Polimorfismo e Classes de Tipos

Inferência de tipos

- Tudo em Haskell tem tipo
- Quando não é explicitado o tipo, Haskell infere
  - É uma boa prática explicitar os tipos

Funções monomórficas

```
upper c
  | 'a' <= c && c <= 'z' = chr (ord 'A' + ord c - ord 'a')
  | otherwise = c
```

- O argumento de `upper` deve ser um caractere, pois senão a comparação `'a' <= c` não poderia ser aplicada
- Dizemos que `upper` é uma função monomórfica, pois ela se aplica somente a um único tipo

```
upper :: Char -> Char
upper c
  | 'a' <= c && c <= 'z' = chr (ord 'A' + ord c - ord 'a')
  | otherwise = c
```

Polimorfismo paramêtrico

```
fst (x, y) = x
```

- Não há nenhuma restrição acerca do tipo das componentes da dupla
- Definição genérica
  - A função atua uniformemente sobre uma família de tipos
- A função é polimórfica
  - O tipo da função é o poli-tipo

    `(t,r) -> t`
  - onde `t` e `r` são variáveis de tipo
- `fst` se aplica a valores cujo tipo tem a forma `(t,r)` e retorna um valor de tipo `t`, onde `t` e `r` são tipos quaisquer

```
fst :: (t, r) -> t
fst (x, y) = x
```

```
fst (3,6) ↝ 3
fst ("ab", 1) ↝ "ab"
fst ((3,'a'), "abcd") ↝ (3,'a')
```

- Temos uma única definição que atua sobre uma família de tipos
- Polimorfismo permite “genericidade”

Exemplos

```
snd :: (t,r) -> r
snd (a,b) = b

identity :: t -> t
identity x = x
```

Quais são os tipos das seguintes funções:

```
head (x:xs) = x

tail (x:xs) = xs
```

```
head :: [t] -> t
head (x:xs) = x

tail :: [t] -> [t]
tail (x:xs) = xs
```

Quais são os tipos mais gerais das seguintes funções:

```
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

drop 0 xs = xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs

length [] = 0
length (x:xs) = 1 + length xs

zip [] ps = []
zip ps [] = []
zip (p:ps) (q:qs) = (p, q) : zip ps qs
```

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

zip :: ...
zip [] ps = []
zip ps [] = []
zip (p:ps) (q:qs) = (p, q) : zip ps qs
```

Sobrecarga

- Um mesmo nome para diferentes entidade

  O nome `(+)` usamos para a soma de `Int`, `Integer`, `Float`, `Double`, ...

  O nome `div` para a divisão inteira de números `Int` e números `Integer`

- Sozinha, é uma mera notação amigável
- Pré-definida e definida pelo usuário
  - Tradicionalmente linguagens só tinham sobrecarga de operadores pré-definidos
- Excesso e mau uso de sobrecarga pode comprometer a

legibilidade dos programas

Sobrecarga vs Polimorfismo

- Sobrecarga
  - Um mesmo identificador denotando diferentes entidade
    `(==)`, `show`, `(+)`, `(<)`, ...

- Polimorfismo
  - Uma única definição opera sobre uma família de tipos

```
fst :: (t,r) -> t
head :: [t] -> t
length :: [t] -> Int
```

Classes de Tipos

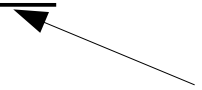
```
elem x [] = False
elem x (y:ys) = x == y || elem x ys
```

- Qual é o tipo de elem?

  `elem :: a -> [a] -> Bool` ?

- A definição requer que o tipo a permita comparação por igualdade `(==)`
- O operador `(==)` não opera em funções, por exemplo
- Precisamos dizer que a variável de tipo a só pode ser instanciada por
- tipos que suportam igualdade `(==)`

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



- `Eq` é a classe de tipos que suporta a operação `(==)` e `(/=)`

Classes e polimorfismo de sobrecarga

```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = x == y || elem x ys
```

- Numa chamada a elem, qual é a função `(==)` que será usada?
  - Depende de qual é o tipo que está sendo instanciado para a
  - Só podemos saber na hora da chamada

- Dynamic binding: resolução do vínculo é feito em tempo de execução

```
elem 3 [2,5]      -- usa (==) de Int
elem 'c' "abcd"   -- usa (==) de Char
```

- Polimorfismo de sobrecarga

Classes de Tipos

- Uma classe de tipos é uma coleção de tipos
  - caracterizada por uma assinatura:

    conjunto de “nomes” de funções/operações com seus tipos

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

  - Cada instância (um tipo) da classe implementa suas próprias funções/operações

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

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

Classes podem ter definições *default*

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

Neste caso, as instâncias só tem a obrigação de dar uma definição para `(==)`

A definição de `(/=)` é opcional

Também é possível

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

Neste caso, as instâncias só tem a obrigação de dar uma definição para `(==)` ou uma para `(/=)`

Também, uma instância pode prover definições para ambos operadores

Instâncias com contextos

Pares do tipo `(a, b)` podem ser comparados por `(==)` caso `a` e `b` sejam instâncias de `Eq`

