CP - Ficha 1

Exercício 1

> Complete a codificação abaixo (em Haskell) das funções $\text{length} :: [a] \to \mathbb{Z}$ e $\text{reverse} :: [a] \to [a]$ que, respetivamente, calculam o comprimento da lista de entrada e a invertem:
>
> ```haskell
> length [] = ...
> length (x:xs) = ...
> reverse [] = ...
> reverse (x:xs) = ...
> ```

Resolução 1

```haskell
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 $\text{take} :: \mathbb{Z} \to [a] \to [a]$ 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:
>
> ```haskell
> take m (take n x ) = take (m . . . n) x
> ```

Resolução 2

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

Exercício 3

> Apresente definições em Haskell das seguintes funções que estudou em PF:
>
> ```haskell
> 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

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

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

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

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

```haskell
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:
>
> $$(f \circ g)\ x = f(g\ x) \tag{F1}$$
>
> Calcule $(f \circ g)\ x$ para os casos seguintes:
>
> $$\begin{cases} f\ x = 2 * x \\ g\ x = x + 1 \end{cases} \quad \begin{cases} f = succ \\ g\ x = 2 * x \end{cases} \quad \begin{cases} f = succ \\ g = length \end{cases} \quad \begin{cases} g\ (x, y) = x + y \\ f = succ \circ (2*) \end{cases}$$
>
> Anime as composições funcionais acima num interpretador de Haskell.

Resolução 4

a)

$$\begin{aligned} f\ x &= 2 * x & \text{Assume-se que: } f &:: \mathrm{Int} \to \mathrm{Int} \\ g\ x &= x + 1 & g &:: \mathrm{Int} \to \mathrm{Int} \end{aligned}$$

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

---

b)

$$f = \text{succ} \qquad \text{Assume-se que: } f :: \text{Int} \to \text{Int}$$
$$g\ x = 2 * x \qquad g :: \text{Int} \to \text{Int}$$

Point-wise notation:

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

Point-free notation:

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

Definição de igualdade extensional:

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

---

c)

$$f = \text{succ} \qquad \text{Assume-se que: } f :: \text{Int} \to \text{Int}$$
$$g = \text{length} \qquad g :: [a] \to \text{Int}$$

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

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)

$$\begin{aligned} g\ (x,y) &= x + y \qquad \text{Assume-se que: } g :: (\mathrm{Int}, \mathrm{Int}) \rightarrow \mathrm{Int} \\ f &= \mathrm{succ} \circ (2*) \qquad f :: \mathrm{Int} \rightarrow \mathrm{Int} \end{aligned}$$

Point-wise notation:

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

Point-free notation:

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

Exercício 5

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

Resolução 5

$$\begin{aligned} ((f \circ g) \circ h)\ x &= (f \circ g)\ (h\ x) && \text{(F1)} \\ &= f\ (g\ (h\ x)) && \text{(F1)} \end{aligned}$$

$$\begin{aligned} (f \circ (g \circ h))\ x &= f\ ((g \circ h)\ x) && \text{(F1)} \\ &= f\ (g\ (h\ x)) && \text{(F1)} \end{aligned}$$

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

Resolução 5 (alternativa)

$$\begin{aligned} \text{Assume-se que} \quad & (f \circ g) \circ h = f \circ (g \circ h) \\ & \equiv \qquad \text{(Igualdade extensional)} \\ & \forall x,\ ((f \circ g) \circ h)\ x = (f \circ (g \circ h))\ x \\ & \equiv \qquad \text{(F1)} \\ & (f \circ g) \circ (h\ x) = f\ ((g \circ h)\ x) \\ & \equiv \qquad \text{(F1)} \\ & f\ (g\ (h\ x)) = f\ (g\ (h\ x)) \\ & \equiv \\ & \text{True} \\ \text{Logo } & (f \circ g) \circ h = f \circ (g \circ h) \end{aligned}$$

Exercício 6

> A função $\mathrm{id} :: a \to a$ é tal que $\mathrm{id}\ x = x$.
>
> Mostre que $f \circ \mathrm{id} = \mathrm{id} \circ f = f$ qualquer que seja f.

Resolução 6

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

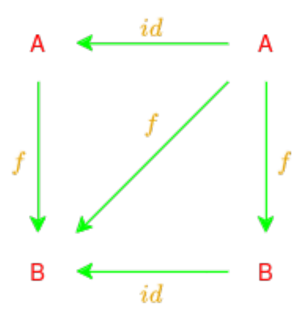
$$
\begin{aligned}
\text{Como } \forall x \in \mathrm{Dom}(f),\ (f \circ \mathrm{id})\ x = (\mathrm{id} \circ f)\ x \\
\text{implica, por ig. extensional, que } f \circ \mathrm{id} = \mathrm{id} \circ f \\
\text{e pela definição de id temos que } f \circ \mathrm{id} = f \\
\text{então } f \circ \mathrm{id}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

Resolução 6 (alternativa)

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

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

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



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:
>
> ```haskell
> 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:
>
> ```haskell
> store 7 [1 .. 10]
> store 11 [1 .. 10]
> ```

```haskell
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]
```

```haskell
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:
>
> ```haskell
> 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"]`

```haskell
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 paramétrico**.