Лекция 4: классы типов

Функциональное программирование на Haskell

Алексей Романов

7 марта 2018

МИЭТ

Классы типов

- Полиморфные функции, которые мы видели в прошлый раз, имеют одно определение для любых параметров типов.
- Часто нужно определить функции по-разному для разных типов.
- Для этого используются классы типов.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

- Eq — название класса.
- a — тип, относящийся к этому классу.
- `(==)` и `(/=)` — функции-члены класса, с определениями по умолчанию.
- Их нужно определить для типа a

Классы типов

- Полиморфные функции, которые мы видели в прошлый раз, имеют одно определение для любых параметров типов.
- Часто нужно определить функции по-разному для разных типов.
- Для этого используются классы типов.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

- Eq — название класса.
- a — тип, относящийся к этому классу.
- (==) и (/=) — функции-члены класса, с определениями по умолчанию.
- Их нужно определить для типа a (достаточно одну).

Экземпляры классов

- Чтобы объявить, что тип `Bool` относится к классу `Eq`:

```
instance Eq Bool where
    -- (==) :: Bool -> Bool -> Bool
    True == True = True
    False == False = True
```

Экземпляры классов

- Чтобы объявить, что тип `Bool` относится к классу `Eq`:

```
instance Eq Bool where
    -- (==) :: Bool -> Bool -> Bool
    True == True = True
    False == False = True
    _ == _ = False
```

- При этом a в сигнатуре членов заменяется на тип, для которого определяется экземпляр.
- Ещё пример:

Экземпляры классов

- Чтобы объявить, что тип `Bool` относится к классу `Eq`:

```
instance Eq Bool where
    -- (==) :: Bool -> Bool -> Bool
    True == True = True
    False == False = True
    _ == _ = False
```

- При этом a в сигнатуре членов заменяется на тип, для которого определяется экземпляр.
- Ещё пример:

```
instance Eq IpAddress where
    -- (==) :: IpAddress -> IpAddress -> Bool
    Ip4Address x == Ip4Address y = x == y
    Ip6Address x == Ip6Address y = x == y
    _ == _ = False
```

- Рекурсивно ли это определение?

Экземпляры классов

- Чтобы объявить, что тип `Bool` относится к классу `Eq`:

```
instance Eq Bool where
    -- (==) :: Bool -> Bool -> Bool
    True == True = True
    False == False = True
    _ == _ = False
```

- При этом a в сигнатуре членов заменяется на тип, для которого определяется экземпляр.
- Ещё пример:

```
instance Eq IpAddress where
    -- (==) :: IpAddress -> IpAddress -> Bool
    Ip4Address x == Ip4Address y = x == y
    Ip6Address x == Ip6Address y = x == y
    _ == _ = False
```

- Это не рекурсия: `==` в правой части относится к другому типу (`String`).

Ограниченный полиморфизм

- Если спросить GHCi про тип ==, получим

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Читается «`a -> a -> Bool` для любого a из `Eq`».
- Часть перед => называется контекстом.
- Его элементы — ограничения. Когда их больше одного, они пишутся в скобках через запятую.

Ограниченный полиморфизм

- Если спросить GHCi про тип ==, получим

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Читается «`a -> a -> Bool` для любого a из `Eq`».
- Часть перед => называется контекстом.
- Его элементы — ограничения. Когда их больше одного, они пишутся в скобках через запятую.
- Типы полиморфных функций, *использующих* == и /=, пишутся аналогично. Определим:

```
foo [] = True
foo [x] = True
foo (x : tail@(y : _)) = x == y && foo tail
```

Какой у неё тип (и смысл)?

Ограниченный полиморфизм

- Если спросить GHCi про тип ==, получим

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Читается «`a -> a -> Bool` для любого a из `Eq`».
- Часть перед => называется контекстом.
- Его элементы — ограничения. Когда их больше одного, они пишутся в скобках через запятую.
- Типы полиморфных функций, *использующих* == и */=*, пишутся аналогично. Определим:

```
foo [] = True
foo [x] = True
foo (x : tail@(y : _)) = x == y && foo tail
```

Какой у неё тип (и смысл)?

