Функциональное программирование с зависимыми типами на языке Idris

Лекция 5. Интерфейсы, модули, пространства имён

В. Н. Брагилевский

27 ноября 2017 г.

Факультет компьютерных наук, НИУ «Высшая школа экономики»

Институт математики, механики и компьютерных наук
имени И. И. Воровича, Южный федеральный университет (Ростов-на-Дону)

Интерфейсы

Интерфейсы

Идея и применение

Типы, интерфейсы и реализации

Тип

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Интерфейс

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

Реализация

```
Show NPair where
  show (MkNPair n m) =
       "(" ++ show n ++ "," ++ show m ++ ")"
```

Использование

```
idris> :type show
show : Show a => a -> String
idris> show (MkNPair 5 10)
"(5,10)" : String
```

Интерфейс Eq

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

- Проверка на равенство
- Определения по умолчанию
- Многие типы реализуют Eq

Интерфейс Ord

```
interface Eq a => Ord a where
  compare : a -> a -> Ordering
  (<) : a -> a -> Bool
  (>) : a -> a -> Bool
  (<=) : a -> a -> Bool
  (>=) : a -> a -> Bool
  max : a -> a -> a
  min : a -> a -> a
```

- Расширяет `Eq`
- Минимальная полная реализация: `compare`
- `data Ordering = LT | EQ | GT`

Простейшие способы применения интерфейсов

```
show : Show a => a -> String

sort : Ord a => List a -> List a


sortAndShow : (Ord a, Show a) =>
                  List a -> String
sortAndShow xs = show (sort xs)
```

Именованные реализации

Реализация Show для Nat по умолчанию

```
idris> show (S (S (S Z)))
"3" : String
```

Именованная реализация

```
[myShowNat] Show Nat where
  show Z = "z"
  show (S k) = strCons 's' (show k)


idris> show @{myShowNat} (S (S (S Z)))
"sssz" : String
```

Именованные реализации (2)

```
f : Show a => a -> String
f a = "Result: " ++ show a

idris> f @{myShowNat} (S Z)
"Result: sz" : String
```

Числовые интерфейсы

```
interface Num a where
  (+) : a -> a -> a
  (*) : a -> a -> a
  fromInteger : Integer -> a

interface Num a => Neg a where
  negate : a -> a
  (-) : a -> a -> a
  abs : a -> a

interface Integral a where
  div : a -> a -> a
  mod : a -> a -> a
```

Интерфейсы для ограниченных типов

```
interface Ord b => MinBound b where
  minBound : b

interface Ord b => MaxBound b where
  maxBound : b
```

Перечисление

```
interface Enum a where
  pred : a -> a
  succ : a -> a
  succ e = fromNat (S (toNat e))
  toNat : a -> Nat
  fromNat : Nat -> a
```

Пример: ориентация локатора

Тип данных

```
data Direction = North | East | South | West
```

Реализация Eq

```
Eq Direction where
  North == North = True
  East == East = True
  South == South = True
  West == West = True
  _ == _ = False
```

Реализация Ord

```
Ord Direction where
  compare a b =
    if a == b then EQ
    else
      case a of
        North => LT
        West => GT
        East => if b == North then GT else LT
        South => if b == West then LT else GT
```

MinBound и MaxBound

```
MinBound Direction where
  minBound = North

MaxBound Direction where
  maxBound = West
```

```
Enum Direction where
  toNat North = 0
  toNat East = 1
  toNat South = 2
  toNat West = 3

  fromNat Z = North
  fromNat (S Z) = East
  fromNat (S (S Z)) = South
  fromNat (S (S (S Z))) = West
  fromNat _ = West

  pred West = South
  pred South = East
  pred _ = North
```

Пример определения собственного интерфейса

Циклическое перечисление

```
interface (Eq t, Enum t, MinBound t, MaxBound t)
          => CEnum t where
  cpred : t -> t
  cpred a = if a == minBound then maxBound
                             else pred a

  csucc : t -> t
  csucc a = if a == maxBound then minBound
                             else succ a
```

Реализация циклического перечисления для Direction

```
CEnum Direction where
```

Правда, эта штука пока не работает:
`https://github.com/idris-lang/Idris-dev/issues/4222`

Интерфейсы

Абстрагирование операций


I
ALGEBRA

Semigroup (Prelude.Algebra)

```
interface Semigroup a where
  (<+>) : a -> a -> a
```

```
idris> "123" <+> "456"
"123456" : String
idris> [1,2,3] <+> [4,5]
[1, 2, 3, 4, 5] : List Integer
idris> Just 5 <+> Just 10
Just 5 : Maybe Integer
idris> Nothing <+> Just 10
Just 10 : Maybe Integer
```

