И тайное становится явным

(на примере оптики)

Bartosz Milewski

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

Введение

Типы данных могут содержать скрытую информацию. Что-то из этого можно извлечь, а что-то спрятать навсегда. Мы собираемся докопаться до сути этого механизма.

Ни один из типов данных не был поврежден при извлечении их «секретов».

Никакое принуждение при этом не использовалось.

Речь идет, конечно же, об `unsafeCoerce`, который не рекомендуется использовать.

Скрытие реализации

Реализация функции, даже если она доступна для просмотра программистом, скрыта от самой программы.

Что за функция, с наводящим на размышления именем double, скрывается внутри?

x	double x
2	4
3	6
-1	-2

Лучшее предположение: скрывается функция удвоения. И, вероятно, она реализована как:

```
double x = 2 * x
```

Как же нам извлечь это скрытое значение? Мы можем просто обратиться к ней, предъявляя в качестве аргумента умножения единицу:

```
double 1
> 2
```

Возможно ли, что она реализован по-другому (при условии, что мы уже проверили ее для всех значений аргумента)? Конечно! Может быть, это прибавление единицы, умножение на два, а затем вычитание двух. Но какой бы ни была фактическая реализация, она должна быть эквивалентна умножению на два. Можно заявить, что данная реализация *изоморфна* умножению на два.

Функторы

Функтор — это тип данных, который скрывает объекты типа `a`. Быть функтором означает, что его содержимое можно изменять с помощью функции. То есть, если имеется функция `a -> b` и *функтор-хранилище*[1] объектов `a`, то можно создать функтор-хранилище объектов `b`. В Haskell класс `Functor` определяется как конструктор типов, оснащенный методом `fmap`:

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

Стандартным примером функтора является список объектов `a`. Реализация `fmap` применяет функцию `g` к каждому своему элементу:

[1] Так (термин «functorful») названа «область» (и «ко-область») функтора (*прим. переводчика*)

```haskell
instance Functor [] where
  fmap g []       = []
  fmap g (a : as) = (g a) : fmap g as
```

То, что является функтором, не гарантирует, что оно на самом деле «содержит» значения типа `a`. Но у большинства структур данных, являющихся функторами, есть определенные средства доступа к их содержимому. Когда они произведут соответствующие действия, можно убедиться, что они изменили свое содержимое, после применения `fmap`. Но существуют и «хитрые» функторы.

Например, реализация `Maybe a` фактически информирует: возможно, у меня есть аргумент `a`, а может быть, и нет. Но, если он имеется, `fmap` заменит его на `b`.

```haskell
instance Functor Maybe where
  fmap g Empty    = Empty
  fmap g (Just a) = Just (g a)
```

Функция, которая производит значения типа `a`, также является функтором. Функция `e -> a` «сообщает»: я выдам значение типа `a`, если вы предоставите мне значение типа `e`. Имея производителя `a`, можно заменить его на производителя `b`, проведя последующую его компоновку с функцией `g :: a -> b`:

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

Существует также самый сложный из функторов — функтор `IO` , который «говорит»: поверьте, у меня есть `a`, но я никак не могу вам сказать, что это такое (если, конечно, вы не смотрите на экран или не открываете файл, на который перенаправляется вывод).

Продолжения

Суть продолжения (как механизма приостановки и последующего возобновления некоторого процесса) можно выразить фразой: «не звоните нам, мы сами вам позвоним». Вместо того, чтобы предоставлять значение типа `a` напрямую, оно просит предоставить ему обработчик — функцию, которая использует `a` и возвращает результат желаемого типа:

```
type Cont a = forall r. (a -> r) -> r
```

Можно предположить, что продолжение либо скрывает значение типа `a`, либо имеет средства для его создания по запросу. На самом деле можно извлечь это значение, вызвав продолжение с тождественной функцией:

```
runCont  :: Cont a -> a
runCont k = k id
```

На самом деле `Cont a` во всех смыслах эквивалентно `a` — оно изоморфно ему. Действительно, учитывая значение типа `a`, можно создать продолжение как замыкание:

```
mkCont  :: a  -> Cont a
mkCont a = \k -> k a
```

Функции `runCont` и `mkCont` являются обратными друг другу, что устанавливает изоморфизм `Cont a ~ a`.

