ЛИНЕЙНЫЕ ЛИНЗЫ В HASKELL

Bartosz Milewski

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

Я всегда считал, что основными проблемами при разработке языка программирования являются управление ресурсами и параллелизм, и эти две проблемы связаны между собой. Если имеется возможность отслеживать право собственности на ресурсы, то можно быть уверенным, что синхронизация не потребуется, если у них один владелец.

Я долгое время пропагандировал управление ресурсами, сначала на C++, а затем на D (см. Приложение 3). Это реализовано в Rust как типы владения, и я с удовлетворением отмечаю, что в Haskell это реализовано как линейные типы.

В Haskell, по сути, решены проблемы конкурентности и параллелизма, перенаправлением мутации на выделенные монады, но управление ресурсами всегда было частью сложной задачи. Основное преимущество линейных типов в Haskell, помимо работы с внешними ресурсами, заключается в обеспечении безопасной мутации и управления памятью без GC (автоматического управления). Потенциально, это может привести к существенному увеличению производительности.

Это описание основано на дискуссиях с Арно Спиваком (Arnaud Spiwack), который рассказал о работе, проделанную им, над линейными типами и линейными линзами, некоторые из которых никогда ранее не документировались.

PDF-версия этого текста, вместе с полным кодом на Haskell, доступна на GitHub.

1 Линейные типы

Что такое линейная функция? Краткий ответ: линейная функция $a \multimap b$ «использует» свой аргумент *ровно один раз*. Однако это не вся правда, поскольку также имеется линейная тождественность $\mathsf{id}_a : a \multimap a$, которое, якобы, не использует a. Более детальный ответ заключается в том, что линейная функция использует свой аргумент ровно один раз, если она сама используется ровно один раз, и ее результат используется ровно один раз.

Осталось определить, что означает «быть использованным». Функция используется, когда она применяется к своему аргументу. Базовое значение, такое как `Int` или `Char`, используется при его вычислении, а алгебраический тип данных используется при сопоставлении с шаблоном, и используются все его сопоставленные компоненты.

Например, чтобы использовать линейную пару `(a, b)`, ее сопоставляют с образцом, а затем используют, как `a`, так и `b`. Чтобы использовать `Either a b`, его сопоставляют с образцом и используют соответствующий компонент, `a` или `b`, в зависимости от того, какая возможность была выбрана.

Как видите, за исключением базовых значений, линейный аргумент подобен горячей картофелине: вы «съедаете» его, передавая кому-то другому.

Так где же остановиться? Вот тут-то и происходит магическая манипуляция: каждый ресурс имеет специальный примитив, от которого происходит избавление, — дескриптор файла закрывается, память освобождается, массив замораживается.

Чтобы уведомить систему типов об уничтожении ресурса, линейная функция вернет значение внутри специального *неограниченного* типа `Ur`. Когда этот тип соответствует шаблону сопоставления, исходный ресурс окончательно уничтожается.

Например, для линейных массивов одним из таких примитивов является `toList`:

$$toList : \mathrm{Array}\, a \multimap \mathrm{Ur}[a]$$

В Haskell, линейные стрелки аннотируются указанием кратности 1:

```
toList :: Array a %1-> Ur [a]
```

Точно так же, магия, в первую очередь, используется для создания ресурса. Для массивов это происходит внутри примитива `fromList`.

$$fromList : [a] \to (\mathrm{Array}\, a \multimap \mathrm{Ur}\, b) \multimap \mathrm{Ur}\, b$$

или, используя синтаксис Haskell:

```
fromList :: [a] -> (Array a %1-> Ur b)
                                %1-> Ur b
```

Тот тип управления ресурсами, который я рекламировал в C++, основан на области действия. Ресурс был инкапсулирован в интеллектуальный указатель, который автоматически уничтожался при выходе из области действия.

В случае линейных типов, роль области видимости играет линейная функция, предоставляемая пользователем; вот продолжение:

```
(Array a %1-> Ur b)
```

Примитив `fromList` обеспечивает использование этой, предоставленной пользователем, функции ровно один раз и возврат ее неограниченного результата. Клиент обязан использовать массив ровно один раз (например, вызвав `toList`). Это обязательство закодировано в типе продолжения, принимаемом `fromList`.

