ФП 2.0 [1][2]

осень 2025

Содержание

[1] Автор Андрей Стоян (`andrey.stoyan.csam@gmail.com`).

[2] Спасибо Илье Колегову за первое внимательное прочтение и кучу комментариев.

Введение

Многие сложные концепции в дизайне языков и программ могут быть поняты как частные случаи некоторых простых фундаментальных принципов, которые, как правило, считаются общеизвестным фольклором, не требующим дополнительных пояснений. Однако, сложность в том, что эти знания рассеяны по книгам, статьям и "культовым" блог-постам, и требуется довольно много времени и сил для восстановления целостной картины.

Цель данного курса — собрать в одном месте такие фольклорные знания и организовать их в некоторую систему. Курс будет явным образом опираться на классические работы, исследующие принципы построения языков, и помогать в их изучении. Просмотр упоминаемых статей является важной частью самостоятельной работы в рамках курса.

Под функциональным программированием, вынесенным даже в заголовок курса, понимается трепетное отношение к понятию эффекта, которое в ФП, в отличие от других школ мысли, не считается аксиоматической данностью, но предметом для изучения, сознательного конструирования и аккуратного обращения. Этот подход оказывается очень полезным для изучения языков, построения могущественных языковых конструкций, а так же является основой для продуктивного стиля программирования. Кроме того, функциональные языки сравнительно просты, в результате чего новые идеи и подходы нередко зарождаются в них и распространяются далее.

В качестве основного языка курса выбран Haskell, так как он, с одной стороны, воплощает в себе многие концепции, часто доведенные до некоторого логического завершения, и достаточно могуществен для кодирования других. С другой стороны, всё ещё является прикладным промышленным языком программирования.

В связи с широтой контекста, данный курс не всегда является глубоким. Так, детали реализации в GHC или теор-категорные основания вещей могут даваться в общем виде и без конкретики. В то же время, в плоскости языкового дизайна через оптику функционального программирования курс пытается быть максимально подробным.

Таким образом, данный курс может быть полезен тем, кто интересуется дизайном языков и красивыми обобщениями программистских концепций, хочет улучшить свои навыки проектирования API, или планирует вести практическую деятельность на функциональных языках.

Пререквизитом к прохождению курса является знание основ функционального программирования: алгебраических типов данных, паттерн-матчинга, свёрток, параметрического полиморфизма, классов типов, базовых монад. Дополнительно будет полезным умение читать типовые дроби, знакомство с полиморфным λ-исчислением и кодированием Чёрча.

1 Воспоминания о ФП

В этом разделе мы вспомним основные концепции функционального программирования и языка Haskell.

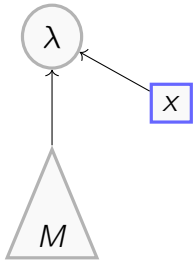
1.1 Термы и редукция

В ФП программы представляют собой выражения. Выполнение программ — редукция таких выражений до более "простых". Выражения можно представлять как в виде линейной записи символов, так и в виде дерева, для понимания которого не требуется знания вспомогательных правил ассоциативности и проч.

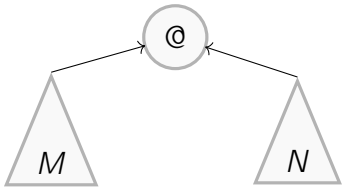
Простейший функциональный язык — λ-исчисление. Выражения в нём называются λ-термами, которые состоят из вершин трёх видов (V — множество валидных идентификаторов, Λ — множество λ-термов):

Переменные $x \in \Lambda$, если $x \in V$

Абстракция $(\lambda x.\, M) \in \Lambda$, если $x \in V, M \in \Lambda$



Аппликация $(M\ N) \in \Lambda$, если $M \in \Lambda, N \in \Lambda$



В произвольном выражении можно заменить некоторый его фрагмент на формальный параметр, который должен быть задекларирован выше по дереву с помощью специальной вершины λ. Вместо формального параметра можно в дальнейшем подставлять различные конкретные параметры с помощью вершины-аппликации @, то есть переиспользовать это выражение для различных целей (например, рис. 1). Редукция как раз определяется как следующее правило переписывания: ищется применение λ-функции к аргументу и в её тело осуществляется подстановка аргумента во все свободные вхождения переменной, связанной лямбдой (рис. 2).

1.2 Типы

Программное обеспечение — это сложно. Поэтому постоянно и неизбежно в программах возникают ошибки. Их можно искать, в том числе, статически, то есть без запуска программы. Одним из видов статического анализа является анализ типов.

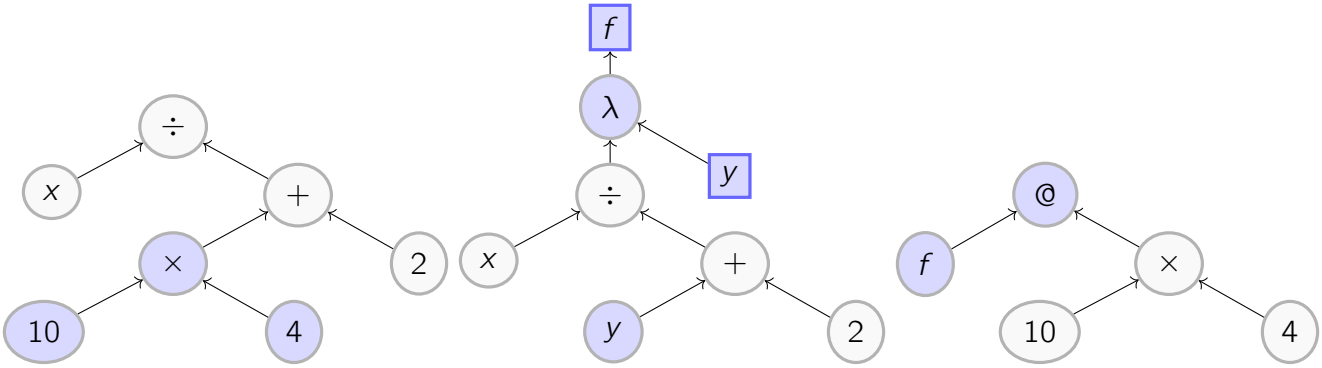


Рис. 1: Выражение с помощью λ вершины преобразуется в функцию одного аргумента.

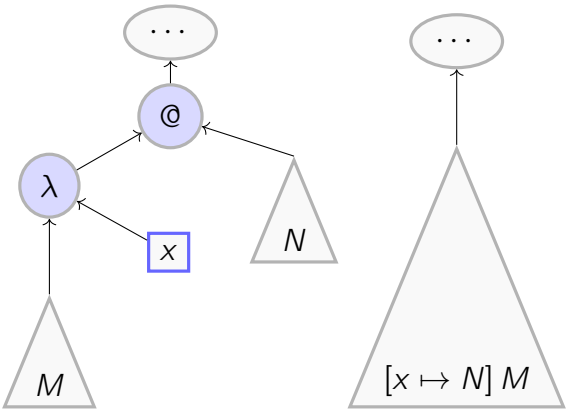


Рис. 2: Редукция переписывает дерево путём подстановки конкретного аргумента вместо формального параметра.

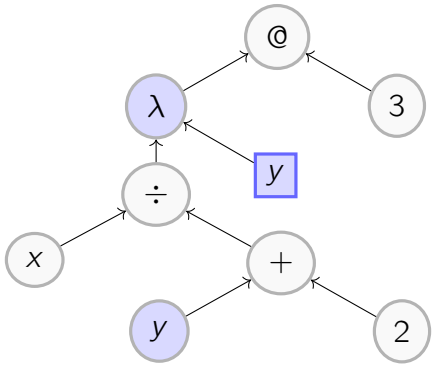


Рис. 3: Дерево соответствующее выражению $(\lambda y.\, x \div (y + 2))\ 3$.

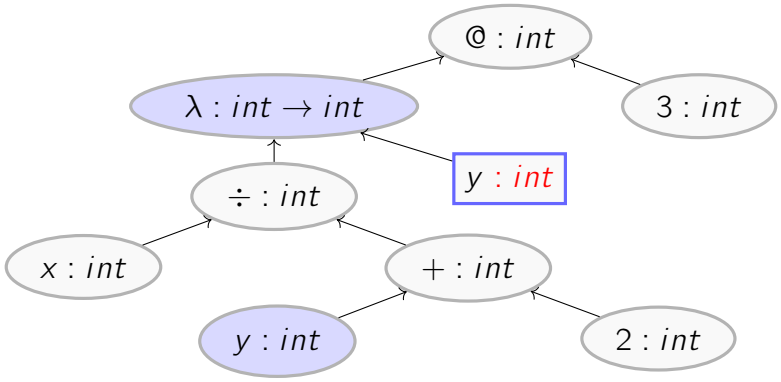


Рис. 4: Дерево выражения $(\lambda y.\, x \div (y + 2))\ 3$ после приписывания типовых меток.

Идея анализа типов состоит в том, что мы каждой вершине дерева программы пытаемся присвоить некоторую синтаксическую метку по определённым правилам. Если каждой вершине метку присвоить можно, то мы считаем, что программа проходит проверку типов, и она "хорошая". Например, на рисунке 3 представлено выражение, а на рисунке 4 каждой вершине приписаны метки в согласие с некоторой системой типов.

Система типов определяет синтаксис типовых меток и правила, по которым их можно приписывать. Синтаксис обычно описывается в классических нотациях а ля BNF, а правила в виде типовых дробей. Например, так выглядят дроби для просто-типизированного λ-исчисления:

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ ctx \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ elim \to \qquad \frac{\{x : \sigma\} \cup \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, M : \sigma \to \tau}\ intro \to$$

Типовые метки имеют чисто-синтаксическую природу, однако их можно проинтерпретировать. Самая популярная интерпретация — воспринимать типовую метку как множество. Так, метке $int \to int$ можно поставить в соответствие множество функций между множествами ограниченных целых чисел.

1.3 Функции в Haskell

В своей основе Haskell представляет собой расширенное типизированное λ-исчисление, дополненное примитивными типами, возможностью декларировать новые имена, структурами данных и классами типов.

Примеры λ-абстракций в REPL окружении GHCi:

```
ghci> (\x -> x + 1) 4
5
```

Можно узнать тип функции в интерпретаторе (в реальности числа полиморфные, но об этом далее):

```
ghci> :t \x -> x + 1
\x -> x + 1 :: Int -> Int
```

Функциям можно давать имена. Именам можно приписывать типы, это рекомендуется делать явно для деклараций на верхнем уровне файлов исходного кода.

```
f :: Int -> Int
f x = x + 1
```

Если имя типа начинается с маленькой буквы, то это не конкретный заранее заданный тип, а типовая переменная, способная принимать различные значения в зависимости от места вызова. Такая возможность называется **параметрическим полиморфизмом**. Так, функция, которая просто возвращает свой аргумент, никак не ограничивает тип аргумента. Но в то же время тип результата должен совпадать с типом аргумента.

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

ghci> :t id 5
id 5 :: Int
```

Функции могут принимать другие функции в качестве аргументов (такие функции называются **функциями высших порядков (higher-order functions)**. Имя функции может состоять из специальных символов, тогда она считается оператором и может применяться к своим операндам в инфиксном стиле:

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

Пример рекурсивной функции, использующей охранные выражения для отличения базового случая рекурсии:

```haskell
factorial :: Int -> Int
factorial n
  | n < 1 = 1
  | otherwise = n * factorial (n - 1)
```

Упражнение 1 *Что выведет запрос* `ghci> :t uncurry (flip const)`*?*

Упражнение 2 *Что выведет запрос* `ghci> :t first . first` *при*

```haskell
first :: (a -> a') -> (a, b) -> (a', b)
```

Упражнение 3 *Реализуйте факториал с помощью техники аккумулирующего параметра.*

1.4 Данные в Haskell

В Haskell есть встроенная возможность объявлять новые типы данных на основании других типов, а так же создавать их экземпляры.

Зададим тип данных, описывающий животных:

```haskell
data Animal
  = Cat String Int
  | Dog String
```

Мы задали тип данных `Animal` и два способа создать значения этого типа: для кошек и собак. `Cat` и `Dog` — это **конструкторы данных**. Они представляют собой функции, реализация которых находится на стороне языка. Они выделяют память под экземпляры данного типа и позиционно размещают компоненты. Кошек мы описываем именем и оставшимся количеством жизней, а собак — только именем.

```haskell
Cat :: String -> Int -> Animal
Dog :: String -> Animal
```

Чтобы воспользоваться информацией, сохранённой в структуре данных, требуется деконструировать её с помощью паттерн-матчинга. Мы сопоставляем значение типа с образцом. Если образец похож на то, как было сконструировано значение, то он выбирается среди других образцов и переменные, задекларированные в нём, начинают ссылаться на соответствующее позиционно содержимое структуры данных:

```haskell
show :: Animal -> String
show animal = case animal of
  Cat name nLifes -> "This is cat " ++ name ++ show nLifes
  Dog name -> "This is dog " ++ name
```

В Haskell есть специальный синтаксис для объявления полей с именованными метками.

```haskell
data Penguin = Penguin { getName :: String, getAge :: Int }
penguin = Penguin { getName = "Andrey", getAge = 500 }
```

Haskell генерирует функции-аксессоры для доступа к полям объекта:

```haskell
ghci> :t getName :: Penguin -> String
```

Часто функции в программировании частичные — при некоторых значениях аргументов они могут вернуть результат, а при некоторых — нет. Давайте моделировать это с помощью специального типа данных. Если есть вещественный результат, будем возвращать его. Если нет, будем возвращать специально выделенное константное значение этого типа.

```haskell
data MaybeD = NothingD | JustD Double
sqrt :: Double -> MaybeD
sqrt x = if x < 0 then NothingD else JustD (calcSqrt x)
```

Можно заметить, что так нам придётся объявлять по типу `MaybeT` для каждого типа `T`. Поэтому Haskell позволяет абстрагироваться в типе, аналогично тому как можно абстрагироваться по значениям в терме.

```haskell
data Maybe a = Nothing | Just a
sqrt :: Double -> Maybe Double
sqrt x = if x < 0 then Nothing else Just (calcSqrt x)
```

Заметьте, что сейчас `Maybe` — это не совсем тип, так как теперь нужно передать типовой параметр, чтобы получить конкретный тип. `Maybe` называют **типовым конструктором**.

Вместе с абстракцией на уровне типов появилась и аппликация типа к типу. А что если дать меньше параметров типовому конструктору, чем ожидается? А что если больше? Контроль за корректностью типовых аппликаций обеспечивает **система кайндов**[3]. Это простейшие “типы для типов”, то есть синтаксические метки, контролирующие корректность записанных программистом типов. Так, обычные типы имеют метку (кайнд) `*`. Типовые конструкторы имеют стрелочные кайнды. Например, `Maybe :: * -> *`. Аппликация типового конструктора к типу подходящего кайнда убирает одну стрелку:

```haskell
ghci> :k Int
Int :: *
ghci> :k Maybe
Maybe :: * -> *
ghci> :k Maybe Int
Maybe Int :: *
```

Кроме совершенно новых типов данных, в Haskell можно объявлять типовые синонимы. Это имена, которые можно использовать вместо других типов, если, например, запись оригинального типа слишком длинная для повсеместного написания.

[3]Иногда в русскоязычной литературе кайнды называют родами типов, но мы не будем так говорить.

