INE 5416/5636 - Paradigmas de programação

Turmas 04208/08238

Prof. Dr. João Dovicchi – dovicchi@inf.ufsc.br

http://www.inf.ufsc.br/˜dovicchi

Classes

Polimorfismo paramétrico: tipos estáticos com abrangência genérica.

Funções ou dados podem ser definidos para lidar com valores semelhantes independente do tipo. Exemplo:

```
juntalst :: [a] -> [a] -> [a]
juntalst x y = x ++ y
```

Nota: x, y e a saída devem ter o mesmo tipo.

Classes

Polimorfismo *ad hoc*:

- caracteres $[0-9]$ são usados para representar números de precisão fixa ou arbitrária
- caracteres de sinais (`*`, `&`, `$` ou +, - etc.) representam operadores para diversos tipos de números ou caracteres.
- a igualdade funciona para números e diversos outros (nem todos) tipos.

Enquanto o polimorfismo *ad hoc* depende mas não define tipos ou erros, no polimorfismo paramétrico o tipo é realmente abstraído.

Classes

HASKELL define classes para lidar com polimorfismo.

Classes

HASKELL define classes para lidar com polimorfismo.

C++ define classes que implementam métodos e interfaces.

Classes

HASKELL define classes para lidar com polimorfismo.

C++ define classes que implementam métodos e interfaces.

As classes tipadas do HASKELL são mais como interfaces genéricas.

Classes

HASKELL define classes para lidar com polimorfismo.

Polimorfismo *ad hoc* (*overloading*)

$\Downarrow$

Polimorfismo paramétrico

Em HASKELL , classes tipadas permitem uma forma estruturada de controlar o polimorfismo *ad hoc*.

Classes

Por exemplo, considere a função `elem`

```
x `elem`  []            = False
x `elem` (y:ys)         = x==y || (x `elem` ys)
```

Deveria ser tipada como: `elem :: a->[a]->Bool`, bem como o operador `(==)` deveria ser `(==) :: a->a->Bool`.

`(==)` $\rightarrow$ sobre qualquer tipo (não é conveniente)

Classes

Por exemplo, considere a função `elem`

```
x `elem`  []          = False
x `elem` (y:ys)       = x==y || (x `elem` ys)
```

Uma restrição pode resolver o problema:

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

"Para todo tipo `a` que seja uma instância da classe `Eq`, `(==)` é do tipo `a->a->Bool`"

```
elem :: (Eq a) => a->[a]->Bool
```

"elem" não é definido para todos os tipos, mas apenas para aqueles que podem ser comparados com relação à igualdade.

Type Classes

Classes em Haskell vs. Classes OOP

- construtor de tipos do sistema
- suporte ao polimorfismo paramérico

Type Classes

Classes em Haskell vs. Classes OOP

- construtor de tipos do sistema
- suporte ao polimorfismo paramérico

Uma variável **v** descrita por uma classe **C** significa que a variável somente pode ser instanciada por um tipo, cujos membros suportam operações associadas a C.

Type Classes

Classes em Haskell vs. Classes OOP

- construtor de tipos do sistema
- suporte ao polimorfismo paramérico

Uma variável **v** descrita por uma classe **C** significa que a variável somente pode ser instanciada por um tipo, cujos membros suportam operações associadas a C.

Atributos e métodos em “pure” Haskell não definem objetos. (ver O’Haskell para definição de classes associadas a objetos).

Declaração de Classes

Forma geral:

```
class <nome> a => <novo_nome> a where
   declaracao 1 [... declaracao N]
```

Exemplo:

```
class (Ord a, Num a) => Real a where
   toRational :: a -> Rational
```

Type Classes

Definem métodos ou conj. operações sobre tipos

Um tipo pode ser uma instância de uma classe e herdar o método de cada operação sobre ele.

Type Classes

Definem métodos ou conj. operações sobre tipos

Um tipo pode ser uma instância de uma classe e herdar o método de cada operação sobre ele.

São organizadas de forma hierárquica

Idéias de superclasses e subclasses que transmitem a herança de métodos.

Type Classes

Definem métodos ou conj. operações sobre tipos

Um tipo pode ser uma instância de uma classe e herdar o método de cada operação sobre ele.

