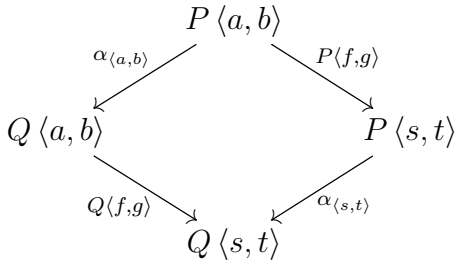
Это следствие (контравариантной) леммы ниндзя-ко-Йонеды, которое также является изоморфизмом, а не равенством.

Категория, в которой категорные законы выполняются с точностью до изоморфизма, называется *бикатегорией*. Обратите внимание, что такая категория должна быть оснащена 2-клетками — морфизмами между морфизмами, с которыми мы уже сталкивались в определении 2-категории. Они нужны для определения изоморфизмов между 1-клетками.

Бикатегория **Prof** имеет (малые) категории в качестве объектов, профункторы как 1-клетки и естественные преобразования как 2-клетки.

Поскольку профункторы являются функторами $\mathcal{C}^{\mathsf{op}} \times \mathcal{D} \to \mathbf{Set}$, между ними применяется стандартное определение естественных преобразований. Это семейство функций, параметризованных объектами $\mathcal{C}^{\mathsf{op}} \times \mathcal{D}$, которые сами являются парами объектов.

Условие естественности преобразования $\alpha_{a,b}$ между двумя профункторами P и Q принимает вид:



для каждой пары стрелок:

$$\langle f : s \to a, g : b \to t\rangle$$

Монады в бикатегории

Мы знаем, что категории, функторы и естественные преобразования образуют 2-категорию **Cat**. Сосредоточимся на одном объекте, категории

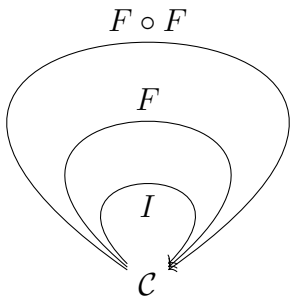
$\mathcal{C}$, которая является 0-клеткой в $\mathbf{Cat}$. 1-клетки, которые начинаются и заканчиваются на этом объекте, образуют обычную категорию, в данном случае это категория функторов $[\mathcal{C},\mathcal{C}]$. Объектами этой категории являются эндо-1-клетки внешней 2-категории $\mathbf{Cat}$. Стрелки между ними — 2-клетки внешней 2-категории.

Эта эндо-1-клеточная категория автоматически снабжена моноидальной структурой. Мы определяем тензорное произведение как композицию 1-клеток — все 1-клетки с одними и теми же источником и целью компонуемы. Моноидальный единичный объект представляет собой тождественную 1-клетку, I. В $[\mathcal{C},\mathcal{C}]$ это произведение представляет собой композицию эндофункторов, а единицей является тождественный функтор.

Если теперь сосредоточить внимание только на одной эндо-1-клетке F, то можно «возвести» ее в квадрат, то есть использовать моноидальное произведение для умножения ее самой на себя. Другими словами, используется 1-клеточная композицию для создания $F \circ F$. Мы говорим, что F является монадой, если можно найти 2-клетки:

$$\mu : F \circ F \to F$$
$$\eta : I \to F$$

которые ведут себя как умножение и единица, то есть они делают ассоциативность и диаграммы единицы коммутативными.



На самом деле, монада может быть определена в произвольной бикатегории, а не только в 2-категории $\mathbf{Cat}$.

Пред-стрелки как монады в Prof

Поскольку **Prof** является бикатегорией, в ней можно определить монаду. Это эндо-профунктор (1-клетка):

$$P : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$$

оснащенный двумя естественными преобразованиями (2-клетки):

$$\mu : P \diamond P \to P$$
$$\eta : \mathcal{C}(-,=) \to P$$

которые удовлетворяют условиям ассоциативности и единицы.

Рассмотрим эти естественные преобразования как элементы концов. Например:

$$\mu \in \int_{\langle a,b\rangle} \mathbf{Set}\left(\int^{x} P\left\langle a,x\right\rangle \times P\left\langle x,b\right\rangle, P\left\langle a,b\right\rangle\right)$$

По ко-непрерывности, это эквивалентно:

$$\int_{\langle a,b\rangle,x} \mathbf{Set}(P\left\langle a,x\right\rangle \times P\left\langle x,b\right\rangle, P\left\langle a,b\right\rangle)$$

Единица есть:

$$\eta \in \int_{\langle a,b\rangle} \mathbf{Set}(\mathcal{C}(a,b), P\left\langle a,b\right\rangle)$$

В Haskell такие профункторные монады называются *пред-стрелками*:

```
class Profunctor p => PreArrow p where
  (»>) :: p a x -> p x b -> p a b
  arr  :: (a -> b) -> p a b
```

`Arrow` — это `PreArrow`, которая также является модулем Тамбары. Мы поговорим о модулях Тамбары в следующей главе.

17.9 Экзистенциальная линза

Первое правило клуба теории категорий — не говорить о внутреннем устройстве объектов. Второе правило гласит, что если необходимо говорить о внутренностях объектов, используйте только стрелки.

Экзистенциальная линза на Haskell

Для объекта, быть составным означает иметь части. Как минимум, должна существовать возможность получить часть такого объекта. Еще лучше, если при этом имеется возможность заменить эту часть на новую. Это в значительной степени определяет линзу:

```
get :: s -> a
set :: s -> a -> s
```

Здесь, `get` извлекает часть `a` из целого `s`, а `set` заменяет эту часть новым `a`. Законы линзы помогают усилить эту картину. И все это делается в терминах стрелок.

Другой способ описания составного объекта состоит в том, чтобы задать его разделение на фокус и остаток. Особенность в том, что, мы желаем знать тип фокуса, но нас совершенно не интересует тип остатка. Все, что нам нужно знать об остатке, это то, что его можно соединить с фокусом для воссоздания всего объекта.

На Haskell мы бы выразили эту идею, используя экзистенциальный тип:

```
data LensE s a where
  LensE :: (s -> (c, a), (c, a) -> s) -> LensE s a
```

Это говорит о том, что существует некоторый неопределенный тип `c` такой, что `s` может быть разбит на произведение `(c, a)` и реконструирован из него.

Версия линзы `get`/`set` может быть получена из экзистенциальной формы:

```
toGet                   :: LensE s a -> (s -> a)
toGet (LensE (l, r))      = snd . l

toSet                   :: LensE s a ->
                                (s -> a -> s)
toSet (LensE (l, r)) s a = r (fst (l s), a)
```

Заметим, что нам не нужно ничего знать о типе остатка. Мы пользуемся тем фактом, что экзистенциальная линза содержит как производителя, так и потребителя `c`, а мы являемся лишь посредниками между ними.

Извлечь остаток «в чистом виде» невозможно, о чем свидетельствует тот факт, что следующий код не будет компилироваться:

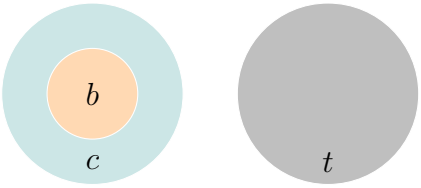
```
getResidue              :: Lens s a -> c
getResidue (Lens (l, r)) = fst . l
```

Экзистенциальная линза в теории категорий

Можно легко перевести новое определение линзы в теорию категорий, выразив экзистенциальный тип в виде ко-конца:

$$\int^{c} \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times a, s)$$

На самом деле, мы можем обобщить его на линзу, меняющую тип, в которой фокус a может быть заменен новым фокусом другого типа b. Замена a на b создаст новый составной объект t:



Линза теперь параметризуется двумя парами: $\langle s, t\rangle$ — для внешних объектов и $\langle a, b\rangle$ — для внутренних объектов. Экзистенциальный остаток c остается скрытым:

$$\mathcal{L}\,\langle s, t\rangle\,\langle a, b\rangle = \int^{c} \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

Произведение под ко-концом — это диагональная часть профунктора, ковариантная по y и контравариантная по x:

$$\mathcal{C}(s, y \times a) \times \mathcal{C}(x \times b, t)$$

Упражнение 17.9.1. *Покажите, что* $\mathcal{C}(s, y \times a) \times \mathcal{C}(x \times b, t)$ *является профунктором в* $\langle x, y\rangle$.

Линза, меняющая тип, на Haskell

На Haskell можно определить меняющую тип линзу как следующий экзистенциальный тип:

```
data LensE s t a b where
  LensE :: (s -> (c, a)) -> ((c, b) -> t) ->
                                  LensE s t a b
```

Как и выше, можно использовать его для получения и установки фокуса:

```
toGet               :: LensE s t a b ->
                                 (s -> a)
toGet (LensE l r)     = snd . l

toSet               :: LensE s t a b ->
                              (s -> b -> t)
toSet (LensE l r) s a = r (fst (l s), a)
```

Простейший пример линзы представляют действия с произведением. Она может извлекать или заменять один компонент произведения, обрабатывая другой как остаток. На Haskell это может быть реализовано так:

```
prodLens :: LensE (c, a) (c, b) a b
prodLens  = LensE id id
```

Здесь, типом объекта является произведение `(c, a)`. Когда мы заменяем `a` на `b`, то получаем целевой тип `(c, b)`. Поскольку источник и цель уже являются произведениями, две функции в определении экзистенциальной линзы — просто тождественности.

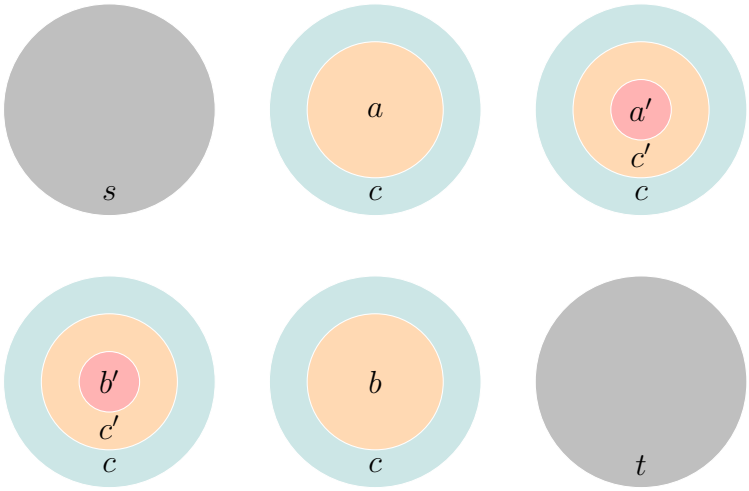
Композиция линз

Главное преимущество использования линз заключается в том, что они компонуемы. Композиция двух линз позволяет увеличить подкомпонент компонента.

Предположим, что мы начинаем с линзы, которая позволяет нам получить доступ к фокусу `a` и заменить его на `b`. Этот фокус является частью целого, описываемого источником `s` и целью `t`.

Также, пусть имеется внутренняя линза, которая может получить доступ к фокусу `a'` внутри целого `a` и заменяет его на `b'` для получения `b`.

Тогда можно построить составную линзу, которая может получить доступ к `a'` и `b'` внутри `s` и `t`. Хитрость заключается в том, чтобы понять, что можно взять в качестве нового остатка произведение двух остатков:




```
compLens             :: LensE a b a' b' ->
                          LensE s t a  b  ->
                            LensE s t a' b'
compLens (LensE l2 r2) (LensE l1 r1)
                       = LensE l3 r3
  where l3 = assoc' . bimap id l2 . l1
        r3 = r1 . bimap id r2 . assoc
```

Левое отображение в новой линзе задается следующей композицией:

$$s \xrightarrow{l_1} (c, a) \xrightarrow{(\mathsf{id}, l_2)} (c, (c', a')) \xrightarrow{assoc'} ((c, c'), a')$$

а правое отображение определяется следующим образом:

$$((c, c'), b') \xrightarrow{assoc} (c, (c', b')) \xrightarrow{(\mathsf{id}, r_2)} (c, b) \xrightarrow{r_1} t$$

Мы использовали ассоциативность и функториальность произведения:

```
assoc                 :: ((c, c'), b') ->
                                        (c, (c', b'))
assoc ((c, c'), b')    = (c, (c', b'))

assoc'                :: (c, (c', a')) ->
                                        ((c, c'), a')
assoc' (c, (c', a'))   = ((c, c'), a')

instance Bifunctor (,) where
    bimap f g (a, b) = (f a, g b)
```

В качестве примера, скомпонуем две линзы произведения:

```
l3 :: LensE (c, (c', a')) (c, (c', b')) a' b'
l3  = compLens prodLens prodLens
```

и применим их к вложенному произведению:

```
x  :: (String, (Bool, Int))
x   = ("Outer", (True, 42))
```

Наша составная линза позволяет не только извлечь самый внутренний компонент:

```
toGet l3 x
> 42
```

но и заменить его значением другого типа (в данном случае, `Char`):

```
toSet l3 x 'z'
> ("Outer",(True,'z'))
```

Категория линз

Поскольку линзы могут быть скомпонованы, может вызвать интерес возможность существования категории, в которой линзы определяют hom-множества.

Действительно, существует категория **Lens**, объекты которой являются парами объектов в $\mathcal{C}$, а стрелки от $\langle s, t\rangle$ к $\langle a, b\rangle$ являются элементами $\mathcal{L}\,\langle s, t\rangle\,\langle a, b\rangle$.

Формула композиции экзистенциальных линз слишком сложна, чтобы ее можно было использовать на практике. В следующей главе мы увидим альтернативное представление линз с использованием модулей Тамбары, в которых композиция — это обычная композиция функций.

17.10 Линзы и расслоения

Существует альтернативный взгляд на линзы на языке пространств расслоений. Проекцию p, задающую расслоение, можно рассматривать как «декомпозицию» пространства E на слои.

В этом представлении p играет роль `get`:

$$p : E \to B$$

База B представляет тип фокуса, а E представляет собой тип совокупности, из которой можно извлечь этот фокус.

Другая часть линзы, `set`, является отображением:

$$q : E \times B \to E$$

Рассмотрим, как это можно интерпретировать с помощью расслоений.

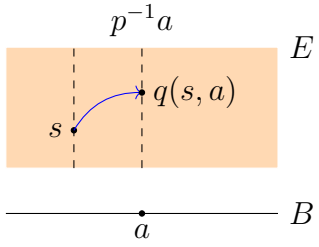
Закон транспортирования

Мы интерпретируем q как «транспортирование» элемента пространства E к новому слою. Новый слой задается элементом из B.

Это свойство транспортирования выражается законом линзы get/set, или законом *транспортирования*, который гласит, что «вы получаете то, что устанавливаете»:

```
get (set s a) = a
```

Говорят, что $q(s, a)$ транспортирует s в новый слой над a:

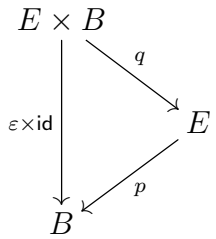


Этот закон можно переписать в терминах p и q:

$$p \circ q = \pi_2$$

где π_2 — вторая проекция произведения.

Эквивалентно, это можно представить в виде коммутативной диаграммы:



Здесь, вместо проекции π_2, мы использовали ко-моноидальную ко-единицу ε:

$$\varepsilon : E \to 1$$

за которым следует закон единицы для произведения. Использование ко-моноида упрощает обобщение этой конструкции на тензорное произведение в моноидальной категории.

Закон тождественности

Следующий закон — set/get или закон *тождественности*. Он выражает простой факт, что «ничего не изменится, если вы установите то, что получите»:

```
set s (get s) = s
```

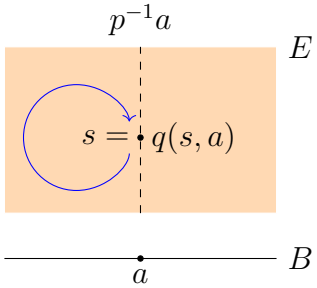
Можно записать его в терминах ко-моноидального ко-умножения:

$$\delta : E \to E \times E$$

Закон «установить/получить» требует, чтобы следующая композиция была тождественностью:

$$E \xrightarrow{\delta} E \times E \xrightarrow{\mathsf{id} \times p} E \times B \xrightarrow{q} E$$

Вот иллюстрация этого закона в пространстве:

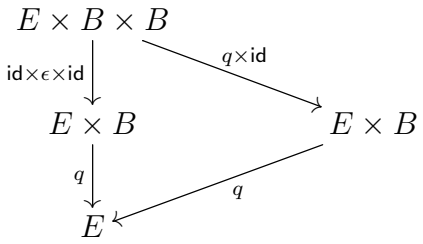


Закон композиции

Наконец, закон set/set, или закон *композиции*. Он означает, что «выигрывает последний set»:

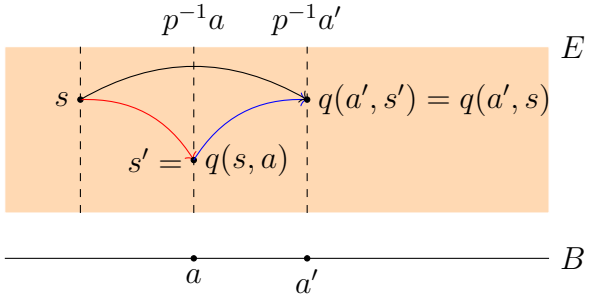
```
set (set s a) a' = set s a'
```

и соответствующая коммутативная диаграмма:



Опять же, чтобы избавиться от среднего B, была использована коединица, а не проекция от произведения.

Вот так выглядит закон set/set в пространстве:

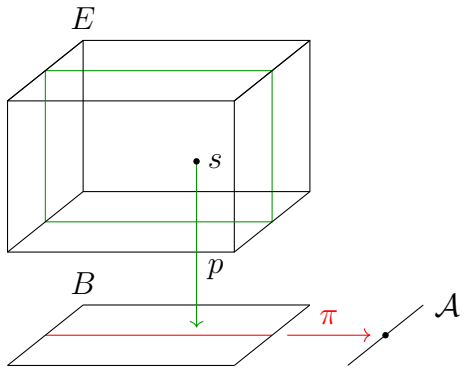


Линза с изменяющимся типом

Линза с изменяющимся типом обобщает транспортирование для действия между пространствами. Мы должны определить целое семейство пространств. Начнем с категории $\mathcal{A}$, объекты которой определяют типы, которые мы будем использовать для фокусов нашей линзы.

Мы конструируем множество B как объединенное множество всех элементов всех типов фокусов. B расслоен над $\mathcal{A}$ — проекцией π, отсылающий элемент B к соответствующему ему типу. Можете представлять себе B как множество объектов категории ко-слоев $1/\mathcal{A}$.

Пространство пространств E — это множество, расслоенное над базой B, с проекцией p. Поскольку сама B расслоена над $\mathcal{A}$, E транзитивно расслоено над $\mathcal{A}$, с составной проекцией $\pi \circ p$. Именно это, более грубое, расслоение разбивает E на семейство пространство. Каждое из этих пространств соответствует различному типу совокупности для данного типа фокуса. Между этими пространствами и будет перемещаться видоизменяющаяся линза.



Проекция p принимает элемент $s \in E$ и производит элемент $b \in B$, тип которого задается посредством π_b. Это — обобщение `get`.

Транспортирование q, соответствующее множеству, принимает элемент $s \in E$ и элемент $b \in B$, и создает новый элемент $t \in E$. Важное замечание состоит в том, что s и t могут принадлежать разным подпространствам из E.

Такое транспортирование удовлетворяет следующим законам.

- Закон **get/set** (транспортирование):

$$p(q(b, s)) = b$$

- Закон **set/get** (тождественность):

$$q(p(s), s) = s$$

- Закон **set/set** (композиция):

$$q(c, q(b, s)) = q(c, s)$$

17.11 Важные формулы

Приведем краткую сводку по исчислению (ко-)концов.

- Непрерывность $\hom$-функтора:

$$\mathcal{D}\left(d, \int_a P\left\langle a, a\right\rangle\right) \cong \int_a \mathcal{D}\left(d, P\left\langle a, a\right\rangle\right)$$

- Ко-непрерывность $\hom$-функтора:

$$\mathcal{D}\left(\int^a P\left\langle a, a\right\rangle, d\right) \cong \int_a \mathcal{D}\left(P\left\langle a, a\right\rangle, d\right)$$

- ниндзя Йонеды:

$$\int_x \mathbf{Set}(\mathcal{C}(a, x), Fx) \cong Fa$$

- ниндзя ко-Йонеды:

$$\int^x \mathcal{C}(x, a) \times Fx \cong Fa$$

- ниндзя Йонеды для контравариантных функторов (предпучков):

$$\int_x \mathbf{Set}(\mathcal{C}(x, a), Gx) \cong Ga$$

- ниндзя ко-Йонеды для контравариантных функторов:

$$\int^x \mathcal{C}(a, x) \times Gx \cong Ga$$

- D-свертка:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb$$

Глава 18

Модули Тамбары

Нечасто случается так, что малоизвестный уголок теории категорий внезапно приобретает известность в программировании. Модули Тамбары обрели новую жизнь в своем применении к профункторной оптике. Они обеспечивают разумное решение проблемы компоновки оптики. Мы видели, что в случае с линзами геттеры (getter — получатель, от `get`) прекрасно компонуются с использованием функциональной композиции, но композиция сеттеров (setter — установщик, от `set`) включает в себя некоторые махинации. Экзистенциальное представление мало помогает. С другой стороны, профункторное представление упрощает композицию.