```haskell
type T a = VeryLongType Int (a -> AnotherLongType a)
```

Если тип данных содержит только один конструктор и только одно поле, то отсутствует необходимость в аллокации новой памяти, содержащей тег конструктора и набор ссылок на поля. В таком случае, в качестве значения такого типа можно всегда просто использовать значение оборачиваемого типа, оставляя новый тип присутствовать исключительно во время компиляции, снижая нагрузку во время исполнения. Для объявления таких типов-обёрток нужно воспользоваться ключевым словом `newtype` вместо `data`:

```haskell
newtype CourseId = CourseId Int64
newtype ModuleId = ModuleId Int64
```

Упражнение 4 *Определите кайнд конструктора типа*

```haskell
data Free f a = Pure a | Free (f (Free f a))
```

1.5 Классы типов в Haskell

Параметрический полиморфизм позволяет использовать один и тот же код для различных типов входных данных. Классы типов же позволяют одному идентификатору ссылаться на разные реализации для разных типов данных (что аналогично механизму перегрузки (overloading) в других языках). Классы типов, как говорят, являются механизмом **специального (ad-hoc) полиморфизма**. Так, мы можем задекларировать символ ==, выбор реализации которого зависит от выбора типа аргументов a:

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

Для каждого типа можно объявить свою собственную реализацию Eq:

```haskell
instance Eq CourseId where
  CourseId x == CourseId y = x == y

instance Eq a => Eq [a] where
  [] == [] = True
  x:xs == y:ys = x == y && xs == ys
```

Теперь в зависимости от конкретного типа a в месте вызова, будет выбрана подходящая реализация для этого типа:

```haskell
ghci> CourceId 1 == CourceId 2
False
ghci> [CourceId 1, CourceId 2] == [CourceId 1, CourceId 2]
True
```

Рассмотренные ранее параметрически-полиморфные функции ничего не могли делать со своими аргументами, кроме как возвращать их в качестве результата или передавать в другие полиморфные функции. Чтобы уметь делать что-то ещё, нужна какая-то дополнительная информация про тип, потому что иначе нет никакой гарантии, что над объектом данного типа можно делать все необходимые операции. Так, функция `suc n = n + 1` не будет работать для строчек, потому что для них, очевидно, не определена операция сложения. Поэтому некорректно будет приписать полиморфный тип `suc :: a -> a`.

Классы типов, в отличие от перегрузки, в том числе являются механизмом ограничения полиморфности функций. Мы можем явно задать, что функция требует не произвольный тип на вход, а произвольный тип, для которого определены обязательно нужные нам операции. Так, для типа `suc` достаточно ограничить тип условием наличия плюса для него (операция обозначаемая символом + объявлена в классе типов `Num`):

```
suc :: Num a => a -> a
```

Упражнение 5 *Реализуйте функцию, проверяющую равенство всех элементов данного списка.*

Упражнение 6 *Реализуйте инстанс полугруппы для функций.*

Упражнение 7 *Реализуйте проверку равенства функций.*

1.6 Монады в Haskell

Класс типов `Functor` объявляется для конструкторов типов и позволяет заменить в некотором контейнере все элементы одного типа на все элементы другого, оставляя структуру контейнера неизменной.

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

instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x:xs) = f x : fmap f xs
```

В Haskell любая функция просто вычисляет результат некоторого типа. Однако в программирования часто требуются функции, которые не только вычисляют результат, но и делают что-то ещё. Например, изменяют какое-то состояние или пишут в консоль. Иными словами, производят побочные эффекты. В любом случае в Haskell мы можем только вернуть из функции только результат, поэтому такие побочные эффекты мы кодируем в качестве дополнительной структуры, оборачивающей чистый результат. Т.е. если функция без побочных эффектов возвращала какой-то тип `a`, то после добавления побочных эффектов в её реализацию, она будет возвращать некоторый тип *вычислений* `f a`.

- Если функция кидает ошибку, то `f = Maybe`.
- Если функция читает глобальное состояние типа `e`, то `f = e -> _`.
- Если функция читает глобальное состояние `s` и обновляет его, то `f = s -> (s, _)`.

Стандартная библиотека Haskell предоставляет несколько классов типов для работы со значениями вида `f a`. Они позволяют абстрагироваться от структуры `f` и работать со значениями `a` внутри, как будто нет никакой дополнительной структуры.

Первый такой класс типов позволяет писать выражения над вычислениями `f a`.

```haskell
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c

instance Applicative Maybe where
  pure :: a -> Maybe a
  pure = Just

  liftA2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
  liftA2 _ Nothing _ = Nothing
  liftA2 _ _ Nothing = Nothing
  liftA2 f (Just x) (Just y) = Just (f x y)
```

Второй класс типов позволяет делать последовательную композицию вычислений в императивном стиле:

```haskell
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b

newtype State s a = State { runState :: s -> (s, a) }

instance Monad (State s) where
  (>>=) :: State s a -> (a -> State s b) -> State s b
  m >>= k = State \s ->
    let (s', x) = runState m s in
    runState (k x) s'
```

Теперь если мы определим базовые операции работы с состоянием, мы сможем писать код в императивном стиле с побочными эффектами.

```haskell
get :: State s s
get = State \s -> (s, s)

put :: s -> State s ()
put newS = State \oldS -> (newS, ())
```

```haskell
example :: State Int Int
example =
  get >>= \x ->
  put 42 >>= \() ->
  get >>= \y ->
  pure (x + y)
```

```
ghci> runState example 1
43
```

Для таких монадических цепочек существует специальный синтаксический сахар:

```haskell
example :: State Int Int
example = do
  x <- get
  put 42
  y <- get
  pure (x + y)
```

Упражнение 8 *Реализуйте `liftA3` через `liftA2`.*

Упражнение 9 *Реализуйте `>>=` через `join` и наоборот.*

Упражнение 10 *Два числа с консоли, поделите одно на другое нацело и распечатайте результат, если остаток не нулевой, распечатайте его тоже.*

2 Параметрический полиморфизм

Никакое нетривиальное свойство программ не может быть алгоритмически проверено[4]. Чтобы оставаться разрешимыми (в смысле проверки типов и/или вывода), многие системы типов жертвуют полнотой и, помимо некорректных программ, отвергают много корректных. В то же время системы типов также стараются предоставлять различные возможности, позволяющие протипизировать как можно больше корректных программ. Одна из них — параметрический полиморфизм.

Под **параметрическим полиморфизмом** мы будем подразумевать возможность кода единообразно работать с произвольными типами данных Strachey [2000], Cardelli and Wegner [1985], что позволяет во многих случаях избегать дублирования кода.

В этой главе мы рассмотрим, как описывают полиморфизм в самом простом виде — в типизированном λ-исчислении. Изучим различные формы параметрического полиморфизма и сопутствующие техники безопасного программирования. Проанализируем возможные способы эффективной реализации параметрического полиморфизма. И в завершение рассмотрим полиморфизм по рантайм-представлению, "полиморфизм по полиморфизму".

2.1 Параметрический полиморфизм в языке

λ-абстракция позволяет обобщать выражения по значениям, каждая абстракция добавляет стрелку в тип выражения. Аппликация же снимает стрелку.

$$\frac{x : \tau, \Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau.\, M : \tau \to \sigma}\ Lam \quad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \sigma}\ App$$

В то же время Λ-абстракция позволяет обобщать выражения по типам, добавляя квантор в тип (П-абстракцию) [Pierce, 2002, глава 23]:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\, \tau}\ TLam \quad \frac{\Gamma \vdash M : \forall\alpha.\, \tau}{\Gamma \vdash M\ \sigma : [\alpha \to \sigma]\ \tau}\ TApp$$

Теперь, например, мы можем дать возможность пользователю выбрать, с каким типом он хочет использовать нашу функцию (применение к типу называют **универсальной аппликацией (universal application)**):

$$\begin{aligned} &id : \forall\alpha.\, \alpha \to \alpha \\ &id = \Lambda\alpha.\, \lambda x : \alpha.\, x \\ &id\ nat : nat \to nat \\ &id\ nat\ 42 : nat \end{aligned}$$

Функция id фактически принимает два аргумента: тип и значение.

В Haskell типовые абстракции и аппликации приписываются неявно механизмом вывода типов. Однако, есть расширения языка, которые позволяют их написать явно: TypeAbstractions,

[4] `https://en.wikipedia.org/wiki/Rice%27s_theorem`

TypeApplications. Это может помочь, например, когда информации из терма не достаточно, чтобы вывести тип. Так, можно явно специализировать `id` на нужный тип:

```
id :: forall a . a -> a
ghci> :t id @Int
id @Int :: Int -> Int
```

Кванторы также приписываются неявно в начале типа, следуя конвенции именования: конкретные типы начинаются с большой буквы, а полиморфные — с маленькой. Аналогично, у пользователя есть возможность явно приписывать `forall`'ы с помощью расширения ExplicitForAll. Это может понадобиться либо за тем, чтобы задать вручную порядок типовых абстракций, либо, чтобы иметь возможность сослаться на абстрагированный тип в теле функции (расширение ScopedTypeVariables).

Полиморфные типы данных задаются с помощью другой конструкции. Если ранее мы управляли типом с уровня термов универсальной аппликацией, то теперь мы хотим управлять типом на уровне типов. Для этого мы вводим λ абстракцию в типах, аппликацию в типах и, соответственно, β-редукцию. Система кайндов (пока) представляет собой простейшую "систему типов для типов" и обеспечивает well-formedness типов и строгую нормализуемость[5]. Например, мы можем написать тип пары, абстрагированный от конкретных типов компонент, чтобы пользователь мог выбрать нужные ему.

$$
\begin{aligned}
&Pair : * \to * \to * \\
&Pair = \lambda \tau^{*}\ \sigma^{*}.\, \forall \gamma.\, (\tau \to \sigma \to \gamma) \to \gamma \\
&pair : \forall \alpha\ \beta.\, \alpha \to \beta \to Pair\ \alpha\ \beta \\
&pair = \Lambda \alpha^{*}\ \beta^{*}.\, \lambda x^{\alpha}\ y^{\beta}.\, (\Lambda \gamma^{*}.\, \lambda f^{\alpha \to \beta \to \gamma}.\, f\ x\ y) \\
&fst : \forall \alpha\ \beta.\, Pair\ \alpha\ \beta \to \alpha \\
&fst = \Lambda \alpha^{*}\ \beta^{*}.\, \lambda p^{Pair\ \alpha\ \beta}.\, p\ \alpha\ (\mathbf{K}\ \alpha\ \beta)
\end{aligned}
$$

В Haskell вычислительную семантику полиморфных типов можно проследить в синонимах типов:

```
type Pair a b = forall c . (a -> b -> c) -> c
intPair :: Pair Int Int -- forall c . (Int -> Int -> c) -> c
```

Обычные конструкторы типов номинативны. Например, `(Int, Int)` или `Maybe Int` никуда далее не вычисляются.

Haskell не позволяет создавать функции на типах по месту с помощью явной типовой лямбды[6] ввиду проблематичности этой конструкции для вывода типов. Однако полноценные функции на типах есть, и мы рассмотрим их далее 3.2. В Scala существует нетривиальный

[5] **Строгая нормализуемость** — любой порядок редукций приводит к нормальной форме.

[6] `https://stackoverflow.com/questions/4069840/lambda-for-type-expressions-in-haskell`

трюк[7][8], который позволяет этого добиться. Scala3, однако, включила эту возможность непосредственно в язык[9].

2.1.1 Эмуляция типовых абстракций и аппликаций (Proxy)

В Haskell расширения, позволяющие вручную задавать типовые аппликации и абстракции появились сравнительно недавно[10]. До этого пользовались следующей техникой.

В стандартной библиотеке определён тип Proxy с одним параметром. Это **фантомный типовой параметр** — значения соответствующего типа не хранятся в структуре данных, он только позволяет размещать дополнительную информацию на уровне типов[11]. Соответственно, неинформативную константу Proxy можно проаннотировать нужным типом и передать в функцию, чтобы специализировать типовой параметр на нужный тип. Или можно принять Proxy и воспользоваться ScopedTypeVariables для типовых сигнатур в паттернах[12].

```
data Proxy a = Proxy

id :: Proxy a -> a -> a
ghci> :t id (Proxy :: Proxy Int)
id (Proxy :: Proxy Int) :: Int -> Int

id (Proxy :: Proxy a) x = (x :: a)
```

Иногда прокси-тип оставляют полиморфным, чтобы пользователь сам мог его задать. Вместо конкретного значения иногда передают специализированное значение $\bot$, а получатель, не зная тип, не сможет его форсировать (однако, любые вхождения $\bot$ в терм слишком настораживают, поэтому это скорее не очень хорошая практика).

```
id :: proxy a -> a -> a
id (_ :: proxy a) x = (x :: a)

ghci> :t id (undefined :: Proxy Int)
id (undefined :: Proxy Int) :: Int -> Int
```

2.1.2 First-class polymorphism

Существует возможность писать функции, которые принимают другие полиморфные функции в качестве аргументов. Типы таких функций называются **типами высшего ранга (higher-**

[7](stackoverflow) Scala type lambdas.

[8]https://stackoverflow.com/questions/9443004/what-does-the-operator-mean-in-scala

[9]https://docs.scala-lang.org/scala3/reference/new-types/type-lambdas.html

[10]TypeApplications, TypeAbstractions.

[11]https://wiki.haskell.org/Phantom_type

[12]Типовый параметр на самом деле имеет полиморфные кайнд `data Proxy (a :: k) = Proxy`, чтобы эта техника работала с типами произвольных кайндов (см. далее 2.1.5.

rank types), их можно использовать с расширением RankNTypes. Так, типовой параметр функции g определяет функция f, а не вызывающий функцию f:

```
f :: (forall a . a -> a) -> (Int, Char)
f g = (g @Int 42, g @Char 'a') -- универсальная аппликация для наглядности
ghci> f (\x -> x)
```

Проблема типов высшего ранга в том, что их вывод неразрешим, то есть глобальный вывод типов Haskell в этом случае перестаёт работать. Но если типы высшего ранга приписать вручную, остальной вывод будет работать как раньше. Например, числа Чёрча имеют высший ранг[13]:

```
suc :: (forall a . (a -> a) -> a -> a) -> (a -> a) -> a -> a
suc n s z = s (n s z)
```

Упражнение 11 *Какой ранг имеет тип* `Int -> (forall a . a -> a)`*?*

От многих проблем сопутствующих типам высших рангов можно избавиться, если создавать для них обёртки. Например, для чисел Чёрча можно создать обёртку `newtype Church`. Теперь код, работающий с обёрткой, может быть протипизирован типами первого ранга, только конструктор имеет тип высшего ранга.

```
newtype Church = Church (forall a . (a -> a) -> a -> a)
(+) :: Church -> Church -> Church -- rank 1
```

Аналогичный код можно написать и в Java (Kotlin):

```
interface Church { fun <a> fold(s: (a) -> a, z: a): a }
fun plus(n: Church, m: Church): Church = object : Church {
    override fun <a> fold(s: (a) -> a, z: a): a = n.fold(s, m.fold(s, z))
}
```

По умолчанию типовые параметры можно специализировать только на конкретные типы. Расширение ImpredicativeTypes позволяет специализировать типовые параметры на полиморфные типы (включающие `forall`'ы внутри себя) — **импредикативное применение**.

```
runST :: (forall s. ST s a) -> a
($) :: forall a b . (a -> b) -> a -> b
foo = runST $ ... -- типизируется только с ImpredicativeTypes
```

Higher-rank типы можно использовать как type-based escape analysis, иначе говоря, не позволять пользователю передавать некоторое значение вовне определённого скоупа. Так, например, Haskell предоставляет эффективную монаду `ST`, позволяющую в рамках ограниченного скоупа работать с мутабельными ячейками памяти Launchbury and Peyton Jones [1995][Maguire, a, 7.2, ST trick]:

[13] https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html

```haskell
newtype ST s a = ST (IO a)
runST :: (forall s. ST s a) -> a

sumTo :: Int -> Int
sumTo n = runST do
  ref <- newSTRef 0
  forM [0..n] \i -> modifySTRef ref (+ i)
  readSTRef ref
```

Заметим, что если попытаться вернуть из `runST` ссылку на мутабельную ячейку, то результирующий тип не пройдёт well-formedness проверку, так как будет содержать фантомный параметр `s`, который не будет нигде связан:

```haskell
newSTRef :: a -> ST s (Ref s a)
ghci> runST (newSTRef 0) :: Ref s Int -- ошибка
```

На практике, чтобы отличать такие локально связанные типовые переменные, используют концепцию уровней[14] Jones [2019].

Типы высших рангов вместе с импредикативным применением образуют **полиморфизм первого класса (first-class polymorphism)**, когда полиморфные типы могут использоваться почти так же свободно, как и любые другие. Классический алгоритм глобального вывода Хиндли-Милнера не справляется (и в общем случае задача неразрешима), так что существует большое количество решений, делающих различные компромиссы. Можно сделать вывод типов локальным, опирающемся только на соседние ноды AST и вспомогательные типовые аннотации Pierce and Turner [2000], Christiansen [2013], Dunfield and Krishnaswami [2019]. Либо же можно попытаться помочь глобальному выводу дополнительной предобработкой (Quick Look[15] Serrano et al. [2020], реализованный в Haskell с недавнего времени) или дополнительными регулирующими конструкциями (FreezeML Emrich et al. [2020]).

