

Programação Funcional

Módulos e Tipos abstratos

2022

Tipos concretos

Até agora definimos um novo tipo de dados começando por listar os seus construtores.

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

```
data Nat = Zero | Succ Nat
```

Esta definição diz-se **concreta** porque se começa por definir a representação de *dados* mas não quais as *operações*.

Tipos abstratos

Em alternativa, podemos começar por especificar as operações que um tipo deve suportar.

Esta especificação diz-se **abstrata** porque omitimos a representação concreta dos dados.

Pilhas I

A pilha é uma estrutura LIFO (“last-in, first-out”): o último valor a ser colocado é o primeiro a ser removido.

Pilhas II

Operações:

push acrescentar um valor ao topo da pilha

pop remover o valor do topo da pilha

top obter o valor no topo da pilha

empty criar uma pilha vazia

isEmpty testar se uma pilha é vazia

Pilhas III

Vamos especificar a pilha como um tipo *Stack* e uma função por cada operação.

```
data Stack a    -- pilha com valores de tipo 'a'
```

```
push :: a -> Stack a -> Stack a
```

```
pop  :: Stack a -> Stack a
```

```
top  :: Stack a -> a
```

```
empty :: Stack a
```

```
isEmpty :: Stack a -> Bool
```

Implementação de um tipo abstrato

Para implementar o tipo abstrato:

- ▶ escolhemos uma representação concreta e implementamos as operações
- ▶ escondemos a representação e permitimos *apenas* usar o tipo e as operações

Vamos usar **módulos** para implementar tipos abstratos em Haskell.

Módulos

- ▶ Um módulo é um conjunto de definições relacionadas (tipos, constantes e funções)
- ▶ Definimos um módulo `Foo` num ficheiro `Foo.hs` com a declaração:

```
module Foo where
```

- ▶ Para usar o módulo `Foo` noutro módulo colocamos uma declaração `import Foo`
- ▶ Podemos listar explicitamente quais as entidades exportadas dum módulo;

```
module Foo(T1, T2, f1, f2, ...) where
```

- ▶ Isto permite esconder partes da implementação

Implementação de pilhas I

```
module Stack (Stack,          -- exportar o tipo
              push, pop, top,  -- e as operações
              empty, isEmpty) where

data Stack a = Stk [a]  -- implementação usando listas

push :: a -> Stack a -> Stack a
push x (Stk xs) = Stk (x:xs)

pop :: Stack a -> Stack a
pop (Stk (_:xs)) = Stk xs
pop _            = error "Stack.pop: empty stack"
```

Implementação de pilhas II

```
top :: Stack a -> a
top (Stk (x:_)) = x
top _           = error "Stack.top: empty stack"
```

```
empty :: Stack a
empty = Stk []
```

```
isEmpty :: Stack a -> Bool
isEmpty (Stk []) = True
isEmpty (Stk _)  = False
```

Usar o tipo abstrato I

Determinar se uma cadeia de caracteres tem os sinais de parêntesis correctamente casados:

- ▶ usamos uma pilha auxiliar com caracteres
- ▶ quando encontramos ' (' empilhamos
- ▶ quando encontramos ') ' verificamos o topo da pilha e retiramos o carácter correspondente
- ▶ se no final a pilha ficar vazia, então os parêntesis estão correctamente casados

Usar o tipo abstrato II

```
import Stack

parent :: String -> Bool
parent str = parentAux str empty

parentAux :: String -> Stack Char -> Bool
parentAux []      stk = isEmpty stk
parentAux (x:xs) stk
  | x == '(' = parentAux xs (push '(' stk)
  | x == ')' = not (isEmpty stk) &&
                top stk == '(' &&
                parentAux xs (pop stk)
  | otherwise = parentAux xs stk
```

(Ver demonstração.)

Usar o tipo abstrato III

- ▶ Apenas usamos as operações exportadas (`empty`, `isEmpty`, `push`, `top`, `pop`)
- ▶ Não podemos usar o construtor de pilhas `Stk` porque este não é exportando
- ▶ Logo: o nosso programa usa pilhas como um tipo abstrato

Propriedades das pilhas I

- ▶ Podemos especificar o comportamento das operações dum tipo abstrato usando equações algébricas
- ▶ Exemplo: para todo elemento x e pilha s

$$\text{pop}(\text{push } x \ s) = s$$

- ▶ Tradução: para qualquer pilha e valor, fazer um *push* seguido de um *pop* deixa a pilha inalterada

Propriedades das pilhas II

Para todos elementos x e pilha s :

$$\text{pop}(\text{push } x \ s) = s \quad (1)$$

$$\text{top}(\text{push } x \ s) = x \quad (2)$$

$$\text{isEmpty } \text{empty} = \text{True} \quad (3)$$

$$\text{isEmpty}(\text{push } x \ s) = \text{False} \quad (4)$$

- ▶ Estas propriedades são válidas intuitivamente
- ▶ Mais tarde veremos que as podemos provar pelas definições das operações

Conjuntos

member testar se um elemento pertence a um conjunto

insert acrescentar um elemento a um conjunto

delete remover um elemento de um conjunto

union união de dois conjuntos

intersection interseção de dois conjuntos

empty conjunto vazio

isEmpty testar se um conjunto é vazio

fromList construir um conjunto a partir duma lista de elementos

toList listar os elementos de um conjunto

Exemplos

```
> insert 1 (insert 2 (insert 1 empty)
fromList [1,2])
```

```
> delete 2 (fromList [1,2,3])
fromList [1,3]
```

```
> member 2 (fromList [1,3,4,6])
False
```

```
> intersection (fromList [1..4]) (fromList [3..6])
fromList [3,4]
```