Ситуация несколько аналогична проблеме компоновки геометрических преобразований в графическом программировании. Например, если вы попытаетесь скомпоновать два вращения вокруг двух разных осей, формула для новой оси и угла будет довольно сложной. Но если вы представляете повороты как матрицы, вы можете использовать матричное умножение; или, если вы представляете их как кватернионы, то можете использовать умножение кватернионов. Профункторное представление позволяет компоновать оптику, используя обычную композицию функций.

18.1 Реконструкция Таннакяна

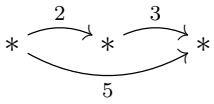
Моноиды и их представления

Теория или представления сами по себе являются наукой. Здесь мы подойдем к этому с категорной точки зрения. Будем рассматривать моноиды. Моноид можно определить как особый объект в моноидальной категории, но его также можно рассматривать и как категорию $\mathcal{M}$ с единственным объектом, скажем $*$. Тогда hom-множество $\mathcal{M}(*,*)$ будет содержать всю необходимую информацию об этом моноиде.

То, что мы назвали «произведением» в моноиде, заменяется композицией морфизмов. По законам категории она ассоциативна и унитальна — тождественный морфизм служит моноидальной единицей.

В этом смысле каждая одно-объектная категория автоматически является моноидом, а все моноиды могут быть превращены в одно-объектные категории.

Например, моноид целых чисел со сложением можно рассматривать как категорию с одним абстрактным объектом $*$ и морфизмом для каждого числа. Чтобы скомпоновать два таких морфизма, вы складываете их номера:



Морфизм, соответствующий нулю, автоматически является тождественным морфизмом.

Можно представить моноид в качестве преобразования множества. Такое представление задается функтором $F : \mathcal{M} \to \mathbf{Set}$. Он отображает единственный объект $*$ к некоторому множеству S, а hom-множество $\mathcal{M}(*,*)$ — к множеству функций $S \to S$. По законам функтора, он отображает тождественность в тождественность, а композицию в композицию, поэтому сохраняет структуру моноида.

Если функтор вполне точен, его образ содержит точно такую же информацию, что и моноид, и ничего более. Но в целом функторы «мухлюют». hom-множество $\mathbf{Set}(S,S)$ может содержать некоторые другие функции, не входящие в образ $\mathcal{M}(*,*)$; и несколько морфизмов в $\mathcal{M}$ могут быть отображены к одной функции.

В крайнем случае, все hom-множество $\mathcal{M}(*,*)$ может быть отображено к тождественному морфизму id_S. Так что, просто взглянув на

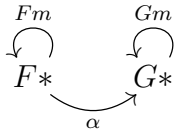
множество S — образ $*$ под функтором F — мы не можем и думать о восстановлении исходного моноида.

Однако, не все потеряно, если нам удастся одновременно рассмотреть все представления данного моноида. Такие представления образуют категорию — функторную категорию $[\mathcal{M}, \mathbf{Set}]$, также известную как ко-предпучковая категория над $\mathcal{M}$. Стрелки в этой категории — это естественные преобразования.

Поскольку исходная категория $\mathcal{M}$ содержит только один объект, условия естественности принимают особенно простой вид. Естественное преобразование $\alpha : F \to G$ имеет только один компонент, функцию $\alpha : F* \to G*$. Для морфизма $m : * \to *$, квадрат естественности выглядит следующим образом:

$$\begin{array}{ccc} F* & \xrightarrow{\alpha} & G* \\ {\scriptstyle Fm}\downarrow & & \downarrow{\scriptstyle Gm} \\ F* & \xrightarrow[\alpha]{} & G* \end{array}$$

что задает отношение между тремя функциями, действующими на два множества:



Это условие естественности выражает то, что:

$$Gm \circ \alpha = \alpha \circ Fm$$

Другими словами, если выбрать элемент x в множестве $F*$, то можно отобразить его к $G*$, используя α, а затем применить преобразование Gm, соответствующее m; или, сначала можно применить преобразование Fm, а затем отобразить результат с помощью α. Итог должен быть одним и тем же.

Такие функции называются *эквивариантными*. Часто Fm называют *действием* m на множестве $F*$. Эквивариантная функция связывает действие на одном множестве с соответствующим действием на другом множестве, используя, либо пред-, либо пост-композицию.

Реконструкция Таннакяна для моноида

Сколько информации необходимо, чтобы восстановить моноид из его представлений? Одного взгляда на множества определенно недостаточно, так как любой моноид может быть представлен на любом множестве. Но если использовать функции, сохраняющие структуру, между этими множествами, то может появиться шанс.

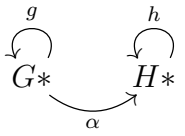
Для функтора $F : \mathcal{M} \to \mathbf{Set}$ рассмотрим множество *всех* функций $\mathbf{Set}(F*, F*)$, т.е. hom-множество $\mathbf{Set}(F*, F*)$. По крайней мере, некоторые из этих функций являются действиями моноида. Это функции вида Fm, где m — стрелка в $\mathcal{M}$. Однако, следует иметь в виду, что в этом hom-множестве может содержаться гораздо больше функций, которые не соответствуют действиям.

Теперь рассмотрим другое множество $G*$, являющееся образом некоторого другого функтора G. В его hom-множестве $\mathbf{Set}(G*, G*)$ можно обнаружить, среди прочего, соответствующие действия вида Gm. Эквивариантная функция, т.е. естественное преобразование в $[\mathcal{M}, \mathbf{Set}]$, согласует любые два связанных действия Fm к Gm.

Теперь представьте, что вы создаете гигантский кортеж, беря по одной функции из каждого множества $\mathbf{Set}(F*, F*)$, для всех функторов $F : \mathcal{M} \to \mathbf{Set}$. Нас интересуют только кортежи, элементы которых связаны. Вот что имеется в виду: если мы выберем $g \in \mathbf{Set}(G*, G*)$ и $h \in \mathbf{Set}(H*, H*)$, и существует естественное преобразование (эквивариантная функция) α между функторами G и H, мы хотим, чтобы эти две функции были связаны:

$$\alpha \circ g = h \circ \alpha$$

или, графически:

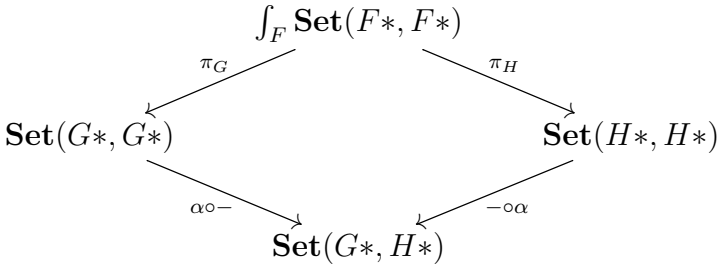


Заметим, что эта корреляция также работает с парами g и h, которые не соответствуют виду $g = Gm$ и $h = Hm$.

Такие кортежи и есть элементы конца:

$$\int_F \mathbf{Set}(F*, F*)$$

условие клина которых обеспечивает искомые ограничения.



Здесь, α — это морфизм в категории функторов $[\mathcal{M}, \mathbf{Set}]$:

$$\alpha : G \to H$$

Это естественное преобразование имеет только одну компоненту, которую мы также обозначим α. Она является эквивариантной функцией между этими двумя представлениями.

Вот некоторые детали. Профунктор под этим концом задается как:

$$P \langle G, H \rangle = \mathbf{Set}(G*, H*)$$

Это есть функтор вида:

$$P : [\mathcal{M}, \mathbf{Set}]^{\mathsf{op}} \times [\mathcal{M}, \mathbf{Set}] \to \mathbf{Set}$$

Рассмотрим его действие на парах морфизмов в $[\mathcal{M}, \mathbf{Set}]$. Для пары естественных преобразований:

$$\alpha : G' \to G$$
$$\beta : H \to H'$$

их поднятие является функцией

$$P \langle \alpha, \beta \rangle : P \langle G, H \rangle \to P \langle G', H' \rangle$$

Подставляя наше определение для P, имеем:

$$P \langle \alpha, \beta \rangle : \mathbf{Set}(G*, H*) \to \mathbf{Set}(G'*, H'*)$$

Мы получаем эту функцию путем пред-композиции с α и пост-композиции с β (эти функции являются единственными компонентами двух естественных преобразований, α и β):

$$P \langle \alpha, \beta \rangle = \beta \circ - \circ \alpha$$

То есть, по заданной функции $f \in \mathbf{Set}(G*, H*)$, получаем функцию $\beta \circ f \circ \alpha \in \mathbf{Set}(G'*, H'*)$.

В условии клина, если мы выбираем g, как элемент из $\mathbf{Set}(G*, G*)$, и h, как элемент из $\mathbf{Set}(H*, H*)$, то воспроизводим наше условие:

$$\alpha \circ g = h \circ \alpha$$

В этом случае, теорема реконструкции Таннакяна выражается в виде:

$$\int_F \mathbf{Set}(F*, F*) \cong \mathcal{M}(*, *)$$

Другими словами, мы можем восстановить моноид по его представлениям. Мы рассмотрим доказательство этой теоремы в контексте более общего утверждения.

Теорема Кэли

В теории групп теорема Кэли утверждает, что каждая группа изоморфна подгруппе группы перестановок. *Группа* — это просто моноид, в котором каждый элемент имеет обратный. Перестановки — это биективные функции, которые отображают множество в себя. Они *переставляют* элементы множества.

В теории категорий теорема Кэли практически встроена в определение моноида и его представлений.

Связь между одно-объектной интерпретацией и более традиционной интерпретацией моноида, основанной на множестве элементов, установить несложно. Мы сделаем это, создавая функтор $F : \mathcal{M} \to \mathbf{Set}$, который отображает $*$ к специальному множеству S, которое эквивалентно hom-множеству: $S = \mathcal{M}(*, *)$. Элементы этого множества отождествляются с морфизмами в $\mathcal{M}$. Определим действие F на морфизмах как пост-композицию:

$$(Fm)n = m \circ n$$

Здесь, m — морфизм в $\mathcal{M}$, а n — элемент из S, который также является морфизмом в $\mathcal{M}$.

Мы можем принять это конкретное представление как альтернативное определение моноида в моноидальной категории $\mathbf{Set}$. Все, что оста-

ется, это реализовать единицу и умножение:

$$\eta : 1 \to S$$
$$\mu : S \times S \to S$$

Единица выбирает элемент в S, который соответствует id_* в $\mathcal{M}(*,*)$. Умножение двух элементов m и n задается элементом, соответствующим $m \circ n$.

В то же время, мы можем рассматривать S как образ $F : \mathcal{M} \to \mathbf{Set}$, и, в этом случае, именно функции $S \to S$образуют представление моноида. В этом суть теоремы Кэли: каждый моноид может быть представлен множеством эндофункций.

В программировании, лучшим примером применения теоремы Кэли является эффективная реализация обращения списка. Напомним наивную рекурсивную реализацию обращения:

```
reverse          :: [a] -> [a]
reverse []        = []
reverse (a : as) = reverse as ++ [a]
```

Она разбивает список на голову и хвост, инвертирует хвост и добавляет к результату синглетон, состоящий из головы. Проблема в том, что каждое добавление должно проходить по растущему списку, что приводит к оценке производительности $O(N^2)$.

Однако, помните, что список — это (свободный) моноид:

```
instance Monoid [a] where
  mempty        = []
  mappend as bs = as ++ bs
```

Можно использовать теорему Кэли, чтобы представить этот моноид как функции на списках:

```
type DList a = [a] -> [a]
```

Чтобы представить список, мы превращаем его в функцию. Это функция (замкнутая), которая добавляет этот список `as` к своему аргументу `xs`:

```
rep   :: [a] -> DList a
rep as = \xs -> as ++ xs
```

Это представление называется *списком различий*.

Чтобы превратить функцию обратно в список, достаточно применить его к пустому списку:

```
unRep  :: DList a -> [a]
unRep f = f []
```

Легко проверить, что представление пустого списка является тождественной функцией, а представление конкатенации двух списков — композицией представлений:

```
rep []         = id
rep (xs ++ ys) = rep xs . rep ys
```

Так что, это точно представление Кэли моноида списка.

Теперь можно преобразовать алгоритм обращения, чтобы получить новое представление:

```
rev          :: [a] -> DList a
rev []        = rep []
rev (a : as) = rev as . rep [a]
```

и вернуть его обратно в список:

```
fastReverse :: [a] -> [a]
fastReverse  = unRep . rev
```

На первый взгляд может показаться, что ничего особенного не сделано, кроме добавления слоя преобразования поверх нашего рекурсивного алгоритма. За исключением того, что новый алгоритм имеет оценку производительности $O(N)$, а не $O(N^2)$. Чтобы убедиться в этом, рассмотрим обращение простого списка `[1, 2, 3]`. Функция `rev` превращает этот список в композицию функций:

```
rep [3] . rep [2] . rep [1]
```

Она делает это за линейное время. Функция `unRep` выполняет эту компоновку, начиная с пустого списка. Но обратите внимание, что каждое `rep` *добавляет* свой аргумент к совокупному результату. В частности, последнее `rep [3]` выполняет:

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

В отличие от привычного добавления, пред-добавление в начале — это операция с постоянным временем, поэтому весь алгоритм выполняется за линейное время.

Другой способ взглянуть на это — понять, что `rev` ставит действия в очередь в порядке элементов списка, начиная с головы. Но очередь функций выполняется в порядке «первым поступил — первым обслужен» (FIFO).

Из-за ленивости Haskell обращение списка с помощью `foldl` имеет аналогичную производительность:

```
reverse = foldl (\as a -> a : as) []
```

Это связано с тем, что `foldl`, перед возвратом результата, проходит по списку слева направо, накапливая функции (замыкания), которые выполняются, по мере необходимости, в порядке FIFO.

Обоснование реконструкции Таннакяна

Моноидальная реконструкция — это частный случай более общей теоремы, в которой вместо одно-объектной категории используется регулярная категория. Как и в случае с моноидом, восстановим hom-множество, только на этот раз это будет обычное hom-множество между двумя объектами. Докажем формулу:

$$\int_{F:[\mathcal{C},\mathbf{Set}]} \mathbf{Set}(Fa, Fb) \cong \mathcal{C}(a, b)$$

Прием заключается в том, чтобы использовать лемму Йонеды для представления действия F:

$$Fa \cong [\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F)$$

и то же самое для Fb. Получаем:

$$\int_{F:[\mathcal{C},\mathbf{Set}]} \mathbf{Set}\left([\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), F), [\mathcal{C}, \mathbf{Set}](\mathcal{C}(b, -), F)\right)$$

Отметим, что эти два множества естественных преобразований являются hom-множествами в $[\mathcal{C}, \mathbf{Set}]$.

Напомним следствие из леммы Йонеды, справедливое для любой категории $\mathcal{A}$:

$$[\mathcal{A}, \mathbf{Set}](\mathcal{A}(x,-), \mathcal{A}(y,-)) \cong \mathcal{A}(y,x)$$

Можно записать это, используя конец:

$$\int_{z:\mathcal{C}} \mathbf{Set}(\mathcal{A}(x,z), \mathcal{A}(y,z)) \cong \mathcal{A}(y,x)$$

В частности, можно заменить $\mathcal{A}$ категорией функторов $[\mathcal{C}, \mathbf{Set}]$. Получаем:

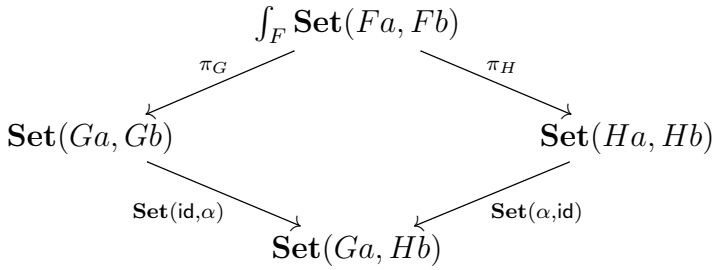
$$\int_{F:[\mathcal{C},\mathbf{Set}]} \mathbf{Set}\left([\mathcal{C}, \mathbf{Set}](\mathcal{C}(a,-), F), [\mathcal{C}, \mathbf{Set}](\mathcal{C}(b,-), F)\right) \cong [\mathcal{C}, \mathbf{Set}]\left(\mathcal{C}(b,-), \mathcal{C}(a,-)\right)$$

Затем можно снова применить лемму Йонеды к правой части и получить:

$$\mathcal{C}(a,b)$$

что и является искомым результатом.

Важно понимать, как структура категории функторов входит в конец через условие клина. Это делается через естественные преобразования. Каждый раз, когда имеется естественное преобразование между двумя функторами $\alpha : G \to H$, следующая диаграмма должна быть коммутативной:



Чтобы получить некоторое представление о реконструкции Таннакяна, можно вспомнить, что **Set**-значные функторы интерпретируются как релевантные для доказательства подмножества. Функтор $F : \mathcal{C} \to \mathbf{Set}$ (ко-предпучок) определяет подмножество объектов (малой категории) $\mathcal{C}$. Скажем, что объект a находится в этом подмножестве, только если Fa

непусто. Каждый элемент Fa можно интерпретировать тогда как доказательство этого.

Но, если рассматриваемая категория не является дискретной, то не все подмножества будут соответствовать ко-предпучкам. В частности, всякий раз, когда имеется стрелка $f : a \to b$, то существует функция $Ff : Fa \to Fb$. Согласно нашей интерпретации, такая функция отображает каждое доказательство того, что a находится в подмножестве, определяемом F, в доказательство того, что b находится в этом подмножестве. Таким образом, ко-предпучки определяют релевантные для доказательства подмножества, совместимые со структурой категории.

Попробуем переосмыслить реконструкцию Таннакяна в том же духе.

$$\int_{F:[\mathcal{C},\mathbf{Set}]} \mathbf{Set}(Fa, Fb) \cong \mathcal{C}(a, b)$$

Элемент левой части является доказательством того, что для каждого подмножества, совместимого со структурой $\mathcal{C}$, если a принадлежит этому подмножеству, то же самое относится и к b. Это возможно только при наличии стрелки от a к b.

Реконструкция Таннакяна на Haskell

Можно сразу перевести этот результат на Haskell. Заменяем конец на `forall`. Левая сторона принимает вид:

```
forall f. Functor f => f a -> f b
```

а правая часть — это функциональный тип `a -> b`.

Мы уже рассматривали полиморфные функции: это были функции, определенные для всех типов, а иногда и для классов типов. Здесь, имеем функцию, определенную для всех функторов. Она действует следующим образом: имея функтор, полный `a`, создается функтор, полный `b`, независимо от используемого функтора. Единственный способ реализовать это (используя параметрический полиморфизм) — скрытный захват этой функцией типа `a -> b` и применение ее с помощью `fmap`.

В самом деле, одно из направлений изоморфизма действует так, чтобы захватить функцию и преобразовать ее в аргумент:

```
toRep      :: (a -> b) -> (forall f. Functor f =>
                                      f a -> f b)
toRep g fa  = fmap g fa
```

В другом направлении используется прием Йонеды:

```
fromRep    :: (forall f. Functor f => f a -> f b)
                                     -> (a -> b)
fromRep g a = unId (g (Id a))
```

где тождественный функтор определяется как:

```
data Id a   = Id a

unId       :: Id a -> a
unId (Id a) = a

instance  Functor Id where
  fmap g (Id a) = Id (g a)
```

Такая реконструкция может показаться тривиальной и бессмысленной. Зачем стремиться заменить тип функции `a -> b` гораздо более сложным типом:

```
type Getter a b = forall f. Functor f => f a
                                      -> f b
```

Однако, поучительно понимать `a -> b` как предшественника всей оптики. Это линза, которая фокусируется на b, представляющего часть a. Она выражает то, что a содержит достаточно информации, в той или иной форме, чтобы построить b. Это — «геттер» или «средство доступа».

Очевидно, функции компонуются. Что интересно, так это то, что представления функторов также компонуются, причем с использованием простой композиции функций, как показано в примере:

```
boolToStrGetter :: Getter Bool String
boolToStrGetter  = toRep (show) . toRep (bool 0 1)
```

Другие оптики компонуются не так просто, но их функторные (и профункторные) представления обладают этой возможностью.

