Декларативное программирование

Семинар №9, группа 22215

Завьялов А.А.

31 октября 2022 г.

Кафедра систем информатики ФИТ НГУ

Классы типов

- Один из механизмов *обобщенного* программирования в Haskell
- Описывают множество типов с одним набором операций

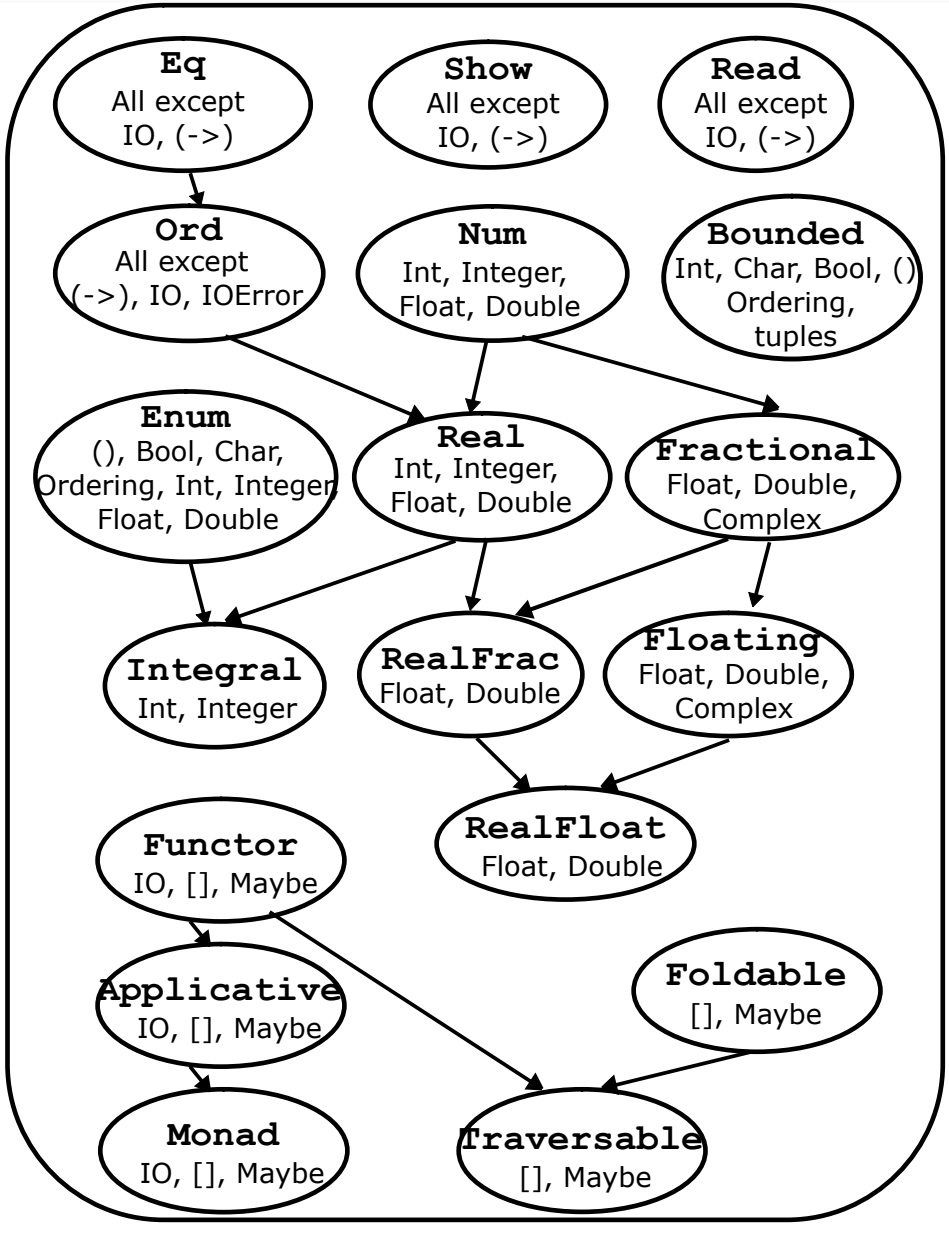
```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a
    {-# MINIMAL compare | (<=) #-}

data Bin = Zero | One deriving (Eq)
instance Ord Bin where
    Zero <= One = TRUE
    x <= y = x == y
```

Стандартные классы типов

- `Eq` – отношение эквивалентности
- `Ord` – отношение *полного* порядка
- `Show` – перевод в строку
- `Read` – чтение из строки
- `Enum` – перечислимое множество
- `Bounded` – множество с точной верхней и нижней границами
- `Num` – арифметические операции
  - `Integral` – целые числа
  - `Real` – вещественные числа