- `foo :: Eq a => [a] -> Bool`

Ограниченный полиморфизм

- Если спросить GHCi про тип ==, получим

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Читается «`a -> a -> Bool` для любого a из `Eq`».
- Часть перед => называется контекстом.
- Его элементы — ограничения. Когда их больше одного, они пишутся в скобках через запятую.
- Типы полиморфных функций, *использующих* == и */=*, пишутся аналогично. Определим:

```
foo [] = True
foo [x] = True
foo (x : tail@(y : _)) = x == y && foo tail
```

Какой у неё тип (и смысл)?

- `foo :: Eq a => [a] -> Bool`
- Функция проверяет, равны ли все элементы списка.

Экземпляры для полиморфных типов

- Попробуем определить `Eq` для `Maybe a`:

Экземпляры для полиморфных типов

- Попробуем определить `Eq` для `Maybe a`:

```
instance Eq (Maybe a) where
    -- (==) :: Maybe a -> Maybe a -> Bool
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

- Верное ли это определение?

Экземпляры для полиморфных типов

- Попробуем определить Eq для `Maybe a`:

```
instance Eq (Maybe a) where
    -- (==) :: Maybe a -> Maybe a -> Bool
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

- Здесь a не обязательно Eq, и поэтому `x == y` не скомпилируется.
- А можно ли так?

```
instance Eq (Maybe a) where
    (==) :: Eq a => Maybe a -> Maybe a -> Bool
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

Экземпляры для полиморфных типов

- Попробуем определить Eq для `Maybe a`:

```
instance Eq (Maybe a) where
    -- (==) :: Maybe a -> Maybe a -> Bool
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

- Здесь a не обязательно Eq, и поэтому `x == y` не скомпилируется.
- А можно ли так?

```
instance Eq (Maybe a) where
    (==) :: Eq a => Maybe a -> Maybe a -> Bool
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

- Нет: эта сигнатура для `(==)` более ограничена, чем заданная классом.

Экземпляры для полиморфных типов с ограничениями

- Правильно так:

```
instance Eq a => Eq (Maybe a) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

- Читаем «`Maybe a` относится к `Eq` только тогда, когда `a` относится к `Eq`».
- Контексты экземпляров подчиняются тем же правилам, что контексты функций.
- Ограничения могут упоминать другие классы, а не только тот, экземпляр которого определяется.
- Не может быть много экземпляров одного класса для одного типа с разными условиями.

Аналоги в других языках

- Как перегрузка функций (или операторов), но глубже встроенная в систему типов.
- Перегруженные функции — просто разные функции, связанные только общим именем.
- Здесь обязательно близкие типы и общий смысл.
- И функции могут быть использованы другими полиморфными функциями.

Аналоги в других языках

- Как перегрузка функций (или операторов), но глубже встроенная в систему типов.
- Перегруженные функции — просто разные функции, связанные только общим именем.
- Здесь обязательно близкие типы и общий смысл.
- И функции могут быть использованы другими полиморфными функциями.
- Не путайте классы типов и классы в ООП, общее только название!
- В C++ есть близкое понятие концепций.

Аналоги в других языках

- Как перегрузка функций (или операторов), но глубже встроенная в систему типов.
- Перегруженные функции — просто разные функции, связанные только общим именем.
- Здесь обязательно близкие типы и общий смысл.
- И функции могут быть использованы другими полиморфными функциями.
- Не путайте классы типов и классы в ООП, общее только название!
- В C++ есть близкое понятие концепций.
- Пока только для документации

Аналоги в других языках

- Как перегрузка функций (или операторов), но глубже встроенная в систему типов.
- Перегруженные функции — просто разные функции, связанные только общим именем.
- Здесь обязательно близкие типы и общий смысл.
- И функции могут быть использованы другими полиморфными функциями.
- Не путайте классы типов и классы в ООП, общее только название!
- В C++ есть близкое понятие концепций.
- Пока только для документации; в C++17 их хотели добавить в сам язык, но не получилось.