Реконструкция Таннакяна с сопряжением

Особенность в обобщении реконструкции Таннакяна состоит в том, чтобы определить конец над некоторой специальной категорией функторов $\mathcal{T}$, сначала применяя забывающий функтор к его функторам. Предположим, что имеется свободное/забывающее сопряжение $F \dashv U$ между двумя функторными категориями, $\mathcal{T}$ и $[\mathcal{C}, \mathbf{Set}]$:

$$\mathcal{T}(FQ, P) \cong [\mathcal{C}, \mathbf{Set}](Q, UP)$$

где Q — функтор в $[\mathcal{C}, \mathbf{Set}]$, а P — функтор в $\mathcal{T}$.

Нашей отправной точкой для реконструкции Таннакяна является следующий конец:

$$\int_{P:\mathcal{T}} \mathbf{Set}\left((UP)a, (UP)s\right)$$

Между прочим, отображение $\mathcal{T} \to \mathbf{Set}$, параметризованное объектом a, и заданное формулой:

$$P \mapsto (UP)a$$

иногда называют *функтором слоя*, поэтому формулу конца можно интерпретировать как множество естественных преобразований между двумя функторами слоя. Концептуально функтор слоя описывает «бесконечно малую окрестность» объекта. Он отображает функторы в множества, но, что более важно, он отображает естественные преобразования в функции. Эти функции исследуют среду, в которой находится объект. В частности, естественные преобразования в $\mathcal{T}$ связаны с условиями клина, определяющими конец (в исчислении, слои пучков играют очень похожую роль).

Как и выше, сначала применим лемму Йонеды, чтобы получить:

$$\int_{P:\mathcal{T}} \mathbf{Set}\left([\mathcal{C}, \mathbf{Set}](\mathcal{C}(a, -), UP), [\mathcal{C}, \mathbf{Set}](\mathcal{C}(s, -), UP)\right)$$

Теперь можно использовать сопряжение:

$$\int_{P:\mathcal{T}} \mathbf{Set}\left(\mathcal{T}(F\mathcal{C}(a, -), P), \mathcal{T}(F\mathcal{C}(s, -), P)\right)$$

В итоге, получаем отображение между двумя естественными преобразованиями в категории функторов $\mathcal{T}$. Далее, выполняем интегрирование, используя следствие из леммы Йонеды, что дает:

$$\mathcal{T}(F\mathcal{C}(s, -), F\mathcal{C}(a, -))$$

Можно применить сопряжение еще раз:

$$\mathbf{Set}(\mathcal{C}(s,-),(U\circ F)\mathcal{C}(a,-))$$

и опять лемму Йонеды:

$$((U\circ F)\mathcal{C}(a,-))s$$

Последнее наблюдение состоит в том, что композиция $U\circ F$ сопряженных функторов является монадой в категории функторов. Обозначим эту монаду через Φ. В результате получается следующее тождество, которое послужит основой для профункторной оптики:

$$\int_{P:\mathcal{T}} \mathbf{Set}((UP)a,(UP)s)\cong(\Phi\mathcal{C}(a,-))s$$

Правая часть — это действие монады $\Phi = U\circ F$ на представимом функторе $\mathcal{C}(a,-)$, который тогда вычисляем в точке s.

Сравните это с более ранней формулой реконструкции Таннакяна, особенно, если мы перепишем ее, в следующем виде:

$$\int_{P:\mathcal{T}} \mathbf{Set}(Fa,Fs)\cong\mathcal{C}(a,-)s$$

Надо иметь в виду, что при выводе оптики мы заменим a и s парами объектов $\langle a,b\rangle$ и $\langle s,t\rangle$ из $\mathcal{C}^{\mathsf{op}}\times\mathcal{C}$. В этом случае наши функторы станут профункторами.

18.2 Профункторные линзы

Наша цель — найти функторное представление для оптик. Мы уже сталкивались с этим раньше, например, линзы, меняющие тип, можно рассматривать как hom-множества в категории **Lens**. Объекты в **Lens** представляют собой пары объектов из некоторой категории $\mathcal{C}$, а hom-множество от одной такой пары, $\langle s,t\rangle$, к другой, $\langle a,b\rangle$, определяется по формуле ко-конца:

$$\mathcal{L}\,\langle s,t\rangle\,\langle a,b\rangle = \int^{c}\mathcal{C}(s,c\times a)\times\mathcal{C}(c\times b,t)$$

Заметим, что пару hom-множеств в этой формуле можно рассматривать как одно hom-множество в категории произведений $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$:

$$\mathcal{L}\left\langle s,t\right\rangle \left\langle a,b\right\rangle = \int^{c} (\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(c \bullet \left\langle a,b\right\rangle, \left\langle s,t\right\rangle)$$

где действие c на пару $\langle a,b\rangle$ определяется как:

$$c \bullet \langle a,b\rangle = \langle c \times a, c \times b\rangle$$

Это — сокращенное обозначение диагональной части более общего действия $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$ на себя, определяемое как:

$$\langle c,c'\rangle \bullet \langle a,b\rangle = \langle c \times a, c' \times b\rangle$$

Это говорит о том, что для представления такой оптики мы должны рассматривать ко-предпучки в категории $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$, то есть надо рассматривать профункторные представления.

Iso

Чтобы быстро проверить эту идею, применим нашу формулу реконструкции к простому случаю $\mathcal{T} = [\mathcal{C}^{\mathsf{op}} \times \mathcal{C}, \mathbf{Set}]$, без дополнительной структуры. В этом случае не нужно использовать забывающие функторы или монаду Ф, и мы просто получаем простое приложение реконструкции Таннакяна:

$$\mathcal{O}\left\langle s,t\right\rangle \left\langle a,b\right\rangle = \int_{P:\mathcal{T}} \mathbf{Set}(P\left\langle a,b\right\rangle, P\left\langle s,t\right\rangle) \cong ((\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(\left\langle a,b\right\rangle, -))\left\langle s,t\right\rangle$$

Правая часть оценивается следующим образом:

$$(\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(\left\langle a,b\right\rangle, \left\langle s,t\right\rangle) = \mathcal{C}(s,a) \times \mathcal{C}(b,t)$$

Эта оптика известна в Haskell как `Iso` (или, адаптер):

```
type Iso s t a b = (s -> a, b -> t)
```

и у нее действительно есть профункторное представление, соответствующее следующему концу:

```
type IsoP s t a b = forall p. Profunctor p =>
                                p a b -> p s t
```

Имея пару функций, легко построить эту профункторно-полиморфную функцию:

```
toIsoP       :: (s -> a, b -> t) -> IsoP s t a b
toIsoP (f, g) = dimap f g
```

Это просто говорит о том, что любой профунктор может быть использован для поднятия пары функций.

И наоборот, можно задать вопрос: как одиночная полиморфная функция может отображать множество $P\langle a, b\rangle$ к множеству $P\langle s, t\rangle$ для *любого* вообразимого профунктора? Единственное, что эта функция знает о профункторе, это то, что он может поднимать пару функций. Следовательно, это должно быть замыкание, которое либо содержит пару функций `(s -> a, b -> t)`, либо может их произвести.

Упражнение 18.2.1. *Реализуйте функцию*:

```
fromIsoP :: IsoP s t a b -> (s -> a, b -> t)
```

Подсказка: *используйте тот факт, что пара тождественных функций может быть использована для построения следующего профунктора*:

```
data Adapter a b s t = Ad (s -> a, b -> t)
```

Профункторы и линзы

Попробуем применить ту же логику к линзам. Нужно найти класс профункторов, чтобы включить его в наше представление профункторов. Предположим, что забывающий функтор U забывает только дополнительную структуру, но не меняет множества, поэтому множество $P\langle a, b\rangle$ совпадает с множеством $(UP)\langle a, b\rangle$.

Начнем с экзистенциального представления. В нашем распоряжении есть объект c и пара функций:

$$\langle f, g\rangle : \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

Требуется построить представление профунктора, поэтому должна существовать возможность отображать множество $P\langle a, b\rangle$ к множеству $P\langle s, t\rangle$. Можно было бы получить $P\langle s, t\rangle$, подняв эти две функции, но только если начать с $P\langle c \times a, c \times b\rangle$. Действительно:

$$P\langle f, g\rangle : P\langle c \times a, c \times b\rangle \to P\langle s, t\rangle$$

Чего нам не хватает, так это отображения:

$$P\langle a, b\rangle \to P\langle c \times a, c \times b\rangle$$

И это именно та дополнительная структура, которая будет требоваться от нашего класса профункторов.

Модуль Тамбары

Профунктор P, оснащенный семейством преобразований:

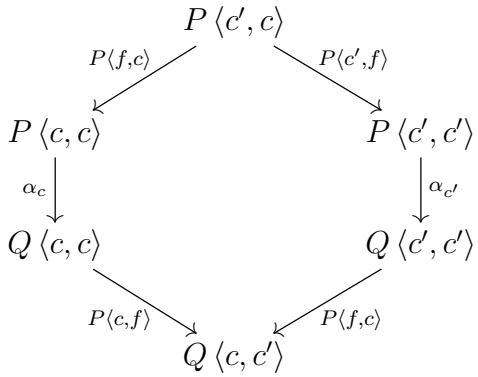
$$\alpha_{\langle a,b\rangle,c} : P\langle a, b\rangle \to P\langle c \times a, c \times b\rangle$$

называется *модулем Тамбары*.

Мы хотим, чтобы это семейство было естественным в a и b, но что мы должны требовать от c? Проблема с c в том, что c появляется дважды, один раз в контравариантном и один раз в ковариантном положении. Таким образом, если мы хотим правильно взаимодействовать со стрелками, такими как $h : c \to c'$, то должны изменить условие естественности. Можно рассмотреть более общий профунктор $P\langle c' \times a, c \times b\rangle$ и рассматривать α как порождающий его диагональные элементы, для которых c' совпадает с c.

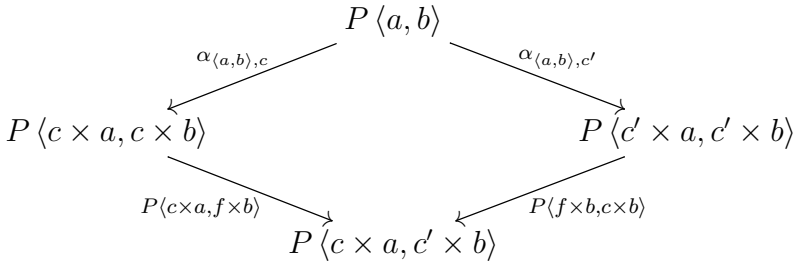
Преобразование α между диагональными частями двух профункторов P и Q называется *ди-естественным преобразованием* (диагонально естественным), если следующая диаграмма является коммутативной,

для любой $f : c \to c'$:



(используется обычное сокращение $P\langle f, c\rangle$, вискеринг, для $P\langle f, \mathsf{id}_c\rangle$).

В нашем случае условие ди-естественности упрощается до:



(здесь, снова $P\langle f \times b, c \times b\rangle$ означает $P\langle f \times \mathsf{id}_b, \mathsf{id}_{c\times b}\rangle$).

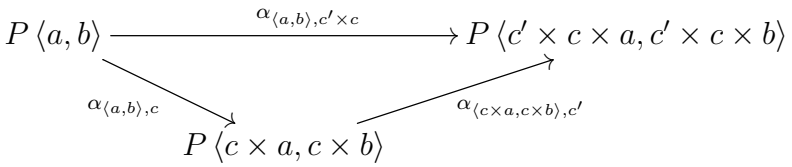
Имеется еще одно условие непротиворечивости модулей Тамбары: они должны сохранять моноидальную структуру. Действие умножения на c имеет смысл в декартовой категории: должно существовать произведение для любой пары объектов, а терминальный объект должен служить единицей умножения. Модули Тамбары должны не нарушать тождества единицы и сохранять умножение. Для единицы (терминального объекта) дополнительно предписываем следующее условие:

$$\alpha_{\langle a,b\rangle,1} = \mathsf{id}_{P\langle a,b\rangle}$$

Для умножения имеем:

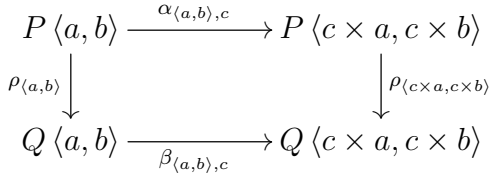
$$\alpha_{\langle a,b\rangle,c'\times c} \cong \alpha_{\langle c\times a,c\times b\rangle,c'} \circ \alpha_{\langle a,b\rangle,c}$$

или, схематически:



(обратите внимание, что произведение ассоциативно с точностью до изоморфизма, поэтому на этой диаграмме имеется скрытый ассоциатор).

Поскольку мы хотим, чтобы модули Тамбары образовывали категорию, то необходимо определить морфизмы между ними. Это естественные преобразования, сохраняющие дополнительную структуру. Допустим, имеется естественное преобразование ρ между двумя модулями Тамбары $\rho : (P, \alpha) \to (Q, \beta)$. Мы можем, либо применить α, а затем ρ, либо сначала выполнить ρ, а затем β. Требуется, чтобы результат был одним и тем же:



Имейте в виду, что структура категории Тамбары закодирована в этих естественных преобразованиях. Они будут определять, через условие клина, форму концов, входящих в определение профункторных линз.

Профункторные линзы

Теперь, когда у нас есть некоторое представление о том, как модули Tambara связаны с линзами, вернемся к основной формуле:

$$\mathcal{L}\,\langle s, t\rangle\,\langle a, b\rangle = \int_{P:\mathcal{T}} \mathbf{Set}((UP)\,\langle a, b\rangle\,, (UP)\,\langle s, t\rangle) \cong (\Phi(\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(\langle a, b\rangle\,, -))\,\langle s, t\rangle$$

Мы завершаем обсуждение категории Тамбары. Единственная отсутствующая часть — это монада $\Phi = U \circ F$ или функтор F, свободно порождающий модули Тамбары.

Оказывается, вместо того, чтобы искать монаду, проще подобрать комонаду. В категории профункторов имеется ко-монада, которая принимает профунктор P и производит другой профунктор ΘP. Вот формула:

$$(\Theta P)\left\langle a, b\right\rangle = \int_c P\left\langle c \times a, c \times b\right\rangle$$

Можно проверить, что это действительно комонада, реализуя ε и δ (`extract` и `duplicate`). Например, ε отображает $\Theta P \to P$, используя проекцию π_1 для терминального объекта (единица декартова произведения).

Что интересно в этой комонаде, так это то, что ее коалгебры являются модулями Тамбары. Опять же, это коалгебры, которые отображают профункторы в профункторы. Это естественные преобразования $P \to \Theta P$. Такое естественное преобразование можно записать как элемент конца:

$$\int_{a,b} \mathbf{Set}(P\left\langle a, b\right\rangle, (\Theta P)\left\langle a, b\right\rangle) = \int_{a,b}\int_c \mathbf{Set}(P\left\langle a, b\right\rangle, P\left\langle c \times a, c \times b\right\rangle)$$

Здесь используется непрерывность $\hom$-функтора, чтобы выдвинуть конец поверх c. Полученный конец кодирует множество естественных (диестественных в c) преобразований, которые определяют модуль Тамбары:

$$\alpha_{\langle a,b\rangle,c} : P\left\langle a, b\right\rangle \to P\left\langle c \times a, c \times b\right\rangle$$

Фактически эти коалгебры являются коалгебрами ко-монад, т.е. совместимы с ко-монадой Θ. Другими словами, модули Тамбары образуют категорию коалгебр Эйленберга-Мура для ко-монады Θ.

Левое сопряжение к Θ есть монада Φ, заданная формулой:

$$(\Phi P)\left\langle s, t\right\rangle = \int^{u,v,c} (\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(c \bullet \left\langle u, v\right\rangle, \left\langle s, t\right\rangle) \times P\left\langle u, v\right\rangle$$

где используется сокращенная запись:

$$(\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(c \bullet \left\langle u, v\right\rangle, \left\langle s, t\right\rangle) = \mathcal{C}(s, c \times u) \times \mathcal{C}(c \times v, t)$$

Это сопряжение можно легко проверить, используя некоторые манипуляции с ко/концом. Отображение-вне ΦP к некоторому профунктору

Q можно записать как конец. Затем ко-концы в Φ можно удалить, используя ко-непрерывность hom-функтора. Наконец, применение леммы ниндзя-Йонеды приводит к отображению в ΘQ. Получаем:

$$[\mathcal{C}^{\mathsf{op}} \times \mathcal{C}, \mathbf{Set}](P\Phi, Q) \cong [\mathcal{C}^{\mathsf{op}} \times \mathcal{C}, \mathbf{Set}](P, \Theta Q)$$

Заменяя Q на P, сразу видим, что множество алгебр для Φ изоморфно множеству коалгебр для Θ. Фактически, они являются монадными алгебрами для Φ. Это означает, что категория Эйленберга-Мура для монады Φ совпадает с категорией Тамбары.

Напомним, что конструкция Эйленберга-Мура разлагает монаду на свободное/забывающее сопряжение. Это именно то сопряжение, которое мы искали, выводя формулу для профункторной оптики. Остается оценить действие Φ на представимом функторе:

$$(\Phi(\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(\langle a, b\rangle, -))\,\langle s, t\rangle = \int^{u,v,c} (\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(c \bullet \langle u, v\rangle, \langle s, t\rangle) \times (\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(\langle a, b\rangle, \langle u, v\rangle)$$

Применяя лемму ко-Йонеды, получаем:

$$\int^{c} (\mathcal{C}^{\mathsf{op}} \times \mathcal{C})(c \bullet \langle u, v\rangle, \langle s, t\rangle) = \int^{c} \mathcal{C}(s, c \times a) \times \mathcal{C}(c \times b, t)$$

что является в точности экзистенциальным представлением линзы.

Профункторные линзы на Haskell

Чтобы определить представление профунктора на Haskell, введем класс профункторов, которые являются модулями Тамбары по отношению к декартовому произведению (далее мы увидим более общие модули Тамбары). В библиотеке Haskell этот класс называется `Strong`. Он также появляется в литературе как `Cartesian`:

```
class Profunctor p => Cartesian p where
  alpha :: p a b -> p (c, a) (c, b)
```

Полиморфная функция `alpha` обладает всеми соответствующими свойствами естественности, гарантированными параметрическим полиморфизмом.

Профункторная линза — это всего лишь синоним функционального типа, который является полиморфным в декартовых профункторах `Cartesian`:

```
type LensP s t a b = forall p. Cartesian
                               p => p a b -> p s t
```

Самый простой способ реализовать такую функцию — начать с экзистенциального представления линзы и применить `alpha`, а затем `dimap` к профункторному аргументу:

```
toLensP                :: LensE s t a b ->
                                  LensP s t a b
toLensP (LensE from to) = dimap from to . alpha
```

Поскольку профункторные линзы — это просто функции, то их можно скомпоновать соответствующим образом:

```
lens1 :: LensP s t x y   -- p s t -> p x y
lens2 :: LensP x y a b   -- p x y -> p a b
lens3 :: LensP s t a b   -- p s t -> p a b
lens3  = lens2 . lens1
```

Также возможно обратное отображение из представления профунктора в set/get-представление линзы. Для этого нужно подобрать профунктор, который можно передать в `LensP`. Получается, что get/set-представление линзы является таким профунктором, когда мы фиксируем пару типов, `a` и `b`. Определим:

```
data FlipLens a b s t = FlipLens (s -> a)
                                 (s -> b -> t)
```

Легко показать, что это действительно профунктор:

```
instance Profunctor (FlipLens a b) where
  dimap f g (FlipLens get set)
    = FlipLens (get . f) (fmap g . set . f)
```

Мало того — это еще и профунктор `Cartesian`:

```
instance Cartesian (FlipLens a b) where
  alpha(FlipLens get set) = FlipLens get'
                                     set'
    where get' = get . snd
          set' = \(x, s) b -> (x, set s b)
```

Теперь можно инициализировать `FlipLens` тривиальной парой геттера и сеттера и передать их в наше профункторное представление:

```
fromLensP   :: LensP s t a b ->
                    (s -> a, s -> b -> t)
fromLensP pp = (get', set')
  where FlipLens get' set'
                = pp (FlipLens
                      id (\s b -> b))
```

18.3 Общая оптика

Модули Тамбары первоначально были определены для произвольной моноидальной категории[1] с тензорным произведением $\otimes$ и единичным объектом I. Их структурные отображения имеют вид:

$$\alpha_{\langle a,b\rangle,c} : P\langle a, b\rangle \to P\langle c \otimes a, c \otimes b\rangle$$

Вы можете легко убедиться, что все законы когерентности переводятся непосредственно в этот случай, и вывод профункторной оптики работает без изменений.

Призмы

С точки зрения программирования имеются две очевидные моноидальные структуры для изучения: произведение и сумма. Мы видели, что произведение порождает линзы. Сумма, или копроизведение, порождает призмы.

