

Tipos Algébricos

Exemplo: `data CCart = Coord Float Float`

Os valores do tipo `CCart` são expressões da forma `(Coord x y)`, em que `x` e `y` são valores do tipo `Float`.

`Coord` pode ser vista como uma função cujo tipo é

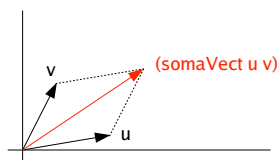
`Coord :: Float -> Float -> CCart`

mas os *constructores* são *funções especiais*, pois não têm nenhuma definição associada.

Expressões como `(Coord 1 3.1)` ou `(Coord 3 0.7)`, não podem ser reduzidas, e são exemplos de valores atômicos do tipo `CCart`.

Exemplo:

Função que soma de dois vectores:

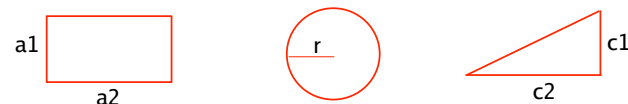


```
somaVect :: CCart -> CCart -> CCart
somaVect (Coord x1 y1) (Coord x2 y2) = Coord (x1+x2) (y1+y2)
```

85

Tipos Algébricos

Exemplo: Um tipo de dados para representar as seguintes figuras geométricas.



```
data Figura = Rectangulo Float Float
            | Circulo Float
            | Triangulo Float Float
```

Cálculo da área de uma figura:

```
area :: Figura -> Float
area (Rectangulo a1 a2) = a1 * a2
area (Circulo r) = pi * r^2
area (Triangulo c1 c2) = c2 * c1 / 2
```

Uma lista com figuras geométricas:

```
lfig = [(Rectangulo 5 3.2), (Circulo 5.7), (Triangulo 4 3)]
```

Note que é o facto de termos definido o tipo de dados `Figura` que nos permite construir esta lista, uma vez que só são aceites *listas homogéneas*.

87

Tipos Algébricos

Exemplo: `data Hora = AM Int Int | PM Int Int`

Os valores do tipo `Hora` são expressões da forma `(AM x y)` ou `(PM x y)`, em que `x` e `y` são valores do tipo `Int`.

Os constructores do tipo `Hora` são:

```
AM :: Int -> Int -> Hora
PM :: Int -> Int -> Hora
```

e podem ser vistos como uma “etiqueta” que indica de que forma os argumentos a que são aplicados devem ser entendidos.

Os *data types* implementam o **co-produto** (ou a **união disjunta**) de tipos.

NOTA: Erradamente, pode parecer que termos como `(AM 5 10)`, `(PM 5 10)` ou `(5,10)` contêm a mesma informação, mas não! Os constructores `AM` e `PM` têm aqui um papel essencial na interpretação que fazemos destes termos.

Exemplo: As funções sobre tipos algébricos geralmente definem-se por *pattern matching*.

```
totalMinutos :: Hora -> Int
totalMinutos (AM h m) = h*60 + m
totalMinutos (PM h m) = (h+12)*60 + m
```

86

Tipos Algébricos

As definições de tipos também podem ser *recursivas*.

Exemplo: O tipo dos números naturais pode ser definido por

```
data Nat = Zero | Suc Nat
```

O tipo `Nat` é definido à custa dos constructores

isto é,

```
Zero :: Nat
Suc :: Nat -> Nat
```

`Zero` é um valor do tipo `Nat`, e
se `n` é um valor do tipo `Nat`, `(Suc n)` é também um valor do tipo `Nat`.

A este género de tipo algébrico dá-se o nome de **tipo recursivo**.

Exemplos:

```
Zero
Suc Zero
Suc (Suc Zero)
```

São números naturais.

```
fromNatToInt :: Nat -> Int
fromNatToInt Zero = 0
fromNatToInt (Suc n) = 1 + (fromNatToInt n)
```

```
somaNat :: Nat -> Nat -> Nat
somaNat Zero n = n
somaNat (Suc n) m = Suc (somaNat n m)
```

88

Tipos Algébricos

O tipo pré-definido `[a]` das listas é um outro exemplo de um *tipo recursivo*.

Exemplo: Poderíamos definir o tipo das listas, através da seguinte definição:

```
data Lista a = Nil
              | Cons a (Lista a)
```

O tipo `(Lista a)` é aqui definido à custa dos contrutores

```
Nil :: Lista a
Cons :: a -> Lista a -> Lista a
```

A lista `[3, 7, 1]` seria representada pela expressão

```
Cons 3 (Cons 7 (Cons 1 Nil))
```

`(Lista a)` é um exemplo de um **tipo polimórfico**.

`Lista` está parameterizada com uma variável de tipo `a`, que poderá ser substituída por um tipo qualquer. (É neste sentido que se diz que `Lista` é um *construtor de tipos*.)

Exemplo:

```
comprimento :: Lista a -> Int
comprimento Nil = 0
comprimento (Cons _ xs) = 1 + comprimento xs
```

89

Expressões CASE

Exemplos:

```
par :: Nat -> Bool
par n = case n of
    Zero       -> True
    (Suc (Suc x)) -> par x
    _          -> False
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p l = case l of
    [] -> []
    (x:xs) -> if (p x) then x : (takeWhile p xs)
                else []
```

Exercícios:

- Defina duas versões da função `impar` (com e sem expressões case).
- Defina uma outra versão da função `takeWhile` utilizando várias equações.