2.1.3 Higher-order/kinded polymorphism

Haskell позволяет также абстрагироваться по типам произвольных кайндов, а не только `Type`, как в `data` декларациях (**higher-order/kinded types (HKT)**[16]), так и в полиморфных функциях. Далее мы встретим немало примеров. Так, `Fix` имеет кайнд `(Type -> Type) -> Type`, а катаморфизм абстрагирован по типу стрелочного кайнда:

```haskell
newtype Fix f = Fix (f (Fix f))
cata :: forall (f :: Type -> Type) a . Functor f => (f a -> a) -> Fix f -> a
```

Далее мы рассмотрим технику, позволяющую типы высших порядков закодировать в языке, их не поддерживающем (см. далее 3.1.8).

[14] https://okmij.org/ftp/ML/generalization.html
[15] (youtube) A Quick Look at Impredicativity (Simon Peyton Jones)
[16] https://serokell.io/blog/kinds-and-hkts-in-haskell

2.1.4 Обобщённые алгебраические типы данных (GADTs)

Обобщённые алгебраические типы данных (generalized algebraic data types, GADTs) позволяют приписывать данным на уровне типов больше информации. В качестве модельного примера возьмём синтаксис крошечного языка программирования. Зададимся целью не допустить возможности конструирования в Haskell некорректных с точки зрения типов синтаксических деревьев.

```
data Expr = Const Int | IsZero Expr | If Expr Expr Expr
```

Как мы знаем, конструкторы данных в Haskell — это обычные функции с той лишь разницей, что их реализация генерируется компилятором (аллокация памяти, размещение полей...). У функций есть тип. Например, `IsZero :: Expr -> Expr`.

В Haskell есть синтаксис определения `data` через задание типов конструкторов[17]. Он совершенно аналогичен рассмотренному ранее, только гораздо более удобен для сложно организованных структур данных. Рассмотренный ранее тип термов `Expr` будет выглядеть следующим образом:

```
data Expr where
  Const :: Int -> Expr
  IsZero :: Expr -> Expr
  If :: Expr -> Expr -> Expr -> Expr
```

Для полиморфных структур данных, на примере списка, используется следующий синтаксис. Имя `elem` нужно исключительно для документации и больше никак его использовать нельзя, оно только маркирует наличие типового параметра и позволяет ему вручную задать кайнд[18].

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

Добавим к `Expr` фантомный типовой параметр `ty`, обозначающий тип Haskell, в который должно быть проинтерпретировано данное выражение, и с помощью GADT зададим конкретные значения `ty` результирующим типам конструкторов. Так, мы говорим, что программа сконструированная с помощью `Const` вычисляется в число, `IsZero` вычисляется в булево значение, а условное выражение — в тип веток:

```
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty

eval :: Expr ty -> ty
```

[17] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/gadt_syntax.html#gadt-style

[18] Кайд можно не писать. Либо можно не писать имена и просто приписать кайнд типовому конструктору: `data List :: Type -> Type where ...`.

Теперь мы можем написать безопасный типизированный интерпретатор. Обратите внимание, что при сопоставлении с образцами конструкторов, у нас уточняется информация о типовом параметре:[19]

```
eval :: Expr ty -> ty
eval = \case
  Const x  -> x          -- ty ~ Int
  IsZero t -> eval t == 0 -- ty ~ Bool
  If c t e -> if eval c then eval t else eval e
```

Далее мы рассмотрим как GADT в Haskell выражаются через более базовые механизмы языка 3.4.4.

2.1.5 Структуры на уровне типов, data promotion

Чтобы обрести больший контроль корректности программ, научимся кодировать произвольные структуры данных на уровне типов. В качестве модельной задачи зададим структуру данных, моделирующую вектор, но с контролем длины.

Для начала определим натуральные числа на уровне типов в стиле Пеано:

```
data Zero
data Suc n
```

Упражнение 12 *Сколько обитателей типа* `Suc (Suc Zero)`*?*

Теперь мы можем задать тип вектора, содержащий информацию о длине:

```
data Vec (size :: Type) (elem :: Type) where
  VNil :: Vec Zero a
  VCons :: a -> Vec n a -> Vec (Suc n) a

example :: Vec (Suc (Suc Zero)) Int
example = VCons 1 (VCons 2 VNil)
```

Для такого типа, например, можно написать безопасную функцию `zip`, работающую только на векторах одинаковой длины:

```
vzip :: Vec n a -> Vec n b -> Vec n (a, b)
vzip VNil VNil = VNil                                    -- n ~ Zero
vzip (VCons x xs) (VCons y ys) = VCons (x, y) (vzip xs ys) -- n ~ Suc n'
```

Заметьте, что в остальных ветках `vzip` должны возникнуть эквивалентности, начинающиеся с различных конструкторов, например, `Zero ~ Suc n`. Поскольку невозможно построить такие аргументы функции, Haskell позволяет соответствующие ветки не рассматривать.

[19] Тут используется удобное расширение LambdaCase, позволяющее не вводить лишние имена.

Упражнение 13 *Напишите функцию добавления в конец элемента вектора. Двигайтесь последовательно, заполняя типовые дыры и отслеживая возникающие эквивалентности.*

Удивительно, но сейчас наш язык типов не типизирован. Действительно, кайнд `Suc` — `Suc :: Type -> Type`, соответственно ничто не мешает написать `Suc (Maybe Int)`. То есть язык кайндов, который должен контролировать типы, слишком беден. В то же время он слишком ограничивающий, поскольку не поддерживает полиморфизм, что дало начало большому количеству дублирований а ля `Typeable (ty :: Type)`, `Typeable1 (ty :: Type -> Type)`...

Современный Haskell имеет расширение TypeData, позволяющее объявлять новые типы и кайнды подобно тому, как `data` позволяет объявлять новые типы.

```
type data Nat = Zero | Suc Nat
```

Теперь вектору можно приписать более точный кайнд:

```
data Vec (size :: Nat) (elem :: Type) where
  VNil :: Vec Zero a
  VCons :: a -> Vec n a -> Vec (Suc n) a
```

Упражнение 14 *Что выведет `ghci> :k Vec`?*

Другим вариантом добиться того же самого является использование DataKinds Yorgey et al. [2012]. Это расширение автоматически продвигает (promotion) все `data` декларации на уровень выше. А именно: любой конструктор типа также становится кайндом, а конструктор данных — конструктором типа. Так, в примере с числами, мы можем задекларировать натуральные числа как обычно и использовать на уровне типов:

```
data Nat = Zero | Suc Nat
ghci> :k Suc :: Nat -> Nat -- тут понятно что Suc используется как тип
```

Поскольку типы и термы в Haskell живут в разных пространствах имён, можно называть конструкторы типов и данных одинаково. Однако если продвинуть такой тип данных, возникнет неоднозначность: мы имеем в виду тип или продвинутый конструктор. Haskell позволяет указать явно, что речь идёт о продвинутом конструкторе с помощью одинарной кавычки.

```
data T = T Nat
ghci> :k T
T :: Type      -- про конструктор типа
ghci> :k 'T
'T :: Nat -> T -- про продвинутый конструктор данных
```

Не любые `data` декларации подходят для продвижения, в то же время `type data` декларации позволяют явно запросить структуру уровня типов и получить внятные ошибки, если декларация написана неправильно.

В случае продвижения полиморфного типа, мы получаем полиморфные кайнды (PolyKinds):

Term	Type	Kind
`Zero`	`Nat`	`Type`
`[Zero, Suc Zero]`	`[Nat]`	`Type`
`[]`	`forall a. [a]`	`Type`
`(:)`	`forall a. a -> [a] -> [a]`	`Type`
	`'Suc 'Zero`	`Nat`
	`'['Zero, 'Suc 'Zero]`	`[Nat]`
	`'[Int, Double]`	`[Type]`
	`'[]`	`forall k. [k]`
	`'(:)`	`forall k. k -> [k] -> [k]`

Рис. 5: Пример продвижений в Haskell.

```
data [a] = [] | (:) a [a]
ghci> :k '(:)
'(:) :: forall k . k -> [k] -> [k]
```

Примеры продвижения различных конструкций можно увидеть в таблице 5.

В качестве примера, зададим гетерогенный список, индексированный типами элементов:

```
data HList (tys :: [Type]) where
  HNil :: HList '[]
  HCons :: ty -> HList tys -> HList (ty ': tys)

example :: HList '[Int, Bool, Double]
example = HCons 42 $ HCons True $ HCons 12.5 HNil
```

Структуры данных тоже могут быть полиморфными по кайндам. Рассмотрим следующий тип `Tagged`, позволяющий дополнить тип значения дополнительным типовым тегом. Кайнд тега может быть произвольным, поэтому, например, можем использовать встроенные в систему типов константы TypeLits (другой пример использования полиморфных кайндов мы видели ранее 2.1.1):

```
newtype Tagged (tag :: k) (a :: Type) = Tagged a
ghci> :t Tagged
Tagged :: forall k (tag :: k) a. a -> Tagged tag a

example :: Tagged ("dbId" :: Symbol) Int
example = Tagged 42
```

Современный Haskell в итоге пришёл к тому, что система типов не делает различий между типами и кайндами (рис. 6). В частности, `Type :: Type`. Это нужно для расширения возможностей Haskell в сторону программирования с зависимыми типами путём добавления несин-

таксических эквивалентностей для кайндов (TypeInType). *System FC* была представлена в работе Weirich et al. [2013][20][21].

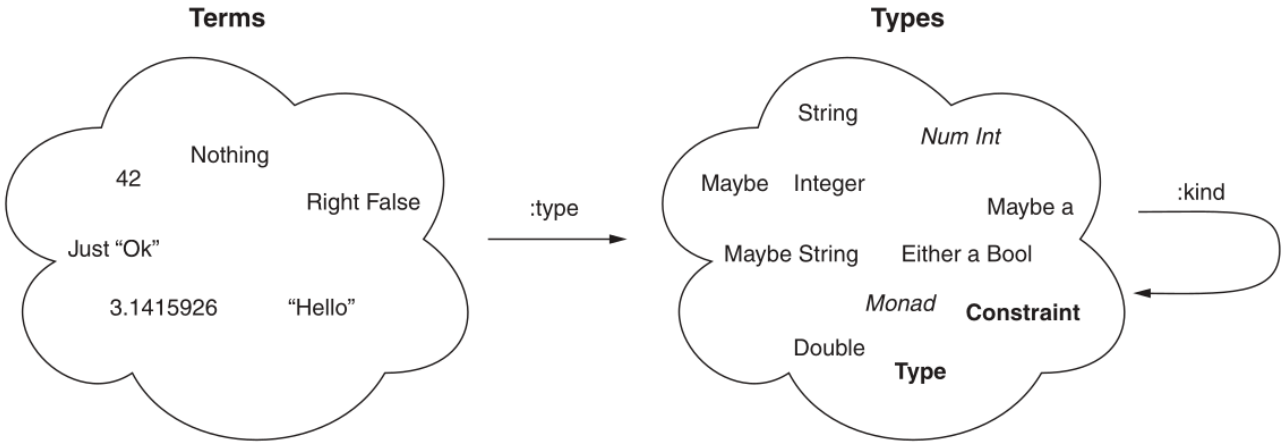


Рис. 6: Типы и канды — одно Bragilevsky.

2.2 Реализация параметрического полиморфизма

Конвенция вызова[22] представляет собой набор соглашений между тем как функция компилируется и как должна вызываться. Например, функция принимает два аргумента, каждый размером в машинное слово, и возвращает один результат размером в машинное слово. Тогда сгенерированный низкоуровневый код этой функции может, например, ожидать, что оба аргумента передаются через специальную пару регистров, а складывать результат он будет в третий. В таком случае вызывающий код обязан предоставить аргументы в правильных регистрах и ожидать результата в некотором третьем, заранее оговоренном регистре.

В общем случае, конвенция вызова функции зависит от типов аргументов и результата. Нужно знать как минимум их размер, чтобы понять, размещать их в регистрах или на стеке. Нужно знать, это указатель (**reference type**) или значение само по себе (**value type**), чтобы понимать, как с ним работать. В структурах данных нужно знать смещения полей.

Таким образом, реализация параметрического полиморфизма в языке — это не тривиальная задача. Разные языки используют различные подходы, все со своими достоинствами и недостатками.

2.2.1 Мономорфизация

Мономорфизация — самый прямолинейный подход, компилируем полиморфные функции и структуры для каждого набора типовых аргументов. Так, если различных наборов типовых

[20](youtube) Мини-курс на русском языке про развитие Haskell в сторону зависимой типизации.
[21](youtube) Мини-курс на русском языке — система вывода типов Haskell.
[22]`https://en.wikipedia.org/wiki/Calling_convention`

аргументов, с которыми эта функция вызывается, например, 100 (что запросто может быть), то её код будет компилироваться сто раз и занимать в бинарнике в сто раз больше места. Так делают, например, C++ и Rust.

На самом деле всё ещё хуже. Если проект многомодульный и состоит из множества единиц компиляции (кусков, которые компилируются отдельно), то одна и та же специализация функции на типовые аргументы будет компилироваться заново во всех единицах компиляции, где такая специализация нужна. А затем, линкер будет заниматься удалением дубликатов, что тоже не самый быстрый и эффективный процесс.

+ Порождаемый код максимально эффективен для каждого типа;
+ Легко на этапе компиляции отрабатывают `is`-проверки значений на принадлежность определённому типу (в остальных подходах с этим всё сложно);
− Время компиляции крайне велико;
− Существенно увеличивается размер результирующего бинарного файла, что может быть критично для некоторых приложений;
− Может неэффективно работать из-за засорения кеша кода в процессоре;
− В интерфейсах не может быть полиморфных методов, так как мы не знаем в месте вызова, к какому именно наследнику относится вызываемый метод, и какой код нужно специализировать (аналогично, не работает higher-rank полиморфзм);
− К полиморфным функциям нельзя динамически линковаться (у них нет кода до специализации);
− В общем случае нельзя поддержать variance, потому что код компилируется для конкретного типа и в общем случае не может работать для произвольного подтипа или супертипа (если reference и value типы могут находиться в одной иерархии подтипизации).

Некоторые языки не делают инстанциацию скрытой деталью реализации языка, а предоставляют её как инструмент пользователям. Так делают, например, C++ и Zig. А именно, это позволяет добиться следующего:

- Если разрешить использовать значения в типах, инстанциация может использоваться как механизм вычислений на этапе компиляции.
- Если отложить проверку ошибок на стадию инстанциирования, то мы получим своего рода статическую утиную типизацию. Это позволит не описывать сложные сигнатуры полиморфных функций. Однако тогда функции для тестирования придётся вручную инстанциировать против всевозможных типов, иначе нельзя понять статически, компилируется она хотя бы против этих типов или нет.

2.2.2 Стирание типа

Можно всё сделать наоборот, унифицировав значения, которые приходят на вход полиморфным функциям и хранятся в полиморфных структурах данных, вместо того, чтобы компилировать код под каждый тип.

Пусть каждое значение будет аллоцировано в куче и передаваться по указателю. Тогда мы сможем переиспользовать один и тот же код для разных типовых аргументов — он просто будет ожидать указатели.

+ Каждая функция компилируется ровно один раз — быстро;
+ Можно динамически загружать новые полиморфные функции и типы и использовать их друг с другом;
+ Гибкость — вариантность, полиморфные методы в интерфейсах, higher-rank types и т.д. просто работают;
− Аллокация в куче и разыменование указателя может очень сильно замедлить код;
− Поскольку информация о типах стирается, нельзя ничего сделать с типовым аргументом, не имея его обитателей (например, запросить рефлексией информацию или сделать `is` проверку).

Такого подхода придерживаются JVM, Haskell и, как правило, другие функциональные языки ввиду его гибкости и скорости компиляции.

