4.2 Classes

Em *Haskell* é possível definir novos tipos e inseri-los na estrutura de classe de tipos. Desta forma é possível obter funções para comparar valores dos novos tipos, funções para visualizar os novos valores, entre outras.

A estrutura de classe do *Haskell* dá conta das propriedades comuns aos diferentes tipos; por exemplo se os tipos admitem a comparação entre os seus membros, se admitem uma ordenação dos seus membros, entre outras.

Na figura 4.1 apresenta-se a estrutura de classes do *Haskell*.

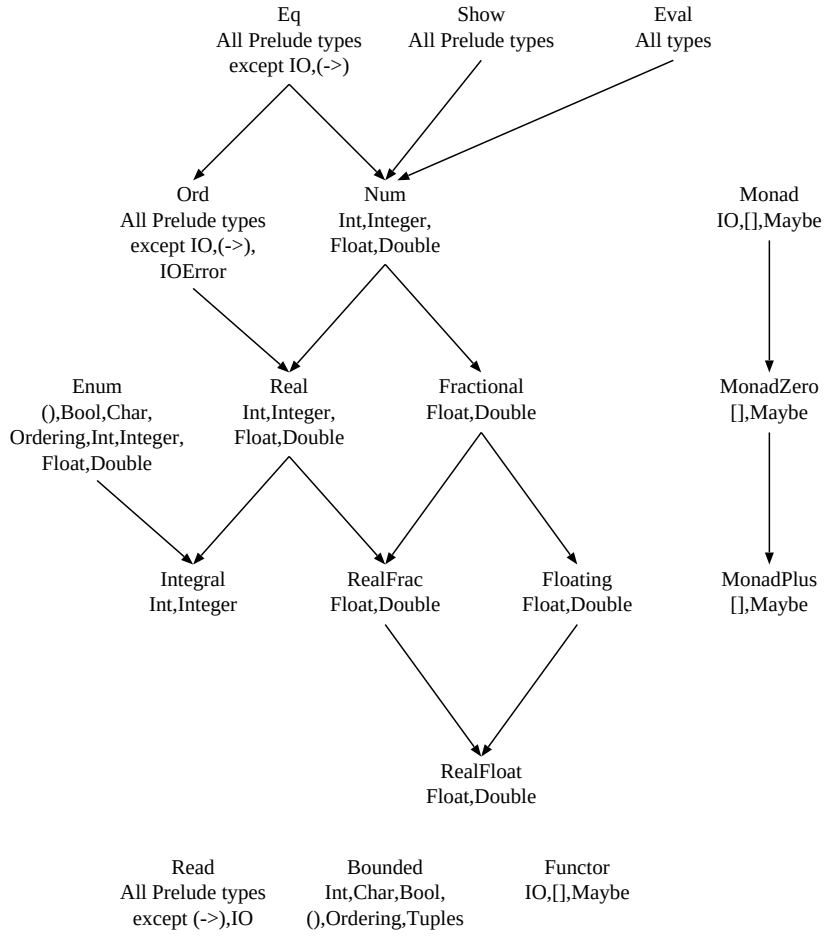

Figura 4.1: Classes Haskell

Tomando o exemplo do tipo `Triângulo`, vejamos como incorporar este novo tipo na estrutura já existente.

Pretende-se ter acesso às funções de visualização e de comparação.

A classe dos tipos para os quais é possível testar a igualdade entre os seus elementos é a classe `Eq`. Para incorporar o novo tipo nesta classe é necessário criar uma nova instância da classe e definir explicitamente o operador de igualdade para os elementos do novo tipo.

```
instance Eq Triangulo where
     NaoETriangulo == NaoETriangulo = True
     NaoETriangulo == Escaleno = False
```

```
NaoETriangulo == Isoscele = False
NaoETriangulo == Equilatero = False
Escaleno == NaoETriangulo = False
Escaleno == Escaleno = True
Escaleno == Isoscele = False
Escaleno == Equilatero = False
Isoscele == NaoETriangulo = False
Isoscele == Escaleno = False
Isoscele == Isoscele = True
Isoscele == Equilatero = False
Equilatero == NaoETriangulo = False
Equilatero == Escaleno = False
Equilatero == Isoscele = False
Equilatero == Equilatero = True
```

A definição da relação `=/=`, não é necessário dado que esta é definida na classe `Eq` à custa da relação `==`. A definição da relação de igualdade desta forma é fastidiosa e, para tipos com mais elementos, rapidamente se torna impraticável.

