Programação Funcional
14ª Aula — Tipos abstratos de dados

Pedro Vasconcelos
DCC/FCUP

2012

Tipos concretos de dados

Até agora definimos um novo tipo de dados listamos os seus construtores.

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

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

Estes tipos dizem-se concretos, porque se define a representação de *dados* mas não as *operações*.

Tipos abstratos de dados

Em alternativa, podemos começar por definir as operações que um tipo deve suportar, sem especificar a representação.

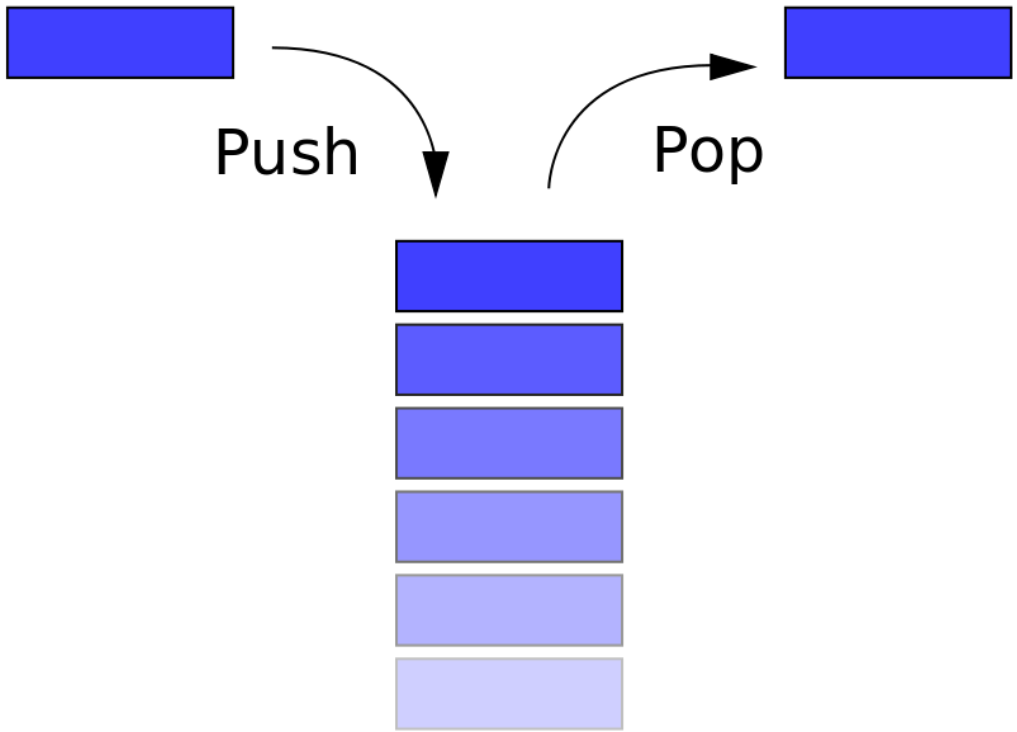
Tipos definidos *apenas* pelas suas operações dizem-se abstratos.

Pilhas

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 (cont.)



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

Pilhas (cont.)

Em Haskell vamos representar pilhas por 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 abstracto vamos:

1. Escolher uma representação concreta e implementar as operações.
2. Ocultar a representação concreta permitindo *apenas* usar as operações.

Em Haskell: vamos usar um módulo.

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

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

data Stack a = Stk [a]          -- representação 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 (cont.)

```
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
```

Usando o módulo `Stack`

Exemplo: calcular o tamanho duma pilha (número de elementos).

```
import Stack

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

Esta função usa *apenas* as operações abstratas sobre pilhas, não a representação concreta.

Ocultação da representação

```
import Stack

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

Esta definição é rejeitada porque o construtor de pilhas `Stk` é *invisível* fora do módulo `Stack` (logo não podemos usar encaixe de padrões).

Também não podemos construir pilhas usando `Stk` — apenas de usar as operações `push` e `empty`.

Propriedades das pilhas

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

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

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

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

$$isEmpty\ empty = True \tag{3}$$

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

Propriedades das pilhas (cont.)

Vamos verificar a propriedade (1) para a implementação com listas; temos $s = Stk\ xs$ em que xs é uma lista.