2 Линейная линза: экзистенциальная форма

Линза абстрагирует идею сосредоточения внимания на части более крупной структуры данных. Он используется для доступа или изменения его *фокуса*. Экзистенциальная форма линзы состоит из двух функций: расщепления источника на фокус и дополнение; а другая заменяет фокус новым значением и создает новое целое. Нас не интересует реальный тип дополнения, поэтому мы сохраняем его как экзистенциальный.

Линейную линзу можно представлять, рассматривая ее источник как ресурс. Акт разделения ее на фокус и дополнение деструктивен: он потребляет свой источник для производства двух новых ресурсов (разбивает одну горячую картофелину `s` на две горячие части: дополнение (кожура) `c` и фокус (сердцевина) `a`).

И обратно, часть, которая перестраивает цель `t`, должна использовать как дополнение `c`, так и новый фокус `b`.

В итоге получим следующую реализацию на Haskell:

```
data LinLensEx a b s t where
  LinLensEx :: (s %1-> (c, a))
                       -> ((c, b) %1-> t)
                       -> LinLensEx a b s t
```

Экзистенциальный тип Haskell соответствует категорному ко-концу, поэтому приведенное выше определение эквивалентно следующему:

$$Labst = \int^{c} (s \multimap c \otimes a) \times (c \otimes b \multimap t)$$

Я использую обозначение «леденца» $(\multimap)$ для hom-множества в моноидальной категории с тензорным произведением $\otimes$.

Важным свойством моноидальной категории является то, что ее тензорное произведение не имеет пары проекций; и единичный объект не является терминальным. В частности, морфизм $s \multimap c \otimes a$ не может быть разложен в произведение двух морфизмов $(s \multimap c) \times (s \multimap a)$.

Однако, в *замкнутой* моноидальной категории можно создать отображение-из тензорного произведения:

$$c \otimes b \multimap t \cong c \multimap (b \multimap t)$$

Поэтому можно переписать экзистенциальную линзу так:

$$Labst = \int^{c} (s \multimap c \otimes a) \times (c \multimap (b \multimap t))$$

а затем применить лемму ко-Йонеды, чтобы получить:

$$s \multimap ((b \multimap t) \otimes a)$$

В отличие от случая стандартной линзы, эту форму нельзя разделить на пару get/set.

Интуиция заключается в том, что линейная линза позволяет использовать объект s, но оставляет за собой обязанность использовать как сеттер (set) $b \multimap t$, так и фокус a. Нельзя просто извлечь a, потому что

это оставит зияющую дыру в объекте. Нужно подключить его к новому объекту b, и это то, что позволяет сделать сеттер.

Перевод этой формулы на Haskell (условно, с обратным расположением пар аргументов):

```
type LinLens s t a b = s %1-> (b %1-> t, a)
```

Манипуляции Йонеды преобразуются в пару функций Haskell. Заметим, что, как и в трюке Ко-Йонеды, экзистенциал c заменяется линейной функцией $b \multimap t$.

```
fromLinLens   :: forall s t a b .
                   LinLens s t a b
                     -> LinLensEx a b s t
fromLinLens h  = LinLensEx f g
  where
    f         :: s  %1-> (b %1-> t, a)
    f          = h
    g         :: (b %1-> t, b) %1-> t
    g (set, b) = set b
```

Обратное отображение можно представить так:

```
toLinLens                    :: LinLensEx
                                  a b s t ->
                                  LinLens s t a b
toLinLens (LinLensEx f g) s =
                case f s of
                  (c, a) ->
                        (\b -> g (c, b), a)
```

3 Профункторное представление

Каждая оптика имеет профункторное представление, и линейная линза не является исключением. Говоря категорно, профунктор — это функтор от категории произведения $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$ к $\mathbf{Set}$. Он отображает пары объектов к множествам, а пары морфизмов — к функциям. Поскольку рассматривается моноидальная категория, морфизмы являются линейными

функциями, а отображения между множествами — регулярными функциями (см. Приложение 1). Таким образом, действие профунтора p на морфизмы является функцией:

$$(a' \multimap a) \to (b \multimap b') \to pab \to pa'b'$$

На Haskell:

```
class Profunctor p where
  dimap :: (a' %1-> a) -> (b %1-> b')
                       -> p a b -> p a' b'
```

Модуль Тамбары (он же, сильный профунктор) — это профунктор, снабженный следующим отображением:

$$\alpha_{abc} : pab \to p(c \otimes a)(c \otimes b)$$

естественным в a и b, ди-естественным в c. На Haskell, это преобразуется в полиморфную функцию:

```
class (Profunctor p) => Tambara p where
  alpha :: forall a b c. p a b
                      -> p (c, a) (c, b)
```

