Кайнды

Кайнды по определению это тип типа

`Int` и `Char` ничем не параметризованы, поэтому *:

```
ghci> :kind Int
Int :: *
ghci> :kind Char
Char :: *
```

Но вот если мы возьмём что-то посложнее, например, `Maybe`:

```
ghci> :kind Maybe
Maybe :: * -> *
```

... то увидим, что у `Maybe` есть один параметр.

НО, если мы займём этот параметр чем-нибудь, например строкой:

```
ghci> :kind Maybe String
Maybe :: *
```

... то снова получим результат, как у примитивных типов. Потому что `Maybe String` больше никаких типовых аргументов принимать не будет, у него всё определено.

Какой, в такое случае, кайнд у `(->)`? Правильно:

```
ghci> :k (->)
(->) :: * -> * -> *
```

... аргумента ведь два.

Для чего они нужны?

Можем указать, какой кайнд должен быть у нашего типа

```
newtype ReaderT r (m :: * -> *) (a :: *) = ReaderT {runReaderT :: r -> m a}
ghi> :kind ReaderT
ReaderT :: * -> (* -> *) -> * -> *
```

... и избежать всяких ошибок.

Declassified kind

Все классы в ***Haskell*** имеют некоторый тип, который завершается `Constraint`:

```
ghci> :k Num
Num :: * -> Constraint
```

`Constraint` показывает, что результат - какой-то конкретный класс, а не произвольный тип данных. У классов всегда тип `Constraint`

Можем создавать элиасы для `Constraint`'ов:

```
type MyConstraints a = (Read a, Num a, Show a)
foo :: MyConstraints a => String -> a -> a
```

`Constraint` не обязательно находится в конце:

```
ghci> type ConstraintEndomorphism p a = p a => a -> a
ghci> :kind ConstraintEndomorphism
ConstraintEndomorphism :: (* -> Constraint) -> * -> *
```

Типовые операторы (-XTypeOperators)

Обычно мы не можем определять типы с помощью операторов:

```
ghci> data a * b = Mult a b
Illegal declaration of a type or class operator '*'
Use TypeOperators to declare operators in type and declarations
```

Но есть специальный Language Extension на такой случай, который как раз позволяет нам определять типы с помощью операторов:

```
ghci> :set -XTypeOperators
ghci> data a * b = Mult a b
ghci> :t (*)
(*) :: Num a => a -> a -> a
ghci> :k (* )
(*) :: * -> * -> *
```

Получается, что тип от кайнда отличается только `constraint`'ом `Num a`.

Более того, конструкторы тоже можно определить в операторном виде.

```
gci> let x = Mult @Int 3 True
ghci> :t x
x :: Int * Bool
```

Грамматика кайндов

- `*` - кайнд;
- `Constraint` - кайнд;
- Если `v` - это терм, то у `v` есть тип, у которого есть кайнд: `v :: (t :: *)`;
- Если `a` и `b` - кайнды, то `(a -> b)` тоже кайнд;
- Типы данных имею кайнд `*` или `k -> *`;
- Классы имеют кайнд `Constraint` или `k -> Constraint`;

* и Constraint - всё?

Короткий ответ - нет.

Мы сами можем определять кайнды. Мотивационный пример:

```
data Z
data S n

data Vec :: * -> * -> * where
    Nil  :: Vec a 2
    Cons :: a -> Vec a n -> Vec a (S n)
```

Мы таким образом задали вектор, в типе которого содержится размер: - либо мы конструируем вектор `Nil`, который имеет нулевой размер; - либо мы конструируем вектор, добавляя к нему один элемент.

То есть, если мы создадим вектор несоответствующего размера, оно упадёт ещё на этапе компиляции, так называемый `type-safe` вектор.

Но с таким конструктором у нас всё ещё есть проблема: мы не указали Constraint'ы для типов, поэтому следующая запись валидна:

```
v3 :: Vec Int Char
v3 = ??
```

Хотим задать ограничение на передаваемый тип:

```
{-# LANGUAGE DataKinds #-}

data Nat = Z | S Nat

data Vec :: * -> Nat -> * where
    Nil  :: Vec a 2
    Cons :: a -> Vec a n -> Vec a (S n)
```

Указанное расширение языка "поднимает" типы данных, которые есть в текущем модуле (создаёт поднятую копию), и мы всё ещё можем использовать Nat как тип данных, а S и Z как конструкторы. Но также мы можем использовать Nat как кайнд, а S и Z как типы.

Но мы **не** можем создавать экземпляр типа Nat, потому что отсутствует term-левел.

