Программирование с зависимыми типами на языке Idris

Лекция 5. Интерфейсы, модули, пространства имён

В. Н. Брагилевский

12 февраля 2017 г.

Computer Science клуб (Санкт-Петербург)

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

```
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
```

Другие интерфейсы из Prelude

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
```

Другие интерфейсы из Prelude (2)

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
```

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 "123") (Just "456")
Just "123" : Maybe String
Idris> (<+>) @{collectJust} (Just "123") (Just "456")
Just "123456" : 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 10
GetMultiplicative 50 : 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 возможнм признаком ошибки и её объяснением (`Either a b`).
- Результат недетерминированного вычисления (`List a`).

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

```
Idris> :t (<$>)
(<$>) : 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 : способ 1

```
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
```

Сложение двух Maybe: способ 2

```
m_add' : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add' a b = pure plus <*> a <*> b

Idris> m_add' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add' (Just 5) Nothing
Nothing : Maybe Nat

Applicative Maybe where
    pure = Just

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

Способ 3 (Idiom Brackets)

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

```
Idris> m_add'' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add'' (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_add''' : Maybe Nat -> Maybe Nat -> Maybe Nat
m_add''' a b = pure (!a + !b)
```

```
Idris> m_add''' (Just 5) (Just 10)
Just 15 : Maybe Nat
Idris> m_add''' (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
```

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> :t (::)
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> :t 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 x = x * 2

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

```
Idris> Foo.x.test 10
20.0 : Double
```

Список литературы

Idris: A Language with Dependent Types. URL: http://www.idris-lang.org/.

- Idris Tutorial: Interfaces
  http://docs.idris-lang.org/en/latest/tutorial/interfaces.html
- Idris Tutorial: Modules and Namespaces
  http://docs.idris-lang.org/en/latest/tutorial/modules.html
- Idris Libraries Source Code
  https://github.com/idris-lang/Idris-dev/tree/master/libs/