Линейная линза $Labst$, сама по себе, является модулем Тамбары, для фиксированных ab. Чтобы это показать, построим отображение:

$$\alpha_{stc} : Labst \to Lab(c \otimes s)(c \otimes t)$$

Раскрывая это проеделение, получаем:

$$\int^{c''} (s \multimap c'' \otimes a) \times (c'' \otimes b \multimap t) \to \int^{c'} (c \otimes s(c' \otimes a) \times (c' \otimes b(c \otimes t)$$

Ввиду ко-непрерывности hom-множества в **Set**, отображение-из ко-конца эквивалентно концу:

$$\int_{c''} \left((s \multimap c'' \otimes a) \times (c'' \otimes b \multimap t) \to \int^{c'} (c \otimes s(c' \otimes a) \times (c' \otimes b(c \otimes t) \right)$$

Учитывая пару линейных стрелок слева, требуется построить ко-конец справа. Это можно сделать, подняв сначала обе стрелки с помощью $(c \otimes -)$. Тогда, получаем:

$$(c \otimes s \multimap c \otimes c'' \otimes a) \times (c \otimes c'' \otimes b \multimap c \otimes t)$$

Можем вставить их в ко-конец справа при $c' = c \otimes c''$.

На Haskell, создадим экземпляр класса `Profunctor` для линейной линзы:

```
instance Profunctor (LinLensEx a b) where
  dimap f' g' (LinLensEx f g) = LinLensEx
                                  (f . f') (g' . g)
```

и экземпляр для `Tambara`:

```
instance Tambara (LinLensEx a b) where
  alpha (LinLensEx f g) =
         LinLensEx (unassoc . second f)
                                   (second g . assoc)
```

Линейные линзы могут быть скомпонованы и имеется тождественная линейная линза:

$$\mathsf{id}_{ab} : Labab = \int^{c} (a \multimap c \otimes a) \times (c \otimes b(b)$$

задаваемая путем инъекции пары $(\mathsf{id}_a, \mathsf{id}_b)$ при $c = I$, моноидальной единицы.

На Haskell можно построить тождественную линзу, используя левый унитор (см. Приложение 1):

```
idLens :: LinLensEx a b a b
idLens  = LinLensEx unlunit lunit
```

Профункторное представление линейной линзы задается концом над модулями Тамбары:

$$Labst \cong \int_{p:Tamb} pab \to pst$$

На Haskell это транслируется к типу функций, полиморфных в модулях Тамбары:

```
type PLens a b s t = forall p. Tambara p =>
                                  p a b -> p s t
```

Преимущество этого представления в том, что оно позволяет компоновать линейные линзы, используя простую композицию функций.

Вот категорное доказательство эквивалентности. Слева направо: для тройки $(c, f : s \multimap c \otimes a, g : c \otimes b \multimap t)$ конструируем:

$$pab \xrightarrow{\alpha_{abc}} p(c \otimes a)(c \otimes b) \xrightarrow{pfg} pst$$

И наоборот, для полиморфной (в модулях Тамбары) функции $pab \to pst$, можно применить ее к тождественной оптике id_{ab} и получить $Labst$.

На Haskell эту эквивалентность подтверждает следующая пара функций:

```
fromPLens  :: PLens a b s t -
> LinLensEx a b s t
fromPLens f = f idLens

toPLens                        :: LinLensEx
                                    a b s t ->
                                     PLens a b s t
toPLens (LinLensEx f g) pab = dimap f g (alpha pab)
```

4 Представление Ван Лаарховена

Подобно обычным линзам, линейные линзы имеют функторно-полиморфную кодировку Ван Лаарховена. Разница в том, что приходится использовать эндофункторы в моноидальной подкатегории, где все стрелки линейны:

```
class Functor f where
  fmap :: (a %1-> b) %1-> f a %1-> f b
```

Как и обычные функторы Haskell, линейные функторы являются сильными. Это следует из того, что замкнутая моноидальная категория является само-обогащенной. Определим прочность как:

```
strength         :: Functor f =>
                        (a, f b) %1-> f (a, b)
strength (a, fb) = fmap (eta a) fb
```

где `eta` – единица каррированного сопряжения (см. Приложение 1).

