

Programação Funcional

13ª Aula — Tipos abstratos

Sandra Alves
DCC/FCUP

2019/20

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 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

Uma *pilha* é uma estrutura de dados que suporta as seguintes operações:

push acrescentar um valor ao topo da pilha;

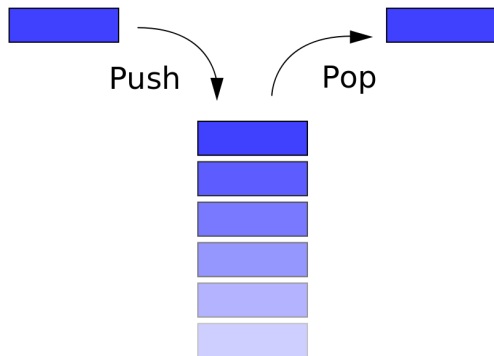
pop remover o valor do topo da pilha;

top obter o valor no topo da pilha;

empty criar uma pilha vazia;

isEmpty testar se uma pilha é vazia.

Pilhas II



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

Pilhas III

Vamos especificar a pilha como um tipo paramétrico `Stack` e uma função para 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:

1. escolher uma representação concreta e implementar as operações.
2. ocultar a representação concreta permitindo *apenas* usar as operações.

Vamos usar **módulos** para encapsular a definição de tipos abstratos.

Módulos

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

```
module Foo where
```

- Para usar o módulo `Foo` colocamos uma declaração
`import Foo`
- Por omissão, todas as definições num módulo são exportadas; podemos restringir as entidades exportadas:

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


Implementação de pilhas I

```
module Stack (Stack,                                -- exportar o tipo
              push, pop, top,                       -- exportar 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
```

Encapsulamento I

```
module Main where
import Stack
```

```
makeStack :: [a] -> Stack a
makeStack xs = Stk xs
```

-- ERRO

```
size :: Stack a -> Int
size (Stk xs) = length xs
```

-- ERRO

O construtor `Stk` não é exportado para fora do módulo; logo:

- não podemos construir pilhas usando `Stk`;
- não podemos usar encaixe de padrões com `Stk`;
- apenas podemos usar as operações exportadas.

Encapsulamento II

Usando *apenas* as operações abstratas sobre pilhas:

```
import Stack

makeStack :: [a] -> Stack a
makeStack xs = foldr push empty xs

size :: Stack a -> Int
size s | isEmpty s = 0
       | otherwise = 1 + size (pop s)
```

Propriedades das pilhas I

Podemos especificar o comportamento das operações dum tipo abstrato usando **equações algébricas**.

Exemplo: qualquer implementação de pilhas deve verificar as condições (1)–(4) para quaisquer valor x e pilha s .

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

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

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

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

Propriedades das pilhas II

Vamos verificar a propriedade (1) para a implementação com listas.

Seja $s = \text{Stk } xs$ em que xs é uma lista.

$$\begin{aligned} & \text{pop } (\text{push } x \overbrace{(\text{Stk } xs)}^s) \\ = & \quad \text{\textcolor{red}{\{pela definição de } push\}} \\ & \text{pop } (\text{Stk } (x : xs)) \\ = & \quad \text{\textcolor{red}{\{pela definição de } pop\}} \\ & \underbrace{\text{Stk } xs}_s \end{aligned}$$

Exercício: verificar as restantes propriedades.

Filas I

Uma *fila* suporta as seguintes operações:

enqueue acrescentar um valor ao fim da fila;

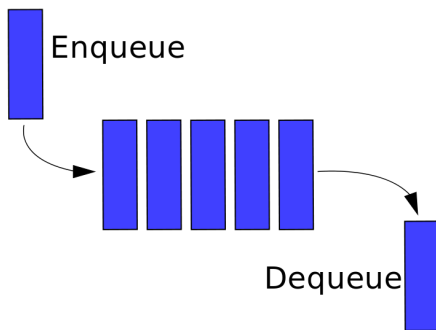
dequeue remover o valor do início da fila;

front obter o valor no início da fila;

empty criar uma fila vazia;

isEmpty testar se uma fila é vazia.

Filas II



A fila é uma estrutura FIFO (“first-in, first-out”): o primeiro valor a ser colocado é o primeiro a ser removido.

Filas III

```
data Queue a -- fila com valores de tipo 'a'
```

```
enqueue :: a -> Queue a -> Queue a
```

```
dequeue :: Queue a -> Queue a
```

```
front :: Queue a -> a
```

```
empty :: Queue a
```

```
isEmpty :: Queue a -> Bool
```

Vamos ver duas implementações:

- uma versão simples usando uma só lista;
- outra mais eficiente usando um par listas.