Аналоги в других языках

- Как перегрузка функций (или операторов), но глубже встроенная в систему типов.
- Перегруженные функции — просто разные функции, связанные только общим именем.
- Здесь обязательно близкие типы и общий смысл.
- И функции могут быть использованы другими полиморфными функциями.
- Не путайте классы типов и классы в ООП, общее только название!
- В C++ есть близкое понятие концепций.
- Пока только для документации; в C++17 их хотели добавить в сам язык, но не получилось.
- Часто сравнивают с интерфейсами в Java/C#, но это подходит хуже:

Аналоги в других языках

- Как перегрузка функций (или операторов), но глубже встроенная в систему типов.
- Перегруженные функции — просто разные функции, связанные только общим именем.
- Здесь обязательно близкие типы и общий смысл.
- И функции могут быть использованы другими полиморфными функциями.
- Не путайте классы типов и классы в ООП, общее только название!
- В C++ есть близкое понятие концепций.
- Пока только для документации; в C++17 их хотели добавить в сам язык, но не получилось.
- Часто сравнивают с интерфейсами в Java/C#, но это подходит хуже: в классах типов есть «статические» и бинарные методы, совсем другой смысл возврата из функций.

Другие важные классы

- Линейно упорядоченные типы:

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

  - `Eq a` — надкласс `Ord`, то есть любой экземпляр `Ord` должен быть и экземпляром `Eq`.

Другие важные классы

- Линейно упорядоченные типы:

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

  - `Eq a` — надкласс `Ord`, то есть любой экземпляр `Ord` должен быть и экземпляром `Eq`.
  - Логически стрелка скорее в другом направлении: a относится к `Ord` $\Longrightarrow$ a относится к `Eq`.

Другие важные классы

- Линейно упорядоченные типы:

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

  - `Eq a` — надкласс `Ord`, то есть любой экземпляр `Ord` должен быть и экземпляром `Eq`.
  - Логически стрелка скорее в другом направлении: `a` относится к `Ord` $\Longrightarrow$ `a` относится к `Eq`.
- Типы, ограниченные сверху и снизу:

```
class Bounded a where
    minBound, maxBound :: a
```

  - Это не подкласс `Ord` потому, что порядок, для которого эти значения — границы, может быть частичным.

Преобразования в строки и обратно

- Типы, значения которых можно превратить в строки:

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

Преобразования в строки и обратно

- Типы, значения которых можно превратить в строки:

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

- Или прочитать из строк:

```
class Read a where ...

read :: Read a => String -> a
```

Преобразования в строки и обратно

- Типы, значения которых можно превратить в строки:

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

- Или прочитать из строк:

```
class Read a where ...

read :: Read a => String -> a
```

- Для нас то, что за многоточиями, неважно, но подробности описаны в документации.

Преобразования в строки и обратно

- Типы, значения которых можно превратить в строки:

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

- Или прочитать из строк:

```
class Read a where ...

read :: Read a => String -> a
```

- Для нас то, что за многоточиями, неважно, но подробности описаны в документации.
- `read` должна быть обратной к `show`.
- Заметьте полиморфизм `read` по возвращаемому типу:

```
Prelude> read "2" :: Integer
2
Prelude> read "2" :: Double
2.0
```

Опасность read

- Что случится, если передать read неподходящий аргумент?

```
Prelude> read "1" :: [Bool]
```

Опасность read

- Что случится, если передать `read` неподходящий аргумент?

  ```
  Prelude> read "1" :: [Bool]
  *** Exception: Prelude.read: no parse
  ```

- Если вы не знаете *точно*, что текст содержит значение, используйте безопасную

  ```
  readMaybe :: Read a => String -> Maybe a
  ```

  из модуля `Text.Read`.

Опасность read

- Что случится, если передать `read` неподходящий аргумент?

  ```
  Prelude> read "1" :: [Bool]
  *** Exception: Prelude.read: no parse
  ```

- Если вы не знаете *точно*, что текст содержит значение, используйте безопасную

  ```
  readMaybe :: Read a => String -> Maybe a
  ```

  из модуля `Text.Read`.

- Это не единственная функция в Prelude с такой проблемой.
- Пакет safe содержит безопасные варианты для всех таких функций (посмотрите его документацию).