Nota: As expressões `if-then-else` são equivalentes à análise de casos no tipo `Bool`.

```
if e then e1
    else e2
```

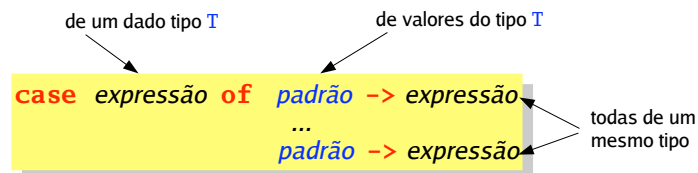
é equivalente a

```
case e of True -> e1
         False -> e2
```

91

Expressões CASE

O Haskell tem ainda uma forma construir expressões que permite fazer **análise de casos** sobre a estrutura dos valores de um tipo. Essas expressões têm a forma:



Exemplos:

```
fria :: Cor -> Bool
fria c = case c of
    Azul -> True
    Verde -> True
    _ -> False
```

```
comprimento :: Lista a -> Int
comprimento l = case l of
    Nil -> 0
    (Cons _ xs) -> 1 + comprimento xs
```

90

A construção de tipos algébricos dá à linguagem Haskell um enorme poder expressivo, pois permite a implementação de:

- tipos enumerados;
- co-produtos (união disjunta de tipos);
- tipos recursivos;
- uma certa forma de *encapsulamento de dados*.

Além disso, os tipos algébricos:

- podem ter uma apresentação escrita própria
- podem ser declarados como instâncias de classes

Nota: Se quiser experimentar os exemplos apresentados atrás, será melhor acrescentar às declarações dos tipos algébricos a indicação: **deriving Show**, para que os valores dos novos tipos possam ser escritos (no formato usual).

Exemplo:

```
data Nat = Zero | Suc Nat
    deriving Show
```

92

O construtor de tipos Maybe

Um tipo algébrico importante, já pré-definido no Prelude é o tipo polimórfico

```
data Maybe a = Nothing | Just a
```

que permite representar a **parcialidade**, podendo ser usado para lidar com situações de excepções e erros.

Exemplos:

```
divisao :: Integer -> Integer -> Maybe Integer
divisao x y | y /= 0 = Just (x `div` y)
            | otherwise = Nothing
```

```
cabeca :: [a] -> Maybe a
cabeca (x:xs) = Just x
cabeca [] = Nothing
```

em casos de excepções o resultado é **Nothing**

Funções que trabalham sobre um tipo **t** terão que ser adaptadas para trabalhar com o tipo **Maybe t**.

Exemplo:

```
soma (Just x) (Just y) = Just (x+y)
soma _ _ = Nothing
```

93

A função **proc** só consegue concluir que um dado BI não ocorre na tabela, ao fim de pesquisar toda a lista.

Esta conclusão poderia ser tirada mais cedo, se que a lista estivesse **ordenada** por BI.

Exemplo: Tendo a garantia de que a lista está ordenada por ordem crescente de BI, podemos definir a função de procura da seguinte forma:

```
proc :: BI -> [(BI, Nome)] -> Maybe Nome
proc n ((x,y):xys) | n > x = proc n xys
                  | n == x = Just y
                  | n < x = Nothing
proc _ [] = Nothing
```

Nos casos de insucesso, esta versão de **proc** é bastante **mais eficiente** do que a versão do slide anterior.

Exercício: Compare o funcionamento das duas versões da função **proc** para o seguinte exemplo:

```
proc 3 [ (bi,"xxxxx") | bi <- [1,5..1000] ]
```

95

Exemplo: A seguinte função que procura o nome associado a um dado número de BI, numa tabela implementada como uma lista de pares.

```
type BI = Integer
type Nome = String
```

```
procura :: BI -> [(BI, Nome)] -> Nome
procura n ((x,y):xys) | n == x = y
                    | otherwise = procura n xys
procura _ [] = error "Não existe!"
```

A função **procura** termina em **erro** caso o BI não exista na tabela. Ou seja, **procura** é uma **função parcial**.

Podemos **totalizar** a função de procura usando o tipo **(Maybe Nome)**.

```
proc :: BI -> [(BI, Nome)] -> Maybe Nome
proc n ((x,y):xys) | n == x = Just y
                  | otherwise = proc n xys
proc _ [] = Nothing
```

Desta forma, se o BI não existir na tabela a função **proc** devolve **Nothing** e nunca termina em erro. Ou seja, **proc** é uma **função total**.

94

Árvores Binárias

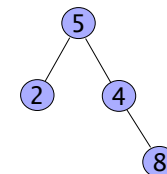
Uma estrutura de dados muito útil para organizar informação são as **árvores binárias**.

O tipo polimórfico das árvores binárias pode definido pelo seguinte tipo recursivo:

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
```

Ou seja, uma árvore binária: ou é vazia; ou é um nodo com um valor e duas sub-árvores.

Exemplo: A árvore de valores inteiros:



é representada pela expressão

```
(Nodo 5 (Nodo 2 Vazia Vazia)
        (Nodo 4 Vazia (Nodo 8 Vazia Vazia)))
```

de tipo **(ArvBin Int)**.

96