[1] На самом деле, модули Тамбары изначально были определены для категории, обогащенной над векторными пространствами.

Мы получаем экзистенциальное представление, просто заменяя произведение суммой в определении линзы:

$$\mathcal{P}\left\langle s,t\right\rangle \left\langle a,b\right\rangle =\int^{c}\mathcal{C}(s,c+a)\times\mathcal{C}(c+b,t)$$

Чтобы упростить это, заметим, что отображение-вне для суммы эквивалентно произведению отображений:

$$\int^{c}\mathcal{C}(s,c+a)\times\mathcal{C}(c+b,t)\cong\int^{c}\mathcal{C}(s,c+a)\times\mathcal{C}(c,t)\times\mathcal{C}(b,t)$$

С помощью леммы ко-Йонеды, можно избавиться от ко-конца, получая:

$$\mathcal{C}(s,t+a)\times\mathcal{C}(b,t)$$

На Haskell это определяет пару функций:

```
match :: s -> Either t a
build :: b -> t
```

Чтобы понять это, сначала переведем на Haskell экзистенциальную форму призмы:

```
data Prism s t a b where
  Prism :: (s -> Either c a) ->
           (Either c b -> t) ->
                    Prism s t a b
```

Здесь, `s` содержит, либо фокус `a`, либо остаток `c`. И наоборот, `t` можно построить, либо из нового фокуса `b`, либо из остатка `c`.

Эта логика отражена в функциях конвертации:

```
toMatch                  :: Prism s t a b ->
                            (s -> Either t a)
toMatch (Prism from to) s =
  case from s of
    Left  c -> Left (to (Left c))
    Right a -> Right a

toBuild                  :: Prism s t a b ->
```

```
                                          (b -> t)
toBuild (Prism from to) b = to (Right b)

toPrism            :: (s -> Either t a)  ->
                                (b -> t)  ->
                                   Prism s t a b
toPrism match build = Prism from to
  where
    from         = match
    to (Left c)  = c
    to (Right b) = build b
```

Профункторное представление призмы почти идентично представлению линзы, за исключением замены произведения на сумму.

Класс модулей Тамбары для типа суммы называется `Choice`, в библиотеке Haskell, или `Cocartesian`, в литературе:

```
class Profunctor p => Cocartesian p where
  alpha' :: p a b -> p (Either c a) (Either c b)
```

Профункторное представление является полиморфным функциональным типом:

```
type PrismP s t a b = forall p. Cocartesian p =>
                                  p a b -> p s t
```

Преобразование из экзистенциальной призмы практически идентично преобразованию из линзы:

```
toPrismP                :: Prism s t a b  ->
                                  PrismP s t a b
toPrismP (Prism from to) = dimap from to . alpha'
```

Опять же, профункторные призмы компонуются с использованием композиции функций.

Траверсали

Траверсаль — это тип оптики, который фокусируется сразу на нескольких фокусах. Например, имеется дерево, которое может иметь ноль или более листьев типа a. Траверсаль должен дать список этих узлов. Это также должно позволить заменить эти узлы новым списком. И вот проблема: длина списка, доставляющего замены, должна соответствовать количеству узлов, иначе возможны аварийные ситуации.

Типо-безопасная реализация траверсали потребовала бы отслеживания размеров списков. Другими словами, для этого потребуются зависимые типы.

На Haskell (небезопасная по типам) траверсаль часто записывается как:

```
type Traversal s t a b = s -> ([b] -> t, [a])
```

при понимании, что размеры двух списков определяются `s` и должны быть одинаковыми.

При переводе траверсалей на категорный язык будем выражать это условие, используя сумму по размерам списка. Подсчитываемый список *размера* n является n-кортежем или элементом a^n, поэтому можно записать:

$$\mathbf{Tr}\left\langle s,t\right\rangle \left\langle a,b\right\rangle =\mathbf{Set}\left(s,\sum_{n}(\mathbf{Set}(b^{n},t)\times a^{n})\right)$$

Мы интерпретируем траверсаль как функцию, которая по заданному источнику s создает экзистенциальный тип, скрывающий n. Она выражает существование n и пары, состоящая из функции $b^n \to t$ и n-кортежа a^n.

Экзистенциальная форма траверсаля должна учитывать тот факт, что вычеты для разных n будут иметь, в принципе, разные типы. Например, можно разложить дерево на n-кортеж листьев a^n и остаток c_n с n пустотами. Таким образом, правильное экзистенциальное представление для траверсаля должно включать ко-конец по всем последовательностям c_n, индексированных натуральными числами:

$$\mathbf{Tr}\left\langle s,t\right\rangle \left\langle a,b\right\rangle =\int^{c^{n}}\mathcal{C}\left(s,\sum_{m}c_{m}\times a^{m}\right)\times\mathcal{C}\left(\sum_{k}c_{k}\times b^{k},t\right)$$

Суммы здесь являются копроизведениями в $\mathcal{C}$.

Один из способов выразить последовательности c_n — интерпретировать их как расслоения. Например, в **Set** мы должны начать с множества C и проекции $p : C \to \mathbb{N}$, где $\mathbb{N}$ — множество натуральных чисел. Точно так же, a^n можно интерпретировать как расслоение свободного моноида на a (множество списков всех a) с проекцией, которая извлекает длину списка.

Или, можно рассматривать c_n как отображения от множества натуральных чисел к $\mathcal{C}$. Фактически, можно рассматривать натуральные числа как дискретную категорию $\mathcal{N}$, и в этом случае, c_n являются функторами $\mathcal{N} \to \mathcal{C}$.

$$\mathbf{Tr}\,\langle s, t\rangle\,\langle a, b\rangle = \int^{c:[\mathcal{N},\mathcal{C}]} \mathcal{C}\left(s, \sum_m c_m \times a^m\right) \times \mathcal{C}\left(\sum_k c_k \times b^k, t\right)$$

Чтобы показать эквивалентность этих двух представлений, перезапишем сначала отображение-вне суммы как произведение отображений:

$$\int^{c:[\mathcal{N},\mathcal{C}]} \mathcal{C}\left(s, \sum_m c_m \times a^m\right) \times \prod_k \mathcal{C}\left(c_k \times b^k, t\right)$$

и тогда воспользуемся каррированным сопряжением:

$$\int^{c:[\mathcal{N},\mathcal{C}]} \mathcal{C}\left(s, \sum_m c_m \times a^m\right) \times \prod_k \mathcal{C}\left(c_k, [b^k, t]\right)$$

Здесь, $[b^k, t]$ — внутренний hom, альтернативное обозначение экспоненциального объекта t^{b^k}.

Следующий шаг — признать, что произведение в этой формуле представляет собой множество естественных преобразований в $[\mathcal{N}, \mathcal{C}]$. Действительно, можно было бы записать его как конец:

$$\prod_k \mathcal{C}\left(c_k, [b^k, t]\right) \cong \int_{k:\mathcal{N}} \mathcal{C}\left(c_k, [b^k, t]\right)$$

Это потому, что конец над дискретной категорией — это просто произведение. В качестве альтернативы мы могли бы записать его как hom-множество в категории функторов:

$$[\mathcal{N}, \mathcal{C}](c_\_, [b^-, t])$$

с заполнителями, заменяющими аргументы двух рассматриваемых функторов:

$$k \mapsto c_k$$
$$k \mapsto [b^k, t]$$

Теперь можно использовать лемму ко-Йонеды в категории функторов $[\mathcal{N}, \mathcal{C}]$:

$$\int^{c:[\mathcal{N},\mathcal{C}]} \mathcal{C}\left(s, \sum_m c_m \times a^m\right) \times [\mathcal{N}, \mathcal{C}](c_\_, [b^-, t]) \cong \mathcal{C}\left(s, \sum_m [b^m, t] \times a^m\right)$$

Этот результат является более общим, чем наша исходная формула, но превращается в нее при ограничении категорией множеств.

Чтобы получить профункторное представление для траверсалей, мы должны более внимательно изучить тип задействованных преобразований. Определим действие функтора $c : [\mathcal{N}, \mathcal{C}]$ на a как:

$$c \bullet a = \sum_m c_m \times a^m$$

Эти действия могут быть скомпонованы, расширением этой формулы, используя законоы дистрибутивности:

$$c \bullet (c' \bullet a) = \sum_m c_m \times (\sum_n c'_n \times a^n)^m$$

Если целевой категорией является **Set**, это эквивалентно следующей D-свертке (для категорий, отличных от **Set**, можно использовать расширенную версию D-свертки):

$$(c \star c')_k = \int^{m,n} \mathcal{N}(m+n, k) \times c_m \times c'_n$$

Это наделяет категорию $[\mathcal{N}, \mathcal{C}]$ моноидальной структурой.

Экзистенциальное представление траверсалей можно записать в терминах действия этой моноидальной категории на $\mathcal{C}$:

$$\mathbf{Tr}\, \langle s, t\rangle\, \langle a, b\rangle = \int^{c:[\mathcal{N},\mathcal{C}]} \mathcal{C}(s, c \bullet a) \times \mathcal{C}(c \bullet b, t)$$

Чтобы вывести профункторное представление траверсалей, мы должны обобщить модули Тамбары на действие моноидальной категории:

$$\alpha_{\langle a,b\rangle,c} : P\,\langle a,b\rangle \to P\,\langle c \bullet a, c \bullet b\rangle$$

Оказывается, что первоначальный вывод профункторной оптики работает и для этих обобщенных модулей Тамбары, и траверсали могут быть записаны как полиморфные функции:

$$\mathbf{Tr}\,\langle s,t\rangle\,\langle a,b\rangle = \int_{P:\mathcal{T}} \mathbf{Set}((UP)\,\langle a,b\rangle\,,(UP)\,\langle s,t\rangle)$$

где конец берется над обобщенным модулем Тамбары.

18.4 Смешанная оптика

Всякий раз, когда имеется некоторое действие моноидальной категории $\mathcal{M}$ на $\mathcal{C}$, можно определить соответствующую оптику. Категория с таким действием называется *актегорией*. Можно пойти еще дальше, рассмотрев два отдельных действия. Предположим, что $\mathcal{M}$ может действовать, как на $\mathcal{C}$, так и на $\mathcal{D}$. Мы будем использовать одно и то же обозначение для обоих действий:

$$\bullet : \mathcal{M} \times \mathcal{C} \to \mathcal{C}$$
$$\bullet : \mathcal{M} \times \mathcal{D} \to \mathcal{D}$$

Тогда можно определить *смешанную оптику* как:

$$\mathcal{O}\,\langle s,t\rangle\,\langle a,b\rangle = \int^{m:\mathcal{M}} \mathcal{C}(s, m \bullet a) \times \mathcal{D}(m \bullet b, t)$$

Эта смешанная оптика имеет профункторные представления в терминах профункторов:

$$P : \mathcal{C}^{\mathsf{op}} \times \mathcal{D} \to \mathbf{Set}$$

и соответствующих модулей Тамбары, которые используют два отдельных действия:

$$\alpha_{\langle a,b\rangle,m} : P\,\langle a,b\rangle \to P\,\langle m \bullet a, m \bullet b\rangle$$

с a объектом из $\mathcal{C}$, b объектом из $\mathcal{D}$ и m объектом из $\mathcal{M}$.

Упражнение 18.4.1. *Что представляет собой смешанная оптика для действия декартова произведения, когда одна из категорий является терминальной категорией? Что, если первая категория — это* $\mathcal{C}^{\mathsf{op}} \times \mathcal{C}$, *а вторая — терминальная категория?*

Глава 19

Расширения Кана

Если теория категорий продолжает повышать уровень абстракции, то это потому, что она направлена на обнаружение шаблонов. Как только шаблоны обнаруживаются, наступает время изучения шаблонов, образованных этими шаблонами, и т.д.

Мы уже видели, как одни и те же повторяющиеся концепции описывались все более и более лаконично на все более и более высоких уровнях абстракции.

Например, сначала мы определили произведение с помощью универсальной конструкции. Затем увидели, что пролеты в определении произведения оказались естественными преобразованиями. Это привело к интерпретации произведения как предела. Затем стало понятно, как можно его определить с помощью сопряжений. Далее, нам удалось объединить его с ко-произведением в одну компактную формулу:

$$(+) \dashv \Delta \dashv (\times)$$

Лао-цзы сказал: «Если вы хотите что-то уменьшить, вы должны сначала позволить этому расшириться».

Расширения Кана поднимают уровень абстракции еще выше. МакЛейн даже высказал предположение: «Все понятия являются расширениями Кана».

19.1 Замкнутые моноидальные категории

Мы знаем, что функциональный объект может быть определен как правый сопряженный к категорному произведению:

$$\mathcal{C}(a \times b, c) \cong \mathcal{C}(a, [b, c])$$

Здесь использовано альтернативное обозначение $[b, c]$ для внутреннего $\hom$ — экспоненциала c^b.

Сопряжение между двумя функторами можно рассматривать как псевдоинверсию одного из них. Они не компонуются к тождественности, но их композиция связана с тождественным функтором через единицу и ко-единицу. Например, если вы достаточно тщательно подготовились к неожиданностям, то ко-единица каррирующего сопряжения:

$$\varepsilon_{b,c} : [b, c] \times b \to c$$

предполагает, что $[b, c]$ воплощает в некотором смысле обратную операцию умножения — играет ту же роль, что и c/b в:

$$c/b \times b = c$$

В типично категорной манере можно задать вопрос: что, если заменить произведение чем-то другим? Очевидное действие, заменяющее его копроизведением, не работает (таким образом, у нас нет аналога вычитания). Но, возможно, существуют и другие хорошо работающие бинарные операции с правым сопряжением.

Естественной установкой для обобщения произведения является моноидальная категория с тензорным произведением $\otimes$ и единичным объектом I. Если имеется сопряжение

$$\mathcal{C}(a \otimes b, c) \cong \mathcal{C}(a, [b, c])\,,$$

будем называть эту категорию *замкнутой моноидальной*. При типичном категорном злоупотреблении обозначениями, если это не приводит к путанице, мы будем использовать тот же символ (пару квадратных скобок) для моноидального внутреннего $\hom$, как и для декартового $\hom$. Существует альтернативное обозначение для правого сопряжения с тензорным произведением:

$$\mathcal{C}(a \otimes b, c) \cong \mathcal{C}(a, b \multimap c)$$

Оно часто используется в контексте линейных типов.

Определение этого внутреннего hom хорошо работает для симметричной моноидальной категории. Если тензорное произведение несимметрично, сопряжение определяет *замкнутую слева* моноидальную категорию. Левый внутренний hom есть сопряжение к функтору «пост-умножения» $(- \otimes b)$. Право-замкнутая структура определяется как правая сопряженная к функтору «пред-умножения» $(b \otimes -)$. Если оба определены, то эта категория называется *би-замкнутой*.

Внутренний hom для D-свертки

В качестве примера рассмотрим симметричную моноидальную структуру в категории ко-предпучков с D-сверткой:

$$(F \star G)x = \int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb$$

Наблюдаем сопряжение:

$$[\mathcal{C}, \mathbf{Set}](F \star G, H) \cong [\mathcal{C}, \mathbf{Set}](F, [G, H]_{D\text{-}conv})$$

Естественное преобразование в левой части можно записать в виде конца:

$$\int_x \mathbf{Set}\left(\int^{a,b} \mathcal{C}(a \otimes b, x) \times Fa \times Gb, Hx\right)$$

Используем ко-непрерывность, для вытягивания ко-концов на верхний уровень:

$$\int_{x,a,b} \mathbf{Set}(\mathcal{C}(a \otimes b, x) \times Fa \times Gb, Hx)$$

Затем можно использовать каррированное сопряжение в **Set** (квадратные скобки обозначают внутренний hom в **Set**):

$$\int_{x,a,b} \mathbf{Set}(Fa, [\mathcal{C}(a \otimes b, x) \times Gb, Hx])$$

Наконец, используем непрерывность hom-множества для перемещения двух концов внутрь hom-множества:

$$\int_a \mathbf{Set}\left(Fa, \int_{x,b} [\mathcal{C}(a \otimes b, x) \times Gb, Hx]\right)$$

Получаем, что правый сопряженный к D-свертке задается выражением:

$$([G,H]_{D\text{-}conv})a = \int_{x,y} [\mathcal{C}(a \otimes x, y), [Gx.Hy]] \cong \int_x [Gx, H(a \otimes x)]$$

Последнее преобразование — это применение леммы Йонеды к **Set**.

Упражнение 19.1.1. *Реализуйте внутренний* hom *для D-свертки в Haskell. Подсказка*: *используйте псевдоним типа.*

Возведение в степень и возведение в ко-степень

В категории множеств, внутренний hom (функциональный объект или экспоненциал) изоморфен внешнему hom (множеству морфизмов между двумя объектами):

$$\mathcal{C}^{\mathcal{B}} \cong Set(\mathcal{B}, \mathcal{C})$$

Таким образом, можно переписать каррированное сопряжение, определяющее внутренний hom в **Set**, следующим образом:

$$\mathbf{Set}(A \times B, C) \cong \mathbf{Set}(A, \mathbf{Set}(BC))$$

Можно обобщить это сопряжение на случай, когда B и C не множества, а объекты в некоторой категории $\mathcal{C}$. Внешний hom в любой категории всегда является множеством. Но левая сторона, это уже не произведение. Вместо этого определим действие множества A на объект b:

$$\mathcal{C}(A \cdot b, c) \cong \mathbf{Set}(A, \mathcal{C}(b, c))$$

также известное как *ко-степень*.

Можно думать об этом действии как о суммировании (извлечении копроизведения) A копий b. Например, если A — двухэлементное множество $\mathbf{2}$, то получаем:

$$\mathcal{C}(\mathbf{2} \cdot b, c) \cong \mathbf{Set}(\mathbf{2}, \mathcal{C}(b, c)) \cong \mathcal{C}(b, c) \times \mathcal{C}(b, c) \cong \mathcal{C}(b + b, c)$$

Другими словами,

$$\mathbf{2} \cdot b \simeq b + b$$

В этом смысле ко-степень определяет умножение в терминах повторяющегося сложения, известное еще со школы.

Если умножить b на hom-множество $\mathcal{C}(b,c)$ и возять ко-конец над всеми b, то результат будет изоморфен c:

$$\int^b \mathcal{C}(b,c) \cdot b \cong c$$

В самом деле, отображения к произвольному x с обеих сторон изоморфны в силу леммы Йонеды:

$$\mathcal{C}\left(\int^b \mathcal{C}(b,c) \cdot b, x\right) \cong \int_b \mathbf{Set}(\mathcal{C}(b,c), \mathcal{C}(b,x)) \cong \mathcal{C}(c,x)$$

Как и можно было ожидать, в **Set**, ко-степень распадается до декартова произведения.

$$\mathbf{Set}(A \cdot B, C) \cong \mathbf{Set}(A, \mathbf{Set}(B,C)) \cong \mathbf{Set}(A \times B, C)$$

Точно так же, можно выразить возведение в степень как многократное умножение. Мы используем ту же правую часть, но на этот раз — отображение-внутрь, для определения *степени*:

$$\mathcal{C}(b, A \pitchfork c) \cong \mathbf{Set}(A, \mathcal{C}(b,c))$$

Можно думать о степени как о перемножении A копий c. Действительно, замена A на $\mathbf{2}$ приводит к:

$$\mathcal{C}(b, 2 \pitchfork c) \cong \mathbf{Set}(\mathbf{2}, \mathcal{C}(b,c)) \cong \mathcal{C}(b,c) \times \mathcal{C}(b,c) \cong \mathcal{C}(bc \times c)$$

Другими словами:

$$2 \pitchfork c \simeq c \times c$$

что позволяет использовать обозначение c^2.

Если мы возводим c в степень hom-множества $\mathcal{C}(c',c)$ и используем этот конец над всеми c, то результат будет изоморфен c':

$$\int_c \mathcal{C}(c',c) \pitchfork c \cong c'$$

Это следует из леммы Йонеды. Действительно, отображения от любого x, по обоим сторонам, изоморфны:

$$\mathcal{C}\left(x, \int_c \mathcal{C}(c',c) \pitchfork c\right) \cong \int_c \mathbf{Set}(\mathcal{C}(c',c), \mathcal{C}(x,c)) \cong \mathcal{C}(x,c')$$

В **Set**, степень убывает до экспоненциальной, что совпадает с hom-множеством:

$$A \pitchfork C \cong C^A \cong \mathbf{Set}(A, C)$$

Это есть следствие симметрии произведения.

$$\mathbf{Set}(B, A \pitchfork C) \cong \mathbf{Set}(A, \mathbf{Set}(B, C)) \cong \mathbf{Set}(A \times B, C) \cong$$
$$\mathbf{Set}(B \times A, C) \cong \mathbf{Set}(B, \mathbf{Set}(A, C))$$

19.2 Обращение функтора

Один аспект преобразований (с потерями) из теории категорий связан с отбрасыванием некоторой информации, другой — позволяет восстанавливать «потерянную» информацию. Мы уже видели примеры восполнения потерянных данных с помощью свободных функторов — сопряжений к забывающим функторам. Расширения Кана предоставляют еще одну группу примеров в этом качестве. Обе возможности позволяют восстановить данные, потерянные функтором, что необратимо.

