Programação Funcional
Aula 13 — Definição de tipos

Pedro Vasconcelos
DCC/FCUP

2021

Declarações de sinónimos

Podemos definir um nome novo para um tipo existente usando uma declaração `type`.

Exemplo (do prelúdio-padrão):

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

Diz-se que esta declaração define um sinónimo.

Declarações de sinónimos (cont.)

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

Exemplo: no jogo da Vida definimos sinónimos:

```
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 (cont.)

As declarações `type` também podem ter parâmetros.

Exemplo: listas de associações entre chaves e valores.

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

idades :: Assoc String Int
idades = [("Pedro", 41), ("João", 27), ("Maria", 19)]

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

Declarações de sinónimos (cont.)

Os sinónimos podem ser usados noutras definições:

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

Mas não podem ser usados recursivamente:

```
type Tree = (Int,[Tree])                              -- ERRO
```

Declarações de novos tipos

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

Exemplo (do prelúdio-padrão):

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

Declarações de novos tipos (cont.)

- A declaração `data` enumera as alternativas de valores do novo tipo
- `True` e `False` são os *construtores* do tipo `Bool`
- Os construtores devem ser únicos (não podem ser usados em tipos diferentes)
- Os nome dos tipos e construtores devem começar por uma letra maiúscula

Declarações de novos tipos (cont.)

Podemos usar novos tipos tal qual como tipos pré-definido na linguagem.

Exemplo: com a declaração

```
data Dir = Esquerda | Direita | Cima | Baixo
```

podemos definir

```
direções :: [Dir]
direções = [Esquerda, Direita, Cima, Baixo]

oposta :: Dir -> Dir
oposta Esquerda = Direita
oposta Direita  = Esquerda
oposta Cima     = Baixo
oposta Baixo    = Cima
```

Construtores com parâmetros

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 l a) = l*a
```

Construtores com parâmetros (cont.)

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

  aplicando argumento 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

Por omissão novos tipos não têm instâncias de classes como `Show`, `Eq` ou `Ord`.

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

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

Igualdade e conversão em texto (cont.)

Podemos pedir definições automaticas destas instâncias usando `deriving` na declaração.

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

Exemplos:

```
> show (Circ 2)
"Circ 2.0"

> Rect 1 (1+1) == Rect 1 2
True
```

Novos tipos com parâmetros

As declarações de novos tipos também podem ter parâmetros.

Exemplo:

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

Podemos usar `Maybe` para definir uma divisão inteira que não dá erros:

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

Modelar informação

Exemplo: items num inventário de uma loja.

```
data Produto
  = Produto Nome Stock PreçoCusto PreçoVenda

type Nome        = String
type Stock       = Int    -- quantidade inteira
type PreçoCusto  = Float  -- preços em euros
type PreçoVenda  = Float

inventário :: [Produto]
inventário =
   [Produto "Haskell for dummies" 4 30.0 40.0,
    Product "The C programming language" 5 35.0 50.0,
    Product "The Art of Computer Programming" 0 55.0 80.0
   ]
```

Processar informação

```
-- Calcular o valor de stock (euros)
valorEmStock :: [Produto] -> Float
valorEmStock items
  = sum [fromIntegral stock*custo
         | Produto _ stock custo _ <- items]

-- Aplicar um desconto a todos os items em stock
desconto :: Float -> [Produto] -> [Produto]
desconto taxa items
  = [Produto nome stock custo (venda*(1-taxa))
     | Produto nome stock custo venda <- items, stock>0]
```

Tipos recursivos

As declarações `data` podem ser *recursivas*.

Exemplo: os números naturais.

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

O tipo `Nat` tem dois construtores:

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

Valores do tipo `Nat`

Alguns valores de `Nat`:

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

Em geral: os valores de `Nat` são obtidos aplicado *n* vezes `Succ` a `Zero`.

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

Podemos pensar nestes valores como representado os naturais $n \geq 0$.

Funções sobre naturais

Podemos definir funções recursivas que convertem entre inteiros e este novo tipo.

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

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

Funções sobre naturais (cont.)

Podemos usar as funções de conversão para somar naturais.

```
add :: Nat -> Nat -> Nat
add n m = natFromInt (intFromNat n + intFromNat m)
```

Funções sobre naturais (cont.)

Em alternativa, podemos definir diretamente a adição 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:

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

Exemplo

Vamos calcular soma de 2 com 1:

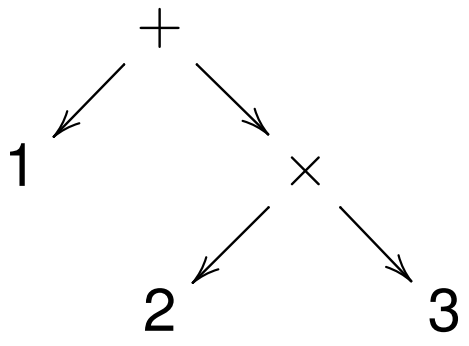
```
  add (Suc (Suc Zero)) (Suc Zero)
=
  Suc (add (Suc Zero) (Suc Zero))
=
  Suc (Suc (add Zero (Suc Zero)))
=
  Suc (Suc (Suc Zero))
```

Árvores sintáticas

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

Exemplo:

$1 + 2 \times 3 \quad \simeq$



Árvores sintáticas (cont.)

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

```
data Expr = Val Int           -- constante
          | Soma Expr Expr    -- soma de 2 sub-expressões
          | Mult Expr Expr    -- multiplicação
```

A árvore no *slide* anterior é:

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

Árvores sintáticas (cont.)

Podemos definir funções sobre árvores de expressões usando encaixe de padrões e recursão.

```
-- contar o tamanho da expressão
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 representado pela expressão
valor :: Expr -> Int
valor (Val n) = n
valor (Soma e1 e2) = valor e1 + valor e2
valor (Mult e1 e2) = valor e1 * valor e2
```