Особую проблему вызывает работа с примитивами и другими value-типами, потому что каждое значение приходится сначала боксить (переносить в кучу), а потом уже использовать в полиморфном контексте. Поэтому языки борются с этим как могут. Некоторые языки урезают диапазоны значений примитивов, чтобы зарезервировать бит, определяющий, это указатель или значение. Код консультируется с этим битом для работы (похоже на 2.2.4). Так делают, например, OCaml и Koka. Агрессивный инлайнинг тоже помогает. Java пытается аккуратно двигаться в сторону возможности мономорфизации[23][24].

2.2.3 Гибридный подход

C# реализует гибридный подход[25]. Они различают значения, хранимые в куче — reference types, и значения, хранимые на стеке — value types. Для первых они генерируют одну специализацию, работающую с указателями. Для каждого набора value-типов они генерируют лениво в рантайме специализации.

То есть следы дженериков в таком подходе есть и промежуточном представлении CIL, и в рантайме.

+ value-типы хранятся и передаются as-is без боксинга;
+ Доступна рефлексия по дженерикам;
+ Небольшое время компиляции;
− Инстанциация в рантайме замедляет исполнение;
− Variance работает только для reference types (что странно — есть “правильная” подтипизация, а есть “неправильная”).

[23] Type Specialization of Java Generics - What If Casts Have Teeth ?
[24] `https://cr.openjdk.org/~jrose/values/parametric-vm.html`
[25] Generics in the runtime (C# programming guide).

```
struct value_witness_table {
  size_t size, align;
  void (*copy_init)(opaque *dst, const opaque *src, type *T);
  void (*copy_assign)(opaque *dst, const opaque *src, type *T);
  void (*move_init)(opaque *dst, opaque *src, type *T);
  void (*move_assign)(opaque *dst, opaque *src, type *T);
  void (*destroy)(opaque *val, type *T);
};
```

Рис. 7: Swift value witness table.

Example:

```
func f<T>(_ t: T) -> T {
  let copy = t
  return copy
}
```

Implementation:

```
void f(opaque *result, opaque *t, type *T) {
  opaque *copy = alloca(T->vwt->size);
  T->vwt->copy_init(copy, t, T);
  T->vwt->move_init(result, copy, T);
  T->vwt->destroy(t, T);
}
```

Рис. 8: Код полиморфной функции, порождаемый компилятором Swift.

2.2.4 Использование виртуальной таблицы свойств типов

Swift[26] вместе с каждым типовым параметром передаёт value witness table (рис. 7). Это таблица со всей необходимой информацией, о типе: размер и выравнивание, что нужно сделать при копировании и перемещении объекта (например, инкрементировать счётчик ссылок). Таким образом, скомпилированный код постоянно обращается к этой таблице и делает виртуальные вызовы функций из неё (рис. 8).

+ Небольшое время компиляции;
+ Предсказуемая эффективность (не приводит к неожиданным паузам в рантайме);
+ Эффективная работа с value-значениями;
+ Высокая гибкость;
+ Информация о типах не стирается;
− Серьёзный константный оверхед на динамические вызовы через таблицу, эффективность очень сильно зависит от компиляторных оптимизаций.

Своего рода реализация параметрического полиморфизма через специальный.

[26](youtube) 2017 LLVM Developers' Meeting: "Implementing Swift Generics"

2.3 Полиморфизм по конвенции вызова

Как мы уже обсуждали выше 2.2.2, параметрический полиморфизм в Haskell реализуется следующим образом: все значения хранятся в куче и передаются в полиморфные функции по указателю. Однако, если для вычислительного кода важна производительность, такой подход не годится ввиду большой нагрузки на подсистему управления памятью и множества индирекций. Поэтому Haskell позволяет также писать код с использованием unboxed значений. А если конвенция вызова не принципиальна, можно по ней абстрагироваться и писать один код для boxed и unboxed значений Eisenberg and Peyton Jones [2017].

2.3.1 Разновидности runtime представлений в Haskell

	Boxed	Unboxed
Lifted	*Int* *Bool*	
Unlifted	$ByteArray_{\#}$	$Int_{\#}$ $Char_{\#}$

Рис. 9: Виды значений в Haskell с примерами Eisenberg and Peyton Jones [2017].

На рисунке 9 можно увидеть классификацию значений в Haskell с примерами типов. **Unboxed типы** — их значения удерживаются и передаются по значению. **Boxed**, соответственно, наоборот, передаются по указателю и хранятся в куче. Обычный `Int` является просто декларацией следующего вида, где `I#` — это обычный конструктор с необычным именем, содержащий unboxed значение.

```
1 data Int = I# Int#
```

Lifted типы — содержат $\perp$ в качестве значения. Иначе говоря, могут содержать отложенные вычисления (это для них специальным образом обеспечивают компилятор и рантайм). **Unlifted типы** — наоборот, не могут быть отложенными. Операции, производящие значения unlifted типов всегда энергичные. Свойство lifted/unlifted называют **levity**. Чтобы распространить дальнейшее изложение на энергичные языки, можно levity заменить на boxity и всё останется справедливым.

`#` в именах типов и функций — это конвенция, показывающая, что где-то рядом происходит работа с unlifted значениями[27].

Также в Haskell есть unboxed кортежи, которых не существует на этапе исполнения. Например, следующая функция как бы возвращает пару значений, но в действительности компилятор может их разместить, например, в паре регистров. Соответственно, паттерн-матчинг по таким кортежам, просто позволяет сослаться на каждое из этих значений.

[27] Нужно подключить расширение MagicHash, чтобы пользоваться # в идентификаторах.

```haskell
divMod# :: Int -> Int -> (# Int, Int #)
case divMod# n k of (# quot, rem #) -> ...
```

Соответственно, нет никакого различия между по-разному вложенными unboxed кортежами:

```haskell
(# A, (# B, C #)) ≡ (# #( A, B #), C #) ≡ (# A, B, C #)
```

2.3.2 Классификация значений по runtime представлению

Значения различных типов могут быть на этапе исполнения устроены по-разному. То есть нам нужна некоторая система классификации типов. Но такая система в Haskell уж есть — кайнды. Опишем в виде структур данных предметную область, а потом продвинем на нужный уровень с помощью DataKinds 2.1.5.

Стандартная библиотека Haskell предоставляет следующие типы данных:

```haskell
TYPE :: RuntimeRep -> Type

data Levity = Lifted | Unlifted

data RuntimeRep = BoxedRep Levity
                | IntRep | DoubleRep
                | TupleRep [RuntimeRep]
                | SumRep [RuntimeRep]
                | ...

type LiftedRep = BoxedRep Lifted

type Type = TYPE LiftedRep
```

`TYPE` — это магический тип, определённый в компиляторе. Он параметризован runtime-представлением значений. Теперь привычный `Type` — это частный случай с boxed lifted значениями.

- `Int :: TYPE (BoxedRep Lifted)` или `:: Type`
- `IntRep` и `DoubleRep` соответствуют представлению численных констант (в зависимости от архитектуры процессора, целые числа и числа с плавающей запятой может быть необходимо располагать в различных специальных регистрах)
  `Int# :: TYPE IntRep`
- `Maybe Int :: Type`
- `Maybe :: Type -> Type`
- `TupleRep` и `SumRep` — unboxed алгебраические типы, представления параметризованы представлениями хранимых значений
  `(# Int, Bool #) :: TYPE (TupleRep '[LiftedRep, LiftedRep])`
- Для простоты, типы вложенных кортежей не унифицируются

```
(# Int#, (# Int, Double# #) #)
  :: TYPE (TupleRep '[IntRep, TupleRep '[LiftedRep, DoubleRep]])
```

2.3.3 Representation polymorphism

Выставив runtime-представление в структуре кайндов, мы теперь можем параметризоваться по ним. Например, кайнд функциональной стрелки выглядит следующим образом[28]:

```
ghci> :k (->)
(->) :: forall {q :: RuntimeRep} {r :: RuntimeRep}. TYPE q -> TYPE r -> Type
```

Упражнение 15 *Подумайте, почему функция имеет boxed тип. Может ли быть иначе? Может ли это быть полезным?*

К сожалению, Haskell выставляет довольно строгое ограничение: связыватели не могут иметь тип, полиморфный по runtime представлению. Можно легко предположить, почему, — нельзя сгенерировать код функции для работы с параметром произвольного рантайм-представления. Это можно решить только мономорфизацией 2.2.1, но Haskell избегает этого подхода[29]. Сообщество также пытается найти другие решения[30] (что-то вроде 2.2.4).

Например, изначально оператор аппликации был обобщён только по возвращаемому типу. Это не порождает проблем, так как вызывающий код сможет вывести представление и сгенерировать подходящий код:

```
($) :: forall r a (b :: TYPE r). (a -> b) -> a -> b
f $ x = f x
```

Однако, было замечено, что для оператора аппликации можно получить другую реализацию, не использующую levity-полиморфное связывание[31]:

```
($) :: forall ra rb (a :: TYPE ra) (b :: TYPE rb). (a -> b) -> a -> b
($) f = f
```

Таким образом, в Haskell полиморфизм по представлениям несколько вырожден и помогает лишь в небольшом количестве случаев, однако немаловажных. Если позволить мономорфизацию по `RuntimeRep` параметрам, получится система аналогичная гибридной реализации параметрического полиморфизма 2.2.3, только с большим контролем со стороны программиста над мономорфизацией.

[28] Выключить упрощения: `ghci> :set -fprint-explicit-foralls -fprint-explicit-runtime-reps`
[29] https://gitlab.haskell.org/ghc/ghc/-/issues/14917
[30] https://mail.haskell.org/pipermail/haskell-cafe/2023-January/135770.html
[31] https://gitlab.haskell.org/ghc/ghc/-/merge_requests/10131

3 Специальный (ad-hoc) полиморфизм

Как-то Joe Fasel в разговоре с Philip Wadler высказал идею того, что перегрузка функций (overloading) должна находить своё отражение в типах. Wadler понял его неправильно Hudak et al. [2007]. Но то, что он понял, — оказалось классами типов Wadler and Blott [1989].

Christopher Strachey ввёл классификацию полиморфизма на две категории Strachey [2000]. Параметрический — один и тот же код работает с данными различных типов. **Специальный (ad-hoc) полиморфизм** — код выбирается в зависимости от типа. Например, один и тот же символ умножения по-разному действует на целые числа и на числа с плавающей точкой.

Перегрузка в языках обозначает возможность назвать несколько функций с различными наборами входных параметров одинаково. В месте вызова компилятор статически определяет по типам аргументов, какую из них действительно следует вызвать.

```
string toString(x: int) { ... }
string toString(fmt: String, d: double) { ... }
```

Классы типов обязуют сначала задекларировать именованную сущность (собственно, класс типов), включающую в себя пачку деклараций функций, которые могут быть перегружены для различных типов.

```
class Show a where
  show :: a -> String

instance Show Int where
  show :: Int -> String
  show = ...
```

Необходимо заметить, что декларация класса типов содержит формальный типовой параметр, по вхождениям которого в тип функции, собственно, выбирается перегрузка. Таких параметров может быть много (MultiParamTypeClasses), они могут иметь стрелочные кайнды. Например, в случае класса типов `Applicative`, выбор реализации операции `pure` будет происходить по типовому конструктору результата, то есть даже не по полноценному типу.

```
class Functor f => Applicative (f :: Type -> Type) where
  pure :: a -> f a
  ...

instance Applicative Maybe where
  pure :: a -> Maybe a
  ...
```

Также, в отличие от перегрузки, классы типов совместимы с параметрическим полиморфизмом. Так, в типе полиморфной функции нельзя указать, что для типа должна присутствовать определённая перегрузка. Классы типов же позволяют ограничить набор возможных типовых аргументов теми, для которых реализован инстанс нужного класса типов:

```haskell
showPrefixed :: Show a => a -> String -> String
```

Если сравнивать классы типов с переопределением (overriding) в ООП языках, то разрешение вызова виртуальной функции происходит с использованием таблицы, хранящейся объекте первого параметра (получателя вызова, receiver). Классы типов же опираются исключительно на тип, поэтому, например, возможно определение констант в классах типов[32]:

```haskell
class Enum a => Bounded a where
  minBound :: a
  maxBound :: a
```

В то же время, классы типов не являются типами, а, скорее, предикатами на типах. Тип удовлетворяет такому предикату, или свойству, если для него есть соответствующий инстанс. Поэтому, в частности, привычный способ в ООП создать гетерогенную коллекцию элементов, имеющих общий интерфейс, напрямую не сработает с классами типов. Например, такой тип не будет корректным: `[Show]`. Мы вернёмся к этой проблеме в 3.4.2.

3.1 Классы типов в языке

Несмотря на поразительное могущество, идея реализации классов типов крайне проста. Она была уже во всей полноте представлена в первой работе Wadler and Blott [1989]. В дальнейших работах уточнялся механизм вывода типов в виде сведения к классической системе типов в стиле Hindley-Milner Hall et al. [1996]. Остальные работы, в основном, предлагают огромное разнообразие различных расширений и приложений Jones et al. [1997].

3.1.1 Словари

Рассмотрим идею реализации классов типов на примере полиморфной сортировки. Сортировка для списка элементов конкретного типа пишется тривиально:

```haskell
sort :: [Int] -> [Int]
sort = \case [] -> []; x:xs -> insert x (sort xs)
  where
    insert x xs = let (l, r) = List.partition (<) x) xs in l ++ x : r)
```

В реализации единственная информация о типе, которой мы пользуемся — порядок на его обитателях. Таким образом, при переходе к полиморфной сортировке, нам нужно принять словарь с предикатами, задающими нужный порядок для данного типа.

```haskell
data OrdDict a = OrdDict { less :: a -> a -> Bool }

sort :: OrdDict a -> [a] -> [a]
```

[32] Современные ООП языки, тем не менее, стремятся поддержать статические функции в интерфейсах, что делает их ближе к классам типов и позволяет делать похожие вещи. Например, Swift.

```haskell
sort d@OrdDict{ less } = \case [] -> []; x:xs -> insert x (sort d xs)
  where
    insert x xs = let (l, r) = List.partition (`less` x) xs in l ++ x : r
```

Теперь, чтобы воспользоваться сортировкой на списке чисел, нужно сконструировать нужный рекорд и вызвать с ним функцию на списке конкретных типов:

```haskell
intOrd :: OrdDict Int
intOrd = OrdDict { less = (<) }
```

```
ghci> sort intOrd [3, 2, 1]
```

Возможна ситуация, когда инстанс для одного типа зависит от инстанса для другого Например, порядок на списках можно получить автоматически, зная порядок на элементах. В случае словарей мы это моделируем функцией между словарями:

```haskell
listDict :: OrdDict a -> OrdDict [a]
listDict d = OrdDict { less = ... less d ... }
```

Теперь мы можем сортировать список списков, конструируя нужный словарь:

```
ghci> sort (listDict intDict) [[3, 2], [2, 1], [0]]
```

Сравнение явной передачи словарей и классов типов можно увидеть в следующей таблице:

1. Определение словаря функций

```haskell
data MyOrd a = MyOrd
  { less :: a -> a -> Bool }
```

2. Экземпляр словаря для конкретного типа
   - Именованное значение

```haskell
intMyOrd :: MyOrd Int
intMyOrd = MyOrd { less = (<) }
```

3. Явный параметр функции

```haskell
sort :: MyOrd a -> [a] -> [a]
```

4. Передаётся пользователем

```haskell
test = sort intMyOrd [3, 2, 1]
```

1. Определение класса типов

```haskell
class MyOrd a where
  less :: a -> a -> Bool
```

2. Объявление типа представителем класса типов
   - Не имеет имени

```haskell
instance MyOrd Int where
  less = (<)
```

3. Неявный параметр функции

```haskell
sort :: MyOrd a => [a] -> [a]
```

4. Передаётся компилятором

```haskell
test = sort [3, 2, 1]
```

Таким образом, словарь — это **свидетель (witness)** или доказательство того, что тип удовлетворяет ограничению.