Правда, имеются две причины, по которым действие функтора может быть необратимым. Во-первых, он может отображать несколько объектов или стрелок в один объект или стрелку. Другими словами, он не является инъективным для объектов или стрелок. Другая причина заключается в том, что его образ может не охватывать всю целевую категорию.

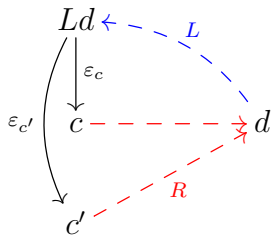
Рассмотрим, например, сопряжение $L \dashv R$. Предположим, что R не является инъективным и сводит два объекта, c и c', к одному объекту d:

$$Rc = d$$
$$Rc' = d$$

L не имеет шансов отменить это. В то же время, он не может отобразить d одновременно к c и c'. Лучшее, что он может сделать, — это отобразить d к «более общему» объекту Ld, от которого идут стрелки, как к c, так и к c'. Эти стрелки необходимы для определения компонентов ко-единицы сопряжения:

$$\varepsilon_c : Ld \to c$$
$$\varepsilon_{c'} : Ld \to c'$$

где Ld является, как $L(Rc)$, так и $L(Rc')$

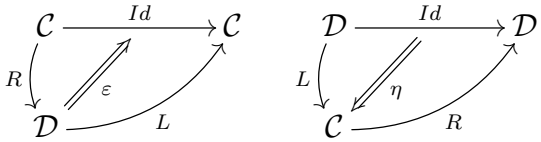


Более того, если R не является сюръективным на объектах, функтор L должен быть каким-то образом определен на тех объектах $\mathcal{D}$, которые не входят в образ R. Опять же, естественность единицы и ко-единицы будет ограничивать возможный выбор, пока имеются стрелки, соединяющие эти объекты с образом R.

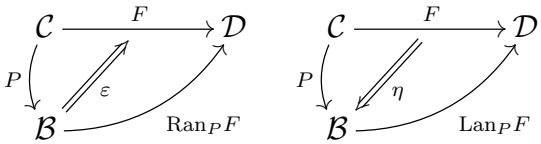
Очевидно, все эти ограничения означают, что сопряжение может быть определено только в очень редких случаях. А расширения Кана еще слабее, чем сопряжения.

Если сопряженные функторы работают как обратные, то расширения Кана ведут себя как дроби.

Это лучше всего видно, если обновить диаграммы, определяющие ко-единицу и единицу сопряжения. На первой диаграмме, L, кажется, играет роль $1/R$. На второй диаграмме, R разыгрывает из себя $1/L$.



Правое расширение Кана $\mathrm{Ran}_P F$ и левое расширение Кана $\mathrm{Lan}_P F$ обобщают их, заменяя тождественный функтор некоторым функтором $F : C \to D$. Кроме этого, расширения Кана играют роль дробей F/P. Концептуально, они отменяют действие P и следуют за этим с действием F.



Как и в случае с сопряжениями, «отмена» не является полной. Композиция $\mathrm{Ran}_P F \circ P$ не воспроизводит F; вместо этого, она связана с ним

через естественное преобразование ε, называемое ко-единицей. Аналогично, композиция $\mathrm{Lan}_P F \circ P$ связана с F через единицу η.

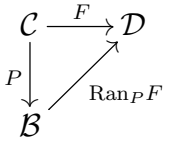
Заметим, что чем больше информации F отбрасывает, тем легче расширениям Кана «инвертировать» функтор P. В этом смысле он должен инвертировать P только «по модулю F».

Интуиция, стоящая за расширениями Кана, сводится к следующему. Начнем с функтора F:

$$\mathcal{C} \xrightarrow{F} \mathcal{D}$$

Существует второй функтор P, который вкладывает категорию $\mathcal{C}$ в другую категорию $\mathcal{B}$. Это вложение может быть с потерями и не-сюръективным. Наша задача — *распространить* определение F на всю категорию $\mathcal{B}$.

В идеальном случае хотелось бы, чтобы следующая диаграмма была коммутативной:



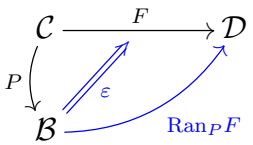
Но это потребовало бы равенства функторов, чего мы пытаемся избежать любой ценой.

Следующим лучшим решением было бы требование наличия естественного изоморфизма между двумя путями на этой диаграмме. Но даже это кажется слишком многовато. И потому, мы, наконец, останавливаемся на требовании, чтобы один путь был деформируемым в другой, что означает существование между ними одностороннего естественного преобразования. Направление этого преобразования различает правое и левое расширения Кана.

19.3 Правое расширение Кана

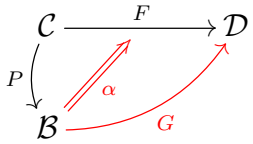
Правое расширение Кана — это функтор $\mathrm{Ran}_P F$, оснащенный естественным преобразованием ε, называемым ко-единицей расширения Кана, определяемого как:

$$\varepsilon : \mathrm{Ran}_P F \circ P \to F$$

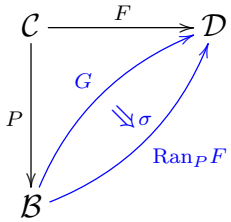


Пара $(\mathrm{Ran}_P F, \varepsilon)$ универсальна среди пар (G, α), где G — функтор $G : \mathcal{B} \to \mathcal{D}$, а α — естественное преобразование:

$$\alpha : G \circ P \to F$$



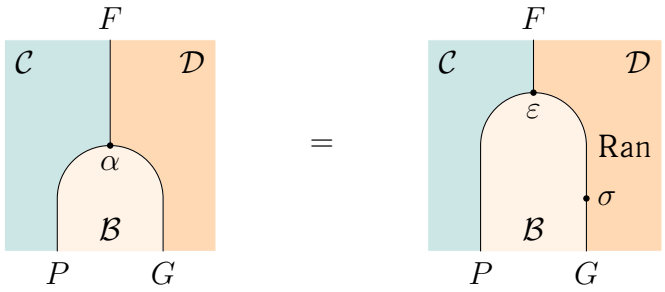
Универсальность означает, что для любого такого (G, α) существует единственное естественное преобразование $\sigma : G \to \mathrm{Ran}_P F$



факторизующее α, то есть:

$$\alpha = \varepsilon \cdot (\sigma \circ P)$$

Заметим, что это есть комбинация вертикальных и горизонтальных композиций естественных преобразований, в которой $\sigma \circ P$ является вискерингом для σ. То же уравнение, изображенное на струнной диаграмме, имеет вид:



По-прежнему, универсальную конструкцию можно обобщить до сопряжения — на этот раз, сопряжения между категориями функторов:

$$[\mathcal{C}, \mathcal{D}](G \circ P, F) \cong [\mathcal{B}, \mathcal{D}](G, \mathrm{Ran}_P F)$$

Для каждого естественного преобразования α, которое является элементом левой части, существует единственное преобразование σ, являющееся элементом правой части.

Другими словами, правое расширение Кана, если оно существует для каждого F, является правым сопряженным к функторной предкомпозиции:

$$(- \circ P) \dashv \mathrm{Ran}_P$$

Компонента ко-единицы этого сопряжения при F есть ε.

Это чем-то напоминает каррированное сопряжение:

$$\mathcal{C}(a \times b, c) \cong \mathcal{C}(a, [b, c])$$

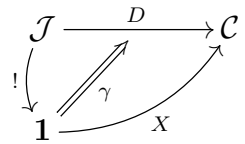
в котором произведение заменяется функторной композицией (конечно, композицию можно считать тензорным произведением только в категории эндофункторов).

Пределы как расширения Кана

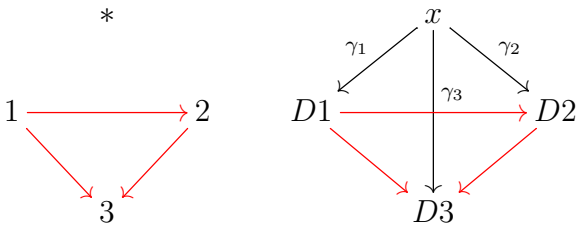
Выше мы определили пределы как универсальные конусы. Определение конуса включает две категории: индексную категорию $\mathcal{J}$, которая определяет форму диаграммы, и целевую категорию $\mathcal{C}$. Диаграмма — это функтор $D : \mathcal{J} \to \mathcal{C}$, который встраивает эту форму в целевую категорию.

Можно ввести третью категорию $\mathbf{1}$: терминальную категорию, содержащую один объект и одну тождественную стрелку. Затем можно использовать функтор X от этой категории к выбранной вершине x конуса в $\mathcal{C}$. Поскольку $\mathbf{1}$ является терминальной категорией в $\mathbf{Cat}$, также имеем единственный функтор от $\mathcal{J}$ к ней, обычным злоупотреблением обозначениями, который обозначим симолом «!».

Оказывается, что пределом D является правое расширение Кана диаграммы D вдоль !. Прежде всего, заметим, что композиция $X \circ !$ отображает форму $\mathcal{J}$ к единственному объекту x, поэтому она выполняет работу постоянного функтора Δ_x. Этим выбирается вершина конуса. Конус с вершиной x является естественным преобразованием γ:

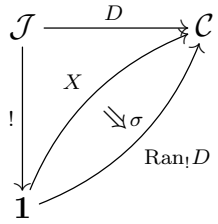


Следующие диаграммы иллюстрируют это. Слева имеются две категории: $\mathbf{1}$ с одним объектом $*$, и $\mathcal{J}$ с тремя объектами, формирующими форму диаграммы. Справа расположен образ D и образ $X \circ !$, который является вершиной x. Три компонента γ соединяют вершину x с диаграммой. Естественность γ заключается в том, что треугольники, образующие стороны конуса, являются коммутативными.



Правое расширение Кана $(\mathrm{Ran}_! D, \varepsilon)$ является таким универсальным конусом. $\mathrm{Ran}_! D$ является функтором от $\mathbf{1}$ к $\mathcal{C}$, поэтому он выбирает объект в $\mathcal{C}$ — вершину $\mathrm{Lim} D$ универсального конуса.

Универсальность означает, что для любой пары (X, γ) существует естественное преобразование $\sigma : X \to \mathrm{Ran}_! D$



факторизующее γ.

σ имеет только одну компоненту σ_*, которая представляет собой стрелку h, соединяющую вершину x с вершиной $\mathrm{Lim} D$. Факторизация:

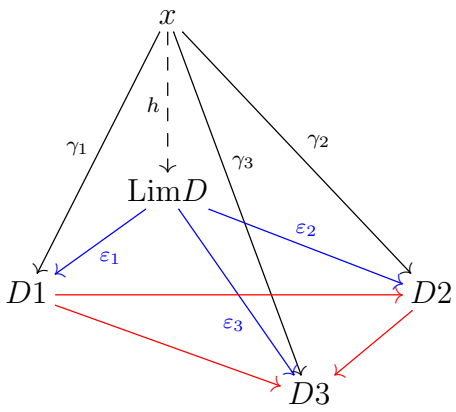
$$\gamma = \varepsilon \cdot (\sigma \circ !)$$

в компонентах выглядит как:

$$\gamma_i = \varepsilon_i \circ h$$

Тогда треугольники на следующей диаграмме являются коммутативны-

ми:



Это условие универсальности делает $\mathrm{Lim}D$ пределом диаграммы D.

Правое расширение Кана как конец

Напомним лемму ниндзя Йонеды:

$$Fb \cong \int_c \mathbf{Set}(\mathcal{B}(b,c), Fc)$$

Здесь, F — ко-предпучок, то есть функтор от $\mathcal{B}$ к $\mathbf{Set}$. Правые расширения Кана F вдоль P обобщают эту формулу:

$$(\mathrm{Ran}_P F)b \cong \int_c \mathbf{Set}(\mathcal{B}(b,Pc), Fc)$$

Это работает для ко-предпучка. В общем случае нас интересует $F : \mathcal{C} \to \mathcal{D}$, поэтому нужно заменить hom-множество в $\mathbf{Set}$ степенью. Таким образом, правое расширение Кана задается следующим концом (конечно, если он существует):

$$(\mathrm{Ran}_P F)b \cong \int_c \mathcal{B}(b,Pc) \pitchfork Fc$$

Доказательство, по сути, создается просто: на каждом шаге нужно сделать только что-то одно. Начнем с сопряжения:

$$[\mathcal{C},\mathcal{D}](G \circ P, F) \cong [\mathcal{B},\mathcal{D}](G, \mathrm{Ran}_P F)$$

и перепишем его, используя концы:

$$\int_c \mathcal{D}(G(Pc), Fc) \cong \int_b \mathcal{D}(Gb, (\mathrm{Ran}_P F)b)$$

Подставляем нашу формулу, получая:

$$\cong \int_b \mathcal{D}\left(Gb, \int_c \mathcal{B}(b, Pc) \pitchfork Fc\right)$$

Используем непрерывность hom-функтора, чтобы сместить конец вперед:

$$\cong \int_b \int_c \mathcal{D}(Gb, \mathcal{B}(b, Pc) \pitchfork Fc)$$

Затем используем определение степени:

$$\int_b \int_c \mathbf{Set}(\mathcal{B}(b, Pc), \mathcal{D}(Gb, Fc))$$

и применяем лемму Йонеды:

$$\int_c \mathcal{D}(G(Pc), Fc)$$

Полученный результат действительно является левой частью сопряжения.

Если F — ко-предпучок, то мощность в формуле для правого расширения Кана убывает до экспоненциала/hom-множества:

$$(\mathrm{Ran}_P F)b \cong \int_c \mathbf{Set}(\mathcal{B}(b, Pc), Fc)$$

Это можно сразу перевести на Haskell:

```
newtype Ran p f b = Ran (forall c.
                           (b -> p c) -> f c)
```

Заметим также, что если у P имеется левый сопряженный, обозначим его P^{-1}, то есть:

$$\mathcal{B}(b, Pc) \cong \mathcal{C}(P^{-1}b, c)$$

то мы могли бы использовать лемму ниндзя Йонеды, чтобы выполнить конец в

$$(\mathrm{Ran}_P F)b \cong \int_c \mathbf{Set}(\mathcal{B}(b, Pc), Fc) \cong \int_c \mathbf{Set}(\mathcal{C}(P^{-1}b, c), Fc)$$

и получить:

$$(\mathrm{Ran}_P F)b \cong (F \circ P^{-1})b$$

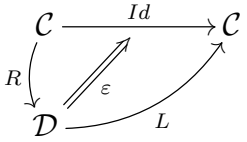
Поскольку сопряжение является ослаблением идеи инверсии, этот результат согласуется с интуитивным предположением, что расширение Кана *инвертирует* P и следует за ним с F.

Левый сопряженный как правое расширение Кана

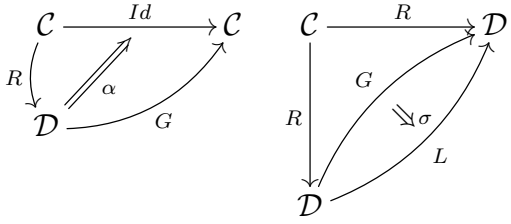
Мы начали с описания расширений Кана как обобщения сопряжений. Глядя на изображения диаграмм, если имеется пара сопряженных функторов $L \dashv R$, мы ожидаем, что левый функтор будет правым расширением Кана тождественности вдоль правого функтора.

$$L \cong \mathrm{Ran}_R Id$$

В самом деле, ко-единица расширения Кана та же, что и ко-единица сопряжения:



Также, надо показать универсальность:



Для этого в нашем распоряжении имеется единица сопряжения:

$$\eta : Id \to R \circ L$$

Построим σкак композицию:

$$G \to G \circ Id \xrightarrow{G \circ \eta} G \circ R \circ L \xrightarrow{\alpha \circ L} Id \circ L \to L$$

Другими словами, определим σ как:

$$\sigma = (\alpha \circ L) \cdot (G \circ \eta)$$

Мы могли бы задать обратный вопрос: если $\mathrm{Ran}_R Id$ существует, является ли он автоматически левым сопряженным к R? Оказывается, для этого нужно еще одно условие: R должно сохранять расширение Кана, то есть:

$$R \circ \mathrm{Ran}_R Id \cong \mathrm{Ran}_R R$$

В следующем разделе мы увидим, что правая часть этого условия определяет ко-плотную монаду.

Упражнение 19.3.1. *Покажите условие факторизации*:

$$\alpha = \varepsilon \cdot (\sigma \circ R)$$

для σ, который был определен выше. Подсказка: *изобразите соответствующие струнные диаграммы и используйте тождество треугольника для сопряжения.*

Ко-плотная монада

Мы видели, что каждое сопряжение $L \dashv F$ порождает монаду $F \circ L$. Оказывается, эта монада является правым расширением Кана F вдоль F. Интересно, что даже если F не имеет левого сопряженного, расширение Кана $\mathrm{Ran}_F F$ по-прежнему является монадой, называемой *ко-плотной монадой*и обозначаемой T^F:

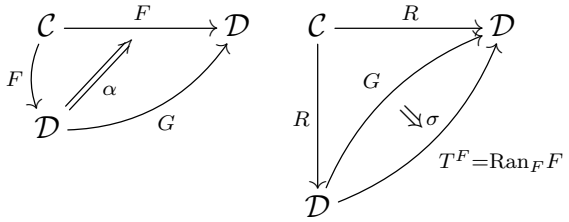
$$T^F = \mathrm{Ran}_F F$$

Если бы мы серьезно относились к интерпретации расширений Кана как дробей, ко-плотная монада соответствовала бы F/F. Функтор, для которого такая «дробь» равна тождественности, называется *ко-плотным*.

Чтобы убедиться, что T^F является монадой, мы должны определить монадическую единицу и умножение:

$$\eta : Id \to T^F$$
$$\mu : T^F \circ T^F \to T^F$$

Оба следуют из универсальности. Для каждой (G, α) имеем σ:



Чтобы получить единицу, заменим G тождественным функтором Id, а α — тождественным естественным преобразованием.

Для получения умножения, заменим G на $T^F \circ T^F$, и заметим, что в нашем распоряжении имеется ко-единица расширения Кана:

$$\varepsilon : T^F \circ F \to F$$

Мы можем определить соответствующее α:

$$\alpha : T^F \circ T^F \circ F \to F$$

как композицию:

$$T^F \circ T^F \circ F \xrightarrow{\mathsf{id} \circ \varepsilon} T^F \circ F \xrightarrow{\varepsilon} F$$

или, используя нотацию вискеринга:

$$\alpha = \varepsilon \cdot (T^F \circ \varepsilon)$$

Соответствующее σ дает нам монадическое умножение.

Теперь покажем, что если мы начнем с сопряжения:

$$\mathcal{C}(Ld, c) \cong \mathcal{D}(d, Fc)$$

то $F \circ L$ даст ко-плотную монаду. Начнем с отображения в $F \circ L$ от произвольного функтора G:

$$[\mathcal{D}, \mathcal{D}](G, F \circ L) \cong \int_d \mathcal{D}(Gd, F(Ld))$$

Это можно переписать, используя лемму Йонеды:

$$\cong \int_d \int_c \mathbf{Set}(\mathcal{C}(Ld, c), \mathcal{D}(Gd, Fc))$$

Здесь замена конца на c приводит к замене c на Ld. Теперь можно использовать сопряжение:

$$\cong \int_d \int_c \mathbf{Set}(\mathcal{D}(d, Fc), \mathcal{D}(Gd, Fc))$$

и выполнить интегрирование ниндзя-Йонеды над d:

$$\cong \int_c \mathcal{D}(G(Fc), Fc)$$

Это, в свою очередь, определяет множество естественных преобразований:

$$\cong [\mathcal{C}, \mathcal{D}](G \circ F, F)$$

Пред-композиция с помощью F является левой сопряженной к правому расширению Кана:

$$[\mathcal{C}, \mathcal{D}](G \circ F, F) \cong [\mathcal{D}, \mathcal{D}](G, \mathrm{Ran}_F F)$$

Это показывает, что $F \circ L$ действительно является ко-плотной монадой для F.

Поскольку каждая монада может быть получена из сопряжения, то отсюда следует, что каждая монада является ко-плотной монадой для этого сопряжения.

Переводя ко-плотную монаду на Haskell, получаем:

```
type Codensity f a = forall c. (a -> f c) -> f c
```

Это очень похоже на монаду продолжения. На самом деле, она превращается в монаду продолжения, если мы выбираем `f` в качестве тождественного функтора. Можно думать о `Codensity` как об обратном вызове `(a -> f c)`, и вызове его, когда становится доступным результат типа `a`.