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

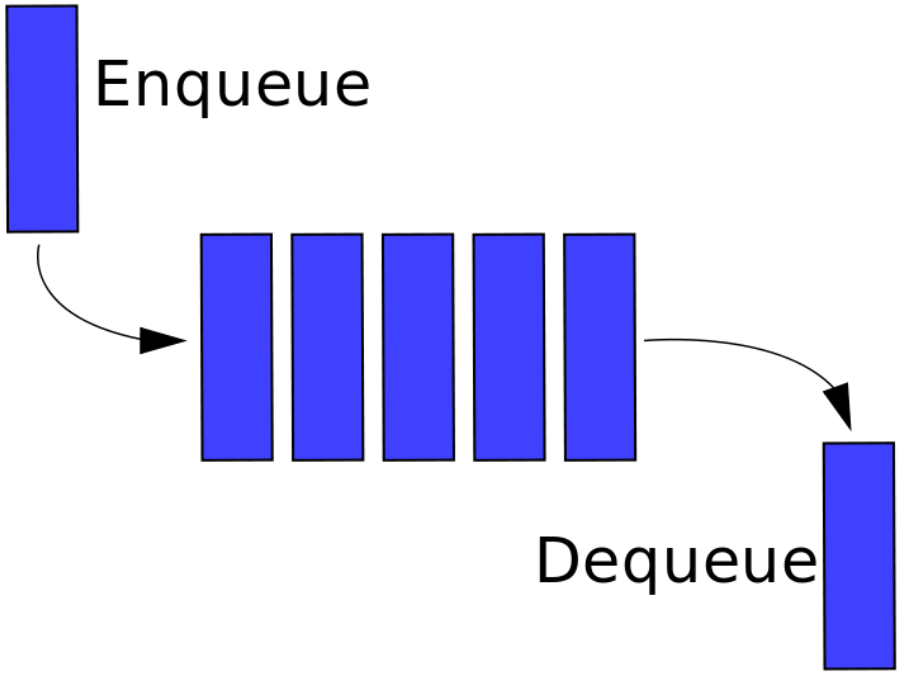
Exercício: verificar as restantes propriedades.

Filas

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 (cont.)



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

Filas (cont.)

```
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)

```
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 ínicio
dequeue (Q (_:xs))  = Q xs
dequeue _           = error "Queue.dequeue: empty queue"
```

Filas (implementação simples) (cont.)

```
front :: Queue a -> a                          -- valor no ínicio
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)

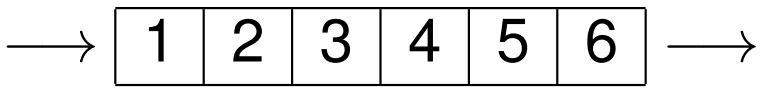
Vamos representar uma fila por um par de listas: a frente e as traseiras.

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) (cont.)

Para retirar um elemento: removemos da lista da frente.

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

Para introduzir um elemento: acrescentamos à lista das traseiras.

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

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

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

Filas (implementação mais eficiente) (cont.)

```
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) (cont.)

```
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?

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

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

$$front\ (enqueue\ x\ (enqueue\ y\ q)) = front\ (enqueue\ y\ q) \tag{6}$$

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

$$dequeue\ (enqueue\ x\ (enqueue\ y\ q)) = enqueue\ x\ (dequeue\ (enqueue\ y\ q)) \tag{8}$$

$$isEmpty\ empty = True \tag{9}$$

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

Exercício: verificar as duas implementações.

Conjuntos

Um *conjunto* é uma coleção de elementos sem ordem e sem repetição; suporta as seguintes operações:

empty	criar o conjunto vazio ($\emptyset$);
single	criar um conjunto com um elemento ($\{x\}$)
union	união de dois conjuntos ($\cup$);
inter	intersecção de dois conjuntos ($\cap$);
diff	diferença entre dois conjuntos ($\setminus$);
member	testar pertença a um conjunto ($\in$).

Conjuntos (cont.)

Em Haskell (1ª tentativa):

```
data Set a -- conjunto com valores de tipo 'a'

empty :: Set a
single :: a -> Set a
union :: Set a -> Set a -> Set a
inter :: Set a -> Set a -> Set a
diff :: Set a -> Set a -> Set a
member :: a -> Set a -> Bool
```

Conjuntos (cont.)

Em Haskell (2ª tentativa):

```
data Set a -- conjunto com valores de tipo 'a'

empty :: Set a
single :: a -> Set a
union :: Eq a => Set a -> Set a -> Set a
inter :: Eq a => Set a -> Set a -> Set a
diff :: Eq a => Set a -> Set a -> Set a
member :: Eq a => a -> Set a -> Bool
```

Necessitamos de *igualdade* entre elementos para testar pertença a um conjunto.

Implementação usando listas

```
module Set (Set, empty, single, union,
            inter, diff, member) where
import Data.List (nub) -- remover repetidos

data Set a = S [a]

empty :: Set a
empty = S []

single :: a -> Set a
single x = S [x]
```

Implementação usando listas (cont.)

```
member :: Eq a => a -> Set a -> Bool
member x (S xs) = x`elem`xs

union :: Eq a => Set a -> Set a -> Set a
union (S xs) (S ys) = S (nub (xs++ys))

⋮
```

Exercício: completar as operações restantes.

Implementação usando árvores binárias

```
module Set (Set, empty, single, union,
            inter, diff, member) where

data Set a = No a (Set a) (Set a) | Vazio

empty :: Set a
empty = Vazio

single :: a -> Set a
single x = No x empty empty
```

Implementação usando árvores binárias (cont.)

```
member :: Ord a => a -> Set a -> Bool
member x Vazio = False
member x (No y esq dir)
   | x==y       = True
   | x<y        = member x esq
   | otherwise = member x dir

⋮
```

Exercício: implementar as operações restantes:

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

NB: necessitamos de *ordem* entre elementos.