Класс перечислимых типов

- `Enum` содержит типы, значения которых образуют последовательность.

```
class Enum a where
    succ, pred :: a -> a
    toEnum   :: Int -> a
    fromEnum :: a -> Int
    enumFrom :: a -> [a] -- и ещё 3 вариации
```

Класс перечислимых типов

- `Enum` содержит типы, значения которых образуют последовательность.

```
class Enum a where
    succ, pred :: a -> a
    toEnum   :: Int -> a
    fromEnum :: a -> Int
    enumFrom :: a -> [a] -- и ещё 3 вариации
```

- Для него есть специальный синтаксис:
  - `[n..]` означает `enumFrom n`;
  - `[n..m]` — `enumFromTo n m`;
  - `[n,m..]` — `enumFromThen n m`;
  - `[n,m..l]` — `enumFromThenTo n m l`.

Класс перечислимых типов

- `Enum` содержит типы, значения которых образуют последовательность.

```
class Enum a where
    succ, pred :: a -> a
    toEnum   :: Int -> a
    fromEnum :: a -> Int
    enumFrom :: a -> [a] -- и ещё 3 вариации
```

- Для него есть специальный синтаксис:
  - `[n..]` означает `enumFrom n`;
  - `[n..m]` — `enumFromTo n m`;
  - `[n,m..]` — `enumFromThen n m`;
  - `[n,m..l]` — `enumFromThenTo n m l`.
- Для `Float` и `Double` есть экземпляры `Enum`, так что можно писать `[1.0..]`:

```
Prelude> [1.2..2.0]
```

Класс перечислимых типов

- `Enum` содержит типы, значения которых образуют последовательность.

```
class Enum a where
    succ, pred :: a -> a
    toEnum   :: Int -> a
    fromEnum :: a -> Int
    enumFrom :: a -> [a] -- и ещё 3 вариации
```

- Для него есть специальный синтаксис:
  - `[n..]` означает `enumFrom n`;
  - `[n..m]` — `enumFromTo n m`;
  - `[n,m..]` — `enumFromThen n m`;
  - `[n,m..l]` — `enumFromThenTo n m l`.
- Для `Float` и `Double` есть экземпляры `Enum`, так что можно писать `[1.0..]`, но они довольно странные:

```
Prelude> [1.2..2.0]
[1.2,2.2]
```

Иерархия числовых классов

- Вернёмся к «числовым типам» из первой лекции.
- Это тоже классы типов, образующие довольно сложную иерархию.

Иерархия числовых классов

- Вернёмся к «числовым типам» из первой лекции.
- Это тоже классы типов, образующие довольно сложную иерархию.

```
class Num a where
    (+), (-), (*), negate, abs, signum,
   fromInteger

class (Num a, Ord a) => Real a where
    toRational

class (Real a, Enum a) => Integral a where
    quot, rem, div, mod, quotRem, divMod
    toInteger

class Num a => Fractional a where
    (/), recip, fromRational
```

Иерархия числовых классов: дробные типы

```
class Fractional a => Floating a where
    exp, log, sqrt, (**), logBase,
    pi, sin, cos, ...

class (Real a, Fractional a) => RealFrac a where
    properFraction :: Integral b => a -> (b, a)
    truncate, round, ceiling, floor ::
        Integral b => a -> b

class (RealFrac a, Floating a) => RealFloat a where
    isNaN, isInfinite :: a -> Bool
    ...

fromIntegral :: (Integral a, Num b) => a -> b
realToFrac :: (Real a, Fractional b) => a -> b
```

- Полная документация.

Числовые литералы

- Литерал `0` означает `fromInteger (0 :: Integer)` и соответственно получает тип

Числовые литералы

- Литерал `0` означает `fromInteger (0 :: Integer)` и соответственно получает тип `Num a => a`.
- Аналогично для дробных: `1.1` представляется как рациональное и превращается в `fromRational ((11 % 10) :: Rational)` типа `Fractional a => a`.
- Здесь:

```
module Data.Ratio where
data Ratio a -- экспортировано без конструктора
type Rational = Ratio Integer
(%) :: Integral a => a -> a -> Ratio a
```

Числовые литералы

- Литерал `0` означает `fromInteger (0 :: Integer)` и соответственно получает тип `Num a => a`.
- Аналогично для дробных: `1.1` представляется как рациональное и превращается в `fromRational ((11 % 10) :: Rational)` типа `Fractional a => a`.
- Здесь:

```
module Data.Ratio where
data Ratio a -- экспортировано без конструктора
type Rational = Ratio Integer
(%) :: Integral a => a -> a -> Ratio a
```

- Использование рациональных чисел для дробных литералов позволяет избежать ошибок округления.