Упражнение 16 *Какой словарь будет соответствовать higher-kinded классу типов* `Functor`*?*

3.1.2 Неявные аргументы

Можно думать так, что слева от `=>` передаются неявные аргументы функций, выводимые компилятором из контекста. То есть, например, не стоит удивляться вхождениям `=>` в типе аргумента, это просто функция с неявным аргументом. Так, следующий код не скомпилируется, потому что в месте использования переменной `y` нет значения типа `Show b`:

```
f :: (Show b => b) -> b
f x = x -- ошибка
```

Можно это значение принять в функции `f`, тогда оно автоматически пропагируется в `y`:

```
f :: Show b => (Show b => b) -> b
f x = x
```

Расширение ImplicitParams даёт возможность делать некоторые аргументы функции неявными. Фактически, это реализация динамического связывания в статическом языке Lewis et al. [2000] (см. далее **??**). Неявные аргументы берутся из скоупа по имени и подставляются автоматически:

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]

sort :: (?cmp :: a -> a -> Bool) => [a] -> [a]
sort = sortBy ?cmp
```

Haskell также предоставляет возможность сохранять словари в структуры данных:

```
data ShowDict a where
  ShowDict :: Show a => ShowDict a

f :: ShowDict b -> (Show b => b) -> b
f d x = case d of ShowDict -> x -- в скоупе доступен инстанс Show b
```

Упражнение 17 *Возможна ли именно такая семантика в энергичном языке? Почему?*

3.1.3 Вывод инстансов

Чтобы вызвать ограниченно-полиморфную функцию, GHC производит вывод инстансов или, иначе говоря, автоматически конструирует свидетелей. Вывод инстансов тесно интегрирован с общей системой вывода типов Haskell Peyton Jones [2019].

В действительности вывод инстансов это не что иное, как *задача населения типа*. Действительно, после трансляции в Core (промежуточное представление в GHC), классы типов представляют собой словари функций. У нас в контексте имеются конкретные словари и функции, позволяющие из одних словарей получать другие. Требуется найти терм, конструирующий словарь нужного типа.

Пусть, например, внутри функции `f :: Show a => ..` происходит вызов ограниченно–полиморфной функции `g :: Show [a] -> ...` То есть, у нас имеется словарь `d1 :: ShowDict a`, а так же функция `d2 :: ShowDict a -> ShowDict [a]`, пришедшая из импортов[33]. Необходимо сконструировать терм типа `ShowDict [a]`. Очевидно, это будет просто аппликации одного к другому: `d2 d1`.

Вывод инстансов происходит рекурсивно. Чтобы вывести `ShowDict [a]`, выводится сначала посылка `ShowDict a`. То есть получается рекурсия по структуре типа. Иначе говоря, вывод инстансов можно эксплуатировать как вычислительный примитив уровня типов. Так, например, мы можем опускать информацию из типов в термы (аналогично GHC.TypeLits):

```
type data Nat = Zero | Suc Nat

class KnownNat (n :: Nat) where
  natVal :: Int

instance KnownNat Zero where
  natVal = 0

instance KnownNat n => KnownNat (Suc n) where
  natVal = 1 + natVal @n

ghci> natVal @(Suc (Suc Zero))
-- выведется natVal {knownSuc (knownSuc knownZero)}
```

В общем случае процесс населения типа, как можно предположить по вычислительной аналогии, неразрешим. Поэтому GHC накладывает большое количество ограничений на вид инстансов, которые гарантируют тотальность вывода. Подробно эти ограничения описаны в Sulzmann et al. [2007a]. Также GHC предоставляет различные расширения, ослабляющие эти ограничения и перекладывающие часть ответственности на плечи программиста[34]. Например, с UndecidableInstances можно легко написать разворот списка типов на этапе компиляции, как и любую другую функцию:

```
class Reverse (acc :: [Type]) (tys :: [Type]) where
  showReverse :: String

instance ShowT acc => Reverse acc '[] where
  showReverse = showTypes @acc

instance Reverse (ty : acc) tys => Reverse acc (ty : tys) where
  showReverse = showReverse @(ty : acc) @tys
```

[33] Инстансы можно импортировать пустым импортом: `import Module ()`.

[34] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/instances.html

```
ghci> showReverse @'[] @'[Char, Int, Double]
```

Можно заметить, что процесс вывода классов типов очень похож на вычисление логических программ, например, на Prolog, только без backtracking'а (перебора различных вариантов решений в поисках подходящего). Как, впрочем, и вывод типов в Haskell Peyton Jones [2019] в целом: собранные по программе эквивалентности можно рассматривать как логическую программу, решение этой системы типовых уравнений — как исполнение этой программы.

Между классами типов и выводом типов существует интересная синергия (рис. 10)[35]. Исходя из термов, выводятся типы. Затем, исходя из типов, выводятся инстансы классов типов. То есть мы пишем какой-то интересный интеллектуальный код, а параллельно с нами компилятор выписывает неинтересный код.

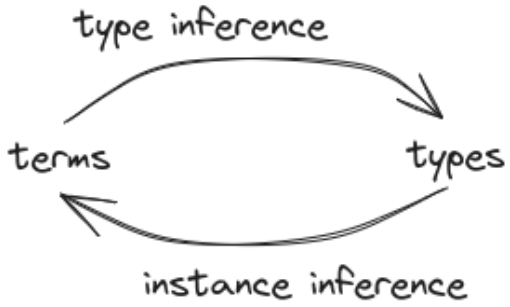


Рис. 10: Классы типов + вывод инстансов = кодогенерация.

Вывод инстансов опирается только на вид "головы" декларации — справа от `=>`, а ограничения слева применяются постфактум. Это можно использовать, чтобы писать более общие инстансы. Так, например, работает **constraint trick**[36], позволяющий резолвить ad-hoc полиморфные функции в параметрически-полиморфном контексте.

3.1.4 Построение типа по значению

После того как мы научились опускать значения из типов, закономерно научиться обратному — поднимать значения в типы. Воспользуемся техникой, описанной в Kiselyov and Shan [2004]. Существует соответствующая библиотека Data.Reflection (см. далее 3.4.7).

В действительности мы, конечно, не можем честно получить синтаксически тип нужного размера, просто потому, что типы существуют строго до стадии исполнения. Однако, как мы знаем, словари классов типов имеют воплощение в рантайме (случай полиморфной рекурсии — как раз пример, когда этого нельзя полностью избежать). Поэтому воспользуемся continuation passing style, который будет подробно рассмотрен далее в главе **??**: вместо

[35] (youtube) Hackett: a metaprogrammable Haskell.

[36] `https://chrisdone.com/posts/haskell-constraint-trick/`

того, чтобы вернуть результат, примем продолжение, умеющее работать с любым типом с KnownNat (пользуемся типовыми абстракциями и аппликациями, см. 2.1)[37]:

```
reify :: Int -> (forall n. KnownNat n => a) -> a
reify n k
  | n <= 0 = k @Zero
  | otherwise = reify (n - 1) \@n' -> k @(Suc n')
```

Продолжение, передаваемое в рекурсивный вызов, захватывает словарь для типа n и конструирует словарь для Suc n.

Наконец, можем написать следующую удивительную тождественную функцию, поднимающую сначала значение в тип, а потом опускающее тип обратно в термы:

```
wonderId :: Int -> Int
wonderId n = reify n (\@t -> natVal @t)
```

3.1.5 Имплиситы и когерентность

Классы типов можно не делать специальным языковым механизмом, но вместо этого предоставлять на языковом уровне неявные параметры и население, достаточные для реализации классов типов.

Так, в Scala существует механизм имплиситов (implicits) Křikava et al. [2019][38]. Параметры функций могут быть помечены ключевым словом implicit, тогда Scala попытается их вывести самостоятельно с помощью доступных в скоупе implicit деклараций. Объявления переменных, функций и конструкторов объектов также могут быть помечены implicit, тогда они будут использоваться при населении. Теперь мы можем смоделировать словарь функций, например, с помощью интерфейсов (которые в Scala называются trait) и ООП синглтонов, чтобы получить классы типов Oliveira et al. [2010]:

```
// Пачка функций.
trait Show[T] {
    def show(x: T): String
}

// Обёртка для удобства вызова.
def show[T](x: T)(implicit ev: Show[T]): String = ev.show(x)

// Объект-синклтон, значение для пачки функций.
implicit object intShow extends Show[Int] {
    def show(x: Int): String = x.toString
}
```

[37] В не самых свежих версиях GHC потребуется воспользоваться техникой Proxy из 2.1.1.

[38] Дизайн неявных параметров в Scala3 изменился (youtube) Scala Implicits Revisited, Martin Odersky.

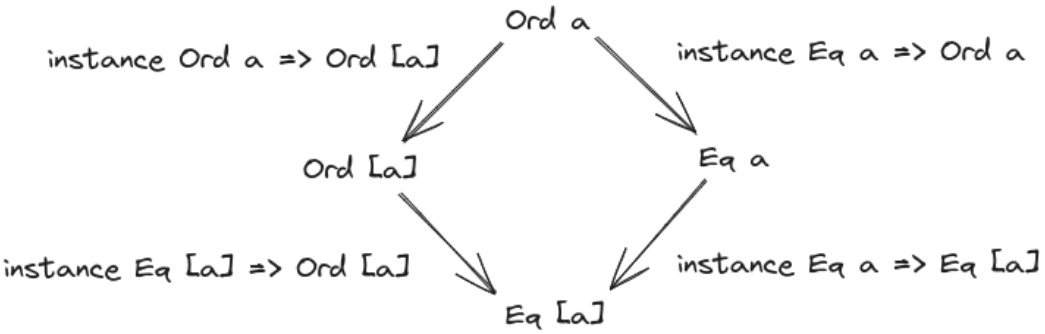


Рис. 11: Когерентность инстансов — диаграмма коммутирует.

```
11 def showAll[T](xs: List[T])(implicit ev: Show[T]): String =
12     xs.map(show(_)).join(", ")
```

Как мы говорили ранее (2.1), работа с типовыми параметрами похожа на работу с обычными. Также, вывод типов можно рассматривать как процесс восстановления пропущенных типовых аппликаций. Заметим, что примерно этим же занимается и механизм вывода имплиситов. Таким образом, при попытке сделать функции с имплиситами функциями первого класса, будут возникать сложности схожие со сложностями first-class полиморфизма (2.1.2).

В языках с зависимыми типами неявные параметры[39] особенно нужны, потому что, например, типы — это ровно такие же параметры функции, как и все остальные. Поэтому вывод типов — это фактически вывод неявных аргументов функций. Более того, зависимые функции, вместе с аргументами часто принимают доказательства каких-то свойств этих аргументов, которые тоже хочется по возможности выводить из контекста автоматически. Такой механизм вывода можно переиспользовать для эмулирования классов типов[40] Devriese and Piessens [2011]. В обратную сторону тоже работает — можно механизмы зависимой типизации эмулировать классами типов McBride [2002].

Как мы увидим далее, неявные параметры сами по себе тоже нужны как статическая аппроксимация динамических свободных переменных для реализации системы эффектов (см. далее **??**). Однако, иметь классы типов отдельной языковой возможностью всё же полезно, несмотря на то, что они, вроде бы, представляют собой те же неявные параметры (пусть и с рекурсивным механизмом населения). Так, можно поддержать важное свойство при соблюдении всех ограничений, т.е. при отсутствии **orphan instances**[41]. **Когерентность инстансов (coherence)** — для одного типа все инстансы данного класса типов, полученные разными способами, неотличимы (рис. 11). Соответственно, не имеет значения происхождение того или иного инстанса. Иначе говоря, об этом можно не думать, это снимает существенное

[39] https://agda.readthedocs.io/en/v2.7.0.1/language/implicit-arguments.html
[40] https://agda.readthedocs.io/en/v2.7.0.1/language/instance-arguments.html
[41] https://stackoverflow.com/questions/3079537/orphaned-instances-in-haskell

количество когнитивной нагрузки и упрощает рефакторинг[42]. В то время как остальные подходы требуют трепетного отношения к контексту вызова, потому что из него может прийти неожиданная реализация.

3.1.6 Правила (rules) и специализация

GHC позволяет прямо в коде, с помощью специально прагмы, указывать оптимизирующие правила переписывания для компилятора[43] Jones et al. [2001]. Например:

```
{-# RULES
  "map/map"    forall f g xs.  map f (map g xs) = map (f . g) xs
  "map/append" forall f xs ys. map f (xs ++ ys) = map f xs ++ map f ys
 #-}
```

Первый закон представляет собой не что иное, как закон функторов. В идеале, мы формулируем законы на этапе дизайна Maguire [b], проверяем их выполнение с помощью property-based testing[44], а потом используем их для оптимизаций.

Можно переписать полиморфную версию функции на специализированную, если типы подходят. Для этого нужно реализовать специализированную версию (совпадение семантики — полностью ответственность программиста) и задать соответствующее правило переписывания:

```
genericLookup :: Ord a => Table a b   -> a   -> b
intLookup     ::          Table Int b -> Int -> b

{-# RULES "genericLookup/Int" genericLookup = intLookup #-}
```

Основной эффект такой оптимизации — гарантированное превращение динамических вызовов функций классов типов в статические (потому что тип известен, следовательно, — и соответствующий ему словарь).

3.1.7 Отступление: дефункционализация

Дефункционализация (defunctionalization) — техника избавления от функций высших порядков в программе[45] Xia. Впервые предложена в Reynolds [1972, 1998].

Идея заключается в том, чтобы заменить каждое создание лямбда-функции вызовом конструктора некоторого алгебраического типа данных. А каждый call-cite функции заменить на вызов специальной first-order функции `apply`, интерпретирующей данный алгебраический тип.

Рассмотрим пример из функции высших порядков `map` и двух колсайтов, создающих лямбда-функции:

[42] Edward Kmett - Type Classes vs. the World.
[43] `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/rewrite_rules.html`
[44] (youtube) John Hughes - Keynote: How to specify it!
[45] `https://en.wikipedia.org/wiki/Defunctionalization`

```haskell
map :: (Int -> Int) -> [Int] -> [Int]
map f = \case [] -> []; x:xs -> f x : map f xs

example1 xs = map (\x -> x + 1) xs
example2 y xs = map (\x -> x * y) xs
```

Для каждого лямбда-литерала заводим по конструктору, хранящему замыкание. Аппликацию заменяем на вызов `apply`. Таким образом получили функцию `map` первого порядка.

```haskell
data Fun = F1 | F2 Int
apply :: Fun -> Int -> Int
apply df x = case df of F1 -> x + 1; F2 y -> x * y

map :: Fun -> [Int] -> [Int]
map df = \case [] -> []; x:xs -> apply df x : map df xs

example1 xs = map F1 xs
example2 y xs = map (F2 y) xs
```

3.1.8 Эмуляция полиморфизма высших порядков

Далеко не во всех языках есть полиморфизм высшего ранга, но иногда он бывает полезен. Самое распространённое его применение — эмуляция классов типов стрелочных кайндов вроде `Monad`.

Заметим, что типовый конструктор кайнда `Type -> Type` — это функция на типах, принимающая один тип, и возвращающая другой. Применим дефункционализацию, чтобы избежать необходимости параметризовать один типовый конструктор другим Xia, Yallop and White [2014].

Поставим в соответствие типовому конструктору `List` тип-“символ” `ListSym` (для примера используем Kotlin):

```kotlin
class ListSym
```

Заведём тип, соответствующий аппликации символа к типу, хранящий оригинальное значение со стёртым типом:

```kotlin
class Apply<Sym, T>(val value: Any)
```

Установим изоморфизм между изначальным типом, полученным типовой аппликацией конструктора, и новой аппликацией символа:[46]

```kotlin
fun <T> List<T>.to(): Apply<ListSym, T> = Apply(this)
fun <T> Apply<ListSym, T>.from(): List<T> = this.value as List<T>
```

[46] Слева от точки в декларации указывается дополнительный аргумент функции с синтаксисом передачи совпадающим с вызовом метода на объекте. Из тела функции на него можно ссылаться с помощью `this`.

Теперь мы можем объявить интерфейс монад и задать реализацию для списка с помощью объекта-синглтона:

```
interface Monad<M> {
    fun <T> pure(x: T): Apply<M, T>
    infix fun <T, R> Apply<M, T>.bind(k: (T) -> Apply<M, R>): Apply<M, R>
}

object ListMonad : Monad<ListSym> {
    override fun <T> pure(x: T): Apply<ListSym, T> = listOf(x).to()
    override fun <T, R> Apply<ListSym, T>.
            bind(k: (T) -> Apply<ListSym, R>): Apply<ListSym, R> =
        this.from().flatMap { k(it).from() }.to()
}
```

И наконец мы можем писать функции над произвольными монадами:[47]

```
fun <M> Monad<M>.go(x: Apply<M, Int>): Apply<M, Int> =
    x bind { it -> pure(it + 1) } bind { it -> pure(it + 2) }

fun test(xs: List<Int>): List<Int> = ListMonad.go(xs.to()).from()
```

Не лишним будет отметить, что результирующий код выглядит несколько чудовищно. Скорее всего, использование этой техники не окупает себя и нужно выбирать другой стиль программирования.

3.2 Семейства

Идея ad-hoc полиморфизма в том, чтобы в зависимости от типа получать различный код. Семейства начинались как продолжение этой идеи в плоскость данных и типов. Так, ассоциированные синонимы типов — различные типы для различных индексов (типовых параметров) Chakravarty et al. [2005a]. Ассоциированные `data` — различные представления для различных индексов Chakravarty et al. [2005b]. В конце концов эти идеи были обобщены до открытых семейств Schrijvers et al. [2008], потом были введены закрытые Eisenberg et al. [2014].

Можно считать, что семейства[48][49] — это типовые конструкторы, задающие множество типов. Конкретный тип из множества можно выбрать, передав типовой параметр, называемый **индексом**. Сравните с обычными полиморфными конструкторами типа, которые ведут себя одинаково вне зависимости от типовых параметров.

Большое количество интересных примеров использования можно найти, например, в Kiselyov et al. [2010].

[47] Аргументы слева от точки умеют самостоятельно запрыгивать в последующие вызовы. Собственно говоря, они являются неявными параметрами.

[48] `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_families.html`

[49] `https://serokell.io/blog/type-families-haskell`

3.2.1 Data families

Data families позволяют выбирать декларацию алгебраического типа в зависимости от типового индекса. Например, для более эффективной реализации структур данных. Это напоминает специализацию шаблонов в C++.

