Programação Funcional
8ª Aula — Definição de tipos

Sandra Alves
DCC/FCUP

2019/20

Declarações de sinónimos I

Podemos dar um nome novo a um tipo existente usando uma declaração de sinónimo.

Exemplo (do prelúdio-padrão):

```
type String = [Char]
```

Declarações de sinónimos II

As declarações de sinónimos são usadas para melhorar legibilidade de programas.

Exemplo:

```
type Pos = (Int,Int)          -- coluna,linha
type Cells = [Pos]            -- colónia
```

Assim podemos escrever

```
isAlive :: Cells -> Pos -> Bool
```

em vez de

```
isAlive :: [(Int,Int)] -> (Int,Int) -> Bool
```

Declarações de sinónimos III

As declarações de sinónimos também podem ter parâmetros.

Exemplo: associações entre chaves e valores.

```
type Assoc ch v = [(ch,v)]          -- tabela de associações

idades :: Assoc String Int
idades = [("Sara", 39), ("João", 27), ("Maria", 19)]

emails :: Assoc String String
emails = [("Sandra", "sandra@dcc.fc.up.pt"),
          ("João", "joao@gmail.com")]
```

Declarações de sinónimos IV

Os sinónimos podem ser usados noutras definições:

```
type Pos = (Int,Int)
type Cells = [Pos]                          -- OK
```

Mas não podem ser usados recursivamente:

```
type List a = (a,List a)                    -- ERRO
```

Declarações de novos tipos I

Podemos definir novos tipos de dados usando declarações `data`.

Exemplo (do prelúdio-padrão):

```
data Bool = True | False
```

Declarações de novos tipos II

- A declaração `data` enumera as alternativas separadas por barras verticais.
- Cada alternativa deve ter um constructor (ex.: `True` e `False`).
- O nome dos tipos e construtores deve ser começar por uma letra maiúscula.
- Cada construtor só pode ser usado num único tipo.

Declarações de novos tipos III

Podemos definir funções sobre novos tipos usando padrões.

Exemplo: um tipo para as direções ortogonais (esquerda, direita, cima, baixo).

```
data Dir = Esq | Dir | Cima | Baixo
```

Vamos definir algumas funções...

Declarações de novos tipos IV

```
contraria :: Dir -> Dir                    -- direção contrária
contraria Esq = Dir
contraria Dir = Esq
contraria Cima = Baixo
contraria Baixo = Cima

mover :: Dir -> Pos -> Pos                 -- deslocar numa direção
mover Esq (x,y) = (x-1,y)
mover Dir (x,y) = (x+1,y)
mover Cima (x,y) = (x,y+1)
mover Baixo (x,y)= (x,y-1)
```

Construtores com parâmetros I

Os construtores podem também ter parâmetros.

Exemplo:

```
data Figura = Circ Float              -- raio
            | Rect Float Float        -- largura, altura

quadrado :: Float -> Figura
quadrado h = Rect h h

area :: Figura -> Float
area (Circ r)   = pi*r^2
area (Rect w h) = w*h
```

Construtores com parâmetros II

- Os construtores podem ter diferentes números de parâmetros
- Os parâmetros podem ser de tipos diferentes
- Podemos usar os construtores de duas formas:

como funções para construir um valor

```
Circ :: Float -> Figura
Rect :: Float -> Float -> Figura
```

em padrões no lado esquerdo de equações

```
area (Circ r)   = pi*r^2
area (Rect w h) = w*h
```

Igualdade e conversão em texto I

Por omissão um novo tipo não tem métodos de igualdade ou conversão para texto.

O interpretador dá erro se tentarmos mostrar ou comparar valores:

```
> Circ 2
ERROR: No instance for (Show Figura)...

> Rect 2 1 == Rect 1 2
ERROR: No instance for (Eq Figura)...
```

Igualdade e conversão em texto II

Podemos definir igualdade e conversão para texto automaticamente usando “`deriving`”:

```
data Figura = Circ Float
            | Rect Float Float
              deriving (Eq, Show)
```

Exemplo de uso:

```
> Circ 2
Circ 2.0

> Rect 2 1 == Rect 1 2
False
```

A igualdade é *sintática*: dois valores são iguais se e só se têm o mesmo construtor e argumentos.

Novos tipos com parâmetros

As declarações de novos tipos também podem ter parâmetros.

Exemplo:

```
data Maybe a = Nothing | Just a       -- do prelúdio-padrão

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n`div`m)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Tipos recursivos I

As declarações `data` podem ser *recursivas*.

Exemplo: os números naturais.

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

Tipos recursivos II

Alguns valores de `Nat`:

```
Zero                          -- zero
Suc Zero                      -- um
Suc (Suc Zero)                -- dois
Suc (Suc (Suc Zero))          -- três
⋮
```

Em geral: n é obtido aplicado n vezes `Succ` a `Zero`.