Числовые литералы

- Литерал `0` означает `fromInteger (0 :: Integer)` и соответственно получает тип `Num a => a`.
- Аналогично для дробных: `1.1` представляется как рациональное и превращается в `fromRational ((11 % 10) :: Rational)` типа `Fractional a => a`.
- Здесь:

```
module Data.Ratio where
data Ratio a -- экспортировано без конструктора
type Rational = Ratio Integer
(%) :: Integral a => a -> a -> Ratio a
```

- Использование рациональных чисел для дробных литералов позволяет избежать ошибок округления.
- Конечно, при преобразовании в `Float` или `Double` они вернутся.

Алгебраические классы

- Для разных структур из общей алгебры или линейной алгебры можно определить соответствующие им классы. В стандартной библиотеке есть
- моноиды в `Prelude`:

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a

x <> y = mappend x y -- синоним для mappend
```

- и полугруппы в `Data.Semigroup`.

Алгебраические классы

- Для разных структур из общей алгебры или линейной алгебры можно определить соответствующие им классы. В стандартной библиотеке есть
- моноиды в `Prelude`:

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a

x <> y = mappend x y -- синоним для mappend
```

- и полугруппы в `Data.Semigroup`.
- Ещё несколько из теории категорий, но это предмет будущих лекций.

Определение функций внутри или вне класса

- При определении функций над классом часто приходится выбирать из двух вариантов:
  - член класса с определением по умолчанию;
  - функция, определённая вне класса.
- Например, `(/=)` можно было бы определить вне класса:

```
(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

Определение функций внутри или вне класса

- При определении функций над классом часто приходится выбирать из двух вариантов:
  - член класса с определением по умолчанию;
  - функция, определённая вне класса.
- Например, `(/=)` можно было бы определить вне класса:

```
(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

- Такую функцию нельзя реализовать по-разному для разных экземпляров.

Определение функций внутри или вне класса

- При определении функций над классом часто приходится выбирать из двух вариантов:
  - член класса с определением по умолчанию;
  - функция, определённая вне класса.
- Например, `(/=)` можно было бы определить вне класса:

```
(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

- Такую функцию нельзя реализовать по-разному для разных экземпляров.
- Если для каких-то типов есть реализация лучше той, что по умолчанию, лучше сделать функцию членом класса.

Определение функций внутри или вне класса

- При определении функций над классом часто приходится выбирать из двух вариантов:
  - член класса с определением по умолчанию;
  - функция, определённая вне класса.
- Например, `(/=)` можно было бы определить вне класса:

```
(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

- Такую функцию нельзя реализовать по-разному для разных экземпляров.
- Если для каких-то типов есть реализация лучше той, что по умолчанию, лучше сделать функцию членом класса.
  - Очевидная проблема: вы можете не знать (или не подумать) о таких типах.

Определение функций внутри или вне класса

- При определении функций над классом часто приходится выбирать из двух вариантов:
  - член класса с определением по умолчанию;
  - функция, определённая вне класса.
- Например, `(/=)` можно было бы определить вне класса:

```
(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

- Такую функцию нельзя реализовать по-разному для разных экземпляров.
- Если для каких-то типов есть реализация лучше той, что по умолчанию, лучше сделать функцию членом класса.
  - Очевидная проблема: вы можете не знать (или не подумать) о таких типах.
- Или если реализация по умолчанию не всегда верна

Определение функций внутри или вне класса

- При определении функций над классом часто приходится выбирать из двух вариантов:
  - член класса с определением по умолчанию;
  - функция, определённая вне класса.
- Например, `(/=)` можно было бы определить вне класса:

```
(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

- Такую функцию нельзя реализовать по-разному для разных экземпляров.
- Если для каких-то типов есть реализация лучше той, что по умолчанию, лучше сделать функцию членом класса.
  - Очевидная проблема: вы можете не знать (или не подумать) о таких типах.
- Или если реализация по умолчанию не всегда верна (но тогда почему это реализация по умолчанию?).

Определение функций внутри или вне класса: случай `Ord`

- Ещё пример: в `Ord` можно было бы оставить только `(<=)` членом класса.
- Или только `compare`.
- Но определение `(<)` и других функций через них не всегда оптимально.
- (Подумайте, почему!)