Вот экземпляр монады:

```
instance Monad (Codensity f) where
  return x = \k -> k x
  m >>= kl = \k -> m (\a -> (kl a) k)
```

Опять же, это почти то же самое, что и монада продолжения:

```
instance Monad (Cont r) where
  return x = Cont (\k -> k x)
  m >>= kl = Cont (\k -> runCont m (\a ->
                              runCont (kl a) k))
```

Вот почему `Codensity` имеет преимущества в производительности, что и стиль передачи продолжения. Поскольку он вкладывает продолжения «наизнанку», его можно использовать для оптимизации длинных цепочек привязок, создаваемых блоками `do` ....

Это свойство особенно важно при работе со свободными монадами, которые накапливают привязки в древовидных структурах. Когда мы, наконец, интерпретируем свободную монаду, эти накопленные привязки требуют обхода постоянно растущего дерева. Для каждой привязки обход начинается с корня. Сравните это с более ранним примером обращения списка, который был оптимизирован путем накопления функций в очереди FIFO. Ко-плотная монада осуществляет такое же улучшение производительности.

19.4 Левое расширение Кана

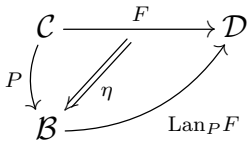
Подобно тому, как правое расширение Кана было определено как правый сопряженный к функторной пред-композиции, левое расширение Кана определяется как левый сопряженный к функторной пред-композиции:

$$[\mathcal{B}, \mathcal{D}](\mathrm{Lan}_P F, G) \cong [\mathcal{C}, \mathcal{D}](F, G \circ P)$$

(существуют и сопряжения к пост-композиции: они называются подъемами Кана).

Альтернативно, $\mathrm{Lan}_P F$ может быть определен как функтор, оснащенный естественным преобразованием, называемым единицей:

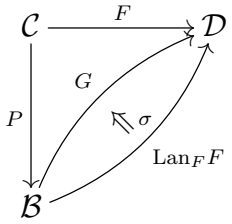
$$\eta : F \to \mathrm{Lan}_P F \circ P$$



Пара $(\mathrm{Lan}_P F, \eta)$ является универсальной, означающее, что для любой другой пары (G, α), где

$$\alpha : F \to G \circ P$$

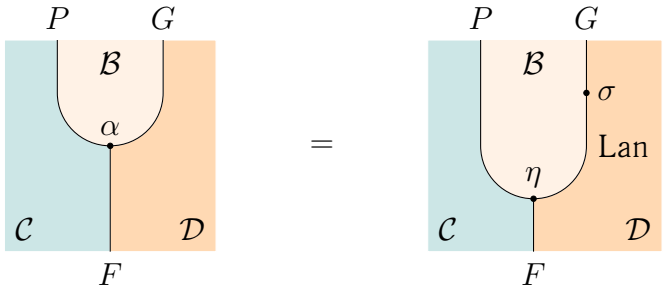
существует единственное отображение $\sigma : \mathrm{Lan}_P F \to G$



которое факторизует α:

$$\alpha = (\sigma \circ P) \cdot \eta$$

или, используя струнные диаграммы:



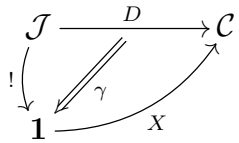
Это устанавливает взаимно однозначное отображение между множествами естественных преобразований. Для каждого α слева существует единственное σ справа:

$$[\mathcal{C}, \mathcal{D}](F, G \circ P) \cong [\mathcal{B}, \mathcal{D}](\mathrm{Lan}_P F, G)$$

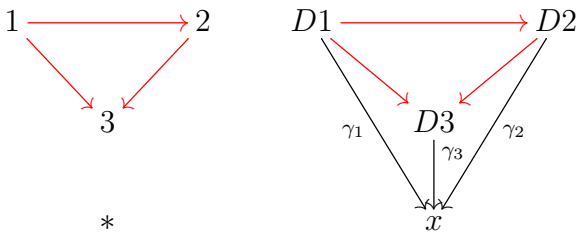
Копределы как расширения Кана

Точно так же, как пределы могут быть определены как правые расширения Кана, так и копределы могут быть определены как левые расширения Кана. Начнем с индексной категории $\mathcal{J}$, которая определяет форму

копредела. Функтор D выбирает эту форму в целевой категории $\mathcal{C}$. Вершина ко-конуса выбирается функтором от терминальной одно-объектной категории $\mathbf{1}$. Естественное преобразование определяет ко-конус от D к X:



Приведем наглядный пример простой формы, состоящей из трех объектов и трех морфизмов (не считая тождественностей). Объект x является образом единственного объекта $*$ под функтором X:



Копредел — это универсальный ко-конус, который задается левым расширением Кана:

$$\mathrm{Colim} D = \mathrm{Lan}_! D$$

Левое расширение Кана как ко-конец

Напомним лемму ниндзя ко-Йонеды. Для каждого ко-предпучка F, имеем:

$$Fb \cong \int^c \mathcal{B}(c, b) \times Fc$$

Левое расширение Кана обобщает эту формулу:

$$(\mathrm{Lan}_P F)b \cong \int^c \mathcal{B}(Pc, b) \times Fc$$

Для общего функтора $F : C \to D$, заменим произведение на ко-степень:

$$(\mathrm{Lan}_P F)b \cong \int^c \mathcal{B}(Pc, b) \cdot Fc$$

Докажем эту формулу, рассмотрев отображение-вне к некоторому функтору G. Представим множество естественных преобразований как конец:

$$\int_b \mathcal{D}\left(\int^c \mathcal{B}(Pc, b) \cdot Fc, Gb\right)$$

Выведем на внешний уровень ко-конец, который превращается в конец:

$$\int_b \int_c \mathcal{D}(\mathcal{B}Pc, b) \cdot Fc, Gb$$

и подставим в определение ко-степени:

$$\int_b \int_c \mathcal{D}((\mathcal{B}Pc, b), \mathcal{D}(Fc, Gb))$$

Теперь можно использовать лемму Йонеды для интегрирования по b, заменив b на Pc:

$$\int_c \mathcal{D}(Fc, G(Pc)) \cong [\mathcal{C}, \mathcal{D}](F, G \circ P)$$

Пока рассматриваемый ко-конец существует, он действительно дает левый сопряженный к функторной пред-композиции:

$$[\mathcal{B}, \mathcal{D}](\mathrm{Lan}_P F, G) \cong [\mathcal{C}, \mathcal{D}](F, G \circ P)$$

Как и ожидалось, в **Set**, ко-степень распадается до декартова произведения:

$$(\mathrm{Lan}_P F)b \cong \int^c \mathcal{B}(Pc, b) \times Fc$$

При переводе этой формулы на Haskell мы заменяем ко-конец на экзистенциальный тип. Символически:

```
type Lan p f b = exists c. (p c -> b, f c)
```

Имеющимися средствами мы бы кодировали экзистенциальность так:

```
data Lan p f e where
  Lan :: (p c -> b) -> f c -> Lan p f b
```

Если у функтора P есть правый сопряженный, обозначим его P^{-1}:

$$\mathcal{B}(Pc, b) \cong \mathcal{C}(c, P^{-1}b)$$

тогда можно использовать лемму ниндзя ко-Йонеды, получая:

$$(\mathrm{Lan}_P F)b \simeq (F \circ P^{-1})b$$

и, тем самым, усиливая интуитивную мысль о том, что расширение Кана инвертирует P и следует за ним с F.

Правый сопряженный как левое расширение Кана

Мы видели, что при наличии сопряжения $L \vdash R$, левый сопряженный связан с правым расширением Кана. Двойственным образом, если правый сопряженный существует, его можно выразить как левое расширение Кана тождественного функтора:

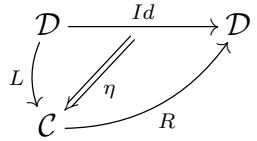
$$R \cong \mathrm{Lan}_L Id$$

Наоборот, если левое расширение Кана тождественности существует и сохраняет функтор L:

$$L \circ \mathrm{Lan}_L Id \cong \mathrm{Lan}_L L$$

то $\mathrm{Lan}_L Id$ является правым сопряженным L (кстати, $\mathrm{Lan}_F F$ называется плотной ко-монадой).

Единица расширения Кана такая же, как единица сопряжения:



Доказательство универсальности аналогично доказательству для правого расширения Кана.

D-свертка как расширение Кана

D-свертка определяется как тензорное произведение в категории копредпучков над моноидальной категорией $\mathcal{C}$:

$$(F \star G)c = \int^{a,b} \mathcal{C}(a \otimes b, c) \times Fa \times Gb$$

Ко-предпучки, то есть функторы в $[\mathcal{C}, \mathbf{Set}]$, также могут быть тензорированы с использованием *внешнего тензорного произведения*. Внешнее произведение двух объектов, вместо создания объекта в той же категории, выбирает объект в другой категории. В нашем случае, результат произведения двух функторов находится в категории ко-предпучков на $\mathcal{C} \times \mathcal{C}$:

$$\bar{\otimes} : [\mathcal{C}, \mathbf{Set}] \times [\mathcal{C}, \mathbf{Set}] \to [\mathcal{C} \times \mathcal{C}, \mathbf{Set}]$$

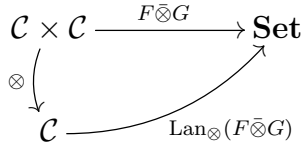
Произведение двух ко-предпучков, действующих на пару объектов в $\mathcal{C} \times \mathcal{C}$, определяется формулой:

$$(F \bar{\otimes} G)\, \langle a, b \rangle = Fa \times Gb$$

Оказывается, D-свертка двух функторов может быть выражена как левое расширение Кана их внешнего произведения вдоль тензорного произведения в $\mathcal{C}$:

$$F \star G \cong \mathrm{Lan}_{\otimes}(F \bar{\otimes} G)$$

Графически:



Действительно, используя формулу ко-конца для левого расширения Кана, получаем:

$$(\mathrm{Lan}_{\otimes}(F \bar{\otimes} G))c \cong \int^{\langle a,b \rangle} \mathcal{C}(a \otimes b, c) \cdot (F \bar{\otimes} G)\, \langle a, b \rangle \cong$$
$$\int^{\langle a,b \rangle} \mathcal{C}(a \otimes b, c) \cdot (Fa \times Gb)$$

Поскольку эти два функтора являются $\mathbf{Set}$-значными, ко-степень распадается до декартова произведения:

$$\cong \int^{\langle a,b \rangle} \mathcal{C}(a \otimes b, c) \times Fa \times Gb$$

и воспроизводит формулу D-свертки.

Расширения Кана и оптика

Рассмотрим актегорию $\mathcal{C}^{\mathsf{op}} \times \mathcal{D}$ с действием моноидальной категории $\mathcal{M}$. Это полное действие является функтором:

$$\begin{aligned} \bullet : \mathcal{M}^{\mathsf{op}} \times \mathcal{M} \times \mathcal{C}^{\mathsf{op}} \times \mathcal{D} &\to \mathcal{C}^{\mathsf{op}} \times \mathcal{D} \\ \langle m, n \rangle \bullet \langle c, d \rangle &= \langle m \bullet c, n \bullet d \rangle \end{aligned}$$

Поскольку мы собираемся взять ко-конец над $m : \mathcal{M}$, то будем использовать сокращение для диагональной части этого действия:

$$m \bullet \langle c, d \rangle = \langle m \bullet c, m \bullet d \rangle$$

С помощью этих обозначений можно выразить общую оптику в терминах левого расширения Кана:

$$\mathcal{O} \langle s, t \rangle \langle a, b \rangle \cong \left(\int^{m:\mathcal{M}} \mathrm{Lan}_{m\bullet} \mathcal{Y}_{\langle a,b \rangle} \right) \langle s, t \rangle$$

где

$$\mathcal{Y}_{\langle a,b \rangle} \langle c, d \rangle = (\mathcal{C}^{\mathsf{op}} \times \mathcal{D})(\langle a, b \rangle, \langle c, d \rangle) = \mathcal{C}(c, a) \times \mathcal{D}(b, d)$$

является функтором Йонеды.

Действительно, по определению, имеем:

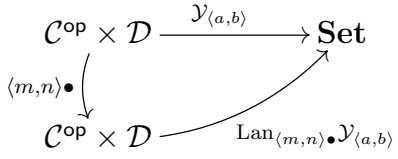
$$\left(\int^{m:\mathcal{M}} \mathrm{Lan}_{m\bullet} \mathcal{Y}_{\langle a,b \rangle} \right) \langle s, t \rangle \cong \int^{m, \langle c,d \rangle} (\mathcal{C}^{\mathsf{op}} \times \mathcal{D})(m \bullet \langle c, d \rangle, \langle s, t \rangle) \cdot \mathcal{Y}_{\langle a,b \rangle} \langle c, d \rangle$$

Теперь можно применить лемму ко-Йонеды, чтобы получить:

$$\cong \int^{m} (\mathcal{C}^{\mathsf{op}} \times \mathcal{D})(m \bullet \langle a, b \rangle, \langle s, t \rangle)$$

что является экзистенциальной формой смешанной оптики.

Рассматриваемые категории изображены на диаграмме:



Вообще, с заменой функтора Йонеды на общий профунктор, этот профунктор-функтор:

$$\Phi P = \int^m \mathrm{Lan}_{m\bullet} P$$

— это монада Пастро-Стрита, которую мы использовали при выводе профункторной оптики.

С помощью этого определения можно вывести комонаду Пастро-Стрита Θ как правый сопряженный к Φ. Будем использовать обозначение Nat для множества естественных преобразований в $[\mathcal{C}^{\mathsf{op}} \times \mathcal{D}, \mathbf{Set}]$. Начнем с:

$$\mathrm{Nat}(\Phi P, Q) = \mathrm{Nat}\left(\int^m \mathrm{Lan}_{m\bullet} P, Q\right)$$

Переносим вперед ко-конец, используя ко-непрерывность hom-функтора:

$$\cong \int_m \mathrm{Nat}(\mathrm{Lan}_{m\bullet} P, Q)$$

и используем сопряжение, определяющее левое расширение Кана:

$$\cong \int_m \mathrm{Nat}(P, Q \circ (m \bullet -))$$

Далее, используем непрерывность hom-функтора, чтобы переместить конец внутрь:

$$\cong \mathrm{Nat}\left(P, \int_m Q \circ (m \bullet -)\right)$$

В результате:

$$\mathrm{Nat}(\Phi P, Q) \cong \mathrm{Nat}(P, \Theta Q)$$

где:

$$(\Theta Q)\,\langle a, b\rangle = \int_m Q(m \bullet \langle a, b\rangle)$$

19.5 Полезные формулы

- Ко-степень:

$$\mathcal{C}(A \cdot b, c) \cong \mathbf{Set}(A, \mathcal{C}(b, c))$$

- Степень:

$$\mathcal{C}(b, A \pitchfork c) \cong \mathbf{Set}(A, \mathcal{C}(b, c))$$

- Правое расширение Кана:

$$[\mathcal{C},\mathcal{D}](G\circ P,F)\cong[\mathcal{B},\mathcal{D}](G,\mathrm{Ran}_P F)$$
$$(\mathrm{Ran}_P F)e\cong\int_c \mathcal{B}(e,Pc)\pitchfork Fc$$

- Правое расширение Кана в **Set**:

$$(\mathrm{Ran}_P F)e\cong\int_c \mathbf{Set}(\mathcal{B}(e,Pc),Fc)$$

- Левое расширение Кана:

$$[\mathcal{B},\mathcal{D}](\mathrm{Lan}_P F,G)\cong[\mathcal{C},\mathcal{D}](F,G\circ P)$$
$$(\mathrm{Lan}_P F)e\cong\int^c \mathcal{B}(Pc,e)\cdot Fc$$

- Левое расширение Кана в **Set**:

$$(\mathrm{Lan}_P F)e\cong\int^c \mathcal{B}(Pc,e)\times Fc$$

Глава 20

Обогащение

Лао-цзы утверждает: «Знать, что у тебя всего достаточно, значит быть богатым».

20.1 Обогащенные категории

Может показаться неожиданным, что определение `Functor` в Haskell не может быть полностью объяснено без некоторого опыта работы с обогащенными категориями. Однако, в этой главе будет предпринята попытка показать, что, по крайней мере концептуально, обогащение не является огромным шагом вперед по сравнению с обычной теорией категорий.

Дополнительной мотивацией для изучения обогащенных категорий является тот факт, что во многих источниках, особенно на веб-сайте nLab, содержится описание понятий в самых общих терминах, которые относятся к терминологии обогащенных категорий. Большинство обычных конструкций можно перевести, просто изменив словарь, заменяя, например, hom-множества на hom-объекты, а категорию $\mathbf{Set}$ — на моноидальную категорию $\mathcal{V}$.

Некоторые расширенные понятия, такие как взвешенные пределы и ко-пределы, оказываются мощными сами по себе, до такой степени, что может возникнуть соблазн заменить изречение МакЛейна «Все понятия являются расширениями Кана» на «Все понятия являются взвешенными (со-)пределами».

Теоретико-множественные основы

Теория категорий очень умеренна в своей основе. Но она опирается на теорию множеств (хотя, и неохотно). В частности, идея hom-множества, определяемого как множество стрелок между двумя объектами, рассматривается в теории множеств как предпосылка к теории категорий. Конечно, стрелки образуют множество только в *локально малой* категории, но это слабое утешение, учитывая, что работа с сущностями, которые слишком велики, чтобы быть множествами, требует еще больше теории.

Было бы неплохо, если бы теория категорий была способна самообновляться, например, заменой hom-множеств более общими объектами. Именно в этом и заключается суть идеи обогащенных категорий. Однако, эти hom-объекты должны происходить из какой-то другой категории, имеющей hom-множества, то есть, в какой-то момент появляется необходимость обращения к теоретико-множественным основам. Тем не менее, возможность замены бесструктурных hom-множеств чем-то другим расширяет возможности моделирования более сложных систем.

Основное свойство множеств состоит в том, что они, в отличие от объектов, не атомарны: они содержат *элементы*. В теории категорий мы иногда говорим об *обобщенных элементах*, которые представляют собой просто стрелки, указывающие на объект; или о *глобальных элементах*, которые являются стрелками от терминального объекта (или, иногда, от моноидальной единицы I). Но самым важным является то, что множества определяют *равенство элементов*.

Практически все, что известно о категориях, можно перевести в область обогащенных категорий. Тем не менее, многие категорные рассуждения включают коммутативные диаграммы, которые выражают равенство стрелок. В обогащенной обстановке отсутствуют стрелки, связывающие объекты, поэтому все соответствующие конструкции придется модифицировать.

hom-объекты

На первый взгляд замена hom-множеств объектами может показаться шагом назад. В конце концов, у множеств есть элементы, а объекты — это бесформенные образования. Однако, богатство hom-объектов закодировано в морфизмах категории. Концептуально, тот факт, что множества

не имеют структуры, означает, что между ними существует множество морфизмов (функций). Меньшее количество морфизмов часто означает наличие большей структуры.

Руководящий принцип в определении обогащенных категорий состоит в том, что должна существовать возможность восстановления обычной теории категорий как частного случая. Ведь $\hom$-множества *являются* объектами категории **Set**. На самом деле, было приложено достаточно усилий, чтобы выразить свойства множеств в терминах функций, а не элементов.

При этом, само определение категории в терминах композиции и тождественности включает морфизмы, которые являются *элементами* $\hom$-множеств. Так что, давайте сначала переформулируем примитивы категории, не прибегая к элементам.

Композиция стрелок может быть определена, в основном, как функция между $\hom$-множествами:

$$\circ : \mathcal{C}(b, c) \times \mathcal{C}(a, b) \to \mathcal{C}(a, c)$$

Вместо того, чтобы говорить о тождественной стрелке, можно использовать функцию от одноэлементного множества:

$$j_a : 1 \to \mathcal{C}(a, a)$$

Это показывает, что если требуется заменитье $\hom$-множества $\mathcal{C}(a, b)$ объектами из некоторой категории $\mathcal{V}$, то необходимо иметь возможность перемножать эти объекты для определения композиции, и нужен какой-то единичный объект для определения тождественности. Можно было бы потребовать, чтобы $\mathcal{V}$ была декартовой, но, на самом деле, и моноидальная категория работает просто отлично. Как мы убедимся, законы единицы и ассоциативности моноидальной категории переходят непосредственно в законы тождественности и ассоциативности для композиции.

