

# Programando com Classe

Felipe A. Lessa

Departamento de Ciência da Computação  
Universidade de Brasília

## Assuntos de hoje

### Classes

“O quê?” e “como?”

Haskell 98

Usando as classes

### Finalizando

Acabou...

## Ops... mas e aquilo?

- ▶ Pessoal, nós vamos pular alguns passos agora =).
- ▶ Poderíamos ainda falar *muita* coisa antes de classes, mas vendo-as agora vocês já vão conhecer toda a base.
- ▶ Sim, nosso curso é um pouco denso hehe.
- ▶ O importante é não esquecer de *praticar*!

## Assuntos de hoje

### Classes

“O quê?” e “como?”

Haskell 98

Usando as classes

### Finalizando

Acabou...

## Classes? Objetos?!

- ▶ Esqueça as classes da programação orientada a objeto!
- ▶ “[Classes em Haskell e em OOP] são similares o suficiente para causar confusão, porém não o suficiente para deixar que você faça analogias com o que você já sabe.”
- ▶ Se for difícil esquecer OOP, imagine uma classe do Haskell como uma interface do Java — será menos danoso =).
- ▶ Então o que são classes, afinal?

## Ah... classes!

- ▶ Uma *classe (de tipo)* é uma forma de garantir que certas operações estão disponíveis para um determinado tipo.
- ▶ Se você sabe que um tipo *instancia* a classe **Num**, então você pode somar dois dados desse tipo.
- ▶ Você pode somar com os tipos **Int**, **Integer** ou **Double**. Você também pode verificar se dois deles são iguais. Com **Bool** você também pode verificar igualdade, mas você não pode somar dois dados do tipo **Bool**.

## Definindo uma classe

- ▶ Uma classe provê operações. Por exemplo:

```
1 class Eq a where
2   (==), (/=) :: a → a → Bool

4   -- Minimal complete definition:
5   --   (==) or (/=)
6   x /= y    = not (x == y)
7   x == y    = not (x /= y)
```

- ▶ Aqui nós definimos que um tipo *a* instancia **Eq** caso ele proveja as operações `(==)` e `(/=)`.
- ▶ Nós também provemos *definições padrão* para ambos, definindo um em termo do outro: assim, para instanciar **Eq** basta implementar um dos dois, se desejar.

## Definindo uma classe II

- ▶ De um modo geral, a definição do corpo uma classe contém:
  - ▶ *Assinaturas de funções*, cada uma definindo uma função que deve ser implementada (observe que “operação” e “função” são a mesma coisa!).
  - ▶ *Definições* para as funções *cuja assinatura foi dada*. Essas definições são usadas para o caso do tipo que instancia a classe não as implementar.
- ▶ “Instanciar”, “implementar”... e como fazer isso?

## Instanciando uma classe

- ▶ Para instanciar é simples. Por exemplo:

```
1  -- Criamos um novo tipo.
2  data Pessoa = Pessoa {nome :: String, idade :: Int}

4  -- Instanciamos Eq.
5  instance Eq Pessoa where
6    (Pessoa nome idade) == (Pessoa nome' idade') =
7      nome == nome' && idade == idade'

▶ Uma definição alternativa:

5  instance Eq Pessoa where
6    (/=) p1 p2 = nome p1 /= nome p2 || idade p1 /= idade p2
```

## Instanciando uma classe II

- ▶ Observe que, para **Eq**, basta definir uma operação que a outra já vem de graça! =)
- ▶ Em geral isso não acontece e você tem, no corpo da declaração de instanciação, várias definições.
- ▶ Na definição acima, usamos (==) e (/=) enquanto definíamos eles? Não, eles foram usados com as instâncias dos tipos **String** e **Int**.
- ▶ Mas nós podemos, sim, ter recursão!