Podemos usar a estrutura de classes e os operadores definidos nas diferentes classes como forma de obter uma definição mais compacta. Para tal vamos começar por estabelecer o tipo `Triângulo` como uma instância da classe `Enum`, a classe dos tipos enumeráveis.

```
instance Enum Triangulo where
  fromEnum NaoETriangulo = 0
  fromEnum Escaleno = 1
  fromEnum Isoscele = 2
  fromEnum Equilatero = 3
  toEnum 0 = NaoETriangulo
  toEnum 1 = Escaleno
  toEnum 2 = Isoscele
  toEnum 3 = Equilatero
```

Podemos agora aproveitar o facto de o tipo `Int` ser uma instância das classes `Eq` e `Ord` (entre outras), e as definições de `fromEnum` e de `toEnum`, para definir o tipo `Triângulo` como uma instância das classe `Eq` e `Ord` de uma forma simplificada.

```
instance Eq Triangulo where
  (x == y) = (fromEnum x == fromEnum y)

instance Ord Triangulo where
  (x < y) = (fromEnum x < fromEnum y)
```

No entanto, e dado que todas estas definições são óbvias, seria de esperar que o *Haskell* providenciasse uma forma automática de estabelecer estas instância de um tipo definido por enumeração. Felizmente tal é possível através da seguinte definição definição do tipo `Triângulo`.

```
data Triangulo = NaoETriangulo | Escaleno | Isoscele | Equilatero
  deriving (Eq,Enum,Ord,Show)
```

Com esta definição são criadas, automaticamente, instâncias do novo tipo nas classes `Eq`, `Enum`, `Ord` e `Show`.

Podemos então avaliar as seguintes expressões:

```
Main> classificar(4,4,4)
```

```
Equilatero
Main> classificar(4,4,4) == classificar(2,2,5)
False
```

4.2.1 Classes Básicas

Vejamos de seguida, mais em pormenor, as classes de base da estrutura de classes do *Haskell*.

Tipos que Admitem Igualdade `Eq`

A classe `Eq` contém todos os tipos cujos elementos podem ser comparados em termos da sua igualdade. A classe possuí os dois métodos seguintes:

```
(==) :: a → a → Bool
```

É de notar que a classe das funções não é uma instância desta classe dado não ser, em geral, possível comparar duas funções em termos da sua igualdade.

Tipos Ordenados

A classe `Ord` é a classe que contém todos os tipos sobre os quais é possível definir uma função de ordenação total. Os tipos nesta classe admitem as funções de comparação usuais:

```
(<) :: a → a → Bool
(≤) :: a → a → Bool
(>) :: a → a → Bool
(≥) :: a → a → Bool
min :: a → a → a
max :: a → a → a
```

Todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, e `Float` são instâncias da classe `Ord`, assim como as listas e os tuplos, sempre que os seus elementos constituintes estejam em classes que sejam instâncias desta classe.

Tipos Visualizáveis e Passíveis de Leitura

A classe `Show` contém todos os tipos para as quais é possível converter os seus elementos numa dada "string" para posterior visualização, ou impressão.

Esta classe contém o seguinte método:

```
show :: a → String
```

A classe `Read` é dual da classe `Show` no sentido que contém todos os tipos para os quais é possível converter uma dada "string" num elemento válido do tipo. A classe possuí o método:

```
read :: String → a
```

Todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, e `Float` são instâncias das classes `Show` e `Read`, assim como as listas e os tuplos, sempre que os seus elementos constituintes estejam em classes que sejam instâncias destas duas classes.