Обогащенные категории

Пусть $\mathcal{V}$ — моноидальная категория с тензорным произведением $\otimes$, единичным объектом I, ассоциатором и двумя унуторами (а также их об-

ратными):

$$\begin{aligned}\alpha &: (a \otimes b) \otimes c \to a \otimes (b \otimes c)\\ \lambda &: I \otimes a \to a\\ \rho &: a \otimes I \to a\end{aligned}$$

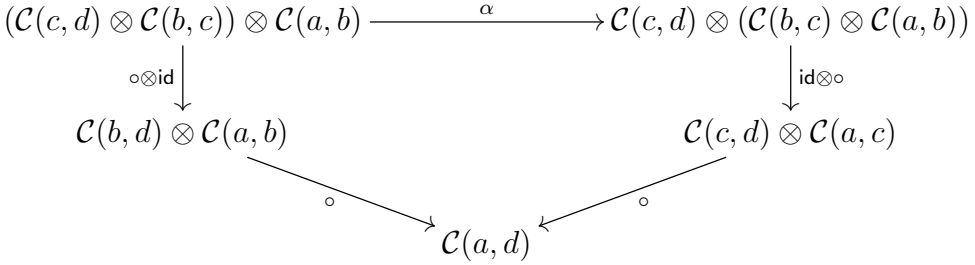
Категория $\mathcal{C}$, обогащенная над $\mathcal{V}$, имеет объекты и, для любой пары объектов a и b, hom-объект $\mathcal{C}(a,b)$. This hom-объект является объектом в $\mathcal{V}$. Композиция определяется с помощью стрелок в $\mathcal{V}$:

$$\circ : \mathcal{C}(b,c) \otimes \mathcal{C}(a,b) \to \mathcal{C}(a,c)$$

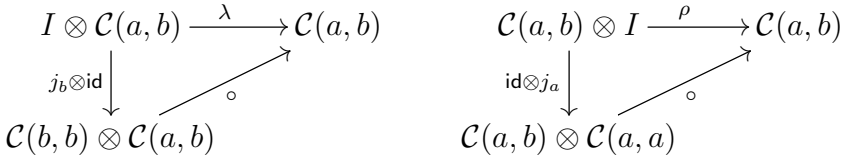
Тождественность определяется стрелкой:

$$j_a : I \to \mathcal{C}(a,a)$$

Ассоциативность выражается через ассоциаторы в $\mathcal{V}$:



Законы единицы выражаются через униторы в $\mathcal{V}$:



Обратите внимание, что это все диаграммы в $\mathcal{V}$, где имеются стрелки, формирующие hom-множества. Мы по-прежнему возвращаемся к теории множеств, но уже на другом уровне.

Категория, обогащенная над $\mathcal{V}$, также называется *$\mathcal{V}$-категорией*. В дальнейшем будем предполагать, что обогащенная категория является симметричной моноидальной, поэтому можно образовывать противоположные и произведение $\mathcal{V}$-категорий.

Категория $\mathcal{C}^{\mathsf{op}}$, противоположная $\mathcal{V}$-категории $\mathcal{C}$, получается перестановкой hom-объектов, то есть:

$$\mathcal{C}^{\mathsf{op}}(a,b) = \mathcal{C}(b,a)$$

Композиция в противоположной категории включает обратный порядок hom-объектов, поэтому она работает только в том случае, если тензорное произведение симметрично.

Также, можно определить тензорное произведение $\mathcal{V}$-категорий; опять же, при условии, что $\mathcal{V}$ симметрична. Произведение двух $\mathcal{V}$-категорий $\mathcal{C} \otimes \mathcal{D}$ имеет, в качестве объектов, пары объектов, по одному из каждой категории. hom-объекты между такими парами определяются как тензорные произведения:

$$(\mathcal{C} \otimes \mathcal{D})(\langle c,d\rangle, \langle c',d'\rangle) = \mathcal{C}(c,c') \otimes \mathcal{D}(d,d')$$

Нам нужна симметрия тензорного произведения, чтобы определить композицию. Действительно, нужно поменять местами два hom-объекта в середине, прежде чем применять две имеющиеся композиции:

$$\circ : (\mathcal{C}(c',c'') \otimes \mathcal{D}(d',d'')) \otimes (\mathcal{C}(c,c') \otimes \mathcal{D}(d,d')) \to \mathcal{C}(c,c'') \otimes \mathcal{D}(d,d'')$$

Тождественная стрелка является тензорным произведением двух тождественностей:

$$I_{\mathcal{C}} \otimes I_{\mathcal{D}} \xrightarrow{j_c \otimes j_d} \mathcal{C}(c,c) \otimes \mathcal{D}(d,d)$$

Упражнение 20.1.1. *Определите композицию и единицу в $\mathcal{V}$-категории $\mathcal{C}^{\mathsf{op}}$.*

Упражнение 20.1.2. *Покажите, что каждая $\mathcal{V}$-категория $\mathcal{C}$ имеет базовую обычную категорию $\mathcal{C}_0$, с теми же объектами, но чьи* hom-*множества задаются (моноидальными глобальными) элементами* hom-*объектов, то есть элементами $\mathcal{V}(I, \mathcal{C}(a,b))$.*

Примеры

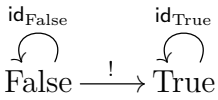
С этой новой точки зрения, обычные категории, которые мы изучали до сих пор, были тривиально обогащены моноидальной категорией $(\mathbf{Set}, \times, 1)$, с декартовым произведением в качестве тензорного произведения и одноэлементным множеством в качестве единицы.

Интересно, что 2-категорию можно рассматривать как обогащенную над **Cat**. Действительно, 1-клетки в 2-категории сами являются объектами в другой категории. 2-клетки — это просто стрелки в этой категории. В частности, обогащается, сама по себе, 2-категория **Cat** малых категорий. Ее hom-объекты — это категории функторов, которые являются объектами в **Cat**.

Предпорядки

Обогащение не всегда означает добавление большего количества сущностей. Иногда, это больше похоже на обеднение, как в случае с обогащением над категорией шагающей стрелки.

В этой категории всего два объекта, которые для целей данной конструкции назовем False и True. Существует единственная стрелка от False к True (не считая тождественных стрелок), которая делает False инициальным объектом, а True — терминальным объектом.



Чтобы превратить это в моноидальную категорию, определим тензорное произведение так, чтобы выполнялось:

$$\mathrm{True} \otimes \mathrm{True} = \mathrm{True}$$

а все остальные комбинации приводили к False. True является моноидальной единицей, так как:

Категория, обогащенная над моноидальной шагающей стрелкой, называется *пред-порядком*. А hom-объект $\mathcal{C}(a, b)$ между любыми двумя объектами может быть, либо False, либо True. Мы интерпретируем True как означающее, что a предшествует b в пред-порядке, что будем записывать как $a \leqslant b$. False означает, что два объекта не сравнимы.

Важное свойство композиции, определяемое:

$$\mathcal{C}(b, c) \otimes \mathcal{C}(a, b) \to \mathcal{C}(a, c)$$

состоит в том, что если оба hom-объекта слева равны True, то и правая часть тоже должна быть True (она не может быть False, потому что нет

стрелки, идущей от True к False). В интерпретации пред-порядка это означает, что отношение $\leqslant$ является транзитивным:

$$b \leqslant c \wedge a \leqslant b \Longrightarrow a \leqslant c$$

По тем же рассуждениям существование тождественной стрелки:

$$j_a : \mathrm{True} \to \mathcal{C}(a, a)$$

означает, что $\mathcal{C}(a, a)$ всегда есть True. В интерпретации пред-порядка это означает, что $\leqslant$ рефлексивно, т.е. $a \leqslant a$.

Заметим, что пред-порядок не исключает циклов, и, в частности, возможно иметь $a \leqslant b$ и $b \leqslant a$ без того, чтобы a было равно b.

Пред-порядок также можно определить, не прибегая к обогащению, как *тонкую категорию* — категорию, в которой между любыми двумя объектами находится не более одной стрелки.

Самообогащение

Любую декартово замкнутую категорию $\mathcal{V}$ можно рассматривать как само-обогащающуюся. Это связано с тем, что любое внешнее hom-множество $\mathcal{C}(a, b)$ можно заменить внутренним hom b^a (объектом стрелок).

Фактически, каждая *моноидально замкнутая* категория $\mathcal{V}$ является само-обогащаемой. Напомним, что в моноидально замкнутой категории имеется hom-функторное сопряжение:

$$\mathcal{V}(a \otimes b, c) \cong \mathcal{V}(a, [b, c])$$

Ко-единица этого сопряжения работает как оценочный морфизм:

$$\varepsilon_{bc} : [b, c] \otimes b \to c$$

Чтобы определить композицию в этой само-обогащаемой категории, требуется стрелка:

$$\circ : [b, c] \otimes [a, b] \to [a, c]$$

Прием заключается в том, чтобы рассмотреть все hom-множество целиком и показать, что из него всегда можно выбрать канонический элемент. Начнем с множества:

$$\mathcal{V}([b, c] \otimes [a, b], [a, c])$$

Можно использовать сопряжение, чтобы переписать это множество как:

$$\mathcal{V}(([b,c] \otimes [a,b]) \otimes a, c)$$

Все, что сейчас нужно сделать, это выбрать элемент этого hom-множества. Сделаем это, создавая следующую композицию:

$$([b,c] \otimes [a,b]) \otimes a \xrightarrow{\alpha} [b,c] \otimes ([a,b]) \otimes a) \xrightarrow{\mathsf{id} \otimes \varepsilon_{ab}} [b,c] \otimes b \xrightarrow{\varepsilon_{bc}} c$$

Мы использовали ассоциатор и ко-единицу сопряжения.

Также требуется стрелка, определяющая тождественность:

$$j_a : I \to [a,a]$$

Опять же, ее можно выбрать как элемент hom-множества $\mathcal{V}(I, [a,a])$. Используем сопряжение:

$$\mathcal{V}(I, [a,a]) \cong \mathcal{V}(I \otimes a, a)$$

Мы знаем, что это hom-множество содержит левый унитор λ, поэтому его можно использовать для определения j_a.

20.2 $\mathcal{V}$-функторы

Обычный функтор отображает объекты к объектам, а стрелки — к стрелкам. Точно так же, *обогащенный* функтор F отображает объект к объектам, но вместо того, чтобы действовать на отдельные стрелки, он должен отображать hom-объекты к hom-объектам. Это возможно только в том случае, если hom-объекты в исходной категории $\mathcal{C}$ принадлежат к той же категории, что и hom-объекты в целевой категории $\mathcal{D}$. Другими словами, обе категории должны быть обогащены одной и той же $\mathcal{V}$. Затем, действие F на hom-объектах определяется с помощью стрелок в $\mathcal{V}$:

$$F_{ab} : \mathcal{C}(a,b) \to \mathcal{D}(Fa, Fb)$$

Для ясности мы указываем пару объектов в нижнем индексе F.

Функтор должен сохранять композицию и тождественность. Их можно выразить в виде коммутативных диаграмм в $\mathcal{V}$:

$$\begin{array}{ccc} \mathcal{C}(b,c)\otimes\mathcal{C}(a,b) & \xrightarrow{\circ} & \mathcal{C}(a,c) \\ {\scriptstyle F_{bc}\otimes F_{ab}}\downarrow & & \downarrow{\scriptstyle F_{ac}} \\ \mathcal{D}(Fb,Fc)\otimes\mathcal{D}(Fa,Fb) & \xrightarrow[\circ]{} & \mathcal{D}(Fa,Fc) \end{array} \qquad \begin{array}{ccc} & I & \\ {\scriptstyle j_a}\swarrow & & \searrow{\scriptstyle j_{Fa}} \\ \mathcal{C}(a,a) & \xrightarrow[F_{aa}]{} & \mathcal{D}(Fa,Fa) \end{array}$$

Отметим, что мы использовали один и тот же символ $\circ$ для двух разных композиций и один и тот же j для двух разных отображений тождественности. Их смысл можно вывести из контекста.

Как и прежде, все диаграммы находятся в категории $\mathcal{V}$.

hom-функтор

hom-функтор в категории, обогащенной над моноидально замкнутой категорией $\mathcal{V}$, является обогащенным функтором:

$$\mathrm{Hom}_{\mathcal{C}} : \mathcal{C}^{\mathsf{op}}\otimes\mathcal{C}\to\mathcal{V}$$

Здесь, чтобы определить обогащенный функтор, с $\mathcal{V}$ надо обращаться как с само-обогащенной категорией.

Понятно, как этот функтор работает с парами объектов:

$$\mathrm{Hom}_{\mathcal{C}}\,\langle a,b\rangle = \mathcal{C}(a,b)$$

Чтобы определить обогащенный функтор, надо определить действие Hom на hom-объектах. Здесь исходной категорией является $\mathcal{C}^{\mathsf{op}}\otimes\mathcal{C}$, а целевой категорией — $\mathcal{V}$, обе они обогащены над $\mathcal{V}$. Рассмотрим hom-объект от $\langle a,a'\rangle$ к $\langle b,b'\rangle$. Действие hom-функтора на этот hom-объект представляет собой стрелку в $\mathcal{V}$:

$$\mathrm{Hom}_{\langle a,a'\rangle\langle b,b'\rangle} : (\mathcal{C}^{\mathsf{op}}\otimes\mathcal{C})(\langle a,a'\rangle,\langle b,b'\rangle)\to\mathcal{V}(\mathrm{Hom}\,\langle a,a'\rangle,\mathrm{Hom}\,\langle b,b'\rangle)$$

По определению категории произведений, источник является тензорным произведением двух hom-объектов. Цель есть внутренний hom в $\mathcal{V}$. Таким образом, имеем стрелку:

$$\mathcal{C}(b,a)\otimes\mathcal{C}(a',b')\to[\mathcal{C}(a,a'),\mathcal{C}(b,b')]$$

Мы можем использовать каррированное сопряжение hom-функтора, для распаковки внутреннего hom:

$$\big(\mathcal{C}(b,a) \otimes \mathcal{C}(a',b')\big) \otimes \mathcal{C}(a,a') \to \mathcal{C}(b,b')$$

Можно построить эту стрелку, переставив произведение и дважды применяя композицию.

В обогащенной настройке самое близкое к определению индивидуального морфизма от a к b — это использование стрелки от единичного объекта. Определим (моноидально-глобальный) элемент hom-объекта как морфизм в $\mathcal{V}$:

$$f : I \to \mathcal{C}(a,b)$$

Можно определить, что значит поднять такую стрелку, используя hom-функтор. Например, сохраняя первый аргумент постоянным, мы бы определили:

$$\mathcal{C}(c,f) : \mathcal{C}(c,a) \to \mathcal{C}(c,b)$$

как композицию:

$$\mathcal{C}(c,a) \xrightarrow{\lambda^{-1}} I \otimes \mathcal{C}(c,a) \xrightarrow{f \otimes \mathsf{id}} \mathcal{C}(a,b) \otimes \mathcal{C}(c,a) \xrightarrow{\circ} \mathcal{C}(c,b)$$

Аналогично, контравариантное поднятие f:

$$\mathcal{C}(f,c) : \mathcal{C}(b,c) \to \mathcal{C}(a,c)$$

может быть определено как:

$$\mathcal{C}(b,c) \xrightarrow{\rho^{-1}} \mathcal{C}(b,c) \otimes I \xrightarrow{\mathsf{id} \otimes f} \mathcal{C}(b,c) \otimes \mathcal{C}(a,b) \xrightarrow{\circ} \mathcal{C}(a,c)$$

Многие знакомые конструкции, изучаемые в обычной теории категорий, имеют свои обогащенные аналоги, в которых произведения заменены тензорными произведениями, а **Set** заменены на $\mathcal{V}$.

Упражнение 20.2.1. *Что является функтором между двумя предпорядками?*

Обогащенные ко-предпучки

Ко-предпучки, то есть **Set**-значные функторы, играют важную роль в теории категорий, поэтому естественно задаться вопросом, каковы их аналоги в обогащенном окружении. Обобщением ко-предпучка является $\mathcal{V}$-функтор $\mathcal{C} \to \mathcal{V}$. Это возможно только в том случае, если $\mathcal{V}$ можно превратить в $\mathcal{V}$-категорию, то есть, когда она моноидально замкнута.

Обогащенный ко-предпучок отображает объект $\mathcal{C}$ к объектам в $\mathcal{V}$, а $\hom$-объекты $\mathcal{C}$ к внутренним $\hom$ в $\mathcal{V}$:

$$F_{ab} : \mathcal{C}(a, b) \to [Fa, Fb]$$

В частности, Hom-функтор является примером $\mathcal{V}$-значного $\mathcal{V}$-функтора:

$$\mathrm{Hom} : \mathcal{C}^{\mathsf{op}} \otimes \mathcal{C} \to \mathcal{V}$$

$\hom$-функтор является частным случаем обогащенного профунктора, который определяется как:

$$\mathcal{C}^{\mathsf{op}} \otimes \mathcal{D} \to \mathcal{V}$$

Упражнение 20.2.2. *Тензорное произведение — это функтор в* $\mathcal{V}$:

$$\otimes : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$$

Покажите, что если $\mathcal{V}$ *моноидально замкнутая, то тензорное произведение определяет* $\mathcal{V}$*-функтор. Подсказка*: *определите его действие на внутренних* $\hom$.

Функториальная прочность и обогащение

Когда мы обсуждали монады, было упомянуто важное свойство, позволяющее им работать в программировании. Эндофункторы, определяющие монады, должны быть прочными, чтобы можно было получить доступ к внешним контекстам внутри монадического кода.

Оказывается, то, как мы определили эндофункторы в Haskell, автоматически делает их прочными. Причина в том, что прочность связана с обогащением, а, как мы видели, декартово замкнутая категория самообогащается. Начнем с некоторых определений.

Функториальная прочность эндофунктора F в моноидальной категории определяется как естественное преобразование с компонентами:

$$\sigma_{ab} : a \otimes Fb \to F(a \oplus b)$$

Имеются несколько довольно очевидных условий согласованности, которые не позволяют прочности нарушать свойства тензорного произведения. Это условие ассоциативности:

$$\begin{array}{ccccc} (a \otimes b) \otimes F(c) & & \xrightarrow{\sigma_{(a\otimes b)c}} & & F((a \otimes b) \otimes c) \\ \downarrow{\scriptstyle \alpha} & & & & \downarrow{\scriptstyle F(a)} \\ a \otimes (b \otimes F(c)) & \xrightarrow[a\otimes\sigma_{bc}]{} & a \otimes F(b \otimes c) & \xrightarrow[\sigma_{a(b\otimes c)}]{} & F(a \otimes (b \otimes c)) \end{array}$$

а это условие единицы:

$$\begin{array}{ccc} I \otimes F(a) & \xrightarrow{\sigma_{Ia}} & F(I \otimes a) \\ & {\scriptstyle \lambda} \searrow & \downarrow{\scriptstyle F(\lambda)} \\ & & F(a) \end{array}$$

В общей моноидальной категории это называется *левой прочностью*, и существует соответствующее определение *правой* прочности. В симметричной моноидальной категории они эквивалентны.

Обогащенный эндофунктор отображает hom-объекты к hom-объектам:

$$F_{ab} : \mathcal{C}(a, b) \to \mathcal{C}(Fa, Fb)$$

Если мы будем считать моноидально замкнутую категорию $\mathcal{V}$ самообогащенной, то hom-объекты являются внутренними hom, поэтому обогащенный эндофунктор оснащен отображением:

$$F_{ab} : [a, b] \to [Fa, Fb]$$

Сравните это с нашим определением `Functor` на Haskell:

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

Функциональные типы, включенные в это определение, `(a -> b)` и `(f a -> f b)`, являются внутренними hom. Так чо, `Functor` действительно является обогащенным функтором.

Обычно, мы не делаем различий между внешними и внутренними hom в Haskell, поскольку их множества элементов изоморфны. Это простое следствие каррированного сопряжения:

$$\mathcal{C}(1 \times b, c) \cong \mathcal{C}(1, [b, c])$$

и того факта, что терминальный объект является единицей декартова произведения.