## Instanciando uma classe III

- ▶ Uma classe *não* é um tipo. Ela é mais um “template” para os tipos implementarem. Isso difere de várias linguagens onde classes podem ser manipuladas como tipos ou como dados. (esqueçam OOP!)
- ▶ A classe **Eq** foi definida no Prelude, enquanto **Pessoa** e sua instância de **Eq** estão num outro arquivo. Os três são totalmente separados e não existe uma regra para onde estarão agrupados. Você pode, por exemplo, criar uma nova classe e instanciá-la para **Int**.
- ▶ Você só pode instanciar classes para tipos criados com **data** ou **newtype**.

## Derivação de classes

- ▶ Certas classes são comuns e óbvias (e.g. a maioria dos tipos que você criar serão membros de **Eq**).
  - ▶ Podemos *derivar* algumas classes automaticamente:
- ```
1  data Pessoa = Pessoa {nome :: String, idade :: Int}
2                                deriving (Eq, Ord, Show)
```
- ▶ Com a definição acima, **Pessoa** é declarada e instanciará as classes **Eq**, **Ord** e **Show**.

## Derivação de classes II

- ▶ As regras de como a instância é criada são definidas pelo padrão Haskell 98. Mas em geral é *bem* óbvio.
- ▶ Por exemplo, na derivação de **Ord** para **Pessoa**, primeiro ele compara os nomes. Se forem iguais, então ele compara as idades.
- ▶ Você só pode usar **deriving** para **Eq**, **Ord**, **Enum**, **Bounded**, **Show** e **Read** (já veremos o que cada é).
- ▶ Não derive as classes que você não precisa! Elas vão gastar tempo e espaço (no código) à toa, principalmente a classe **Read** que é complexa.

## Herança de classes

- ▶ Uma classe pode herdar de outra (esqueça OOP!).
- ▶ Em Haskell, “C herdar de D” significa que para instanciar **C** é obrigatório instanciar **D**.
- ▶ Chamamos de “herança” porque as operações da classe herdada passam a estar disponíveis para a classe herdeira.

## Herança de classes II

- ▶ Por exemplo:

```
1 class (Eq a) => Ord a where
2   compare      :: a -> a -> Ordering
3   (<), (<=), (>=), (>) :: a -> a -> Bool
4   max, min     :: a -> a -> a
5   -- Algumas definicoes padrao foram cortadas daqui.
```

- ▶ Acima é definida a classe **Ord** (que está no **Prelude**).
- ▶ Essa classe herda de **Eq** (é o que diz “(**Eq** a) =>”).
- ▶ Para escrever => use =>.

## Dentro do interpretador

- ▶ O interpretador pode te ajudar com o comando “: info”:

```
1 Prelude> :info Eq
2 class Eq a where
3   (==) :: a -> a -> Bool
4   (/=) :: a -> a -> Bool
5   -- Defined in GHC.Base
6 instance (Eq a) => Eq (Maybe a) -- Defined in Data.Maybe
7 instance (Eq a, Eq b) => Eq (Either a b) -- Defined in Data.Either
8 instance Eq Integer -- Defined in GHC.Num
9 instance Eq Float -- Defined in GHC.Float
10 instance Eq Double -- Defined in GHC.Float
11 instance Eq () -- Defined in GHC.Base
12 instance Eq Char -- Defined in GHC.Base
13 instance Eq Int -- Defined in GHC.Base
14 instance Eq Bool -- Defined in GHC.Base
15 instance Eq Ordering -- Defined in GHC.Base
16 instance (Eq a) => Eq [a] -- Defined in GHC.Base
```

## Dentro do interpretador II

- ▶ Também funciona com tipos além de funções:

```
1 Prelude> :i []
2 data [] a = [] | a : [a]           -- <wired into compiler>
3 instance (Eq a) => Eq [a] -- Defined in GHC.Base
4 instance Monad [] -- Defined in GHC.Base
5 instance Functor [] -- Defined in GHC.Base
6 instance (Ord a) => Ord [a] -- Defined in GHC.Base
7 instance (Read a) => Read [a] -- Defined in GHC.Read
8 instance (Show a) => Show [a] -- Defined in GHC.Show
```

## Assuntos de hoje

### Classes

“O quê?” e “como?”