Лемма Йонеды

Рассмотрим вариацию на тему продолжений. Подобно продолжению, следующая функция принимает обработчик для `a`, но вместо некоторого `x`, она создает функтор-хранилище всех `x`:

```
type Yo f a = forall x. (a -> x) -> f x
```

Так же, как продолжение скрывало некоторое значение типа `a`, этот тип данных скрывает функтор-хранилище всех `a`. Можно легко получить это функтор-хранилище, используя тождественную функцию в качестве обработчика:

```
runYo  :: Functor f => Yo f a -> f a
runYo g = g id
```

И наоборот, по заданному функтор-хранилищу всех `a`, можно реконструировать `Yo f a`, определив замыкание, над которым `fmap`-ы являются обработчиком:

```
mkYo   :: Functor f => f a -> Yo f a
mkYo fa = \g -> fmap g fa
```

Опять же, две функции, `runYo` и `mkYo`, являются обратными друг другу, что устанавливает очень важный изоморфизм, называемый *леммой Йонеды*:

```
Yo f a ~ f a
```

Оба продолжения и лемма Йонеды определяются как полиморфные функции. `forall x` в их определении означает, что они используют одну и ту же формулу для всех типов (это называется параметрическим полиморфизмом). Функция, работающая для *любого типа*, не может делать никаких предположений о свойствах этого типа. Все, что она может сделать, это посмотреть, как упакован этот тип: передается ли он внутри списка, функции или каким-то другим способом. Другими словами, такая функция может использовать информацию о форме, в которой передается полиморфный аргумент.

Экзистенциальные типы

Можно не говорить об экзистенциальных типах, если не упоминать Жан-Поль Сартра (Jean-Paul Sartre)[2].

[2]Основа его философии — идея экзистенциализма: «существование предшествует сущности» (*прим. переводчика*)

Экзистенциальный тип данных отражает то, что тип существует, но неизвестно какой. Собственно, тип был известен в момент его создания, но затем все его следы были стерты. Это возможно только в том случае, если конструктор данных сам является полиморфным. Он принимает любой тип и тут же забывает характеристику этого типа.

Вот пример крайности: экзистенциальная черная дыра. Что бы ни попало в нее (через конструктор `BH`), оно никогда не сможет выбраться.

```
data BlackHole = forall a. BH a
```

Даже фотон не может покинуть черную дыру:

```
bh :: BlackHole
bh  = BH "Photon"
```

Известны типы данных, конструкторы которых можно отменить — например, с помощью сопоставления с образцом. В теории, типы определяются правилами введения и исключения, предписывающими, как конструировать и как деконструировать типы.

Но экзистенциальные типы стирают тип аргумента, который был передан (полиморфному) конструктору, поэтому их нельзя деконструировать. Однако не все потеряно. В физике рассматривается излучение Хокинга, выходящее из черной дыры. В программировании, даже если нет возможности заглянуть в экзистенциальный тип, можно извлечь некоторую информацию об окружающей его структуре.

Рассмотрим пример: известно, что имеется список, но какой?

```
data SomeList = forall a. SomeL [a]
```

Оказывается, для отмены полиморфного конструктора можно использовать полиморфную функцию. Существуют функции, которые действуют на списки произвольного типа, например `length`:

```
length :: forall a. [a] -> Int
```

Использование полиморфной функции для «отмены» полиморфного конструктора не раскрывает экзистенциальный тип:

```
len           :: SomeList -> Int
len (SomeL as) = length as
```

Действительно, это работает:

```
someL :: SomeList
someL  = SomeL [1..10]
> len someL
> 10
```

Извлечение хвоста списка также является полиморфной функцией. Можно использовать его в `SomeList`, не раскрывая тип `a`:

```
trim                :: SomeList -> SomeList
trim (SomeL [])      = SomeL []
trim (SomeL (a: as)) = SomeL as
```

Здесь, хвост (непустого) списка сразу прячется внутри `SomeList`, скрывая, таким образом, тип `a`.

