sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Funções Polimórficas

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação

UFSJ
Universidade Federal de São João del-Rei

Introdução

As funções do Haskell podem ter um cabeçalho para indicar o tipo dos parâmetros de entrada e saída.

Além de limitar e as possíveis entradas e a saída, o cabeçalho das funções também serve como documentação, uma vez que deixa explícito a interface das funções.

Por exemplo, a função:

```
divisao :: Int -> Int -> Int
divisao x y = div x y
```

Deixa claro qual é a interface de entrada e saída da função.

Introdução

Mas o Haskell também permite que as funções sejam implementadas sem a definição de um cabeçalho.

Nesse caso, o Haskell realiza a **inferência de tipos**, ou seja, identifica automaticamente qual o tipo dos parâmetros de entrada e saída com base nas operações que são realizadas nela.

Por exemplo, se a função `divisao` for implementada somente como:

```
divisao x y = div x y
```

O Haskell identifica que a, b e a saída da função são do tipo Int porque a operação div só aceita como entradas o tipo Int e também retorna esse tipo.

Introdução

Em algumas situações, é interessante que uma função aceite mais de um tipo para cada um dos seus parâmetros de entrada e saída, para evitar que ela precise ser implementada repetidas vezes.

Por exemplo, ao invés de :

```
somai :: Int -> Int -> Int
somai x y = x + y
somaf :: Float -> Float -> Float
somaf x y = x + y
```

Por exemplo, pode-se implementar somente:

```
soma x y = x + y
```

Pois o operador + permite qualquer tipo numérico.

Introdução

Polimorfismo é a capacidade de uma função se comportar de forma diferente a depender do contexto.

Funções polimórficas são aquelas que permitem mais de um tipo para seus parâmetros.

Elas não devem ser confundidas com **funções sobrecarregadas**, que consistem em duas ou mais funções com o mesmo nome, implementadas dentro da mesma unidade de código, porém com entradas e/ou saídas diferentes.

Funções Polimórficas

Como já foi visto, uma função implementada sem cabeçalho também pode ser polimórfica, desde que suas operações internas sejam.

A função soma é polimórfica porque o operador + aceita qualquer tipo numérico.

```
soma x y = x + y
```

```
> soma 1 4
5
> soma 1.5 6.2
7.7
```

Funções Polimórficas

Mas se a intenção é criar funções polimórficas, a melhor forma é definir seus cabeçalhos usando **tipos polimórficos**.

Por exemplo, a função soma pode ser implementada como:

```
soma :: a -> a -> a
soma x y = x + y
```

Nessa função, a é uma **variável de tipo**, ou seja, uma variável que representa um tipo, enquanto x e y são variáveis que representam valores.

Como x e y possuem o mesmo tipo e a saída da função também, então usa-se a mesma variável em todos os parâmetros.

Funções Polimórficas

```
tamanho :: [a] -> Int
tamanho [] = 0
tamanho (_:r) = 1 + tamanho r

inverte :: [a] -> [a]
inverte [] = []
inverte (x:r) = inverte r ++ [x]

mymap :: (a -> b) -> [a] -> [b]
mymap _ [] = []
mymap f (x:r) = f x : mymap f r

myfst :: (a,b) -> a
myfst (x,_) = x
```

Tipo Maybe

Haskell não tem um valor nulo, como `null` de Java e `NULL` de C/C++.

Então, o que fazer quando uma função não deve retornar valor algum dada uma determinada entrada?

Por exemplo:

```
fatorial :: Int -> Int
fatorial n = product [1..n]
```

Como definir que a função fatorial não deve retornar nada quando recebe um número negativo?

Tipo Maybe

Para situações como essa, Haskell oferece o tipo **Maybe**.

Esse tipo **sempre** é usado junto com outro tipo e **normalmente** como parâmetro de saída das funções.

Uma função que possui o tipo Maybe como saída pode retornar nada (`Nothing`) ou simplesmente (`Just`) um valor.

```
fatorial :: Int -> Maybe Int
fatorial n =
  | n < 0 = Nothing
  | otherwise = Just (product [1..n])
```

Tipo Maybe

Cuidado que um valor do tipo `Maybe x` não é compatível com um valor do tipo `x`.

Por exemplo, a expressão `6 * (fatorial 5)` resulta em erro porque não é possível multiplicar 6, que é do tipo `Int`, com o resultado de `fatorial 5`, que é do tipo `Maybe Int`.