Haskell 98

Usando as classes

### Finalizando

Acabou...

## Conhecendo algumas classes

- ▶ Vamos agora conhecer algumas classes que foram definidas dentro do Haskell 98 Report.
- ▶ Em geral não vamos mostrar as definições padrão.
- ▶ Vocês já conhecem bem a classe **Eq**.
- ▶ A classe **Ord**, que acaba de ser vista, é para tipos que possuem uma ordenação entre eles.

## Escrevendo e lendo

- ▶ As classes **Show** e **Read** provêem uma forma de passar para e de **String** numa representação válida no Haskell (ou seja, **read . show == id**).

```
1 class Show a where
2   showsPrec      :: Int → a → ShowS
3   show           :: a → String
4   showList       :: [a] → ShowS
5                   -- Minimal complete definition: show or showsPrec

7 class Read a where
8   readsPrec      :: Int → ReadS a
9   readList       :: ReadS [a]
10                  -- Minimal complete definition: readsPrec
```

- ▶ Em geral essas classes são apenas derivadas.

## Enumerações

- Classe para enumerações (como prometido =):

```
1 class Enum a where
2   succ, pred      :: a → a
3   toEnum          :: Int → a
4   fromEnum        :: a → Int
5   enumFrom        :: a → [a]      -- [n..]
6   enumFromThen    :: a → a → [a]  -- [n,n'..]
7   enumFromTo      :: a → a → [a]  -- [n..m]
8   enumFromThenTo  :: a → a → a → [a] -- [n,n'..m]

10      -- Minimal complete definition:
11      --   toEnum, fromEnum
```

## Tipos limitados

- A classe abaixo é para tipos que possuem limites inferior e superior (enumerações, alguns tipos numéricos).

```
1 class Bounded a where
2   minBound :: a
3   maxBound :: a
```

## Números em geral

- A classe **Num** é implementada por todos os tipos numéricos do Haskell:

```
1 class (Eq a, Show a) => Num a where
2   (+), (-), (*) :: a → a → a
3   negate       :: a → a
4   abs, signum  :: a → a
5   fromInteger  :: Integer → a

7      -- Minimal complete definition:
8      --   All, except negate or (-)
```

## Mais números

- Números que podem ser transformados em **Rational**:

```
1 class (Num a, Ord a) => Real a where
2   toRational :: a → Rational
```

- Números inteiros:

```
1 class (Real a, Enum a) => Integral a where
2   quot, rem      :: a → a → a
3   div, mod       :: a → a → a
4   quotRem, divMod :: a → a → (a,a)
5   toInteger      :: a → Integer

7      -- Minimal complete definition:
8      --   quotRem, toInteger
```

## Racionais

- ▶ Números que suportam divisão:

```
1 class (Num a) => Fractional a where
2   (/)          :: a -> a -> a
3   recip        :: a -> a
4   fromRational :: Rational -> a

6   -- Minimal complete definition:
7   --   fromRational and (recip or (/))
```

## Ponto-flutuante

- ▶ Números de ponto flutuante (i.e.  $\mathbb{R}$ , não apenas  $\mathbb{Q}$ ):

```
1 class (Fractional a) => Floating a where
2   pi          :: a
3   exp, log, sqrt :: a -> a
4   (**), logBase :: a -> a -> a
5   sin, cos, tan  :: a -> a
6   asin, acos, atan :: a -> a
7   sinh, cosh, tanh :: a -> a
8   asinh, acosh, atanh :: a -> a

10   -- Minimal complete definition:
11   --   pi, exp, log, sin, cos, sinh, cosh
12   --   asin, acos, atan
13   --   asinh, acosh, atanh
```

## Functors

- ▶ Classe usada para tipos que podem ser mapeados:

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

- ▶ Por exemplo, para listas o tipo da função `fmap` é igual ao tipo de `map`.

- ▶ Leis que devem ser satisfeitas:

```
1 fmap id == id
2 fmap (f . g) == fmap f . fmap g
```

- ▶ Outro tipo que instancia `Functor` é `Maybe`.