```
instance Eq a, Eq b ==> Eq (a, b) where
  (==) :: a -> a -> Bool
  (x1, x2) == (y1, y2) = x1 == y1 && x2 == y2
```

Classes derivadas

Uma classe pode depender de outra

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

Podemos interpretar que a Classe `Ord` herda a assinatura de `Eq`

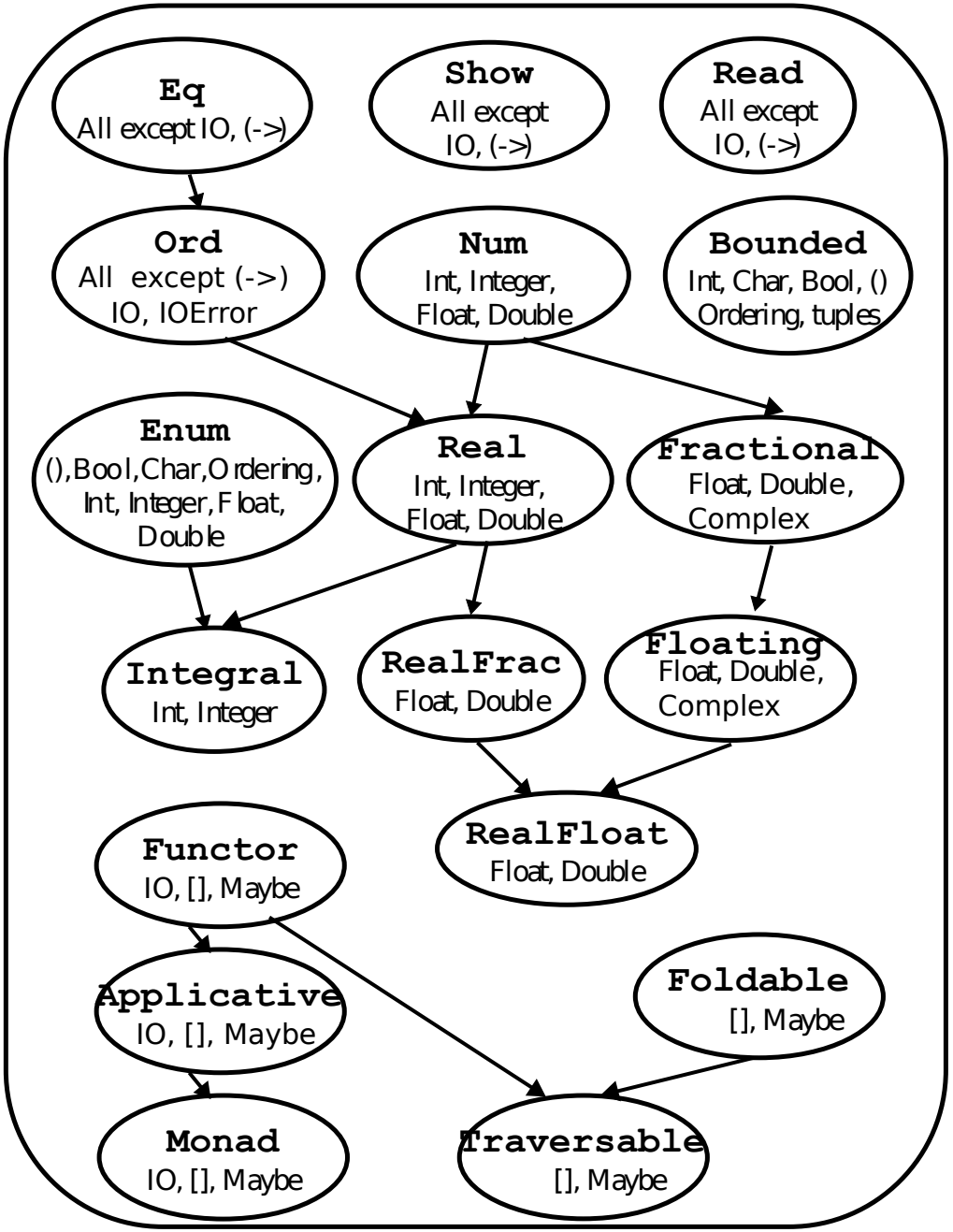
Quais são os tipos mais gerais de

```
iSort

qSort
```

```
iSort :: Ord [a] => [a] -> [a]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)

ins :: Ord [a] => a -> [a] -> [a]
ins x [] = [x]
ins x (y:ys)
  | x < y      = x : y : ys
  | otherwise = y : ins x ys
```

```
ins x [] = [x]
ins x zs@(y:ys)
  | x < y      = x : zs
  | otherwise = y : ins x ys
```

Classes pré-definidas



Qual é o tipo mais geral de

```
pares :: Integral a => [a] -> [a]
pares xs = [ x | x <- xs, mod x 2 == 0 ]
```

Exercícios

- Considere a seguinte função

```
shift ((x, y), z) = (x, (y, z))
```

  Qual é o seu tipo mais geral?

- Considere a seguinte função

```
zip' [] ps = []
zip' ps [] = []
zip' (p:ps) (q:qs) = (p, q) : zip' qs ps
```

  O que ela calcula? Qual é seu tipo mais geral?

- Defina uma função `numEqual` que pegue uma lista `xs` de items e um item `x` e retorne o número de vezes que `x` ocorre dentro de `xs`. Qual é o tipo da sua função? Como poderia usar `numEqual` para definir `elem`?

- Defina a função `oneLookupFirst` que pega uma lista de pares e um item. Digamos que o tipo dos pares é `(a, b)`, e que o tipo do item é `a`. A função retorna a segunda componente do primeiro par cuja primeira componente é igual ao item. Qual é o tipo mais geral da função?
  Defina a função `oneLookupSecond` que retorna a primeira componente do primeiro par cuja segunda componente é igual ao item. Qual é o tipo mais geral da função?

- Considere a seguinte função

  ```
  misterio y x = [ show z | z <- x, elem z y ]
  ```

  Qual é eu seu tipo mais geral?