Isso pode ser resolvido com a função **`fromJust`**, fornecida pela biblioteca **`Data.Maybe`**.

```
fatorial :: Int -> Maybe Int
fatorial 0 = Just 1
fatorial n
  | n < 0 = Nothing
  | otherwise = Just (n * fromJust (fatorial (n-1)))
```

Tipo Maybe

Outras funções da biblioteca **Data.Maybe**:

- **fromMaybe**: recebe um valor do tipo x e um valor do tipo Maybe x. Retorna o primeiro valor se o segundo for Nothing. Caso contrário, retorna o equivalente a fromJust do segundo valor.

```
> fromMaybe 0 (Just 15)
15
```

- **isJust** e **isNothing**: retornam True se o valor passado for, respectivamente, Just x ou Nothing.

```
> isJust (Just 2)
True
```

- **catMaybes**: recebe uma lista do tipo Maybe x e retorna uma lista do tipo x para todo valor diferente de Nothing da lista original.

```
> catMaybes [Just 1, Nothing, Nothing, Just 3]
[1,3]
```

- **mapMaybe**: semelhante a map, mas recebe uma função `x -> Maybe x`.

```
> mapMaybe fatorial [3,-4,5]  --fatorial -4 dá Nothing
[6, 120]                      --então é ignorado
```

Introdução às Classes de Tipos

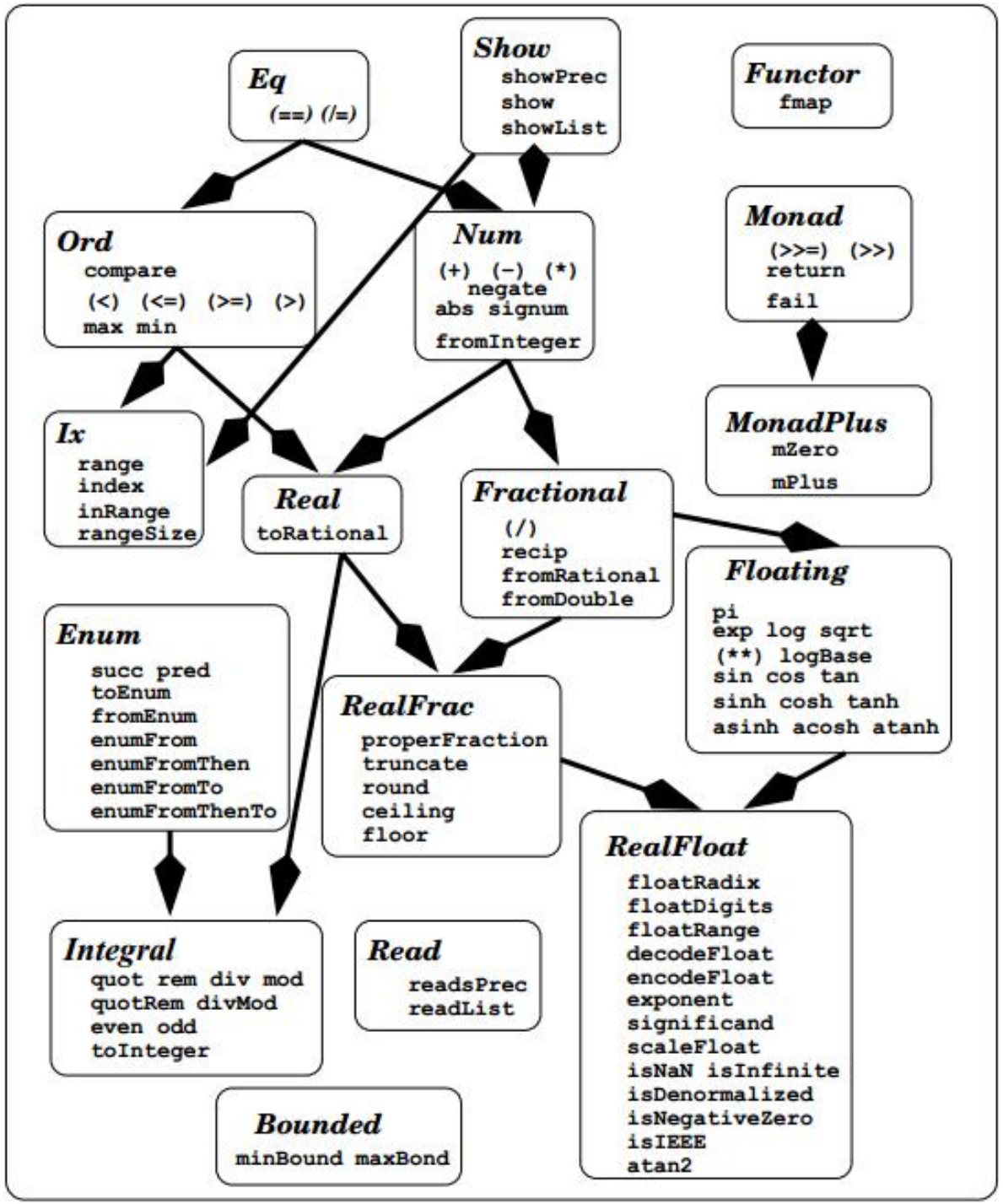
Uma **classe de tipos (*typeclass*)** é uma interface que define um comportamento.

Uma classe de tipos não é um tipo, mas todo tipo faz parte de pelo menos uma classe de tipos.

Se um tipo faz parte de uma classe de tipos, quer dizer que esse tipo suporta e implementa o comportamento especificado por essa classe de tipos.

