

CP - Ficha 1 - 18/09/2024

Exercício 1

Complete a codificação abaixo (em Haskell) das funções $\text{length} :: [a] \rightarrow \mathbb{Z}$ e $\text{reverse} :: [a] \rightarrow [a]$ que, respetivamente, calculam o comprimento da lista de entrada e a invertem:

```
length [] = ...  
length (x:xs) = ...  
reverse [] = ...  
reverse (x:xs) = ...
```

Resolução 1

```
length :: [a] -> Int  
length [] = 0  
length (x:xs) = 1 + length xs  
  
reverse :: [a] -> [a]  
reverse [] = []  
reverse (x:xs) = reverse xs ++ [x]
```

Exercício 2

A função $\text{take} :: \mathbb{Z} \rightarrow [a] \rightarrow [a]$ é tal que $\text{take } n \ x$ é o mais longo prefixo da lista x cujo comprimento não excede n . Complete a seguinte formulação de uma propriedade da função take :

```
take m (take n x) = take (m . . . n) x
```

Resolução 2

```
take m (take n x) = take (m `min` n) x
```

Exercício 3

Apresente definições em Haskell das seguintes funções que estudou em PF:

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
uncurry :: (a -> b -> c) -> (a, b) -> c
curry :: ((a, b) -> c) -> a -> b -> c
flip :: (a -> b -> c) -> b -> a -> c
```

Resolução 3

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs)
  | f x = x : filter f xs
  | otherwise = filter f xs
```

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Exercício 4

A **composição** de funções define-se, em Haskell, tal como na matemática:

$$(f \circ g) x = f(g x) \quad (\text{F1})$$

Calcule $(f \circ g) x$ para os casos seguintes:

$$\left\{ \begin{array}{l} f x = 2 \times x \\ g x = x + 1 \end{array} \right\} \quad \left\{ \begin{array}{l} f = succ \\ g x = 2 \times x \end{array} \right\} \quad \left\{ \begin{array}{l} f = succ \\ g = length \end{array} \right\} \quad \left\{ \begin{array}{l} g(x, y) = x + y \\ f = succ \circ (2 \times) \end{array} \right.$$

Anime as composições funcionais acima num interpretador de Haskell.

Resolução 4

a)

$$\begin{array}{ll} f x = 2 \times x & \text{Assume-se que: } f :: \text{Int} \rightarrow \text{Int} \\ g x = x + 1 & g :: \text{Int} \rightarrow \text{Int} \end{array}$$

$$\begin{aligned} (f \circ g) x &= f (g x) && (\text{F1}) \\ &= f (x + 1) && (\text{Def. } g) \\ &= 2 \times (x + 1) && (\text{Def. } f) \\ &= 2x + 2 \end{aligned}$$

b)

$$\begin{array}{ll} f = \text{succ} & \text{Assume-se que: } f :: \text{Int} \rightarrow \text{Int} \\ g \ x = 2 \times x & g :: \text{Int} \rightarrow \text{Int} \end{array}$$

Point-wise notation:

$$\begin{aligned} (f \circ g) \ x &= f \ (g \ x) && \text{(F1)} \\ &= f \ (2 \times x) && \text{(Def. } g) \\ &= \text{succ} \ (2 \times x) && \text{(Def. } f) \\ &= 2x + 1 \end{aligned}$$

Point-free notation:

$$(f \circ g) = \text{succ} \circ (2 \times)$$

Definição de igualdade extensional:

$$\begin{aligned} f = \text{succ} &\equiv \forall x, \ f \ x = \text{succ} \ x && \text{(Igualdade extensional)} \\ g = (2 \times) &\equiv \forall x, \ g \ x = 2 \times x && \text{(Igualdade extensional)} \end{aligned}$$

c)

$$\begin{array}{ll} f = \text{succ} & \text{Assume-se que: } f :: \text{Int} \rightarrow \text{Int} \\ g = \text{length} & g :: [a] \rightarrow \text{Int} \end{array}$$

$$\begin{aligned} (f \circ g) \ x &= f \ (g \ x) && \text{(F1)} \\ &= f \ (\text{length} \ x) && \text{(Def. } g) \\ &= \text{succ} \ (\text{length} \ x) && \text{(Def. } f) \end{aligned}$$