Filas (implementação simples) I

```
module Queue (Queue,
               enqueue, dequeue,
               front, empty, isEmpty) where

data Queue a = Q [a]                -- representação por uma lista

enqueue :: a -> Queue a -> Queue a    -- coloca no fim
enqueue x (Q xs) = Q (xs ++ [x])

dequeue :: Queue a -> Queue a          -- remove do início
dequeue (Q (_:xs)) = Q xs
dequeue _          = error "Queue.dequeue: empty queue"
```

Filas (implementação simples) II

```
front :: Queue a -> a                                -- valor no início
front (Q (x:_)) = x
front _         = error "Queue.front: empty queue"

empty :: Queue a
empty = Q []

isEmpty :: Queue a -> Bool
isEmpty (Q []) = True
isEmpty (Q _ ) = False
```

Observações

As operações *dequeue* e *front* retiram a cabeça da lista, logo executam em **tempo constante** (independente do comprimento da fila).

A operação *enqueue* acrescenta um elemento ao final da lista, logo executa em **tempo proporcional ao número de elementos da fila**.

Será que podemos fazer melhor?

Filas (implementação mais eficiente) I

Vamos representar uma fila por um par de listas: a **frente** e as **traseiras**.

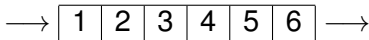
A lista da frente está pela **ordem de saída** da fila, enquanto a lista das traseiras está por **ordem de chegada** à fila.

Exemplos:

$([6, 5, 4], [1, 2, 3])$

$([6, 5, 4, 3, 2], [1])$

são duas representações da fila



Filas (implementação mais eficiente) II

Para **retirar um elemento**: removemos da lista da frente.

$$(x : fr, tr) \xrightarrow{\text{dequeue}} (fr, tr)$$

Para **introduzir um elemento**: acrescentamos à lista das traseiras.

$$(fr, tr) \xrightarrow{\text{enqueue } x} (fr, x : tr)$$

Temos ainda de **normalizar** o resultado quando a lista da frente fica vazia.

$$([], tr) \xrightarrow{\text{norm}} (\text{reverse } tr, [])$$

Filas (implementação mais eficiente) III

```
module Queue (Queue,  
              enqueue, dequeue,  
              front, empty, isEmpty) where  
data Queue a = Q ([a],[a])  -- par frente, traseiras
```

-- normalização (operação interna)

```
norm :: ([a],[a]) -> ([a],[a])
```

-- implementação das operações de filas

```
enqueue :: a -> Queue a -> Queue a
```

```
dequeue :: Queue a -> Queue a
```

```
front :: Queue a -> a
```

```
empty :: Queue a
```

```
isEmpty :: Queue a -> Bool
```

Filas (implementação mais eficiente) IV

```
module Queue (Queue,
               enqueue, dequeue,
               front, empty, isEmpty) where

data Queue a = Q ([a],[a])  -- par frente, traseiras

-- normalização (operação interna)
norm :: ([a],[a]) -> ([a],[a])
norm ([],tr) = (reverse tr, [])
norm (fr,tr) = (fr,tr)

-- implementação das operações de filas
enqueue :: a -> Queue a -> Queue a
enqueue x (Q (fr,tr)) = Q (norm (fr, x:tr))
```


Filas (implementação mais eficiente) V

```
dequeue :: Queue a -> Queue a
dequeue (Q (x:fr,tr)) = Q (norm (fr,tr))
dequeue _              = error "Queue.dequeue: empty queue"
```

```
front :: Queue a -> a
front (Q (x:fr, tr)) = x
front _              = error "Queue.front: empty queue"
```

```
empty :: Queue a
empty = Q ([],[])
```

```
isEmpty :: Queue a -> Bool
isEmpty (Q ([],_)) = True
isEmpty (Q (_,_))  = False
```

Observações

As operações *enqueue* e *dequeue* executam em tempo constante acrescido do tempo de normalização.

A operação de normalização executa no pior caso em **tempo proporcional ao comprimento** da lista das traseiras.

Porque é então esta solução mais eficiente?

Justificação (informal)

- A normalização executa em tempo n apenas após n operações em tempo constante
- Média amortizada: cada operação executa em tempo constante

Propriedades das filas

$$\text{front}(\text{enqueue } x \text{ empty}) = x \quad (5)$$

$$\begin{aligned} \text{front}(\text{enqueue } x (\text{enqueue } y \text{ q})) = \\ \text{front}(\text{enqueue } y \text{ q}) \end{aligned} \quad (6)$$

$$\text{dequeue}(\text{enqueue } x \text{ empty}) = \text{empty} \quad (7)$$

$$\begin{aligned} \text{dequeue}(\text{enqueue } x (\text{enqueue } y \text{ q})) = \\ \text{enqueue } x (\text{dequeue}(\text{enqueue } y \text{ q})) \end{aligned} \quad (8)$$

$$\text{isEmpty empty} = \text{True} \quad (9)$$

$$\text{isEmpty}(\text{enqueue } x \text{ q}) = \text{False} \quad (10)$$

Exercício: verificar as duas implementações.