Иерархия некоторых стандартных классов типов



Полугруппы, моноиды

Класс типов `Semigroup`

- Полугруппа — множество с заданной на нём **ассоциативной** операцией $(S, \cdot)$.

```
class Semigroup a where
    (<>) :: a -> a -> a
```

Примеры

- Списки и операция конкатенации (++)
- Упорядоченное множество с операцией минимума (максимума) из двух элементов

Свойство полугруппы (semigroup law)

- `x <> (y <> z) = (x <> y) <> z` — ассоциативность

Класс типов `Monoid`

- Моноид — полугруппа с нейтральным элементом (''единицей'').

```
class Semigroup a => Monoid a where
    mempty  :: a
    mappend :: a -> a -> a
    mappend = (<>)
    mconcat :: [a] -> a
```

Свойства моноида (monoid laws)

- `x `mappend` mempty = x`
- `mempty `mappend` x = x`
- `x <> (y <> z) = (x <> y) <> z` — ассоциативность

Примеры моноидов

- Целые числа, 0 и операция сложения
- Целые числа, 1 и операция умножения
- Булевы значения, False и логическое "или"
- Булевы значения, True и логическое "и"
- Списки, пустой список и операция конкатенации
- Ограниченное снизу (сверху) множество, операция минимума (максимума) из двух и нижняя (верхняя) грань

Практическое применение моноидов

Ассоциативность решает

```
foldl mappend mempty col = foldr mappend mempty col
fold mappend mempty [] = mempty
mconcat = foldr mappend mempty
```

- Ассоциативность $\equiv$ порядок вычислений не важен
- Свёртка над списком моноидов – дополнительная гарантия, что вы получите нужный результат
  - при условии, что выполнены законы моноидов

Функторы, в том числе, аппликативные

Тип данных `Maybe`

`Maybe` моделирует ситуацию, когда значение может отсутствовать:

```
data Maybe a  =  Nothing | Just a
```

Возможные подходы к обработке отсутствующих значений

- Элемента нет – бросаем исключение, затем обрабатываем (C#, Java)
- Элемента нет – возвращаем `null` (Алгол, C, C++, C#, Java, *etc*)
- Элемент есть – возвращаем `Just elem`, элемента нет – возвращаем `Nothing` (Haskell, Rust, Swift, Kotlin, *большинство современных языков*)

Работа с `Maybe`

Использование `Maybe` делит иерархию типов программы на обычные (`a`, `[a]`, `Int`, `...`) и типы, значение которых может отсутствовать (`Maybe a`, `Maybe [a]`, `Maybe Int`, `...`)

Очевидный подход к использованию `Maybe`

```
showFromMaybe :: Show a => Maybe a -> String
showFromMaybe Nothing = ""
showFromMaybe (Just x) = show x
```

Круто, модно, но есть одна проблема...

Работа с `Maybe`

Есть `Just 42`, нужно прибавить к значению внутри единицу.

Не вопрос!

```
addOneToMaybe :: Maybe Int -> Maybe Int
addOneToMaybe Nothing = Nothing
addOneToMaybe (Just x) = Just (succ x)
```

Но...

- Неужели всегда нужно писать так много кода?
- Неужели нужно всё выносить в отдельные функции?
- Как такие функции комбинировать между собой?

Functors to the rescue!

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

>fmap succ (Just 42)
Just 43
>fmap succ Nothing
```

- обратите внимание: `f` – тип с параметром
- `fmap` выполняет операцию `a -> b` внутри типа `f`, и переводит значение типа `f a` в значение типа `f b`
- можно считать, что мы преобразовали функцию `a -> b` в функцию `f a -> f b` ("подняли" (англ. – lift) вычисление в функтор)

Функторы

Законы функторов

- `fmap id == id` – закон тождественности (Identity)
- `fmap (f . g) == fmap f . fmap g` – закон композиции (Composition)

Использование функторов

```
>fmap (+1) [1,2,3]
[2,3,4]
>(+1) <$> [1,2,3]
[2,3,4]
> (+1) $ 2
3
```

Типы, реализующие класс `Functor`

- `Maybe`
- `[]`
- `(,)` (!)
- И многие другие...

Где функторы бессильны...

```haskell
-- Унарная операция (a -> b), всё ok
(+1) <$> Just 1

-- Бинарная операция, ???
(+) <$> Just 1 ??? Just 2

fmap (+) :: (Num a, Functor f) => f (a -> a)
-- С значениями вида f (a -> b)
-- уже ничего не можем сделать...
```

...найдется дело для аппликативных функторов!

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


-- Бинарная операция? Легко!
(+) <$> Just 1 <*> Just 2
-- Just 3
```

Типы, реализующие класс `Applicative`

- `Maybe`
- `[]`
- ~~`(,)`~~ Почему?
- И многие другие...

Список как аппликативный функтор

Моделирует недетерминированные вычисления:

QuickCheck и проверка свойств

Q&A