Рассмотрим ещё один пример

```
data Nat = Zero | Succ Nat

ghci> :t Succ
Succ :: Nat -> Nat
ghci> :k Nat
Nat :: *
```

Но мы не можем узнать кайнд у Succ, потому что это конструктор:

```
ghci> :k Succ
Not in scope: type constructor or class 'Succ'
A data constructor of that name is in scope; did you mean DataKinds?
```

Но если подключить расширение:

```
ghci> :set -DataKinds
ghci> :k Succ
Succ :: Nat -> Nat
```

Успех. Теперь умеем узнавать кайнд конструкторов:

```
ghci> type Two = `Succ (`Succ Zero) -- перед промоутными типами лучше ставить штрихи для у
ghci> :k Two
Two :: Nat
```

На самом деле мы также научились промоутить листы: у которых [] выступают как конструктор:

```
ghci> :k '[]
'[] :: [k]
```

Под k может быть любой кайнд, k - это переменная

```
ghci> :k '[Int, Bool]
'[Int, Bool] :: [*]
```

– так как внутри листа каждый тип имеет кайнд *. Аналогично:

```
ghci> :k '[Maybe, Either String]
'[Maybe, Either String] :: [* -> *]
```

Ну и так как операторы тоже промоутнулись после добавления расширений DataKinds и TypeOperators:

```
ghai> :k (Int ': '[Bool])
(Int ': '[Bool]) :: [*]
```

Главное, чтобы кайнд в листах был одинаковый у всех элементов.

Ещё немного про TypeSafe вектора

Имеем вектор:

```
data Vec :: * -> Nat -> * where
    Nil  :: Vec a
    (:>) :: a -> Vec a n -> Vec a ('S n)
```

И имеем функцию, которая принимает два вектора и “зипует” их:

```
zipV :: Vec a n -> Vec b n -> Vec (a, b) n
zipV       Nil       Nil = Nil
zipV (x :> xs) (y :> ys) = (x, y) :> zipV xs ys
```

Заметим, что zipV принимает два вектора **одинакового** размера n и отдаёт в качестве результат вектор такого же размера.

Если два вектора одного размера, то всё ок:

```
ghci> let x = 3 :> Nil
ghci> let y = True :> Nil
ghci> :t x
x :: Num a => Vec a ('S 'Z)
ghci> :t y
y :: Vec Bool ('S 'Z)
ghci> :t zipV x y
zipV x y :: Num a => Vec (a, Bool) ('S 'Z)
```

Но если разного:

```
ghci> :t zipV x (4 :> y)
• Couldn't match type ''s 'z' with ''z'
Expected type: Vec Bool ('S 'Z)
Actual type: Vec Bool ('S ('S 'Z))
• In the second argument of 'zipV', namely '(4 :> y)' In the expression: zipV x (4 :> y)
```

... то всё сломается.

Таким образом, мы узнаем о проблеме ещё в компайл тайме.

Гетерогенные листы

```
data HList :: [*] -> * where
      HNil :: HList '[]
      (:^) :: a -> HList t -> HList (a ': t)

infixr 2 :^
```

Тип такого вектора обновляется при добавлении нового элемента

```haskell
foo0 :: HList '[]
foo0 = HNil

foo1 :: HList '[Int]
foo1 = 3 :^ HNil

foo2 :: HList '[Int, Bool]
foo2 = 5 :^ False :^ HNil
```

Инстансы для такого вектора можно определить следующим образом:

```haskell
instance Show (HList '[]) where
    show _ = "H[]"
instance (Show e, Show (HList l)) => Show (HList (e ': l)) where
    show (x :^ 1) = let 'H':'[':s = show l
                    in "H[" ++ show x ++ (if s == "]" then s else ", " ++ s)
```

```
ghci> foo0
H[]
ghci> foo2
H[5, False]
```

Type Level Symbols

Вместо того чтобы пользоваться абстракциями в виде Z и S для обозначения размера вектора, мы можем импортировать числовой тип из `GHC.TypeLits`. Тогда наш вектор примет вид:

```haskell
import GHC. TypeLits

data Vec :: * -> Nat -> * where
    Nil  :: Vec a 0
    (:>) :: a -> Vec a n -> Vec a (n + 1)
```

Где 0 - это, соответственно, Z, а `n + 1` - S

Что такое `n + 1`?

Рассмотрим + подробнее:

```
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :k (+)
(+) :: Nat -> Nat -> Nat
ghci> :i (+)
class Num a where
...
infixl 6 +
type family (+) (a :: Nat) (b :: Nat) :: Nat
infixl 6 +
```

Что ещё за `type family`? Разложим всё по порядку.

Type-level functions

```haskell
newtype Foo bar = MkFoo { unFoo :: bar }
```

Foo - это type-level функция, которая принимает bar и возвращает bar типа Foo:

```
MkFoo :: bar -> Foo bar   (term level)
Foo :: * -> *             (type level)
```

Раз уж мы уже умеем поднимать типы в кайнд, давайте поднимем функцию Foo в тип:

```
data Foo a where
    Foo :: Int -> Foo Int
    Bar :: Char -> Foo Double

Foo :: Int -> Foo Int        (term level)
Bar :: Char -> Foo Double    (term level)
Foo :: * -> *                (type level)
```

Теперь Foo - это type-level функция, которая принимает тип a и, основываясь на нём, возвращает либо Foo Int, либо Foo Double

Можем ли мы написать что-то типа этого?

```
type fun Foo a where
  | a == Char = Double
  | otherwise = a
```

То есть мы хотим паттерн матчинг на типах (кайндах). Для этого и используется специальная конструкция, называемая type family:

```
type family Foo bar :: * where -- закрытый типовой конструктор
    Foo Char = Double
    Foo b = b
```

Здесь Foo - это type-level функция, которая принимает тип bar и возвращает либо Double, либо bar, в зависимости от того, что такое bar.