## Monads

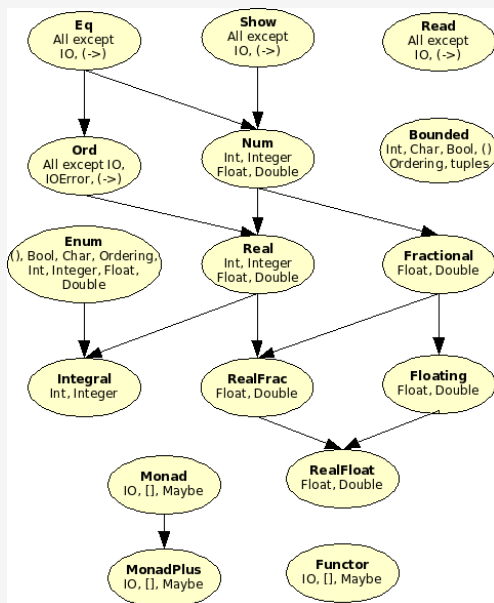
- ▶ Os monads são bem simples mas o seu uso é engenhoso. Dedicaremos uma aula só para esta classe mais tarde:

```
1 class Monad m where
2   (>=) :: m a -> (a -> m b) -> m b
3   (>>) :: m a -> m b -> m b
4   return :: a -> m a
5   fail    :: String -> m a

7   -- Minimal complete definition:
8   --   (>=), return
```

- ▶ Também há leis para monads, mas não falaremos agora.

## Diagrama pego do Haskell 98 Report



## Assuntos de hoje

### Classes

“O quê?” e “como?”

Haskell 98

Usando as classes

### Finalizando

Acabou...

## Classes? Funções

- ▶ É simples usar uma classe. Basta utilizar uma das funções que ela implementa. Por exemplo,

```
1 plus x y = x + y
```

- ▶ Aqui nós usamos (+) que está em **Num**.

- ▶ O tipo de plus então será

```
1 plus :: (Num a) => a -> a -> a
```

## Múltiplas classes

- ▶ E para a função **media** abaixo?

```
1 media a b = (a + b) / 2
```

- ▶ Usamos (+) e (/), que são, respectivamente, das classes **Num** e **Fractional**.

- ▶ Podemos então escrever

```
1 media :: (Num a, Fractional a) => a -> a -> a
```

- ▶ Todavia observe que **Fractional** herda de **Num**, portanto

```
1 media :: Fractional a => a -> a -> a
```

- ▶ As duas definições estão corretas, porém a 2ª é melhor.



## Múltiplas classes II

- ▶ Não existe limite para o uso de classes:

```
1 foo :: (Num a, Show a, Show b) => a -> a -> b -> String
2 foo x y t = show x ++ "_plus_" ++ show y ++ "_is_" ++
3             show (x+y) ++ "__" ++ show t
```

## Assuntos de hoje

### Classes

“O quê?” e “como?”

Haskell 98

Usando as classes

### Finalizando

Acabou...

## Pratiquem, pessoal!

- ▶ Hoje a aula foi curta e grossa: vimos muita coisa em poucos slides. Isso significa que vocês podem acabar esquecendo de pontos importantes!
- ▶ É interessante praticar, então.
  - ▶ Crie desafios e tente resolvê-los.
  - ▶ Reimplemente um algoritmo que você já conhece.
  - ▶ Faça um trabalho de CIC em Haskell =).

## E estudem também

- ▶ Agora não há nada a temer. Vocês já sabem tudo que é necessário para entender o **Prelude**.
- ▶ Não é necessário decorar, mas *no mínimo* leia-o.
- ▶ O ideal seria entender cada função e tentar usá-la no interpretador algumas vezes. Quem fizer isso será recompensado (com um grande conhecimento).
- ▶ Link: <http://haskell.org/ghc/docs/latest/html/libraries/base-3.0.0.0/Prelude.html> (clicável no PDF — não copiem hehe)