Note que a composição $g \circ f$ não é possível, pois a função f retorna um valor do tipo `Int`, enquanto a função g espera um argumento do tipo `[a]`.

d)

$$\begin{array}{ll} g(x, y) = x + y & \text{Assume-se que: } g :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \\ f = \text{succ} \circ (2 \times) & f :: \text{Int} \rightarrow \text{Int} \end{array}$$

Point-wise notation:

$$\begin{aligned} (f \circ g) x &= f (g (x, y)) && \text{(F1)} \\ &= f (x + y) && \text{(Def. } g) \\ &= (\text{succ} \circ (2 \times)) (x + y) && \text{(Def. } f) \\ &= \text{succ} (2 \times (x + y)) \\ &= 2x + 2y + 1 \end{aligned}$$

Point-free notation:

$$\begin{aligned} g(x, y) &= x + y \\ &= (+) x y \\ &= \text{uncurry } (+) (x, y) \\ &\equiv && \text{(Igualdade extensional)} \\ g &= \text{uncurry } (+) \\ \text{Logo } (f \circ g) &= \text{succ} \circ (2 \times) \circ (\text{uncurry } (+)) \end{aligned}$$

Exercício 5

Mostre que $(f \circ g) \circ h = f \circ (g \circ h)$, quaisquer que sejam f, g e h .

Resolução 5 (dedução)

$$((f \circ g) \circ h) x = (f \circ g) (h x) \quad (\text{F1})$$

$$= f (g (h x)) \quad (\text{F1})$$

$$(f \circ (g \circ h)) x = f ((g \circ h) x) \quad (\text{F1})$$

$$= f (g (h x)) \quad (\text{F1})$$

Assim, conclui-se que $\forall x, ((f \circ g) \circ h) x = (f \circ (g \circ h)) x$
e, por ig. extensional, temos que $(f \circ g) \circ h = f \circ (g \circ h)$

Resolução 5 (indução)

$$\begin{array}{ll} \text{Hipótese de indução} & (f \circ g) \circ h = f \circ (g \circ h) \\ & \equiv \quad \quad \quad (\text{Igualdade extensional}) \end{array}$$

$$\begin{array}{ll} & \forall x, ((f \circ g) \circ h) x = (f \circ (g \circ h)) x \\ & \equiv \quad \quad \quad (\text{F1}) \end{array}$$

$$\begin{array}{ll} & (f \circ g) \circ (h x) = f ((g \circ h) x) \\ & \equiv \quad \quad \quad (\text{F1}) \end{array}$$

$$\begin{array}{ll} & f (g (h x)) = f (g (h x)) \\ & \equiv \end{array}$$

True

$$\text{Logo } (f \circ g) \circ h = f \circ (g \circ h)$$

Exercício 6

A função $\text{id} :: a \rightarrow a$ é tal que $\text{id } x = x$.

Mostre que $f \circ \text{id} = \text{id} \circ f = f$ qualquer que seja f .

Resolução 6

$$\begin{aligned}(f \circ \text{id}) x &= f (\text{id } x) && \text{(F1)} \\ &= f x && \text{(Def. id)} \\ &= \text{id } (f x) && \text{(Def. id)} \\ &= (\text{id} \circ f) x && \text{(F1)}\end{aligned}$$

Como $\forall x \in \text{Dom}(f), (f \circ \text{id}) x = (\text{id} \circ f) x$
implica, por ig. extensional, que $f \circ \text{id} = \text{id} \circ f$
e pela definição de id temos que $f \circ \text{id} = f$
então $f \circ \text{id} = \text{id} \circ f = f$

Resolução 6 (alternativa)

$$\begin{aligned}(f \circ \text{id}) x &= f (\text{id } x) && \text{(F1)} \\ &= f x && \text{(Def. id)} \\ &\equiv && \text{(Igualdade extensional)} \\ f \circ \text{id} &= f\end{aligned}$$

$$\begin{aligned}(\text{id} \circ f) x &= \text{id } (f x) && \text{(F1)} \\ &= f x && \text{(Def. id)} \\ &\equiv && \text{(Igualdade extensional)} \\ \text{id} \circ f &= f\end{aligned}$$

Logo $f \circ \text{id} = \text{id} \circ f = f$

Exercício 7

Considere o seguinte problema:

(...) For each **list of calls** stored in an old mobile phone (eg. numbers dialed, SMS messages, lost calls), the **store** operation should work in a way such that:

a) the more recently a **call** is made the more accessible it is;

b) no number appears twice in a list;

c) only the most recent 10 entries in each list are stored.