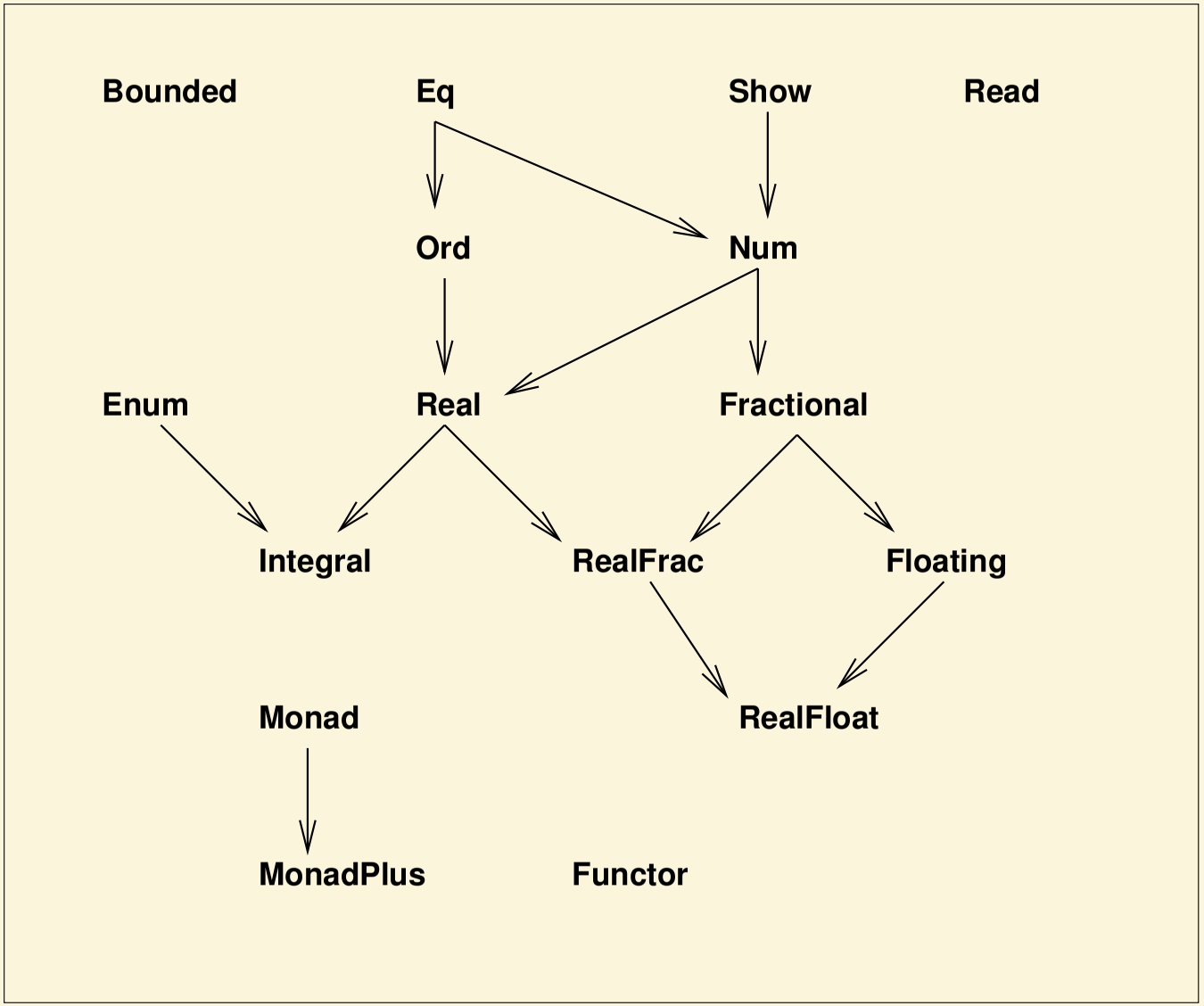
São organizadas de forma hierárquica

Idéias de superclasses e subclasses que transmitem a herança de métodos.

Não definem objetos

Haskell não possui a noção de objeto ou de alteração de estado. Isto torna os métodos mais seguros por ser fortemente tipados. São checadas pelo compilador Qualquer tentativa de aplicar um método a um valor que não pertence à classe é detectado pelo compilador.

Type Classes: hierarquia



Type Classes

Classes primitivas

Bounded: estabelece limites mínimos e máximos para instanciar tipos: `Int, Char, Bool, (), Ordering` e n-uplas.