Полугруппа для Maybe a

```
Semigroup (Maybe a) where
  Nothing   <+> m = m
  (Just x)  <+> _ = Just x

[collectJust] Semigroup a
                    => Semigroup (Maybe a) where
  Nothing   <+> m       = m
  m         <+> Nothing = m
  (Just m1) <+> (Just m2) = Just (m1 <+> m2)
```

```
idris> (<+>) (Just "1") (Just "4")
Just "1" : Maybe String
idris> (<+>) @{collectJust} (Just "1") (Just "4")
Just "14" : Maybe String
```

Monoid (Prelude.Algebra)

```
interface Semigroup a => Monoid a where
  neutral : a
```

```
idris> the String neutral
"" : String
idris> the (Maybe Nat) neutral
Nothing : Maybe Nat
```

Числовые моноиды

Обёртки для Nat

```
record Additive where
  constructor GetAdditive
  _ : Nat

record Multiplicative where
  constructor GetMultiplicative
  _ : Nat
```

Числовые моноиды

```
idris> the Additive neutral
GetAdditive 0 : Additive
idris> the Multiplicative neutral
GetMultiplicative 1 : Multiplicative
idris> GetAdditive 5 <+> GetAdditive 10
GetAdditive 15 : Additive
idris> GetMultiplicative 5 <+> GetMultiplicative 2
GetMultiplicative 10 : Multiplicative
```

Числовые моноиды: реализация

```
Semigroup Additive where
  left <+> right = GetAdditive $ left' + right'
    where
      left'  : Nat
      left'  = case left of
                  GetAdditive m => m

      right' : Nat
      right' = case right of
                  GetAdditive m => m

Monoid Additive where
  neutral = GetAdditive Z
```

Что может быть параметром интерфейса?

```
interface InterfaceName a where
  ...
```

- Type
- Type-значная функция (произвольный конструктор типа) — в этом случае необходимо явно указывать полный тип

Интерфейсы

Абстрагирование контекста

Functor

Определение функтора

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

Что может быть значением в контексте?

- Значение с возможным признаком ошибки (`Maybe a`).
- Значение с возможным признаком ошибки и её объяснением (`Either a b`).
- Результат недетерминированного вычисления (`List a`).
- Значение, полученное с помощью ввода-вывода (`IO`).

Идея функтора

Функтор позволяет изменять значение в контексте без изменения самого контекста, то есть *контекст абстрагируется*.

Использование реализаций функтора

```
idris> map (+1) (Just 1)
Just 2 : Maybe Integer
idris> map (+1) Nothing
Nothing : Maybe Integer
idris> the (Either String Integer)
                        (map (+1) (Right 5))
Right 6 : Either String Integer
idris> map (+1) (Left "some mistake")
Left "some mistake" : Either String Integer
idris> map (+1) [1,2,3,4,5]
[2, 3, 4, 5, 6] : List Integer
```

Использование реализаций функтора (2)

Синоним для map

```
(<$>) : Functor f => (a -> b) -> f a -> f b
```

```
idris> negate <$> (Just 1)
Just -1 : Maybe Integer
```

Applicative: абстрагирование применения функции

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

Определение Applicative

```
interface Functor f
     => Applicative (f : Type -> Type) where
   pure  : a -> f a
   (<*>) : f (a -> b) -> f a -> f b
```

```
idris> pure max <*> (Just 2) <*> (Just 3)
Just 3 : Maybe Integer
idris> max <$> (Just 2) <*> (Just 3)
Just 3 : Maybe Integer
```

Сложение двух `Maybe Nat`

Способ 1: сопоставление с образцом

```
m_add : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add (Just n) (Just m) = Just (n + m)
m_add _ _ = Nothing
```

```
idris> m_add (Just 5) (Just 10)
Just 15 : Maybe Nat
idris> m_add (Just 5) Nothing
Nothing : Maybe Nat
```

Способ 2: аппликативный стиль

```
m_add2 : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add2 a b = plus <$> a <*> b
```

```
idris> m_add2 (Just 5) (Just 10)
Just 15 : Maybe Nat
idris> m_add2 (Just 5) Nothing
Nothing : Maybe Nat
```

```
Applicative Maybe where
    pure = Just

    (Just f) <*> (Just a) = Just (f a)
    _        <*> _        = Nothing
```