Considere ainda a seguinte proposta de resolução que usa a composição de funções, uma por cada requisito do problema:

```
store :: Eq a => a -> [a] -> [a]
store c = take 10 . nub . (c:) -- (F2)
--      (a)      (b)      (c)
```

Resolução 7

a)

Usando a definição (F1) tantas vezes quanto necessário, avalie as expressões:

```
store 7 [1 .. 10]
store 11 [1 .. 10]
```

```
store 7 [1 .. 10] = take 10 . nub . (7:) $ [1 .. 10]
                  = take 10 . nub . (7:) $ [1,2,3,4,5,6,7,8,9,10]
                  = take 10 . nub $      [7,1,2,3,4,5,6,7,8,9,10]
                  = take 10 $            [7,1,2,3,4,5,6,8,9,10]
                  =                      [7,1,2,3,4,5,6,8,9,10]
```

```
store 11 [1 .. 10] = take 10 . nub . (11:) $ [1 .. 10]
                   = take 10 . nub . (11:) $ [1,2,3,4,5,6,7,8,9,10]
                   = take 10 . nub $      [11,1,2,3,4,5,6,7,8,9,10]
                   = take 10 $            [11,1,2,3,4,5,6,7,8,9,10]
                   =                      [11,1,2,3,4,5,6,7,8,9]
```


b)

Suponha que alguém usou a mesma abordagem ao problema, mas enganou-se na ordem das etapas:

```
store c = (c:) . take 10 . nub
```

Qual é o problema desta solução? Que requisitos (a,b,c) viola?

Por esta ordem, a função viola dois dos requisitos definidos:

- **b)** Devido a inserção do elemento `c` ser após a chamada a função `nub`, a função `store` não garante que não existem elementos repetidos.
 - **c)** Como a inserção do elemento `c` é feita depois da chamada a `take 10`, a função `store` não garante que apenas as 10 entradas mais recentes são guardadas.
-

c)

E se o engano for como escreve a seguir?

```
store c = nub . (c:) . take 10
```

Conclua que a composição não é mesmo nada comutativa - a ordem entre as etapas de uma solução composicional é importante!

Neste caso, a função `store` viola o requisito **c)**, como a função `take 10` é chamada antes de ser feita a inserção de `c`, em certos casos a função `store` retorna uma lista com 11 elementos, explicitamente quando é adicionado um elemento `c` a uma lista de 10 elementos sem repetições e `c` não pertence a essa lista.

Exercício 8

Voltando a agora à definição *certa* (F2), suponha que submete ao seu interpretador de Haskell a expressão:

```
store "Maria" ["Manuel", "Tia Irene", "Maria", "Augusto"]
```

Que espera do resultado? Vai dar erro? Tem que mexer em (F2) para funcionar? Que propriedade da linguagem é evidenciada neste exemplo?

Resolução 8

Resultado esperado: ["Maria", "Manuel", "Tia Irene", "Augusto"]

```
store "Maria" ["Manuel", "Tia Irene", "Maria", "Augusto"]
= take 10 . nub . ("Maria":) $ ["Manuel", "Tia Irene", "Maria", "Augusto"]
= take 10 . nub $               ["Maria", "Manuel", "Tia Irene", "Maria", "Augusto"]
= take 10 $                     ["Maria", "Manuel", "Tia Irene", "Augusto"]
=                               ["Maria", "Manuel", "Tia Irene", "Augusto"]
```

Não ocorre erro, pois a função `store` foi definida para lidar com listas de qualquer tipo, desde que este seja uma instância de `Eq`, visto que é usada a função `nub`, que requer a comparação de elementos.

Em Haskell, `String` (uma lista de caracteres) é uma instância de `Eq`, o que significa que `nub` pode operar corretamente sobre ela.

Deste modo, não é necessário alterar a função `store` para que esta funcione corretamente.

A propriedade de Haskell evidenciada neste exemplo é o **polimorfismo (ad-hoc)**.