Согласно этому определению, кодирование линейных линз Ван Лаарховена выглядит следующим образом:

```
type VLL s t a b = forall f.
                   Functor f =>
                   (a %1-> f b) ->
                        (s %1-> f t)
```

Об эквивалентности двух кодировок свидетельствует пара функций:

```
toVLL      :: LinLens s t a b -> VLL s t a b
toVLL lns f = fmap apply . strength
                         . second f . lns

fromVLL      :: forall s t a b. VLL s t a b ->
                                LinLens s t a b
fromVLL vll s = unF (vll (F id) s)
```

Здесь, функтор `F` определяется как линейная пара (тензорное произведение):

```
data F a b x where
  F :: (b %1-> x) %1-> a %1-> F a b x

unF          :: F a b x %1-> (b %1-> x, a)
unF (F bx a) = (bx, a)
```

с очевидной реализацией `fmap`

```
instance Functor (F a b) where
  fmap f (F bx a) = F (f . bx) a
```

Категорный вывод представления Ван Лаарховена содержится в Приложении 2.

5 Линейная оптика

Линейные линзы — лишь один пример более общей линейной оптики. Линейная оптика определяется действием моноидальной категории $\mathcal{M}$ на (возможно, ту же самую) моноидальную категорию $\mathcal{C}$:

$$\bullet : \mathcal{M} \times \mathcal{C} \to \mathcal{C}$$

В частности, можно определить линейные призмы и линейные траверсали, используя действия с помощью копроизведения или степенных рядов.

Экзистенциальная форма задается как:

$$Oabst = \int^{m:\mathcal{M}} (s \multimap m \bullet a) \times (m \bullet b \multimap t)$$

Существует соответствующее представление Тамбары со следующей структурой Тамбары:

$$\alpha_{abm} : pab \to p(m \bullet a)(m \bullet b)$$

Кстати, два hom-множества в определении оптики могут относиться к двум разным категориям, поэтому можно смешивать линейные и нелинейные стрелки в одной оптике.

6 Приложение 1: замкнутая моноидальная категория в Haskell

С появлением линейных типов, внутри Haskell скрываются две основные категории. У них одни и те же объекты — типы Haskell — но в моноидальной категории меньше стрелок. Это линейные стрелки $a \multimap b$. Они могут быть скомпонованы:

```
(.) :: (b %1-> c) %1-> (a %1-> b)
                       %1-> a %1-> c
(f . g) x = f (g x)
```

и для каждого объекта существует тождественная стрелка:

```
id  :: a %1-> a
id a = a
```

В общем случае, тензорное произведение в моноидальной категории является бифунктором: $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. В Haskell тензорное произведение $\otimes$ отождествляется со встроенным произведением (a, b). Разница в том, что в рамках моноидальной категории это произведение не имеет проекций. Не существует линейной стрелки $(a, b) \multimap a$ или $(a, b) \multimap b$. Следовательно, не существует диагонального отображения $a \multimap (a, a)$, а единичный объект $()$ не является терминальным: отсутствует стрелка $a \multimap ()$.

Определим действие бифунктора на пару линейных стрелок целиком в пределах моноидальной категории:

```
class Bifunctor p where
  bimap  :: (a %1-> a') %1-> (b %1-> b')
                         %1-> p a b
                         %1-> p a' b'
  first  :: (a %1-> a') %1-> p a b %1-> p a' b
  first f = bimap f id
  second :: (b %1-> b') %1-> p a b %1-> p a b'
  second  = bimap id
```

Произведение сам по себе является экземпляром этого линейного бифунктора:

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

Тензорное произведение должно удовлетворять условиям связности — законам ассоциативности и единицы:

```
assoc               :: ((a, b), c) %1->
                                (a, (b, c))
assoc   ((a, b), c) = (a, (b, c))
unassoc             :: (a, (b, c)) %1->
                                ((a, b), c)
unassoc (a, (b, c)) = ((a, b), c)

lunit               :: ((), a) %1-> a
lunit   ((), a)      = a
unlunit             :: a %1-> ((), a)
unlunit a            = ((), a)
```

В Haskell тип стрелок между любыми двумя объектами также является объектом. Категория, в которой это справедливо, называется замкнутой. Эта идентификация является следствием каррированного сопряжения между произведением и функциональным типом. В замкнутой моноидальной категории существует соответствующее сопряжение между тензорным произведением и объектом линейных стрелок. Отображение-из тензорного произведения эквивалентно отображению-в функциональный объект. В Haskell об этом свидетельствует пара отображений:

```
curry           :: ((a, b) %1-> c) %1->
                           (a %1-> (b %1-> c))
curry   f x y    = f (x, y)

uncurry         :: (a %1-> (b %1-> c)) %1->
                                   ((a, b) %1-> c)
uncurry f (x, y) = f x y
```