Implementação I

- ▶ Vamos representar conjuntos por árvores de pesquisa dos elementos
- ▶ Permite inserção e procura de valores mais eficiente do que uma lista
- ▶ Necessitamos de uma ordem total entre os valores

```
data Set a = Empty
           -- conjunto vazio
           | Node a (Set a) (Set a)
           -- elemento, sub-conjunto dos menores,
           -- sub-conjunto dos maiores
```

Implementação II

Colocamos todas as definições num módulo que exporta

- ▶ o tipo *Set* e as operações;
- ▶ mas não os construtores *Empty* e *Node*.

```
module Set
  (Set,
   empty, insert, delete, member,
   union, intersection, fromList, toList) where
```

Implementação III

```
member :: Ord a => a -> Set a -> Bool
```

```
member x Empty = False
```

```
member x (Node y left right)
```

```
    | x == y = True
```

```
    | x > y  = member x right
```

```
    | x < y  = member x left
```

```
insert :: Ord a => a -> Set a -> Set a
```

```
insert x Empty = Node x Empty Empty
```

```
insert x (Node y left right)
```

```
    | x == y = Node y left right
```

```
    | x > y  = Node y left (insert x right)
```

```
    | x < y  = Node y (insert x left) right
```

```
delete :: Ord a => a -> Set a -> Set a
```

```
... -- ver aula 15 (árvores de pesquisa)
```

Implementação IV

```
empty :: Set a  
empty = Empty
```

```
isEmpty :: Set a -> Bool  
isEmpty Empty = True  
isEmpty _      = False
```

```
toList :: Set a -> [a]  
toList Empty = []  
toList (Node x l r) = toList l ++ [x] ++ toList r
```

```
fromList :: Ord a => [a] -> Set a  
fromList = foldr insert Empty
```

Exercício

Implementar as operações em falta (união e interseção).

```
union :: Ord a => Set a -> Set a -> Set a
```

```
intersection :: Ord a => Set a -> Set a -> Set a
```

Algumas propriedades das operações

$$\text{member } x \text{ empty} = \text{False} \quad (5)$$

$$\text{member } x (\text{insert } x \ a) = \text{True} \quad (6)$$

$$\text{member } x (\text{insert } y \ a) = \text{member } x \ a \quad (x \neq y) \quad (7)$$

$$\text{insert } x (\text{insert } x \ a) = \text{insert } x \ a \quad (8)$$

$$\text{insert } x (\text{insert } y \ a) = \text{insert } y (\text{insert } x \ a) \quad (9)$$

Tabelas de associações

- ▶ A cada *chave* pode ter associado um único *valor*
- ▶ Também conhecidas como *mapas* ou *dicionários*

insert inserir uma nova chave e valor

delete remover uma chave (se existir)

lookup procurar valor pela uma chave

empty tabela vazia

fromList construir a partir de uma lista de pares

toList listar os pares

Exemplos

```
> let tbl = insert 'a' 1 (insert 'b' 2 empty)
fromList [(a',1), ('b',2)]
```

```
> lookup 'b' tbl
Just 2
```

```
> lookup 'c' tbl
Nothing
```

```
> insert 'b' 3 tbl
fromList [('a',1), ('b',3)]
```

Implementação I

- ▶ Vamos representar associações por árvores de pesquisa ordenadas pelas chaves
- ▶ Cada chave está associada a um valor
- ▶ Necessitamos de ordem total apenas para as chaves

```
data Map k v = Empty
              -- ^ tabela vazia
              | Node k v (Map k v) (Map k v)
              -- ^ chave, valor e sub-árvores
```

Implementação II

Novamente colocamos as definições num módulo exportando apenas o tipo e as operações.

```
module Map
  (Map,
   empty, insert, delete, lookup,
   fromList, toList) where
```

Implementação III

```
insert :: Ord k => k -> v -> Map k v -> Map k v
insert x v Empty = Node x v Empty Empty
insert x v (Node y u l r)
  | x == y = Node x v l r
  | x > y  = Node y u l (insert x v r)
  | x < y  = Node y u (insert x v l) r

delete :: Ord k => k -> Map k v -> Map k v
...    -- ver aula 15 (árvores de pesquisa)
```

Implementação IV

- ▶ A função *lookup* pode não encontrar um valor
- ▶ Usamos o tipo *Maybe* (do prelúdio-padrão) para contemplar essa possibilidade:

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

```
lookup :: Ord k => k -> Map k v -> Maybe v
lookup x Empty = Nothing    -- não encontrou
lookup x (Node y v l r)
  | x == y = Just v         -- encontrou
  | x > y  = lookup x r
  | x < y  = lookup x l
```

Algumas propriedades das operações

$$\text{lookup } x \text{ empty} = \text{Nothing} \quad (10)$$

$$\text{lookup } x (\text{insert } x \ v \ a) = \text{Just } v \quad (11)$$

$$\text{lookup } x (\text{insert } y \ v \ a) = \text{lookup } x \ a \quad (x \neq y) \quad (12)$$

$$\text{insert } x \ v (\text{insert } x \ u \ a) = \text{insert } x \ v \ a \quad (13)$$

$$\text{insert } x \ v (\text{insert } y \ u \ a) = \text{insert } y \ u (\text{insert } x \ v \ a) \quad (x \neq y) \quad (14)$$

Nomes

- ▶ Usamos o mesmo nome para operações em diferentes tipos abstratos
- ▶ Se quisermos usar vários módulos devemos importar com nomes qualificados
- ▶ Também útil para distinguir das funções do prelúdio

```
import qualified Set
import qualified Map
```

```
Set.empty      -- conjunto vazio
Map.empty      -- Map vazio
Set.insert     -- inserir em conjuntos
Map.insert     -- inserir em Map
Map.lookup     -- procurar em Map
lookup         -- procurar em listas (do prelúdio)
```