```
Suc (Suc (... (Suc Zero)...))       -- n aplicações
```

Tipos recursivos III

Usando recursão, podemos definir funções que convertem entre inteiros e naturais:

```
int2nat :: Int -> Nat
int2nat 0       = Zero
int2nat n | n>0 = Suc (int2nat (n-1))

nat2int :: Nat -> Int
nat2int Zero   = 0
nat2int (Suc n)= 1+nat2int n
```

Tipos recursivos IV

Uma definição completa de `int2nat` usando o tipo `Maybe Nat`:

```
int2nat:: Int -> Maybe Nat
int2nat 0       = Just Zero
int2nat n | n>0 = Just (Suc n')
          | otherwise = Nothing
          where (Just n') = (int2nat (n-1))
```

Tipos recursivos V

Podemos usar as funções de conversão para somar naturais.

```
add :: Nat -> Nat -> Nat
add n m = int2nat (nat2int n + nat2int m)
```

Tipos recursivos VI

Em alternativa, podemos definir a soma usando recursão sobre naturais.

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

Estas duas equações traduzem as seguintes igualdades algébricas:

$$0 + m = m$$
$$(1 + n) + m = 1 + (n + m)$$

Tipos recursivos VII

Exemplo:

```
  add (Suc (Suc Zero)) (Suc Zero)
=
  Suc (add (Suc Zero) (Suc Zero))
=
  Suc (Suc (add Zero (Suc Zero)))
=
  Suc (Suc (Suc Zero))
```

Árvores sintáticas I

Podemos representar expressões por uma *árvore sintática* em que os operadores são os *nós* e as constantes são as *folhas*.

Exemplo:

$$1 + 2 \times 3 \quad \simeq \quad \begin{tikzcd} & + \arrow[dl] \arrow[dr] & & \\ 1 & & \times \arrow[dl] \arrow[dr] & \\ & 2 & & 3 \end{tikzcd}$$

Árvores sintáticas II

As árvores podem ser representadas em Haskell por um tipo recursivo.

```
data Expr = Val Int                -- constante
          | Soma Expr Expr         -- nó +
          | Mult Expr Expr         -- nó ×
```

A árvore no *slide* anterior é:

```
Soma (Val 1) (Mult (Val 2) (Val 3))
```

Árvores sintáticas III

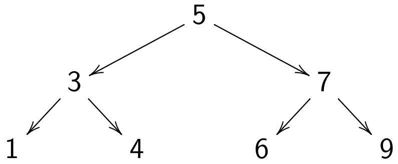
Exemplos de funções sobre árvores de expressões.

```
-- contar o número de folhas
tamanho :: Expr -> Int
tamanho (Val n) = 1
tamanho (Soma e1 e2) = tamanho e1 + tamanho e2
tamanho (Mult e1 e2) = tamanho e1 + tamanho e2

-- calcular o valor
valor :: Expr -> Int
valor (Val n) = n
valor (Soma e1 e2) = valor e1 + valor e2
valor (Mult e1 e2) = valor e1 * valor e2
```

Árvores binárias I

Também podemos usar árvores binárias para facilitar a organização e pesquisa de informação.



Árvores binárias II

Podemos representar árvores binárias de inteiros por um tipo recursivo.

```
data Arv = Folha Int
         | No Arv Int Arv
```

A árvore no *slide* anterior seria representa por:

```
No (No (Folha 1) 3 (Folha 4))
   5
   (No (Folha 6) 7 (Folha 9))
```

Árvores binárias III

Podemos agora definir uma função recursiva para procurar um valor numa árvore.

```
ocorre :: Int -> Arv -> Bool
ocorre m (Folha n)       =  n==m
ocorre m (No esq n dir) = (n==m ||
                            ocorre m esq ||
                            ocorre m dir)
```

Árvores binárias IV

Numa árvore ordenada todos os nós têm valores inferiores na sub-árvore esquerda e superiores na sub-árvore direira. Nesse caso podemos simplificar a pesquisa:

```
ocorre' :: Int -> Arv -> Bool
ocorre' m (Folha n)       = n==m
ocorre' m (No esq n dir) | n==m = True
                         | m<n  = ocorre' m esq
                         | m>n  = ocorre' m dir
```

Esta definição é mais eficiente: percorre apenas os nós num *caminho da raiz até uma folha* em vez de *todos* os nós da árvore.