Способ 3: Idiom Brackets

```
m_add3 : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add3 a b = [| a + b |]
```

```
idris> m_add3 (Just 5) (Just 10)
Just 15 : Maybe Nat
idris> m_add3 (Just 5) Nothing
Nothing : Maybe Nat
```

- Idiom brackets — синтаксический сахар для Applicative
- `[| f a1 ...an |]` переводится в
  `pure f <*> a1 <*> ... <*> an`

Способ 4: !-нотация

```
m_add4 : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add4 a b = pure (!a + !b)
```

```
idris> m_add4 (Just 5) (Just 10)
Just 15 : Maybe Nat
idris> m_add4 (Just 5) Nothing
Nothing : Maybe Nat
```

И ещё чуть-чуть абстракции!

```
a_add : (Semigroup a, Applicative f) =>
          f a -> f a -> f a
a_add a b = [| a <+> b |]

idris> a_add (Just "123") (Just "456")
Just "123456" : Maybe String
idris> a_add (Just (getAdditive 5))
             (Just (getAdditive 10))
Just (getAdditive 15) : Maybe Additive
```

Функция a_add и контекст списка

```
idris> :let xs = map getMultiplicative [1,2,3]
defined
idris> :let ys = map getMultiplicative [5,6]
defined

idris> a_add xs ys
[getMultiplicative 5, getMultiplicative 6,
 getMultiplicative 10, getMultiplicative 12,
 getMultiplicative 15, getMultiplicative 18]
      : List Multiplicative
```

Что здесь происходит?

```
Applicative List where
  pure x = [x]

  fs <*> vs = concatMap (\f => map f vs) fs
```

```
idris> :doc concatMap
Prelude.Foldable.concatMap : Foldable t => Monoid m =>
    (a -> m) -> t a -> m

    Combine into a monoid the collective results of applying
    a function to each element of a structure
```

- Список — это Functor
- Моноидная операция — это конкатенация
- Список — это Foldable

Интерфейс Foldable

```
interface Foldable (t : Type -> Type) where
  foldr : (elt -> acc -> acc) -> acc
            -> t elt -> acc
  foldl : (acc -> elt -> acc) -> acc
            -> t elt -> acc

foldr (°) acc [x1,x2,...,xn]
    === x1 ° (x2 ° ... (xn ° acc)...)

foldl (°) acc [x1,x2,...,xn]
    === (...((acc ° x1) ° x2)...) ° xn
```

От функторов к монадам

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b

interface Functor f
     => Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b

interface Applicative m
     => Monad (m : Type -> Type) where
    (>>=)  : m a -> (a -> m b) -> m b
```

Модули и пространства имён

Программа как набор модулей

Файл “ModA.idr”

```
module ModA

-- интерфейсы и реализации
-- типы и функции
```

File “ModB.idr”

```
module ModB

-- интерфейсы и реализации
-- типы и функции
```

File “program.idr”

```
module Main

import ModA
import ModB

main : IO ()
main = ...
```

Модули в подкаталогах

Файл “Utils/Mod.idr”

```
module Utils.Mod

-- ...
```

Файл “program.idr”

```
module Main

import Utils.Mod

main : IO ()
main = ...
```

Полностью квалифицированные имена

```
idris> :type (::)
ForeignEnv.(::) : (ffi_types f t, t) ->
                  FEnv f xs -> FEnv f (t :: xs)
Prelude.List.(::) : elem -> List elem -> List elem
Prelude.Stream.(::) :
       a -> Lazy' LazyCodata (Stream a) -> Stream a

idris> :type Prelude.List.(::)
(::) : elem -> List elem -> List elem
```

Модификаторы экспорта

- Можно экспортировать имена, конструкторы, реализацию интерфейсов
- Модификаторы private/export/public для экспорта
- Директива `%access` для правил экспорта по умолчанию

Явные пространства имён

```
module Foo

namespace x
  test : Double -> Double
  test a = a * 2

namespace y
  test : String -> String
  test s = s ++ s
```

```
idris> test 10
20.0 : Double
idris> test "aaa"
"aaaaaa" : String
idris> Foo.x.test 10
20.0 : Double
```

Список литературы


Brady, Edwin (March, 2017). *Type-Driven Development with Idris*. Manning Publications.


The Idris Tutorial. URL: `http://docs.idris-lang.org/en/latest/tutorial/index.html`.