```
data family XList elem
data instance XList () = IntList Int
data instance XList Bool = BoolList ByteArray
```

Единственный способ работать с data family — разместить реализацию в классе типов, чтобы она сопровождала значение с того момента, когда типовой индекс конкретный.

```
class XListOp elem where
  xelem :: elem -> XList elem -> Bool

instance XListOp () where
  xelem () (IntList size) = size > 0

instance XList Bool where
  xelem key (ByteArray bs) = ...

xelemAll :: XListOp elem => XList elem -> [elem] -> Bool
xelemAll xs = all (`xelem` xs)
```

К сожалению, в отличие от специализации шаблонов, нельзя задать определение по-умолчанию вроде `data instance XList a = AList [a]`.

3.2.2 Synonym families

Synonym families (семейства типов) фактически являются функциями на типах. Они бывают открытыми, закрытыми и ассоциированными. Открытые семейства открыты в том смысле, что инстансы можно писать отдельно от декларации, подобно классам типов. Закрытые, наоборот, полностью описываются в одном месте, кейсы упорядочены сверху вниз, что несколько ослабляет ограничения на паттерны.

```
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus Zero m = m
  Plus (Suc n) m = Suc (Plus n m)
```

Чтобы посмотреть, во что вычисляется нечто на уровне типов, можно воспользоваться следующей командой в ghci:

```
ghci> :k! Plus (Suc Zero) (Suc (Suc Zero))
Plus (Suc Zero) (Suc (Suc Zero)) :: Nat
= Suc (Suc (Suc Zero))
```

Ассоциированные семейства работают аналогично, только объявляются в рамках некоторого класса типов, являясь некоторой функциональной альтернативой FunctionalDependencies Jones [2000] (которые выглядят скорее реляционно). Иначе говоря, позволяют поставить в соответствие типу, для которого написан инстанс, другой тип. Например, коллекции — тип её элементов:

```
class Container c where
  type Elem c
  elements :: c -> [Elem c]

instance Container [a] where
  type Elem [a] = a
  elements = id

instance Container ByteString where
  type Elem ByteString = Word8
  elements = ByteString.unpack
```

В современных языках часто встречаются ассоциированные семейства под видом ассоциированных типов[50]. Так, Swift сильно полагается на ассоциированные типы, вовсе не поддерживая дженерики в протоколах (интерфейсах)[51]. В то же время Scala пытается отслеживать, в присутствии экзистенциальных типов (интерфейсов), из какого именно значения пришёл тот или иной ассоциированный тип с помощью path-dependent types Amin et al. [2014].

3.2.3 Инъективные семейства

Семейства типов отличаются от типовых конструкторов примерно так же, как функции от конструкторов данных. Конструкторы пассивны и не редуцируются (`Maybe Int`), в то время как функции вычисляются в какой-то результат (e.g. `F Int` $\sim$ `Bool`). В частности, как и функции, семейства не обязательно инъективны.

Для типовых конструкторов, зная, что сконструированные ими типы эквивалентны, можно вывести, что типовые аргументы эквивалентны тоже. Например:

```
Maybe a ∼ Maybe b ⇒ a ∼ b
```

Очевидно, что для классов типов это свойство по умолчанию не выполняется:

[50] Ассоциированные типы являются фактически экзистенциальными типами, их связывает с ассоциированными семействами логический процесс сколемизации.

[51] (youtube) 2017 LLVM Developers' Meeting: "Implementing Swift Generics"

```haskell
type family NonInjective a where
  NonInjective Int = Double
  NonInjective Char = Double
```

Haskell предоставляет явный синтаксис для объявления инъективных семейств типов, напоминающий функциональные зависимости в классах типов (TypeFamilyDependencies) Stolarek et al. [2015]. Конечно, компилятор проверит, что реализация инъективна. Синтаксис требует связать результат именем через равенство и указать, аналогично FunctionalDependencies, что результат определяет какие-то из типовых индексов семейства:

```haskell
type family InjectiveB a b = r | r -> b
  ...
```

3.2.4 Семейства первого класса

Помимо инъективности, классы типов также не обязательно обладают свойством **generavity**, критически важным для вывода типов, — один и тот же результат не обязательно получен из того же самого семейства:

```
f a ∼ g a ⇒ f ∼ g
```

Вместе инъективность и generavity — **matchability**. Когда Haskell работает с типом стрелочного кайнда, он подразумевает, что этот тип matchable. Соответственно, семейства не могут передаваться в качестве параметров, а все их вхождения должны быть полностью применёнными ко всем аргументам (fully saturated). Либо нужно явно указать, что семейство возвращает конструктор (в этом конкретном месте семантика зависит от переноса аргументов направо от `::`):

```haskell
type family ToCtor (s :: Symbol) :: Type -> Type where
  ToCtor "maybe" = Maybe
  ToCtor "identity" = Identity
```

Одним из способов обойти это ограничение является дефункционализация семейств Xia, Eisenberg and Stolarek [2014]. Как мы и обсуждали ранее 3.1.7, вместо функции первого класса заводится некоторый символ, обозначающий её, и функция интерпретации, умеющая сделать действие, соответствующее этому символу. В данном случае функцией интерпретации будет открытое семейство `Apply` [Maguire, a, глава 10].

В Haskell ведутся работы[52] по устранению saturation ограничения Kiss et al. [2019]. Для этого нужно различать matchable и unmatchable типовые функции. Это предлагается делать дополнительным индексированием стрелочных кайндов: $\to\equiv\to^{M}$ и $\twoheadrightarrow\equiv\to^{U}$. И, конечно, эти индексы могут быть полиморфными.

[52] GHC proposal: Unsaturated Type Families.

```haskell
data Matchability = Matchable | Unmatchable

hMap
  :: forall (m :: Matchability) (c :: Type -> Constraint)
   . forall (f :: Type ->^m Type) (as :: [Type])
   . All as c => (forall a. c a => a -> f a) -> HList as -> HList (Map f as)
```

3.3 Кайнд `Constraint`

Давно появлялись предложения добавить в GHC поддержку синонимов для констреинтов, семейств констреинтов и т.д Orchard and Schrijvers [2010]. В итоге был предложен[53] и реализован[54] некоторый механизм унификации типов и констреинтов. Таким образом, всё, что работало для типов, стало работать и для констреинтов.

В GHC с ConstraintKinds был добавлен специальный кайнд `Constraint`:

- Класс типов конструирует констреинт: `Monad :: (Type -> Type) -> Constraint`;
- Эквивалентность является констреинтом: `(a ~ b) :: Constraint`;
- Пустой кортеж констреинтов является констреинтом: `() :: Constraint`;
- Кортеж констреинтов является констреинтом: `(Eq a, a ~ b) :: Constraint`.

Теперь, например, мы можем реифицировать словарь как объект языка:[55]

```haskell
data Dict (c :: Constraint) where
  Dict :: c => Dict c
```

Вспомним гетерогенный список, рассмотренный ранее 2.1.5:

```haskell
data HList (tys :: [Type]) where
  HNil :: HList '[]
  HCons :: ty -> HList tys -> HList (ty : tys)
```

Эта структура данных является first-class аналогом variadic generics в C++ или Swift[56] (собственно, смысл вариадиков — не работать `HList`-подобными структурами напрямую). Например, мы можем написать `map` для такой структуры, если все типы удовлетворяют определённому ограничению. Для этого сначала реализуем семейство, генерирующее кортеж констреинтов для каждого типа из списка:

```haskell
type family All (c :: k -> Constraint) (tys :: [k]) :: Constraint where
  All c '[] = ()
  All c (ty : tys) = (c ty, All c tys)

-- All Show [Int, Double] ~ (Show Int, (Show Double, ()))
```

[53] https://gitlab.haskell.org/ghc/ghc/-/wikis/kind-fact
[54] http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/
[55] https://hackage.haskell.org/package/constraints-0.14.2/docs/src/Data.Constraint.html#Dict
[56] https://github.com/swiftlang/swift-evolution/blob/main/proposals/0398-variadic-types.md

Теперь можем реализовать `map`:

```
hmap :: forall c res tys . All c tys
     => (forall ty . c ty => ty -> res) -> HList tys -> [res]
hmap f = \case
  HNil -> []
  HCons x xs -> f x : hmap @c f xs

ghci> hmap @Show show (HCons (1 :: Int) $ HCons 'a' HNil)
```

Больше такого рода упражнений в гетерогенных конструкциях можно найти в de Vries and Löh [2014].

Констреинты также могут быть параметрически-полиморфными[57] Bottu et al. [2017]:

```
data Rose f x = Rose x (f (Rose f x))

instance (Eq a, forall b. Eq b => Eq (f b)) => Eq (Rose f a) where
  Rose x1 rs1 == Rose x2 rs2 = x1 == x2 && rs1 == rs2
```

Также, как и типы, констреинты параметризованы представлением[58]:

```
data CONSTRAINT (a :: RuntimeRep)
type Constraint = CONSTRAINT LiftedRep
```

3.4 Использование ad-hoc полиморфизма

Часто языки, имеющие что-то напоминающее классы типов, стремятся выразить через них как можно больше других языковых возможностей и полезных техник. Оказывается, это на удивление мощный механизм. Рассмотрим в этом параграфе некоторые избранные примеры.

3.4.1 Сериализация

Классическим примером использования классов типов является сериализация. Проблема в том, что десериализация производится, когда самого объекта ещё нет (ущербный Java подход заполнения объекта дефолтными значениями с последующей мутацией мы не рассматриваем). Поэтому нет возможности написать ООП интерфейс Serializable.

Стандартная библиотека сериализации в Kotlin[59] предоставляет сущность `KSerializer`, которая является интерфейсом для отдельного объекта-сериализатора нашего типа (для эффективности тут используется CPS в виде потоков событий encoder и decoder, вернёмся к этому подходу далее **??**):

[57] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/quantified_constraints.html
[58] https://hackage.haskell.org/package/base-4.21.0.0/docs/GHC-Exts.html#t:CONSTRAINT
[59] https://github.com/Kotlin/kotlinx.serialization

```kotlin
interface KSerializer<T> {
    fun serialize(encoder: Encoder, value: T)
    fun deserialize(decoder: Decoder): T
}
```

Очевидно, чтобы сконструировать сериализатор полиморфного типа, нужны сериализаторы типов-параметров. Что уже отчётливо напоминает классы типов.

```kotlin
class PairSerializer(
    keySerializer: KSerializer<K>,
    valueSerializer: KSerializer<V>,
): KSerializer<Pair<K, V>> { ... }
```

Однако в Kotlin нет классов типов, а значит создавать сериализаторы придётся вручную. С этим, однако несколько помогает пачка технологий: `inline fun`, `reified` дженерики, рефлексия и компиляторный плагин библиотеки, который, однако не безопасен с точки зрения типов.

3.4.2 Экзистенциальные типы

Можно задать тип данных с полиморфным конструктором и мономорфным типом. С его помощью можно, например, заполнить гомогенную коллекцию гетерогенными элементами.

```haskell
data Any where
  Any :: forall a . a -> Any  -- логически эквивалентно (exists a . a) -> Any

list :: [Any]
list = [Any 42, Any "Hello", Any (Just Nothing)]
```

В месте деконструирования `Any`, будет доступно значение некоторого неизвестного типа. Очевидно, с таким значением ничего сделать нельзя. Однако, помимо значения, можно положить в структуру данных свидетельство о том, что этот неизвестный тип удовлетворяет некоторому классу типов. Подобно fat pointers в Rust.

```haskell
data Has (c :: Type -> Constraint) where
  Has :: c a => a -> Has c
```

Значение типа `Has` свидетельствует о том, что существует некоторый населённый тип `a`, который принадлежит определённому классу типов. Например, рассмотрим `Show`:

```haskell
showAll :: [Has Show] -> String
showAll = List.intercalate "," . map \(Has x) -> show x
```

В общем случае, чтобы элиминировать такой тип данных нужны типы высших рангов:

```haskell
foldHas :: Has c -> (forall a . c a => a -> b) -> b
foldHas (Has x) k = k x
```

Подробнее можно посмотреть в [Maguire, a, глава 7] и [Pierce, 2002, глава 24].

3.4.3 Разрешение имён

Процесс разрешения имён (name resolution) в языках программирования определяет, с какой программной сущностью связать то или иное употребление имени. Разрешение имён рассматривает импорты, иерархию пространств имён и скоупов, типы выражений... Как правило, это сложный процесс, неотделимый от вывода типов.

Однако, в GHC стадия разрешения имён довольно простая и отрабатывает до вывода типов. На её тривиальную суть намекает её название — Renamer — она просто переписывает имена в программе на fully-qualified имена, опираясь на импорты.

С одной стороны, простота — это хорошо. С другой — строгое отделение от вывода типов накладывает неприятное ограничение: типы не могут участвовать в разрешении имён. Наиболее остро эта проблема стоит с метками полей в рекордах. Приходится называть все поля в модуле по-разному, чтобы избежать клешей.

Чтобы заставить разрешение имён зависеть от типов, Haskell снова прибегает к классам типов. А именно, определяется класс типов `IsLabel`, который зависит от символа и ожидаемого типа:

```
class IsLabel (s :: Symbol) a where
  fromLabel :: a
```

Для вызова `fromLabel` есть синтаксический сахар (OverloadedLabels):

```
#name ≡ fromLabel @"name"
```

Теперь разрешение имени `name` будет учитывать тип[60]:

```
data Pet = Pet { name :: String }
instance IsLabel "name" (Pet -> String) where
  fromLabel Pet{ name } = name

data Person = Person { name :: String, pets :: [Pet] }
instance IsLabel "name" (Person -> String) where
  fromLabel Person{ name } = name

ghci> #name pet
```

3.4.4 Несинтаксические типовые эквивалентности, System FC

Современный Haskell является синтаксически богатым языком, который, однако, несмотря не многообразие конструкций, транслируется в маленький типизированный внутренний язык. Это язык *System* F_C Sulzmann et al. [2007b], расширяет *System F* (2.1) несинтаксическими эквивалентностями типов. Оказывается, этого достаточно, чтобы поддержать такие

[60] Чтобы избавиться от ошибки переопределения, нужно включить NoFieldSelectors.

возможности Haskell как обобщённые алгебраические типы, ассоциированные семейства типов, функциональные зависимости и т.д.

А именно, вводится встроенный констреинт $\sim$, свидетельствующий о эквивалентности двух типов[61]. Например, тип функции `id` может быть записан таким странным образом:

```
f :: forall a b . a ~ b => a -> b
f = id
```

На самом деле это функция от четырёх параметров: двух типовых параметров, коерции и аргумента. Коерция — это значение размера 0, автоматически выводимое компилятором, которое является свидетельством того, что два соответствующих типа эквивалентны.

Например, GADT из 2.1.4 рассахаривается следующим образом:

```
data Expr ty where
  Const :: Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty
-- транслируется в
data Expr ty where
  Const :: forall ty . ty ~ Int => Expr ty
  IsZero :: forall ty . ty ~ Bool => Expr Int -> Expr ty
  If :: forall ty . Expr Bool -> Expr ty -> Expr ty -> Expr ty
```

И после паттерн-матчинга по конструкторам, констреинт эквивалентности попадёт в ветку и позволит системе вывода типов сделать необходимые переписывания.

Про вывод типов при наличии локальных предположений можно почитать в классической статье *OutsideIn*(X) Vytiniotis et al. [2011].

Очевидно, что Haskell может населить констреинт эквивалентности следуя рефлексивности, симметричности и транзитивности. Также, компилятор может генерировать новые аксиомы (пользователь напрямую свои аксиомы записать не может). Например, по семейству типов компилятор генерирует аксиомы равенства аппликированного конструктора семейства результирующим типам:

```
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus Zero m = m
  Plus (Suc n) m = Suc (Plus n m)
-- раскроется в
axiom Plus Zero m ~ m
axiom Plus (Suc n) m ~ Suc (Plus n m)
```

[61] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/equality_constraints.html

3.4.5 Коерции и роли

Haskell имеет поддержку **generative type abstractions** в виде `newtype` деклараций. Эта техника позволяет задавать доменно-специфичные типы, которые во время исполнения не отличимы от оборачиваемых типов, но позволяют различать их. Так, мы можем ввести обёртки для чисел, которые в предметной области представляют собой идентификаторы различных сущностей. Теперь система типов не даст их перепутать.