Tipos Numéricos

As classes `Num`, `Integral` e `Fraccional` contém todos os tipos numéricos. A classe `Num` contém os métodos:

```
(+) :: a → a → → a
(-) :: a → a → → a
(*) :: a → a → → a
negate :: a → a
abs :: a → a
signum :: a → a
```

O método `negate` devolve o valor negativo de um dado número, o método `abs`, o seu valor absoluto, e `signum` o seu sinal: 1 para positivo, -1 para negativo.

A operação de divisão não está presente dado ser necessário distinguir entre divisão de números inteiros e divisão de números reais, as quais vão ser implementadas em duas classes distintas.

A classe `Integral` contém todos os métodos da classe `Num` (ver Figura 4.1), definindo ainda os seguintes métodos:

```
div :: a → a → a
mod :: a → a → a
```

os quais definem as operações usuais de divisão inteira e do resto da divisão inteira.

A classe `Fraccional`, está numa posição distinta da classe `Integral` no grafo das classes Haskell (Figura 4.1) estando no entanto abaixo da classe `Num` herdando desta todos os seus métodos e extendendo-a com os seguintes métodos:

```
(/) :: a → a → a
recip :: a → a
```

O tipo `Float` é uma instância da classe `Fraccional`. O método `/` define a operação de divisão usual e o método `recip` define o recíproco de um número fraccional.

4.3 Definição Paramétrica

A definição de novos tipos passa não só pela construção de co-produtos com construtores 0-ários, mas também pela construção de co-produtos, produtos e exponenciações com construtores n-ários. Por exemplo:

```
data Complexos = Par (Float,Float)
  deriving (Eq,Show)
```

define o tipo `Complexo` com um construtor binário que constrói um complexo como um par de reais.

De uma forma mais geral podemos definir tipos paramétricos, isto é tipos em cuja definição entram variáveis de tipo, e que desta forma não definem um só tipo mas um "classe" de tipos. Por exemplo, o tipo `Pares` pode ser definido do seguinte modo:

```
data Pares a b = ConstPares (a,b)
 deriving (Eq,Show)
```

com a definição das funções de projecção a serem definidas através da associação de padrões dados pelo construtor de pares `ConstPares`.

```
projX :: Pares a b → a
projX (ConstPares (x,y)) = x

projY :: Pares a b → b
projY (ConstPares (x,y)) = y
```

4.4 Definição Recursiva

É possível estender as definições de tipos ao caso das definições recursivas. Por exemplo o tipo `Lista` pode ser definido do seguinte modo:

```
data Lista a = Vazia | Lst (a,Lista a)
    deriving (Eq,Show)
```

Nesta caso temos dois construtores: `Vazia`, um construtor 0-ário, e `Lst` um construtor binário.

As definições de funções sobre elementos deste novo tipo podem ser definidas através de associação de padrões, tendo em conta os dois construtores disponibilizados.

Por exemplo a função que calcula o comprimento de uma lista.

```
comprimento :: (Lista a) → Int
comprimento Vazia = 0
comprimento (Lst(x,l)) = 1+comprimento(l)
```

como exemplo de aplicação desta função temos:

```
Main> comprimento(Lst(1,Lst(2,Lst(3,Vazia))))
3
```

Outro exemplo de uma definição paramétrica e recursiva é nos dada pela definição de árvores binárias.

```
data ArvBin a = Folha a | Ramo (ArvBin a,ArvBin a)
 deriving (Show)
```

A definição da função de travessia `inordem` é nos dada por:

```
inordem :: (ArvBin a) → [a]
inordem (Folha x) = [x]
inordem (Ramo(ab1,ab2)) = (inordem ab1 ) ++ (inordem ab2)
```

um exemplo de utilização é o seguinte:

```
Main> inordem(Ramo(Ramo(Folha 1,Folha 4),Folha 3))
[1, 4, 3]
```