Законы классов типов

- Рассмотрим такой экземпляр типа:

```
data Weird = Weird Int

instance Eq Weird where
    Weird x == Weird y = x /= y
```

- Даже не зная смысла `Weird` мы видим, что что-то здесь не так. Что?

Законы классов типов

- Рассмотрим такой экземпляр типа:

```
data Weird = Weird Int

instance Eq Weird where
    Weird x == Weird y = x /= y
```

- Даже не зная смысла `Weird` мы видим, что что-то здесь не так.
- Это определение нерефлексивно, то есть `Weird 1 /= Weird 1`.
- У всех классов типов (почти) есть законы.
- То есть наличия функций типов, указанных в определении, недостаточно;
- для корректных экземпляров должны выполняться определённые законы.

Законы классов типов

- Например, для Eq законы такие:

Законы классов типов

- Например, для Eq законы такие:

$$\forall\ \texttt{x :: a. x == x}$$

$$\forall\ \texttt{x, y :: a. x == y} \implies \texttt{y == x}$$

$$\forall\ \texttt{x, y, z :: a. x == y} \wedge \texttt{y == z} \implies \texttt{x == z}$$

Законы классов типов

- Например, для Eq законы такие:

$$\forall\ \texttt{x :: a. x == x}$$

$$\forall\ \texttt{x, y :: a. x == y} \implies \texttt{y == x}$$

$$\forall\ \texttt{x, y, z :: a. x == y} \wedge \texttt{y == z} \implies \texttt{x == z}$$

$$\forall\ \texttt{x, y :: a; Eq b; f :: a -> b. x == y} \implies$$

Законы классов типов

- Например, для Eq законы такие:

```
∀ x :: a. x == x
∀ x, y :: a. x == y ⟹ y == x
∀ x, y, z :: a. x == y ∧ y == z ⟹ x == z

∀ x, y :: a; Eq b; f :: a -> b. x == y ⟹
    f x == f y
```

Законы классов типов

- Например, для Eq законы такие:

```
∀ x :: a. x == x
∀ x, y :: a. x == y ⟹ y == x
∀ x, y, z :: a. x == y ∧ y == z ⟹ x == z

∀ x, y :: a; Eq b; f :: a -> b. x == y ⟹
    f x == f y
```

- Законы могут говорить и о связи между классами. Например, если тип одновременно Ord и Bounded, то естественно требовать

```
∀ x :: a. x <= maxBound
∀ x :: a. x >= minBound
```

Законы классов типов

- Например, для Eq законы такие:

```
∀ x :: a. x == x
∀ x, y :: a. x == y ⟹ y == x
∀ x, y, z :: a. x == y ∧ y == z ⟹ x == z

∀ x, y :: a; Eq b; f :: a -> b. x == y ⟹
    f x == f y
```

- Законы могут говорить и о связи между классами. Например, если тип одновременно Ord и Bounded, то естественно требовать

```
∀ x :: a. x <= maxBound
∀ x :: a. x >= minBound
```

- Компилятор не может проверить законы.
- Поэтому ответственность за их соблюдение лежит на программисте.

Автоматический вывод экземпляров

- Почти все определения (==) устроены одинаково:

Автоматический вывод экземпляров

- Почти все определения (==) устроены одинаково:
  - Типы всех полей всех конструкторов должны быть Eq;
  - Значения с одинаковым конструктором равны, если равны все их поля (или полей нет);
  - Значения с разными конструкторами не равны;
- Почти, но не все:

Автоматический вывод экземпляров

- Почти все определения (==) устроены одинаково:
  - Типы всех полей всех конструкторов должны быть Eq;
  - Значения с одинаковым конструктором равны, если равны все их поля (или полей нет);
  - Значения с разными конструкторами не равны;
- Почти, но не все: для рациональных чисел не так.

Автоматический вывод экземпляров

- Почти все определения `(==)` устроены одинаково:
  - Типы всех полей всех конструкторов должны быть `Eq`;
  - Значения с одинаковым конструктором равны, если равны все их поля (или полей нет);
  - Значения с разными конструкторами не равны;
- Почти, но не все: для рациональных чисел не так.
- Чтобы не писать такие определения каждый раз, после `data` или `newtype` ставят `deriving (Eq)`.
- `deriving` также работает для `Ord`, `Show`, `Read`, `Enum` и `Bounded`.
- Порядок на значениях для `Ord`, `Enum` и `Bounded` задаётся порядком конструкторов в определении.