Но следующее не будет компилироваться, потому что «обнажает» `a`:

```
bad           :: SomeList -> a
bad (SomeL as) = head as
```

Производитель/Потребитель

Экзистенциальные типы часто определяются с помощью пар производитель/потребитель. Производитель может создавать значения скрытого типа, а потребитель может их потреблять. Роль клиента экзистенциального типа состоит в том, чтобы активировать производителя (например, предоставив некоторый ввод) и передать результат (не глядя на него) непосредственно потребителю.

Рассмотрим простой пример. Производитель — это просто значение скрытого типа `a`, а потребитель — функция, потребляющая этот тип:

```
data Hide b = forall a. Hide a (a -> b)
```

Все, что может сделать клиент, — это сопоставить потребителя с производителем:

```
unHide           :: Hide b -> b
unHide (Hide a f) = f a
```

Вот как можно использовать этот экзистенциальный тип (здесь `Int` — видимый тип, а `Char` — скрытый):

```
secret :: Hide Int
secret  = Hide 'a' (ord)
```

Функция `ord` — это потребитель, который превращает символ в его ASCII-код:

```
> unHide secret
> 97
```

Лемма ко-Йонеды

Существует двойственность между полиморфными типами и экзистенциальными типами. Она коренится в двойственности между кванторами всеобщности («для всех», ∀) и кванторами существования («существует», ∃).

Лемма Йонеды — это утверждение о полиморфных функциях. Двойственная ему лемма Ко-Йонеды — это утверждение об экзистенциальных типах. Рассмотрим следующий тип, который объединяет производителя всех `x` (функтор-хранилище всех `x`) с потребителем (функция, которая преобразует все `x` во все `a`):

```
data CoYo f a = forall x. CoYo (f x) (x -> a)
```

Что этот тип данных скрытно кодирует? Единственное, что может сделать клиент `CoYo`, — это применить потребителя к производителю. Поскольку производитель имеет форму функтора, применение действует через `fmap`:

```
unCoYo             :: Functor f => CoYo f a -> f a
unCoYo (CoYo fx g) = fmap g fx
```

Результатом является функтор-хранилище всех `a`. И наоборот, для функтор-хранилища, содержащего значения типа `a`, можно сформировать `CoYo`, путем сопоставления с тождественной функцией:

```
mkCoYo   :: Functor f => f a -> CoYo f a
mkCoYo fa = CoYo fa id
```

Эта пара взаимообратных `unCoYo` и `mkCoYo`, свидетельствует об изоморфизме

```
CoYo f a ~ f a
```

Другими словами, `CoYo f a` тайно скрывает функтор-хранилище, полное значений `a`.

Контравариантные потребители

Неформальным терминам производитель и потребитель можно придать более строгое значение. Производитель — это тип данных, который ведет себя как функтор. Функтор оснащен функцией `fmap`, которая позволяет превратить производителя всех `a` в производителя всех `b` с помощью функции `a -> b`.

И наоборот, чтобы превратить потребителя `a` в потребителя `b`, нужна функция, действующая в обратном направлении, `b -> a`. Эта идея закодирована в определении *контравариантного функтора*:

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

Существует также контравариантная версия леммы ко-Йонеды, которая меняет местами роли производителя и потребителя:

```
data CoYo' f a = forall x. CoYo' (f x) (a -> x)
```

Здесь, `f` контравариантный функтор, поэтому `f x` — потребитель всех `x`. Ему соответствует производитель всех `x`, функция `a -> x`.