```
newtype ModuleId = ModuleId Int64
newtype CourceId = CourceId Int64
```

Существует крайне недооценённая практика программирования[62], когда у нас в программе есть чёткая граница, на которой происходит парсинг данных из внешнего мира. После неё сырые неструктурированные данные обогащаются структурой и принимают смысл внутри предметной области. Либо же мы отвергаем эти данные как некорректные. В оставшейся же части программы мы уже пользуемся типизированными данными, свойства которых уже установлены и гарантированы. Например, мы можем быть уверены, что число `ModuleId` строго больше нуля.

Однако, если у нас есть коллекция обёрнутых данных, а мы хотим с ней поработать как с коллекцией сырых, то нам придётся трансформировать коллекцию, несмотря на то, что эта трансформация ничего не делает. Оптимизатор Haskell не справится её элиминировать, потому что работает с типизированным промежуточным представлением и не сможет избавиться от преобразования, меняющего тип.

```
newtype Csv = Csv { unCsv :: String }

concatC :: [Csv] -> Csv
concatC = Csv . concat . map unCsv
```

Поэтому в Haskell есть механизм безопасных коерций между типами, у которых одинаковое представление во время исполнения. Это реализовано с помощью магического класса типов `Coercible`. Его имплементирует компилятор автоматически (см. рис. 12).

```
class Coercible from to where
  coerce :: from -> to
```

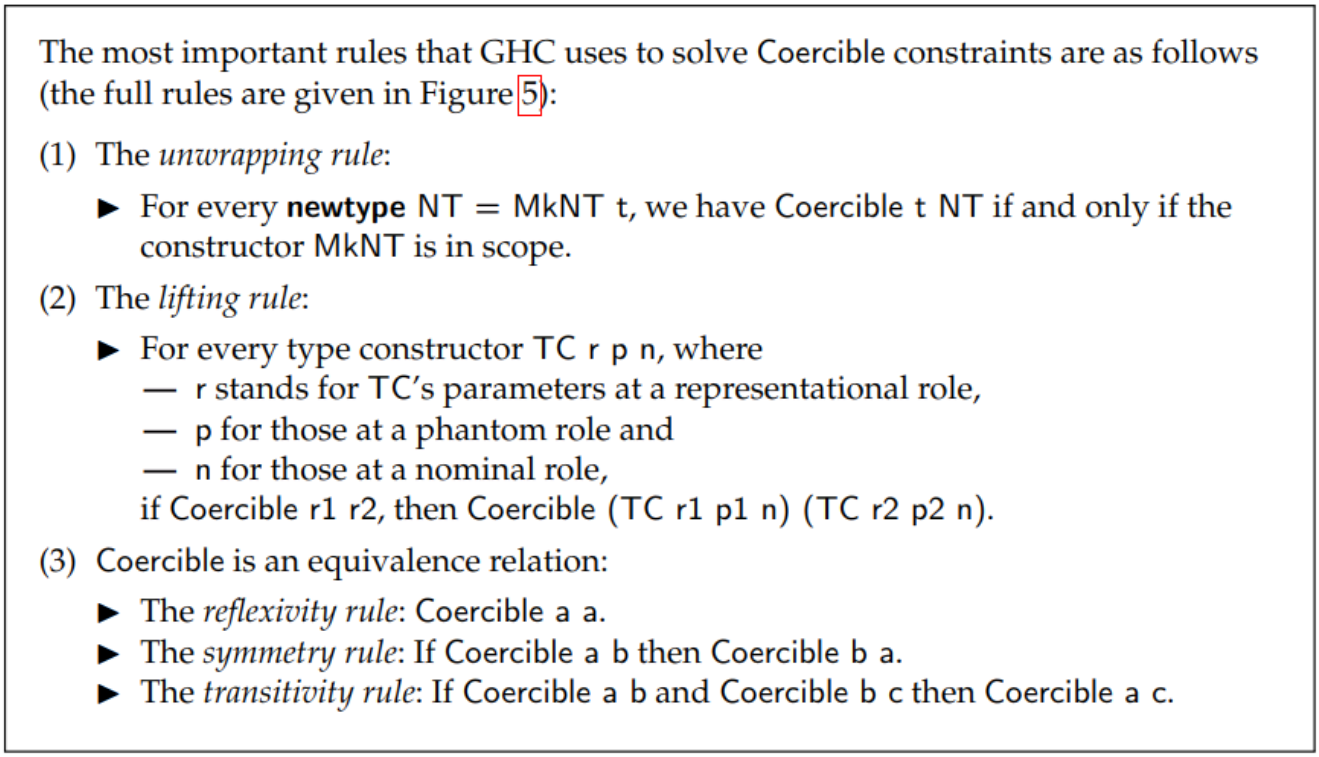
Теперь, можем избавиться от лишней трансформации списка:

```
concatC :: [Csv] -> Csv
concatC = coerce concat
```

Безопасность коерций обеспечивает **система ролей**. Каждый типовой параметр имеет специальное свойство — роль.

Роль phantom имеют фантомные типовые параметры. Их можно свободно коерсить (нет пререквизитов для инстансов `Coercible`):

[62] https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/



The most important rules that GHC uses to solve Coercible constraints are as follows (the full rules are given in Figure 5):

(1) The *unwrapping rule*:

- For every **newtype** NT = MkNT t, we have Coercible t NT if and only if the constructor MkNT is in scope.

(2) The *lifting rule*:

- For every type constructor TC r p n, where
  - r stands for TC's parameters at a representational role,
  - p for those at a phantom role and
  - n for those at a nominal role,

  if Coercible r1 r2, then Coercible (TC r1 p1 n) (TC r2 p2 n).

(3) Coercible is an equivalence relation:

- The *reflexivity rule*: Coercible a a.
- The *symmetry rule*: If Coercible a b then Coercible b a.
- The *transitivity rule*: If Coercible a b and Coercible b c then Coercible a c.

Рис. 12: Принципы построения инстансов `Coercible` Breitner et al. [2014].

```haskell
data Phantom h = Phantom
data NestedPhantom b = MkNP [Phantom b] | SomethingElse

instance Coercible (Phantom a) (Phantom b)
instance Coercible (NestedPhantom a) (NestedPhantom b)
```

Типовой параметр имеет роль representational, если типовой конструктор можно коэрсить только при условии, что можно коэрсить аргументы:

```haskell
data Maybe a = Nothing | Just a
instance Coerce a b => Coerce (Maybe a) (Maybe b)
```

Роль nominal имеет типовой параметр, если типовой конструктор можно коэрсить только при условии, что аргументы эквивалентны. Это требуется, если типовой аргумент индексирует семейство или констреинт.

```haskell
type family F a
data Applied a = Applied (F a)
instance (a ~ b) => Coercible (Applied a) (Applied b)

data ShowDict a where
```

```
  ShowDict :: Show a => a -> ShowDict a
instance (a ~ b) => Coercible (ShowDict a) (ShowDict b)
```

Иногда компилятор выводит неправильную роль типовому параметру. Например, если инварианты структуры зависят на конкретную имплементацию какого-то класса типов для типового аргумента, что совершенно не видно в декларации самого типа. В таком случае, роли можно указать явно:

```
type role Map nominal representational
data Map k v = ...
```

Подробнее можно прочитать в Breitner et al. [2014] и [Maguire, a, глава 8].

3.4.6 Type reflection

Рефлексия — это языковой механизм получения информации о типах во время исполнения (на уровне термов). Звучит знакомо, и действительно, Haskell реализует этот механизм через классы типов Peyton Jones et al. [2016].

Библиотека предоставляет магический класс типов `Typeable`, который реализуется компилятором для каждого конкретного типа через `deriving`. Чтобы получить информацию о типе, в скоупе должен быть инстанс `Typeable` для этого типа. Структура типа представлена типом-суммы `TypeRep`, который предоставляет возможность дополнительного типового контроля с помощью обобщённых алгебраических типов данных и типовых тегов.

```
class Typeable a where
  typeRep :: TypeRep a
```

Например, следующим образом можно получить имя конструктора типа:

```
typeName :: forall a. Typeable a => String
typeName = tyConName $ typeRepTyCon $ typeRep $ Proxy @a

ghci> typeName @Int
```

Упражнение 18 *Объявите класс типов, который позволяет распечатать список типов.*

С помощью структуры представления типа и экзистенциальных типов в Haskell можно эмулировать динамическую типизацию. А именно: любой тип может быть преобразован в `Dynamic`, а потом безопасно преобразован обратно.

```
data Dynamic where
  Dynamic :: Typeable a => a -> Dynamic

fromDynamic :: Typeable a => Dynamic -> Maybe a
```

Это может быть полезно, например, для определения гетерогенного хранилища ключ-значение:

```
data Store = Map Key Dynamic
data Ref ty = Ref Key
get :: Typeable ty => Store -> Ref ty -> Maybe ty
```

3.4.7 Data reflection

Как мы обсуждали ранее, в свойство когерентности гарантирует, что каждому типу в Haskell соответствует ровно один инстанс определённого класса типов. И единственный способ объявить инстанс в Haskell — декларацией на верхнем уровне, то есть он не может зависеть ни от каких локальных данных. Однако в Haskell есть библиотека Data.Reflection[63], которая позволяет создавать локальные инстансы для свежих, чёрной магией сгенерированных[64], типов.

Она пользуется идеей "поднятия значений в типы", обсуждённой нами ранее (см. 3.1.4), но в несколько более общем виде. Вместо заведения классов типов вида `Known_`, вводится один класс типов, индексированный типом термов `terms`, которые спускаются из типов:

```
class Reifies ty terms | ty -> terms where
  reflect :: Proxy ty -> terms
```

Также, с помощью следующей функции, библиотека позволяет сгенерировать свежий тип и инстанс `Reifies`, который по этому свежему типу возвращает данное значение типа `a` (переданное первым аргументом). Поскольку он передаётся в функцию высшего ранга, свежий тип не может утечь из скоупа 2.1.2:

```
reify :: a -> (forall fresh . Reifies fresh a => Proxy fresh -> res) -> res
```

Чтобы воспользоваться нестандартным инстансом класса типов для некого типа `a`, нужно объявить новый тип (например, с помощью `newtype`), содержащий данный, и написать для него нужный инстанс (см, например, Down). Мы не хотим объявлять по новой декларации для каждого случая, поэтому заведём обёртку, похожую на Data.Tagged, которая позволяет добавлять фантомный типовой тег к типу значения. Варьируя тег, можно получить сколь угодно много типов, оборачивающих данный.

```
newtype Wrapped tag a = Wrapped { unwrap :: a }
```

Объявим тип обёртки `Wrapped tag a` представителем нужного класса типов. Код для реализации будем с помощью `reflect` получать по типу тега в виде честного словаря.

[63] https://www.tweag.io/blog/2017-12-21-reflection-tutorial/
[64] https://www.schoolofhaskell.com/user/thoughtpolice/using-reflection

```haskell
data ReifiedOrd a = ReifiedOrd { compare :: a -> a -> Ordering }

instance Reifies tag (ReifiedOrd a) => Ord (Wrapped tag a) where
  compare = coerce $ compare $ reflect $ Proxy @tag
```

Наконец, можем вызвать функцию сортировки, подменив локально порядок на обратный:

```haskell
sort :: Ord a => [a] -> [a]

sortReverse :: forall a . Ord a => [a] -> [a]
sortReverse xs =
  let dict = ReifiedOrd { compare = flip compare } in
  reify dict \(Proxy :: Proxy fresh) ->
    coerce $ sort @(Wrapped fresh a) $ coerce xs
```

3.4.8 Открытые структуры

В динамических языках можно создавать объекты на ходу, последовательно дописывая в них содержимое, и не вводя предварительно декларацию. В Haskell тоже так можно, используя пары для произведений и `Either` для сумм. Например, можно добавить новое поле, создав новую пару: `(oldObj, newField)`.

Однако, такая реализация не оптимальна как с точки зрения эффективности (о более эффективных реализациях можно почитать в [Maguire, a, глава 11]), так и с точки зрения удобства использования. А именно — порядок полей имеет значение и на типах в Haskell нет отношения подтипизации (например, нельзя передать значение с меньшим количеством полей или вариантов). Но можно заметить, что констреинты лишены этих недостатков. Поэтому можно организовывать тип структуры данных, например, таким образом:

```haskell
(Int, Double) заменяем на (Member Int d, Member Double d) => Prod d
```

Далее мы вернёмся снова к открытым суммам, потому что они являются важной частью типизации расширяемых интерпретаторов Swierstra [2008].

3.4.9 Исключения и открытая иерархия

Важный аспект работы с ошибками заключается в том, что многие из них обрабатываются единообразно. Таким образом, ошибки должны образовывать иерархию наподобие той, которая в ООП языках получается с помощью наследования, чтобы иметь возможность реагировать сразу на группу ошибок одним кодом. Так, возникает задача моделирования подобной иерархии в Haskell.

Более того, статически типизированные ошибки это активная область исследований, будем говорить об этом в рамках систем эффектов (см. далее **??**). Классические исключения же

динамически типизированные. Особенно хорошо этот вариант подходит для ошибок программиста, которые по-хорошему не должны обрабатываться в программе кроме как закрытием ресурсов.

Поддержка исключений присутствует в системе исполнения Haskell как простого и привычного способа обработки исключительных ситуаций: ошибок программиста, исполнения неполного паттерн-матчинга, асинхронных системных сигналов Marlow et al. [2001]... Исключения динамически типизированные и образуют иерархию Marlow [2006]. Если породить исключение может и чистый код, так как $\bot$, по семантике Haskell, населяет любой тип, то поймать исключение можно только[65] в `IO` Jones [2001], используя специальные примитивы языка.

Чтобы сделать тип исключением, нужно объявить инстанс `Exception` для него:

```
class (Typeable a, Show a) => Exception a where
  toException :: a -> SomeException
  toException = SomeException

  fromException :: SomeException -> Maybe a
  fromException (SomeExcetion e) = cast e
```

Где `SomeException` — это экзистенциальная обёртка наподобие `Dynamic` (см. 3.4.6), в которую заворачивается конкретный тип исключения. Ловя `SomeException`, можно поймать любое исключение (`cast` всегда сработает).