Типовые семьи также могут быть открытыми:

```
type family Foo bar :: * -- открытый типовой конструктор
type instance Foo Char = Double
type instance Foo Int = Int
```

Отличие открытой в том, что для открытой семьи новые инстансы могут быть определены где угодно. Но в открытой типовой семье мы теряем вохможность создавать неявный типовой конструктор:

```
type family Foo bar :: *
type instance Foo Char = Double
type Foo b = b -- нельзя. такое только в закрытом
```

Ещё вот так можно:

```
class Foo p where
    type AType p :: *
    data BType p :: *

    make :: AType p -> BType p

instance Foo Int where
    type AType Int = Int
    data BType Int = B Integer

    make = B . toInteger
```

Это нужно, когда мы хотим менять тип результата в зависимости от типового параметра, при этом не используя сам типовой параметр.

Таким образом, с помощью `type family`, мы по сути определили свой `from maybe`:

```
type family FromMaybe
    (m :: Maybe *) (def :: *) :: * where
  FromMaybe ('Just t) def = t
  FromMaybe 'Nothing def = def
```

Где `def` - дефолтное значение.

Также есть расширение `PolyKinds`, которое позволяет зафиксировать один тип вместо *:

```
{-# LANGUAGE PolyKinds #-}

type family FromMaybe
    (m :: Maybe k) (def :: k) :: k where
  FromMaybe ('Just t) def = t
  FromMaybe 'Nothing def = def
```

Ограничения `type family`

Пусть есть некоторая типовая семья:

```
type family Foo bar :: * where
  Foo Char = Double
  Foo b = b

show' :: Show (Foo a)
      => Foo a -> String
show' = show
```

Здесь получим ошибку:

```
• Couldn't match type 'Foo a0' with 'Foo a'
  Expected type: Foo a -> String
  Actual type: Foo a0 -> String
  NB: 'Foo' is a non-injective type family
```

Компилятор не понимает, что такое `a`. Можем добавить `TypeFamilyDependencies`:

```
{-# LANGUAGE TypeFamilyDependencies #-}

type family Foo a = r | r -> a where
  Foo Char = Double
  Foo a = a
```

Но тогда мы не можем однозначно определить тип `a`, потому как если он `Double`, то из первого мы получаем, что он `Char`.

Поэтому надо однозначно определить все значения:

```
{-# LANGUAGE TypeFamilyDependencies #-}

type family Foo a = r | r -> a where
  Foo Char = Double
  Foo Int = Int
```

```
ghci> show' (1 :: Int)
" 1"
ghci> (show' :: _) (1 :: Double)
• Found type wildcard for 'Foo Char -> String'
```

Интересные примеры использования

Union из union package

```
data Union (f :: u -> *) (as :: [u]) where
  This :: !(f a) -> Union f (a ': as)
  That :: !(Union f as) -> Union f (a ': as)
```

Rec из vynil package

```
data Rec :: (u -> *) -> [u] -> * where
  RNil :: Rec f '[]
  (:&) :: !(f r) -> !(Rec f rs) -> Rec f (r ': rs)
```

Free monads

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

Free монада - это абстракция над мультиступенчатыми вычислениями, где каждый следующий шаг требует некоторый контекст, чтобы продолжить.

Пример:

```
data Action a =
  PrintLn String a | ReadInt (Int -> a)

justPrint :: Free Action Int
justPrint = Free $ PrintLn "poid" (Pure 5)

readIncPrint :: Free Action ()
readIncPrint =
  Free $ ReadInt $ \i ->
    Free $ PrintLn (show $ i + 1) (Pure ())

runAction :: Free Action a -> IO a
runAction (Pure x) = pure x
runAction (Free (PrintLn s cont)) =
  putStrLn s *> runAction cont
runAction (Free (ReadInt cont)) = do
  i <- fmap read getLine
   runAction (cont i)

ghci> runAction readIncPrint
5
6
```

Если дошли до Pure, то просто возвращаем его. Иначе - смотрим, что за действие и продолжаем, в зависимости от того, что за действие.

Почему монада?

Потому что мы получаем бесплатный монадный инстанс, при условии, что завернули её в Free.

Перепишем предыдущий пример в функтор для наглядности:

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

instance Functor f => Monad (Free f) where
  return = pure
  Pure a >>= f = f a
  Free m >>= f = Free ((>>= f) <$> m)

data Action a =
    PrintLn String a | ReadInt (Int -> a)
    deriving Functor

printLn :: String -> Free Action ()
printLn s = Free $ PrintLn s (Pure())

readInt :: Free Action Int
readInt = Free $ ReadInt Pure

readIncPrint :: Free Action ()
readIncPrint = do
  i <- readInt
  println $ show (i + 1)
```

Is Free actually free?

- >>= is O(d) for d being a depth of computation
- Interpreting step-by-step is much worse than compilation
- You can not formalize something with two continuations, like concurrently

Вывод: нет. Free монады никто не использует.