Каждое сопряжение также определяет пару единицы и ко-единицы естественных преобразований:

```
eta         :: a %1-> b %1-> (a, b)
eta   a b    = (a, b)

apply       :: (a %1-> b, a) %-> b
apply (f, a) = f a
```

Можно, например, использовать единицу для реализации бесточечного отображения линз:

```
toLinLens                  :: LinLensEx a b s t
                              -> LinLens s t a b
toLinLens (LinLensEx f g) = first ((g .) . eta)
                                              . f
```

Наконец, замечание об определении профунтора в Haskell:

```
class Profunctor p where
  dimap :: (a' %1-> a) -> (b %1-> b')
                       -> p a b -> p a' b'
```

Отметим смешивание двух типов стрелок. Оно связано с тем, что профунктор определяется как отображение $\mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathbf{Set}$. Здесь, $\mathcal{C}$ — моноидальная категория, поэтому стрелки в ней линейные. Но `p a b` — это просто множество, а отображение `p a b -> p a' b'` — это просто регулярная функция. Аналогично, тип:

```
(a' %1-> a)
```

рассматривается в $\mathcal{C}$ не как объект, а как множество линейных стрелок. Фактически, это hom-множество само по себе является профунктором:

```
newtype Hom a b = Hom (a %1-> b)

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

Как вы могли заметить, существуют разные определения, которые расширяют обычные концепции Haskel до линейных типов. Поскольку нет смысла вводить их заново и давать каждому из них новые имена, линейные расширения записываются с использованием множественного полиморфизма. Например, наиболее общая функция каррирования записывается так:

```
curry :: ((a, b) %p -> c) %q -> a %p -> b
                                %p -> c
```

охватывающая четыре различные комбинации кратностей.

7 Приложение 2: представление Ван Лаарховена

Начнем с определения функториальной прочности в моноидальной категории:

$$\sigma_{ab} : a \otimes Fb \multimap F(a \otimes b)$$

В замкнутой моноидальной категории каждый функтор является сильным. Для начала, можно каррировать σ. Таким образом, предстоит построить:

$$a \multimap (Fb \multimap F(a \otimes b))$$

В нашем распоряжении имеется ко-единица каррированного сопряжения:

$$\eta_{ab} : a \multimap (b \multimap a \otimes b)$$

Можно применить η_{ab} к a и поднять полученное отображение $(b \multimap a \otimes b)$, чтобы получить $Fb \multimap F(a \otimes b)$.

Теперь запишем представление Ван Лаарховена как конец отображения двух линейных hom-множеств:

$$\int_{F:[\mathcal{C},\mathcal{C}]} (a \multimap Fb) \to (s \multimap Ft)$$

Используем лемму Йонеды, чтобы заменить $a \multimap Fb$ множеством естественных преобразований, записанных как конец над x:

$$\int_F \int_x ((b \multimap x) \multimap (a \multimap Fx)) \to (s \multimap Ft)$$

Можно это де-каррировать:

$$\int_F \int_x ((b \multimap x) \otimes a \multimap Fx) \to (s \multimap Ft)$$

и применить лемму ниндзя-Йонеды к категории функторов, чтобы получить:

$$s \multimap ((b \multimap t) \otimes a)$$

Здесь, лемма ниндзя-Йонеды работает с функторами высшего порядка, такими как $\Phi_{st}F = (s \multimap Ft)$. Это можно записать как:

$$\int_F \int_x (Gx \multimap Fx) \to \Phi_{st}F \cong \Phi_{st}G$$

8 Приложение 3: учебная программа по управлению ресурсами

Приведу ссылки на некоторые из моих сообщений в блоге и статей об управлении ресурсами и его применении в параллельном программировании.

(1) Сильные указатели и управление ресурсами в C++,

- Part 1, 1999
- Part 2, 2000

(2) Walking Down Memory Lane, 2005 (с Андреем Александреску)

(3) unique ptr—How Unique is it?, 2009

(4) Unique Objects, 2009

(5) Многопоточность без гонок, 2009

- Part 1: Ownership
- Part 2: Owner Polymorphism

(6) Edward C++ hands, 2013