Как и выше, можно установить изоморфизм

```
CoYo' f a ~ f a
```

определив пару функций:

```
unCoYo'              :: Contravariant f => CoYo' f a -> f a
unCoYo' (CoYo' fx g) = contramap g fx

mkCoYo'   :: Contravariant f => f a -> CoYo' f a
mkCoYo' fa = CoYo' fa id
```

Экзистенциальная линза

Линза абстрагирует средство для фокусировки на части более крупной структуры данных. В функциональном программировании мы имеем дело с неизменяемыми данными, поэтому, чтобы что-то изменить, мы должны разложить большую структуру на фокус (часть, которую мы модифицируем) и остаток (остальное). Затем можно воссоздать измененную структуру данных, объединив новый фокус со старым остатком. Важное замечание состоит в том, что нас не волнует точный тип остатка. Это описание переводится непосредственно в следующее определение:

```
data Lens' s a =
  forall c. Lens' (s -> (c, a)) ((c, a) -> s)
```

Здесь, `s` — тип большей структуры данных, `a` — тип фокуса, а экзистенциально скрытый `c` — тип остатка. Линза строится из пары функций, первая из которых декомпозирует `s`, а вторая перекомпоновывает ее.

Для линзы, можно построить две функции, которые не раскрывают тип остатка. Первая называется `get`. Она извлекает фокус:

```
toGet               :: Lens' s a -> (s -> a)
toGet (Lens' frm to) = snd . frm
```

Вторая, называемая `set`, заменяет фокус новым значением:

```
toSet               :: Lens' s a -> (s -> a -> s)
toSet (Lens' frm to) = \s a -> to (fst (frm s), a)
```

Обратите внимание, что доступ к остатку невозможен. Следующее не будет компилироваться:

```
bad               :: Lens' s a -> (s -> c)
bad (Lens' frm to) = fst . frm
```

Но откуда известно, что пара из геттера и сеттера — это именно то, что скрыто в экзистенциальном определении линзы? Чтобы показать это, надо использовать лемму ко-Йонеды. Во-первых, необходимо идентифицировать производителя и потребителя `c` в нашем экзистенциальном определении. Для этого, заметьте, что функция, возвращающая пару `(c, a)`, эквивалентна паре функций, одна из которых возвращает `c`, а другая — `a`. Таким образом, можно переписать определение линзы в виде тройки функций:

```
data Lens' s a =
  forall c. Lens' (s -> c) (s -> a) ((c, a) -> s)
```

Первая функция является производителем всех `c`, поэтому остальные будут определять потребителя. Напомним контравариантную версию леммы о ко-Йонеде:

```
data CoYo' f s = forall c. CoYo' (f c) (s -> c)
```

Мы можем определить контравариантный функтор, который является потребителем всех `c`, и использовать его в нашем определении линзы. Этот функтор параметризуется двумя дополнительными типами `s` и `a`:

```
data F s a c = F (s -> a) ((c, a) -> s)
```

Это позволяет переписать линзу, используя представление ко-Йонеды с заменой `f` на (частично примененную) `F s a`:

```
type Lens' s a = CoYo' (F s a) s
```

Теперь можно использовать изоморфизм `CoYo' f s ~ f s`. Подставив в определение `F`, получим:

```
Lens' s a       ~ CoYo' (F s a) s
CoYo' (F s a) s ~ F s a s
F s a s         ~ (s -> a) ((s, a) -> s)
```

Мы распознаем две функции как геттер и сеттер. Таким образом, экзистенциальное представление линзы действительно изоморфно паре геттер/сеттер.