Оказывается, в само-обогащенной категории $\mathcal{V}$ каждый прочный эндофунктор автоматически обогащается. Действительно, чтобы показать, что функтор F обогащен, нужно определить отображение между внутренними hom, то есть элементами hom-множества:

$$F_{ab} \in \mathcal{V}([a, b], [Fa, Fb])$$

Используя сопряжение hom, это изоморфно:

$$\mathcal{V}([a, b] \otimes Fa, Fb)$$

Можно построить это отображение, компонуя прочность и ко-единицу сопряжения (оценочный морфизм):

$$[a, b] \otimes Fa \xrightarrow{\sigma_{[a,b]a}} F([a, b] \otimes a) \xrightarrow{\epsilon_{ab}} Fb$$

И наоборот, каждый обогащенный эндофунктор в $\mathcal{V}$ является прочным. Чтобы показать это, нужно определить естественное преобразование с компонентами, являющимися элементами следующих hom-множеств:

$$\sigma_{ab} \in V(a \otimes Fb, F(a \otimes b))$$

Напомним определение единицы сопряжения hom, ко-оценочного морфизма:

$$\eta_{ab} : a \to [b, a \otimes b]$$

Используя единицу и ко-единицу вместе с действием обогащенного функтора на внутренний hom, построим следующую композицию:

$$a \otimes Fb \xrightarrow{\eta_{ab} \otimes Fb} [b, a \otimes b] \otimes Fb \xrightarrow{F_{b,a\otimes b} \otimes Fb} [Fb, F(a \otimes b)] \otimes Fb \xrightarrow{\epsilon} F(a \otimes b)$$

Это можно перевести непосредственно на Haskell:

```
strength :: Functor f => (a, f b) -> f (a, b)
strength  = eval . bimap fmap id . bimap coeval id
```

со следующими определениями `eval` и `coeval`:

```
eval       :: (a -> b, a) -> b
eval (f, a) = f a

coeval     :: a -> (b -> (a, b))
coeval a    = \b -> (a, b)
```

Поскольку каррирование и оценивание встроены в Haskell, можно еще больше упростить эту формулу:

```
strength        :: Functor f => (a, f b)
                                -> f (a, b)
strength (a, bs) = fmap (a, ) bs
```

20.3 $\mathcal{V}$-естественные преобразования

Обычное естественное преобразование между двумя функторами F и G от $\mathcal{C}$ к $\mathcal{D}$ представляет собой выбор стрелок из hom-множеств $\mathcal{D}(Fa, Ga)$. В обогащенном окружении стрелки отсутствуют, поэтому лучшее, что можно сделать, это использовать единичный объект I для выбора. Определим компонент *$\mathcal{V}$-естественного преобразования* при a как стрелку:

$$\nu_a : I \to \mathcal{D}(Fa, Ga)$$

Условие естественности несколько сложное. Стандартный квадрат естественности включает поднятие произвольной стрелки $f : a \to b$ и равенство следующих композиций:

$$\nu_b \circ Ff = Gf \circ \nu_a$$

Рассмотрим hom-множества, которые участвуют в этом выражении. Мы поднимаем морфизм $f \in \mathcal{C}(a, b)$. Компоновки в обеих частях уравнения являются элементами из $\mathcal{D}(Fa, Gb)$.

Слева имеется стрелка $\nu_b \circ Ff$. Сама композиция представляет собой отображение от произведения двух hom-множеств:

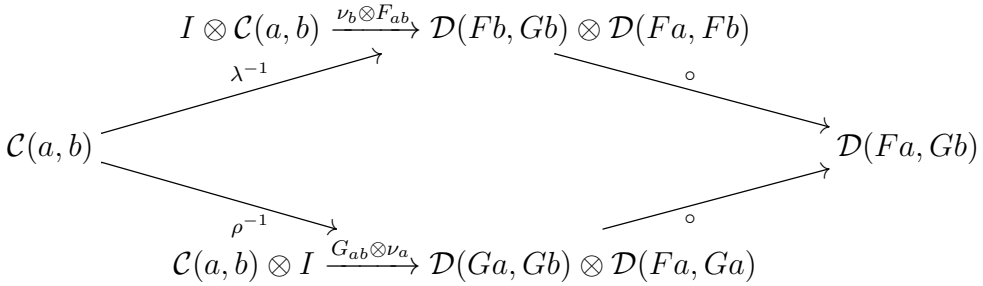
$$\mathcal{D}(Fb, Gb) \times \mathcal{D}(Fa, Fb) \to \mathcal{D}(Fa, Gb)$$

Аналогичным образом, справа имеем $Gf \circ \nu_a$, которая есть композиция:

$$\mathcal{D}(Ga, Gb) \times \mathcal{D}(Fa, Ga) \to \mathcal{D}(Fa, Gb)$$

В расширенном окружении приходится работать с hom-объектами, а не с hom-множествами, и выбор компонентов естественного преобразования осуществляется с помощью единицы I. Всегда можно произвести единицу из «разряжения», используя инверсию левого или правого унитора.

В целом, условие естественности выражается в виде коммутативной диаграммы:



Это также работает для обычной категории, где можно проследить два пути через эту диаграмму, сначала выбрав f из $\mathcal{C}(a, b)$. Затем можно использовать ν_b и ν_a для выбора компонентов естественного преобразования. Мы, также, поднимаем f, используя, либо F, либо G. Наконец, используем композицию, чтобы воспроизвести выражение для естественности.

Эта диаграмма может быть дополнительно упрощена, если воспользоваться нашим предыдущим определением действия hom-функтора на глобальных элементах hom-объектов. Компоненты естественного преобразования определяются как такие глобальные элементы:

$$\nu_a : I \to \mathcal{D}(Fa, Ga)$$

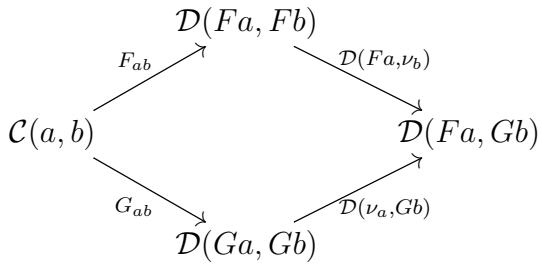
В нашем распоряжении два подъема:

$$\mathcal{D}(d, \nu_b) : \mathcal{D}(d, Fb) \to \mathcal{D}(d, Gb)$$

и:

$$\mathcal{D}(\nu_a, d) : \mathcal{D}(Ga, d) \to \mathcal{D}(Fa, d)$$

Получаем нечто, больше похожее на знакомый квадрат естественности:



$\mathcal{V}$-естественные преобразования между двумя $\mathcal{V}$-функторами F и G образуют множество, которое обозначим $\mathcal{V}\text{-nat}(F, G)$.

Ранее мы видели, что в обычных категориях множество естественных преобразований может быть записано как конец:

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_a \mathcal{D}(Fa, Ga)$$

Оказывается, что концы и ко-концы могут быть определены и для обогащенных профункторов, так что эта формула работает и для обогащенных естественных преобразований. Отличие состоит в том, что вместо множества естественных преобразований $\mathcal{V}\text{-nat}(F, G)$, определяется объект естественных преобразований $[\mathcal{C}, \mathcal{D}](F, G)$ в $\mathcal{V}$.

Такое определение (ко-)конца $\mathcal{V}$-профунктора $P : \mathcal{C} \otimes \mathcal{C}^{\mathsf{op}} \to \mathcal{V}$ аналогично определению, которое мы встречали для обычных профункторов. Например, конец — это объект e в $\mathcal{V}$, снабженный сверх-естественным преобразованием $\pi : e \to P$, которое является универсальным среди таких объектов.

20.4 Лемма Йонеды

Обычная лемма Йонеды включает **Set**-значный функтор F и множество естественных преобразований:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(c, -), F) \cong Fc$$

Чтобы распространить ее на обогащенную установку, рассмотрим $\mathcal{V}$-значный функтор F. Как и прежде, будем использовать тот факт, что

можно рассматривать категорию $\mathcal{V}$ как само-обогащенную, поскольку она замкнута, поэтому можно говорить о $\mathcal{V}$-значных $\mathcal{V}$-функторах.

Слабая версия леммы Йонеды имеет дело с множеством $\mathcal{V}$-естественных преобразований. Следовательно, мы должны превратить и правую часть в множество. Это делается путем взятия (моноидально-глобальных) элементов Fc. Получаем:

$$\mathcal{V}\text{-nat}(\mathcal{C}(c,-),F) \cong \mathcal{V}(I,Fc)$$

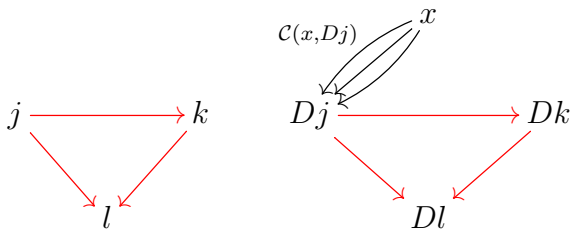
Сильная версия леммы Йонеды работает с объектами $\mathcal{V}$ и использует конец над внутренним hom в $\mathcal{V}$ для представления объекта естественных преобразований:

$$\int_x [\mathcal{C}(c,x),Fx] \cong Fc$$

20.5 Взвешенные пределы

Пределы (и копределы) строятся вокруг коммутативных треугольников, поэтому их нельзя сразу перевести в обогащенное окружение. Проблема в том, что конусы конструируются из «проводов», то есть индивидуальных морфизмов. Можно представлять себе hom-множества толстыми жгутами проводов, каждый из которых имеет нулевую толщину. При построении конуса вы выбираете один провод из hom-множества. Придется заменить провода на что-нибудь потолще.

Рассмотрим диаграмму, представляющую собой функтор D от индексной категории $\mathcal{J}$ к целевой категории $\mathcal{C}$. Провода для конуса с вершиной x выбираются из hom-множеств $\mathcal{C}(x,Dj)$, где j — объект из $\mathcal{J}$.



Этот выбор j-го провода можно описать как функцию от одноэлементного множества 1:

$$\gamma_j : 1 \to \mathcal{C}(x,Dj)$$

Можно постараться собрать эти функции в естественное преобразование:

$$\gamma : \Delta_1 \to \mathcal{C}(x, D-)$$

где Δ_1 — постоянный функтор, отображающий все объекты $\mathcal{J}$ к одноэлементному множеству. Условия естественности гарантируют, что треугольники, образующие стороны конуса, являются коммутативными.

Множество всех конусов с вершиной x задается тогда множеством естественных преобразований:

$$[\mathcal{J}, \mathbf{Set}](\Delta_1, \mathcal{C}(x, D-))$$

Эта переформулировка приближает нас к обогащенной установке, поскольку она перефразирует проблему в терминах hom-множеств, а не отдельных морфизмов. Можно было бы начать с рассмотрения, как $\mathcal{J}$, так и $\mathcal{C}$, обогащенных над $\mathcal{V}$, и в этом случае D был бы $\mathcal{V}$-функтором.

Остается только один вопрос: как определить постоянный $\mathcal{V}$-функтор $\Delta_x : \mathcal{C} \to \mathcal{D}$? Его действие на объекты очевидно: он отображает все объекты из $\mathcal{C}$ к одному объекту x в $\mathcal{D}$. Но что он должен делать с hom-объектами?

Обычный постоянный функтор Δ_x отображает все морфизмы из $\mathcal{C}(a, b)$ к тождественности из $\mathcal{D}(x, x)$. Однако в обогащенном окружении, $\mathcal{D}(x, x)$ — это объект без внутренней структуры. Даже если бы это была единица I в $\mathcal{V}$, нет гарантии, что для каждого hom-объекта $\mathcal{C}(a, b)$ можно будет найти стрелку к I; а даже если она и была бы, то, что она не является единственной. Другими словами, нет оснований полагать, что I — терминальный объект.

Решение заключается в том, чтобы «размазать сингулярность»: вместо использования постоянного функтора для выбора одного провода мы должны использовать какой-то другой «взвешивающий» функтор $W : \mathcal{J} \to \mathcal{V}$ для выбора более толстого «цилиндра». Такой утяжеленный конус с вершиной x является элементом множества естественных преобразований:

$$[\mathcal{J}, \mathbf{Set}](W, \mathcal{C}(x, D-))$$

Взвешенный предел, также известный как *индексированный предел*, $\lim^W D$, затем определяется как универсальный взвешенный конус. Это означает, что для любого взвешенного конуса с вершиной x существует

единственный морфизм от x к $\lim^W D$, факторизующий его. Факторизация гарантируется естественностью изоморфизма, определяющего взвешенный предел:

$$\mathcal{C}(x, \lim^W D) \cong [\mathcal{J}, \mathbf{Set}](W, \mathcal{C}(x, D-))$$

Регулярный невзвешенный предел часто называют *конусным* пределом, и он соответствует использованию постоянного функтора в качестве веса.

Это определение можно почти дословно перевести в обогащенную настройку, заменив $\mathbf{Set}$ на $\mathcal{V}$:

$$\mathcal{C}(x, \lim^W D) \cong [\mathcal{J}, \mathcal{V}](W, \mathcal{C}(x, D-))$$

Разумеется, значение символов в этой формуле изменено. Обе стороны теперь являются объектами в $\mathcal{V}$. Левая часть — это $\hom$-объект в $\mathcal{C}$, а правая часть — это объект естественных преобразований между двумя $\mathcal{V}$-функторами.

Двойственным образом, взвешенный ко-предел определяется естественным изоморфизмом:

$$\mathcal{C}(\operatorname{colim}^W D, x) \cong [\mathcal{J}^{\mathsf{op}}, \mathcal{V}](W, \mathcal{C}(D-, x))$$

Здесь, ко-предел взвешивается функтором $W : \mathcal{J}^{\mathsf{op}} \to \mathcal{V}$ от противоположной категории.

Взвешенные (ко-)пределы, как в обычных, так и в обогащенных категориях, играют фундаментальную роль: их можно использовать для переформулировки многих знакомых конструкций, таких как (ко-)концы, расширения Кана и т.д.

20.6 Концы как взвешенные пределы

Конец имеет много общего с произведением или, в более общем смысле, с пределом. Если приглядеться повнимательнее, проекции $\pi_x : e \to P\langle a, a\rangle$ образуют стороны конуса; за исключением того, что вместо коммутативных треугольников появляются клинья. Оказывается, можно выразить концы как взвешенные пределы. Преимущество этой формулировки в том, что она работает и в обогащенной среде.

Мы видели, что конец $\mathcal{V}$-значного $\mathcal{V}$-профунктора можно определить, используя более фундаментальное понятие сверх-естественного преобразования. Это, в свою очередь, позволило определить объект естественных преобразований, что позволило, далее, определить взвешенные пределы. Теперь можно продолжить и расширить определение конца для работы с более общим $\mathcal{V}$-функтором смешанной вариантности со значениями в $\mathcal{V}$-категории $\mathcal{D}$:

$$P : \mathcal{C}^{\mathsf{op}} \otimes \mathcal{C} \to \mathcal{D}$$

Будем использовать этот функтор в качестве диаграммы в $\mathcal{D}$.

В этой ситуации момент математики начинают беспокоиться о размерностях. В конце концов, мы встраиваем целую категорию — приведенную в соответствие — как единую диаграмму $\mathcal{D}$. Чтобы избежать проблем с размерностью, мы просто предполагаем, что $\mathcal{C}$ является малой; то есть ее объекты образуют множество.

Мы хотим взять взвешенный предел этой диаграммы, определенной посредством P. Вес должен быть $\mathcal{V}$-функтором $\mathcal{C}^{\mathsf{op}} \otimes \mathcal{C} \to \mathcal{V}$. В нашем распоряжении всегда есть один такой функтор, hom-функтор $\mathrm{Hom}_{\mathcal{C}}$. Будем использовать его для определения конца как взвешенного предела:

$$\int_c P\langle c, c\rangle = \lim^{\mathrm{Hom}} P$$

Сначала, убедимся, что эта формула работает в обычной (**Set**-обогащенной) категории. Поскольку концы определяются своим свойством отображения-внутрь, рассмотрим отображение от произвольного объекта d к взвешенному пределу и воспользуемся стандартным приемом Йонеды, чтобы показать изоморфизм. По определению, имеем:

$$\mathcal{D}(d, \lim^{\mathrm{Hom}} P) \cong [\mathcal{C}^{\mathsf{op}} \times \mathcal{C}, \mathbf{Set}](\mathcal{C}(-,=), \mathcal{D}(d, P(-,=))$$

Можно переписать множество естественных преобразований как конец над парами объектов $\langle c, c'\rangle$:

$$\int_{\langle c,c'\rangle} \mathbf{Set}(C(c,c'), \mathcal{D}(d, P\langle c, c'\rangle))$$

Используя теорему Фубини, это эквивалентно повторяющемуся концу:

$$\int_c \int_{c'} \mathbf{Set}(C(c,c'), \mathcal{D}(d, P\langle c, c'\rangle))$$

Теперь можно применить лемму ниндзя Йонеды, чтобы выполнить интегрирование по c'. В результате имеем:

$$\int_c \mathcal{D}(d, P\langle c, c\rangle) \cong \mathcal{D}(d, \int_c P\langle c, c\rangle)$$

где была использована непрерывность, чтобы подтолкнуть конец под $\hom$-функтором. Поскольку d было произвольным, заключаем, что для обычных категорий:

$$\lim^{\mathrm{Hom}} P \cong \int^c P\langle c, c\rangle$$

Это оправдывает использование взвешенного предела для определения конца в обогащенном случае.

Аналогичная формула работает для ко-конца, за исключением того, что ко-предел используется с $\hom$-функтором в противоположной категории $\mathrm{Hom}_{\mathcal{C}^{\mathrm{op}}}$, в качестве веса:

$$\int_c P\langle c, c\rangle = \mathrm{colim}^{\mathrm{Hom}_{\mathcal{C}^{\mathrm{op}}}} P$$

Упражнение 20.6.1. *Покажите, что для обычных* **Set***-обогащенных категорий определение взвешенного ко-предела для ко-конца воспроизводит предыдущее определение. Подсказка*: *используйте свойство отображения-вне для ко-конца.*

Упражнение 20.6.2. *Покажите, что, пока существуют обе стороны, следующие тождества выполняются в обычных* (**Set***-обогащенных*) *категориях* (*их можно обобщить на обогащенную среду*):

$$\lim^W D \cong \int_{j:\mathcal{J}} Wj \pitchfork Dj$$
$$\mathrm{colim}^W D \cong \int^{j:\mathcal{J}} Wj \cdot Dj$$

Подсказка: *Используйте отображение-внутрь/вне с приемом Йонеды и определением степени и ко-степени.*

20.7 Расширения Кана

Мы видели, как выражать пределы и копределы в качестве *расширений* Кана, используя функтор от сингулярной категории 1. Взвешенные пределы позволяют избавиться от сингулярности, а разумный выбор веса позволяет выразить расширения Кана в терминах взвешенных пределов.

Сначала, разработаем формулу для обычных, **Set**-обогащенных категорий. Правое расширение Кана определяется как:

$$(\mathrm{Ran}_P F)e \cong \int_c \mathcal{B}(e, Pc) \pitchfork Fc$$

Будем рассматривать отображение в него от произвольного объекта d. Вывод следует путем применением простых шагов, в основном за счет расширения определений.

Начнем с:

$$\mathcal{D}(d, (\mathrm{Ran}_P F)e)$$

и подставим определение расширения Кана:

$$\mathcal{D}(d, \int_c \mathcal{B}(e, Pc) \pitchfork Fc)$$

Используя непрерывность hom-функтора, можно вытащить (наружу) конец:

$$\int_c \mathcal{D}(d, \mathcal{B}(e, Pc) \pitchfork Fc)$$

Затем используем определение степени:

$$\mathcal{D}(d, A \pitchfork d') \cong \mathbf{Set}(A, \mathcal{D}(d, d'))$$

получая:

$$\mathcal{D}(d, \int_c \mathcal{B}(e, Pc) \pitchfork Fc) \cong \int_c \mathbf{Set}(\mathcal{B}(e, Pc), \mathcal{D}(d, Fc))$$

Это можно записать как множество естественных преобразований:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{B}(e, P-), \mathcal{D}(d, F-))$$

Взвешенный предел также определяется через множество естественных преобразований:

$$\mathcal{D}(d, \lim^W F) \cong [\mathcal{C}, \mathbf{Set}](W, D(d, F-))$$

приводя к конечному результату:

$$D(d, \lim^{B(e,P-)} F)$$

Поскольку d было произвольным, можно использовать прием Йонеды, чтобы заключить:

$$(\mathrm{Ran}_P F)e = \lim^{B(e,P-)} F$$

Эта формула становится определением правого расширения Кана в обогащенной среде.

Точно так же, левое расширение Кана можно определить как взвешенный ко-предел:

$$(\mathrm{Lan}_P F)e = \mathrm{colim}^{B(e,P-)} F$$

Упражнение 20.7.1. *Выведите формулу левого расширения Кана для обычных категорий.*

20.8 Полезные формулы

- Лемма Йонеды:
$$\int_x [\mathcal{C}(c,x), Fx] \cong Fc$$
- Взвешенный предел:
$$\mathcal{C}(x, \lim^W D) \cong [\mathcal{J}, \mathcal{V}](W, \mathcal{C}(x, D-))$$
- Взвешенный ко-предел:
$$\mathcal{C}(\mathrm{colim}^W D, x) \cong [\mathcal{J}^{\mathsf{op}}, \mathcal{V}](W, \mathcal{C}(D-, x))$$
- Правое расширение Кана:
$$(\mathrm{Ran}_P F)e = \lim^{B(e,P-)} F$$
- Левое расширение Кана:
$$(\mathrm{Lan}_P F)e = \mathrm{colim}^{B(e,P-)} F$$

Предметный указатель