```
data SomeException where
  SomeException :: Exception a => a -> SomeException

instance Exception SomeException
```

Система исполнения Haskell предоставляет интринсики для кидания и ловли исключений, обернём их для поддержки любого `Exception` типа:

```
throw :: Exception e => e -> a
throw = primThrow (toException e)

catch :: Exception e => IO a -> (e -> IO a) -> IO a
catch io handler = io `primCatch` \e -> case fromException e of
  Nothing -> throw e
  Just e' -> handler e'

ghci> throw "error" `catch` (e :: String) -> putStrLn e
```

В простейшем случае свой тип исключения можно реализовать в две строчки. Чтобы его поймать, нужно либо ловить сам этот тип, либо `SomeException`, потому что для них обоих `fromException` на объекте вида `SomeException MyError` вернёт `Just`[66].

[65] (stackoverflow) Why can Haskell exceptions only be caught inside the IO monad?

[66] `instance Exception SomeException where fromException = Just`

```haskell
data MyError = MyError deriving (Show, Typeable)
instance Exception MyError
```

Добавим исключение ArithException и ещё один более общий тип исключений между ним и SomeException — SomeArithException (таким образом, будет три способа поймать ArithException). Для этого сделаем SomeArithException экзистенциальной обёрткой, а каждое исключение типа ArithException будем автоматически оборачивать в неё. В fromException на каждом уровне вложенности будем пытаться получить оборачивающий конструктор рекурсивным вызовом.

```haskell
data SomeArithException where
  SomeArithException :: Exception a => a -> SomeArithException

-- SomeException - базовый (реализация по умолчанию)
instance Exception ArithException

data DivisionByZero = DivisionByZero deriving Show
instance Exception DivisionByZero where
  toException = toException . SomeArithException
  fromException e = do
    SomeArithException e' <- fromException e
    cast e'
```

Так, во время бросания DivisionByZero будет конструироваться объект вида:

```haskell
SomeException (SomeArithException DivisionByZero)
```

Реализация fromException для конкретного типа умеет убедиться в наличии соответствующего конструктора в результирующем объекте исключения.

3.4.10 Легковесные частичные стек-трейсы

Забавной эксплуатацией классов типов в Haskell являются легковесные частичные стек-трейсы[67]. Вообще для сбора трейсов нужна поддержка рантайма, что в случае Haskell усложняется ещё и тем, что модель вычислений, редукция графов, реальных трейсов не содержит и их приходится эмулировать. Мы же получим трейсы без поддержки рантайма.

В стандартной библиотеке определён констреинт GHC.Stack.HasCallStack, позволяющий получить информацию о месте вызова функции. Эту информацию фактически размещает компилятор в процессе вывода инстансов. Если в месте вызова доступна информация с уровня выше, компилятор распространяет её дальше. Таким образом, доступна информация только на определённую глубину стека вызовов.

[67] https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/callstack.html

```haskell
myHead :: HasCallStack => [a] -> a
myHead []     = error "empty"
myHead (x:xs) = x

bad :: Int
bad = myHead []
```

```
ghci> bad
*** Exception: empty
CallStack (from HasCallStack):
  error, called at Bad.hs:8:15 in main:Bad
  myHead, called at Bad.hs:12:7 in main:Bad
  -- no information about bad call site here
```

HasCallStack — это просто имплисит (см. 3.1.2), про который знает компилятор:

```haskell
type HasCallStack = (?callStack :: CallStack)
```

3.4.11 Кастомизируемые ошибки типизации

При программировании сложных с точки зрения типов библиотек, желательно предоставлять пользователям более информативные ошибки типизации, чем ошибки по умолчанию. Для этого в GHC есть механизм в GHC.TypeLits, позволяющий сконструировать специальный тип, информация из которого попадёт в сообщение об ошибке. Например, этот тип можно вернуть из synonym family при некорректном наборе аргументов. Или же можно воспользоваться constraint trick (см. 3.1.3) и разместить такой тип в качестве посылки в инстансе. Если инстанс подошел и компилятор начал обрабатывать ограничения слева, значит, что-то пошло не так [Maguire, a, глава 12].

```haskell
instance (TypeError
  ( Text "Attempting to show a function of type "
    :<>: Text "'" :<>: ShowType (a -> b) :<>: Text "'"
    :$$: Text "Did you forget to apply an argument?"
  )) => Show (a -> b) where
  show = undefined -- реализация не важна, до исполнения дело не дойдёт
```

4 Типы данных

В этой главе собраны некоторые общие знания о типах. Также, мы получим различные эквивалентные представления рекурсивных типов данных (иначе говоря, коллекций). Многие концепции являются частными случаями этого многообразия.

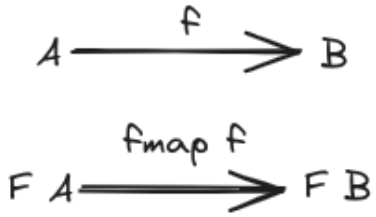
Разделы 4.1, **??** в основном следуют [Maguire, a, глава 1].

4.1 Вариантность

В этом параграфе мы будем рассматривать тему с точки зрения программирования [Maguire, a, глава 3], не отдавая должного теории категорий. Восполнить пробел можно с помощью замечательной статьи, написанной в жанре пьесы Hinze et al. [2012].

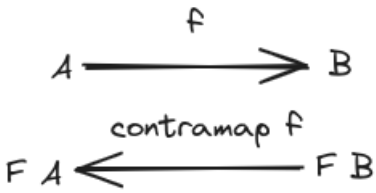
Ковариантный функтор — пара из некоторого типового конструктора `F` и операции на функциях `fmap :: (a -> b) -> (F a -> F b)`. Плюс законы о том, что `fmap` уважает `id` и композицию.

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



Контравариантный функтор — пара из типового конструктора и операции на функциях, разворачивающей стрелку. Плюс соответствующие законы.

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



Типовой конструктор можно объявить ковариантным или контравариантным функтором (или никаким из них) относительно некоторого типового параметра в зависимости от вида

декларации соответствующих конструкторов данных. А именно, от знака позиций, в которых входит этот типовой параметр в тип.

Разовьём интуитивное понимание знаков позиций. Тип `A` входит в положительной позиции в `B` если его значение можно извлечь из `B`. И наоборот, тип `A` входит в отрицательной позиции, если его значение нужно, наоборот, предоставить. Рассмотрим знаки позиций типов в базовых типовых конструкторах:

Тип	знак позиции `A`	знак позиции `B`
`Either A B`	$+$	$+$
`(A, B)`	$+$	$+$
`A -> B`	$-$	$+$

Действительно, из суммы и произведения можно извлечь компоненты с помощью паттерн-матчинга, а из стрелки можно получить правый тип аппликируя её к аргументу. В то же время значение типа слева от стрелки нужно предоставить.

На плюс и минус действуют интуитивные алгебраические законы при рассмотрении более сложных типов. Рассмотрим на примере `f :: ((A, B) -> C) -> (D, E)`.

- Плюс на плюс даёт плюс. Действительно, нужно лишь применить две элиминации вместо одной, чтобы получить заветный тип. В нашем примере, чтобы получить `D`, нужно сначала аппликировать функцию, а потом разобрать пару.
- Плюс на минус (и наоборот) даёт минус. Действительно, `C` нам нужно предоставить: `f (\ab -> provideC)`.
- Минус на минус даёт плюс. Пару `(A, B)` нам предоставляют: `f (\ab -> ...)`.

Упражнение 19 *Убедитесь что плюс на минус даёт минус.*

Возвращаясь к функторам, если типовой параметр входит в декларацию только в положительных позициях, типовой конструктор можно объявить ковариантным функтором относительно этого параметра. Если в только в отрицательных — контравариантным функтором. Если в обоих, то никаким функтором объявить нельзя. Соответственно, будем называть типовые параметры ковариантными, контравариантными и инвариантными.

Упражнение 20 *Объявите* `instance Contravariant F` *для* `data F a = L (a -> ()) | R Int`.

Таким образом, можно понимать ковариантный функтор как вычисление, результат которого можно пост-обработать, а контравариантный функтор — как вычисление, аргументы которого можно пред-обработать.

Тип от двух положительных параметров можно объявить **бифунктором**:

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

Тип от двух параметров, положительного и отрицательного, — **профунктором**:

```haskell
class Profunctor p where
  dimap :: (c -> a) -> (b -> d) -> p a b -> p c d
```

Профункторы являются некоторыми обобщениями функциональной стрелки. Например, если у нас есть SQL запрос, который по данным возвращает результат, его можно объявить профунктором с семантикой — добавить пред-обработку входных данных и пост-обработку выходных:

```haskell
dimap serialize deserialize (query :: Sql Text Text) :: Sql Age [User]
```

Также понятие вариантности часто встречается в объектно ориентированных языках (да и вообще в теории подтипизации) для обозначения возможности дополнить отношение подтипизации на полиморфные типы.

Действительно, **отношение подтипизации** `B <: A` говорит о том, что значение типа `B` безопасно использовать в позиции, где ожидается значение типа `A`. Иначе говоря, существует функция `upcast :: B -> A`. Если типовой конструктор `F a` ковариантен относительно параметра `a`, то по `upcast` найдётся `upcast' :: F B -> F A`. То есть отношение подтипизации также автоматически включает `F B <: F A`. Контравариантный случай аналогично.

Упражнение 21 *Убедитесь в вашем любимом языке с поддержкой вариантности, что минус на минус даёт плюс.*

To be continued...

Список литературы

Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13:11–49, 2000. URL https://facweb.cdm.depaul.edu/smitsch/courses/csc447fa23/assets/articles/strachey-fundamental-concepts-in-programming-languages.pdf.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985. URL https://doi.org/10.1145/6041.6042.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

John Launchbury and Simon L Peyton Jones. State in haskell. *Lisp and symbolic computation*, 8(4): 293–341, 1995. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/state-lasc.pdf.

Sandy Maguire. *Thinking with Types, Type-Level Programming in Haskell*. a. URL https://leanpub.com/thinking-with-types/?

Simon Peyton Jones. Type inference as constraint solving: how ghc's type inference engine actually works. Keynote talk at Zurihac 2019, 2019.

Benjamin C Pierce and David N Turner. Local type inference. *Acm transactions on programming languages and systems (toplas)*, 22(1):1–44, 2000. URL https://doi.org/10.1145/345099.345100.

David Raymond Christiansen. Bidirectional typing rules: A tutorial. 2013. URL https://davidchristiansen.dk/tutorials/bidirectional.pdf.

Jana Dunfield and Neelakantan R Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019. URL https://doi.org/10.1145/3290322.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. A quick look at impredicativity. *Proceedings of the ACM on Programming Languages*, 4 (ICFP):1–29, 2020. URL https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/.

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–437, 2020. URL https://link.springer.com/article/10.1208/s12249-010-9382-3.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, 2012. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/p53-yorgey.pdf.

Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System fc with explicit kind equality. *ACM SIGPLAN Notices*, 48(9):275–286, 2013. URL https://doi.org/10.1145/2544174.2500599.

Vitaly Bragilevsky. *Haskell in Depth*. Manning. URL https://www.manning.com/books/haskell-in-depth.

Richard A Eisenberg and Simon Peyton Jones. Levity polymorphism. *ACM SIGPLAN Notices*, 52 (6):525–539, 2017. URL https://doi.org/10.1145/3140587.3062357.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007. URL https://www.researchgate.net/profile/Simon-Peyton-Jones/publication/221501761_A_history_of_Haskell_Being_lazy_with_class/links/0c960517e31f50f743000000/A-history-of-Haskell-Being-lazy-with-class.pdf.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989. URL https://doi.org/10.1145/75277.75283.

Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996. URL https://doi.org/10.1145/227699.227700.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *In Haskell Workshop*, 1997. URL https://courses.cs.washington.edu/courses/cse590p/06sp/multi.pdf.

Jeffrey R Lewis, John Launchbury, Erik Meijer, and Mark B Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, 2000. URL https://doi.org/10.1145/325694.325708.

Simon Peyton Jones. Type inference as constraint solving: how ghc's type inference engine actually works, 2019. URL https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

Martin Sulzmann, Gregory J Duck, Simon Peyton-Jones, and Peter J Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of functional programming*, 17(1): 83–129, 2007a. URL https://doi.org/10.1017/S0956796806006137.

Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, 2004. URL https://d1wqtxts1xzle7.cloudfront.net/43582096/Functional_pearl_implicit_configurations20160310-32037-1bu6179-libre.pdf.

Filip Křikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–28, 2019. URL https://doi.org/10.1145/3360589.

Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *ACM Sigplan Notices*, 45(10):341–360, 2010. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d30d65ca9ce7891352024a5c71ebe0ae8c41f7ac.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. *ACM SIGPLAN Notices*, 46(9):143–155, 2011. URL https://archive.alvb.in/msc/thesis/reading/typeclasses-agda_Devriese.pdf.

Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002. URL https://doi.org/10.1017/S0956796802004355.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, volume 1, pages 203–233, 2001. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2001/09/rules.pdf.

Sandy Maguire. *Algebra-Driven Design, Elegant Solutions from Simple Building Blocks*. b. URL https://leanpub.com/algebra-driven-design/.

Li-yao Xia. *Defunctionalization*. URL https://poisson.chat/aquarium/defunctionalization.pdf.

John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972. URL https://doi.org/10.1145/800194.805852.

John C Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11: 355–361, 1998. URL https://doi.org/10.1023/A:1010075320153.

Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *International Symposium on Functional and Logic Programming*, pages 119–135. Springer, 2014. URL https://www.cl.cam.ac.uk/~jdy22/papers/lightweight-higher-kinded-polymorphism.pdf.

Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *ACM SIGPLAN Notices*, 40(9):241–253, 2005a. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/at-syns.pdf.

Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, 2005b. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/assoc.pdf.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2008/01/icfp2008.pdf.

Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. *ACM SIGPLAN Notices*, 49(1):671–683, 2014. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/popl137-eisenberg.pdf.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. *Reflections on the Work of CAR Hoare*, pages 301–331, 2010. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/typefun.pdf?from=https://research.microsoft.com/~simonpj/papers/assoc-types/fun-with-type-funs/typefun.pdf&type=exact.

Mark P Jones. Type classes with functional dependencies. In *European Symposium on Programming*, pages 230–244. Springer, 2000. URL https://doi.org/10.1007/3-540-46425-5_15.

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. *Acm Sigplan Notices*, 49(10):233–249, 2014. URL http://lampwww.epfl.ch/~amin/dot/fpdt_post.pdf.

Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. Injective type families for haskell. *ACM SIGPLAN Notices*, 50(12):118–128, 2015. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1070&context=compsci_pubs.

Richard A Eisenberg and Jan Stolarek. Promoting functions to type families in haskell. *ACM SIGPLAN Notices*, 49(12):95–106, 2014. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1000&context=compsci_pubs.

Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in haskell. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–26, 2019. URL https://dl.acm.org/doi/pdf/10.1145/3341706.

Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In *International Symposium on Functional and Logic Programming*, pages 56–71. Springer, 2010. URL https://kar.kent.ac.uk/57498/1/constraint-families.pdf.

Edsko de Vries and Andres Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 83–94, 2014.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C d S Oliveira, and Philip Wadler. Quantified class constraints. *ACM SIGPLAN Notices*, 52(10):148–161, 2017. URL https://www.pure.ed.ac.uk/ws/portalfiles/portal/42495988/quantcc.pdf.

Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, 2007b. URL https://doi.org/10.1145/1190315.1190324.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of functional programming*, 21(4-5): 333–412, 2011. URL https://doi.org/10.1017/S0956796811000098.

Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 189–202, 2014. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1013&context=compsci_pubs.

Simon Peyton Jones, Stephanie Weirich, Richard A Eisenberg, and Dimitrios Vytiniotis. A reflection on types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 292–317. Springer, 2016. URL https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1002&context=compsci_pubs.

Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008. URL https://www.cs.tufts.edu/~nr/cs257/archive/wouter-swierstra/DataTypesALaCarte.pdf.

Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 274–285, 2001. URL https://classes.cs.uchicago.edu/archive/2007/spring/32102-1/papers/p274-marlow.pdf.

Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 96–106, 2006. URL https://archive.alvb.in/msc/03_infoafp/papers/2012-12-18_HoorCollege_MarlowExtensibleExceptions_dk.pdf.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES*, 180:47–96, 2001. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2e6c9d76f9cb690dc18019fc894ba9572a8c2812.

Ralf Hinze, Jennifer Hackett, and Daniel WH James. Functional pearl: F for functor. 2012. URL www.cs.ox.ac.uk/people/daniel.james/functor/functor.pdf.