Автоматический вывод экземпляров

- Почти все определения `(==)` устроены одинаково:
  - Типы всех полей всех конструкторов должны быть `Eq`;
  - Значения с одинаковым конструктором равны, если равны все их поля (или полей нет);
  - Значения с разными конструкторами не равны;
- Почти, но не все: для рациональных чисел не так.
- Чтобы не писать такие определения каждый раз, после `data` или `newtype` ставят `deriving (Eq)`.
- `deriving` также работает для `Ord`, `Show`, `Read`, `Enum` и `Bounded`.
- Порядок на значениях для `Ord`, `Enum` и `Bounded` задаётся порядком конструкторов в определении.
- Разные расширения GHC позволяют `deriving` для других классов.

Классы типов и `newtype`

- Для каждой пары класса и типа можно определить только один экземпляр.
- А что делать, если осмысленных экземпляров больше одного?
- Например, числа — моноиды по операциям + и *.

Классы типов и newtype

- Для каждой пары класса и типа можно определить только один экземпляр.
- А что делать, если осмысленных экземпляров больше одного?
- Например, числа — моноиды по операциям + и *.
- В таком случае мы можем завернуть существующий тип в newtype и определить для нового:

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Ord, Read, Show, Bounded, Num)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x + y)
```

Классы типов и newtype

- Для каждой пары класса и типа можно определить только один экземпляр.
- А что делать, если осмысленных экземпляров больше одного?
- Например, числа — моноиды по операциям + и *.
- В таком случае мы можем завернуть существующий тип в newtype и определить для нового:

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Ord, Read, Show, Bounded, Num)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x + y)

newtype Product a = Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded, Num)

instance Num a => Monoid (Product a) where ...
```

Классы типов и newtype

- Если заметили, `Num` не было в списке классов, для которых есть автоматический вывод экземпляров.
- Для `newtype` можно вывести любой класс с помощью расширения `GeneralizedNewtypeDeriving`.

Классы типов и newtype

- Если заметили, Num не было в списке классов, для которых есть автоматический вывод экземпляров.
- Для newtype можно вывести любой класс с помощью расширения GeneralizedNewtypeDeriving.
- Выведенное определение каждой функции вызывает ту же функцию завёрнутого типа с вставкой и разбором конструкторов где нужно:

```
instance Num a => Num (Sum a) where
    Sum x + Sum y = Sum (x + y)
    Sum x * Sum y = Sum (x * y)
    negate (Sum x) = Sum (negate x)
    ...
```

Классы типов и модули

- `Класс(..)` в списке импорта/экспорта означает класс со всеми методами, как для типа.
- Поскольку экземпляры не имеют имён, они экспортируются из модуля всегда.
- И импортируются всегда, когда модуль упоминается в списке импорта.

Классы типов и модули

- `Класс(..)` в списке импорта/экспорта означает класс со всеми методами, как для типа.
- Поскольку экземпляры не имеют имён, они экспортируются из модуля всегда.
- И импортируются всегда, когда модуль упоминается в списке импорта.

- Из-за этого существование экземпляров для пары класс-тип не в одном модуле — проблема.
- Поэтому экземпляры обычно определяются либо в том модуле, где определён тип, либо в том, где определён класс.
- Остальные называются сиротами и по возможности их создания следует избегать.
- Это делается с помощью того же трюка с `newtype`.

Ограничения классов типов в стандартном Haskell и расширения GHC

- В стандартном Haskell есть несколько ограничений на объявления классов типов и экземпляров.
- А в GHC расширений, которые их снимают.
- Все перечислять не будем, они есть в User's Guide, раздел 10.8.
- Сейчас достаточно знать, что если что-то не компилируется из-за отсутствия `-XFlexibleInstances`, `-XFlexibleContexts` или `-XInstanceSigs`, их спокойно можно включить.
- `-XUndecidableInstances`, `-XOverlappingInstances`, `-XIncoherentInstances` опаснее, но в этом курсе в любом случае не понадобятся.