Esse conceito é parecido com o de interface de Java, mas apesar do nome e outras semelhanças, não tem relação com a orientação a objetos.

Classes de Tipos Predefinidas



Introdução às Classes de Tipos

Classe de Tipos	Descrição	Principais Operadores
`Bounded`	Tipos que possuem um limite inferior e superior.	`maxBound minBound`
`Enum`	Tipos que possuem uma sequência.	`succ pred`
`Eq`	Tipos que suportam teste por igualdade.	`==  /=`
`Fractional`	Tipos fracionários.	`/`
`Integral`	Tipos inteiros.	`quot div mod rem`
`Num`	Tipos numéricos.	`+ - * abs signum negate`
`Ord`	Tipos ordenados.	`> >= < <=`
`Read`	Tipos que podem ser obtidos a partir de String.	`read`
`Show`	Tipos que podem ser transformados em String.	`show`

Introdução às Classes de Tipos

A inferência de tipos faz uso das classes de tipos para inferir os tipos das funções.

Por exemplo, quando não possui cabeçalho, a função soma aceita como entrada e saída qualquer tipo numérico, ou seja, Haskell infere que o tipo dos parâmetros dessa função é a classe de tipos **`Num`**.

```
soma x y = x + y
```

Isso ocorre porque o operador + foi definido na classe `Num`.

Introdução às Classes de Tipos

Se a função realizar operações de tipos diferentes, a inferência de tipos escolhe o tipo mais restrito.

Por exemplo, na função `fun`, o operador + pertence à classe `Num` e o operador / pertence à classe Fractional, que é mais restrita. Portanto, Haskell infere a função fun pode ser usada com qualquer tipo pertencente à classe Fractional.

```
Prelude> let fun x y = (x + y) / y
Prelude> :t fun
fun :: Fractional a => a -> a -> a
Prelude>
```

Tipos Qualificados

As classes de tipos são úteis para limitar a faixa de tipos válidos para uma função polimórfica.

O operador `=>` é usado no cabeçalho das funções para indicar que um tipo polimórfico está limitado a uma determinada classe de tipo.

Quando isso ocorre, o tipo polimórfico da função passa a ser chamado de **tipo qualificado**.

Por exemplo, a função `divisao` deve funcionar para todos os tipos inteiros, mas não os tipos fracionários.

```
divisao :: Integral a => a -> a -> a
divisao x y = div x y
```

Sobrecarga de Funções

Ocorre quando duas ou mais funções na mesma unidade de código possuem o mesmo identificador, mas com entrada/saída distintas.

Também é chamado de **polimorfismo ad hoc**.

Haskell não aceita ter mais de uma função com mesmo nome na mesma unidade de código. Por exemplo, o código abaixo gera o erro indicado em vermelho.

```
soma :: Int -> Int -> Int
soma x y = x + y

soma :: Float -> Float -> Float
soma x y = x + y
```

```
classesdetipos.hs:8:1:
    Duplicate type signatures for ‘soma’
    at classesdetipos.hs:5:1-4
       classesdetipos.hs:8:1-4

classesdetipos.hs:9:1:
    Multiple declarations of ‘soma’
    Declared at: classesdetipos.hs:6:1
                 classesdetipos.hs:9:1
```

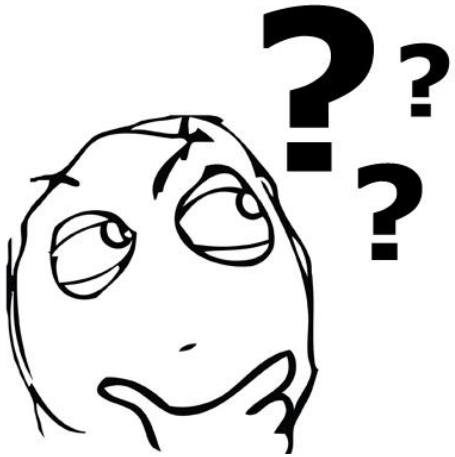
Sobrecarga de Funções

Contudo, a sobrecarga de funções é simulada por meio das funções polimórficas.

Por exemplo, ao invés de ter várias versões, o operador + é implementado usando a classe de tipos `Num`.

```
(+) :: Num a => a -> a -> a
```

Qual é a principal vantagem e a principal desvantagem de ter funções polimórficas e não sobrecarga de funções?



Exercícios

1. Crie funções que fazem a mesma coisa que as funções take e drop usando tipos polimórficos.

2. Crie uma função que faz a mesma coisa que o operador ++ usando tipos polimórficos.

3. Crie uma função que faz a mesma coisa que a função last usando tipos polimórficos e retornando Nothing caso receba uma lista vazia.

4. Crie uma função que recebe uma lista e retorne o menor elemento. Use tipos qualificados e trate a possibilidade da lista ser vazia.

5. Crie uma função que implementa o algoritmo bubblesort sobre uma lista de qualquer tipo ordenado.