Линза с изменяющимся типом

Простая линза, с которой мы имели дело до сих пор, позволяет заменить фокус новым значением того же типа. Но вообще-то, новый фокус может быть и другого типа. В этом случае тип всего также изменится. Таким образом, линза, меняющая тип, имеет ту же функцию декомпозиции, но другую функцию рекомпозиции:

```
data Lens s t a b =
  forall c. Lens (s -> (c, a)) ((c, b) -> t)
```

Как и выше, эта линза изоморфна паре `get`/`set`, где `get` извлекает `a`:

```
toGet               :: Lens s t a b -> (s -> a)
toGet (Lens frm to) = snd . frm
```

а `set` заменяет фокус новым значением типа `b` для получения `t`:

```
toSet               :: Lens s t a b -> (s -> b -> t)
toSet (Lens frm to) = \s b -> to (fst (frm s), b)
```

Прочая оптика

Преимущество экзистенциального представления линз состоит в том, что оно легко обобщается на другую оптику. Идея состоит в том, что линза разлагает структуру данных на пару типов `(c, a)`, а пара — это тип-произведение, формально $c \times a$,

```
data Lens s t a b =
  forall c. Lens (s -> (c, a))
                 ((c, b) -> t)
```

Призма делает то же самое для тип-суммы. Сумма $c + a$ записывается в Haskell как `Either c a`. Имеем:

```
data Prism s t a b =
  forall c. Prism (s -> Either c a)
                  (Either c b -> t)
```

акже, можно комбинировать сумму и произведение в так называемом *аффинном* типе $c_1 + c_2 \times a$. Полученная оптика имеет два возможных остатка, `c1` и `c2`:

```
data Affine s t a b =
  forall c1 c2. Affine (s -> Either c1 (c2, a))
                       (Either c1 (c2, b) -> t)
```

Список оптик можно продолжать и продолжать.

Профункторы

Производитель может быть объединен с потребителем в единую структуру данных, называемую профунктором. Профунктор параметризуется двумя типами; то есть `p a b` является потребителем всех `a` и производителем всех `b`. Можно превратить потребителя `a` и производителя `b` в потребителя всех `s` и производителя всех `t`, используя пару функций, первая из которых работает в противоположном направлении:

```
class Profunctor p where
  dimap :: (s -> a) -> (b -> t) -> p a b -> p s t
```

Стандартным примером профунктора является функциональный тип `p a b = a -> b`. Действительно, можно определить для него функцию `dimap`, предварительно скомпоновав ее с помощью одной функции, а после, скомпоновать с помощью другой:

```
instance Profunctor (->) where
  dimap in out pab = out . pab . in
```

Профункторная оптика

Мы рассмотрели функции, которые были полиморфны по типам. Но полиморфизм не ограничивается типами. Вот определение функции, полиморфной по профункторам:

```
type Iso s t a b = forall p. Profunctor p =>
  p a b -> p s t
```

Эта функция принимает любого производителя всех `b`, использующего все `a`, и превращает его в производителя всех `t`, использующего все `s`. Поскольку она ничего больше не знает о своем аргументе, единственное, что может сделать эта функция, — это применить к нему `dimap`. Но для `dimap` требуется пара функций, поэтому эта профункторно-полиморфная функция должна скрывать такую пару:

```
s -> a
b -> t
```

Действительно, по такой паре можно реконструировать ее реализацию:

```
mkIso    :: (s -> a) -> (b -> t) -> Iso s t a b
mkIso g h = \p -> dimap g h p
```

Вся остальная оптика имеет соответствующую реализацию в виде профункторно-полиморфных функций. Основное преимущество этих представлений состоит в том, что они могут компоноваться с помощью простой композиции функций.

Итоговые тезисы

- Производители и потребители соответствуют ковариантным и контравариантным функторам.
- Экзистенциальные типы двойственны полиморфным типам.
- Экзистенциальная оптика объединяет производителей и потребителей в одном пакете
- В такой оптике производители декомпозируют, а потребители перекомпоновывают данные.
- Функции могут быть полиморфными по типам, функторам или профункторам.