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

Type Classes

Classes primitivas

Eq: trata os métodos de igualdade e desigualdade, instanciando todos os tipos, exceto `IO` e funções

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

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

Type Classes

Classes primitivas

Enum: trata os métodos de enumerabilidade e instancia os tipos: `()`, `Bool`, `Char`, `Int`, `Integer`, `Ordering`, `Float` e `Double`.

```
class Enum a where
   succ, pred :: a -> a
   toEnum     :: Int -> a
   fromEnum   :: a -> Int
   enumFrom   :: a -> [a]
   enumFromThen   :: a -> a -> [a]
   enumFromTo     :: a -> a -> [a]
   enumFromThenTo :: a -> a -> a -> [a]
   ... etc
```

Type Classes

Classes primitivas

Show e **Read**: definem os métodos de conversão de valores para caracteres e vice-versa. Instanciam todos os tipos, exceto `IO` e funções.

Type Classes

Classes primitivas

Show e **Read**: definem os métodos de conversão de valores para caracteres e vice-versa. Instanciam todos os tipos, exceto `IO` e funções.

Monad: define os métodos de operação sobre mônadas e instancia os tipos `IO`, `Maybe` e `[ ]`.

> Mônadas são entidades matemáticas definidas por funções homológicas e será estudada à parte no capítulo sobre IO.

Functor: define os métodos sobre tipos que podem ser mapeados, e instancia `[ ]`, `IO` e `Maybe`.

Type Classes

Classes secundárias

Ord: define os métodos para tipos de dados totalmente ordenáveis. Instancia todos os tipos, exceto funções, `IO` e `IOError`.

```
class (Eq a) => Ord a where
    compare                  :: a -> a -> Ordering
    (<), (<=), (>=), (>)     :: a -> a -> Bool
    max, min                 :: a -> a -> a
    compare x y | x == y      = EQ
                | x <= y      = LT
                | otherwise   = GT
    x <= y  = compare x  y  /=  GT
    x <  y  = compare x  y  ==  LT
    x >= y  = compare x  y  /=  LT
    x >  y  = compare x  y  ==  GT
    -- Note that (min x y,    max x y) = (x,y) or (y,x)
    max x y | x <= y      =   y
            | otherwise   =   x
    min x y | x <= y      =   x
            | otherwise   =   y
```

Type Classes

Classes secundárias

Num: define os métodos para operações com números e instancia os tipos `Int`, `Integer`, `Float` e `Double`.

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)     :: a -> a -> a
    negate            :: a -> a
    abs, signum       :: a -> a
    fromInteger       :: Integer -> a
        -- Minimal complete definition:
        --       All, except negate or (-)
    x - y             = x + negate y
    negate x          = 0 - x
```

Type Classes

Classes secundárias

Real, Integral, Fractional, Floating, RealFrac e **RealFloat**: definem os métodos numéricos de operações. Real instancia `Int`, `Integer`, `Float` e `Double`, enquanto as outras classes instanciam apenas `Float` e `Double`.

Tipos de dados

Booleanos (`Bool`), Caracteres (`Char, String`), Numéricos (Int, Integer, Float e Double), algébricos (n-uplas) e abstratos (funções, `Maybe, Functor` etc.).

`String` equivale a `[Char]` e o contrutor nulo é uma n-upla vazia, representadopor `()`.

Tipos de dados

Booleanos (`Bool`): `False | True`

Caracteres (`Char`): 16 bits de representação Unicode

Numéricos:

- `Int`: inteiros de -229 a (229 -1)
- `Integer`: Inteiros de precisão arbitrária
- `Float`: número real de precisão simples (32 bits)
- `Double`: número real de precisão dupla (64 bits)

Tipos de dados

Algébricos: coleção de valores organizados, tais como vetores, matrizes, duplas, triplas, n-uplas (ordenadas e não ordenadas).

Tipos de dados

Algébricos: coleção de valores organizados, tais como vetores, matrizes, duplas, triplas, n-uplas (ordenadas e não ordenadas).

Listas: tipos algébricos formados de 2 construtores `:` e `[ ]`.

Tuplas: tipos algébricos formados de 2 construtores `,` e `( )`.

tuplas podem ser de qualquer tamanho (geralmente até 15 em algumas implementações do Haskell).

Tipos de dados

Algébricos: coleção de valores organizados, tais como vetores, matrizes, duplas, triplas, n-uplas (ordenadas e não ordenadas).

Listas: tipos algébricos formados de 2 construtores `:` e `[ ]`.

Tuplas: tipos algébricos formados de 2 construtores `,` e `( )`.

tuplas podem ser de qualquer tamanho (geralmente até 15 em algumas implementações do Haskell).

Funções são tipos abstratos de dados, bem como `Maybe`, `Either`, `Ordering` e `( )`.