Princípios de Programação

Exercícios

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

2019/2020

Construção de tipos e de classes de tipos

1. Defina um tipo de dados que descreva as seguintes formas geométricas: círculo, rectângulo e triângulo. Escreva funções para calcular o perímetro e para verificar se uma figura é regular (uma forma é regular se todos os seus ângulos são iguais e todos os lados são iguais).

2. Considerando o tipo de dados `Nat`:

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

   escreva as seguintes funções:

   (a) `add :: Nat -> Nat -> Nat`, a soma de dois naturais,
   (b) `monus :: Nat -> Nat -> Nat`, a subtracção natural, i.e., se o 2º natural for maior que o 1º, a função é igual a zero,
   (c) `pred :: Nat -> Nat`, calcule o predecessor do natural dado. Esta função deverá estar indefinida para o natural zero,
   (d) `sub :: Nat -> Nat -> Nat`, a diferença de dois naturais. Use a função `pred`. Esta função deverá estar indefinida para valores do primeiro argumento menores que o segundo argumento,
   (e) `mult :: Nat -> Nat -> Nat`, o produto de dois naturais. Utilize a função `add`,
   (f) `pot :: Nat -> Nat -> Nat`, a potência do 1º natural elevado ao 2º natural,
   (g) `fact :: Nat -> Nat`, o factorial do natural dado,
   (h) `remnat :: Nat -> Nat -> Nat`, o resto da divisão inteira entre o 1º e o 2º natural,

(i) `quotnat :: Nat -> Nat -> Nat`, o quociente da divisão inteira entre o 1º e o 2º natural,
(j) `lessThan :: Nat -> Nat -> Bool`, verifica se o 1º natural é menor que o 2º natural.

3. Para o exercício sobre conjuntos da secção anterior, escolha um tipo de dados mais apropriado, um tipo de dados que melhore a complexidade (o o-grande, $\mathcal{O}$) das várias operações. Reescreva depois o módulo de modo a esconder a estrutura do tipo de dados.

4. Para o exercício sobre mapas da secção anterior, escreva um tipo de dados `Map` mais apropriado, um tipo de dados que melhore a complexidade (o o-grande, $\mathcal{O}$) das várias operações. Reescreva depois o módulo de modo a esconder a estrutura do tipo de dados.

5. Utilizando o tipo de dados

```
data Tree a = EmptyTree | Node (Tree a) a (Tree a)
```

escreva as funções abaixo.

(a) `empty :: Tree a` , uma árvore vazia
(b) `size :: Tree a -> Int`, o número de nós na árvore.
(c) `depth :: Tree a -> Int`, a profundidade da árvore. A profundidade de uma árvore vazia é zero; aquela de uma árvore não vazia é um mais o máximo das profundidades das sub árvores.
(d) `flatten :: Tree a -> [a]`, a lista dos elementos da árvore visitados pelo percurso prefixo. O percurso prefixo de uma árvore não vazia visita primeiro o elemento do nó, depois a sub árvore esquerda e finalmente a sub árvore direita.
(e) `isPerfect :: Tree a -> Bool`, a árvore é perfeita? Uma árvore vazia é considerada perfeita. Uma árvore não vazia diz-se perfeita se as duas sub árvores são perfeitas e têm o mesmo número de nós. Numa primeira fase, resolva este exercício recorrendo à função `size`. Analise a sua complexidade. Desenhe depois uma solução que não percorra a árvore mais do que uma vez.
(f) `invert :: Tree a -> Tree a`, a árvore onde cada sub árvore esquerda é trocada pela sub árvore direita.
(g) `makeTree :: [a] -> Tree a`, a árvore sintetizada a partir de uma lista de elementos da seguinte forma: a cabeça da lista é a raiz da árvore. Dos restantes elementos, a 1ª metade constrói recursivamente a sub árvore da esquerda e a 2ª metade a sub árvore da direita.
(h) `isIn :: Eq a =>a -> Tree a -> Bool` que verifique se um dado elemento consta de uma árvore.

(i) `allIn :: Eq a =>Tree a -> Tree a -> Bool` que verifique se todos os elementos de uma dada árvore constam de uma outra árvore.

6. Reescreva as funções acima recorrendo à seguinte função `fold`.

```
fold :: (b -> a -> b -> b) -> b -> Tree a -> b
fold _ e EmptyTree = e
fold f e (Node l x r) = f (fold f e l) x (fold f e r)
```

7. Torne o tipo de dados `Tree` instância da classe **`Eq`**. Para efeitos deste exercício duas árvores são iguais se contiverem os mesmos elementos.

8. Torne o tipo de dados `Tree` instância da classe **`Show`**. A conversão de uma árvore numa **`String`** deverá ser tal que a árvore

```
Node (Node EmptyTree "cao" (Node EmptyTree "gato"
  EmptyTree)) "peixe" (Node EmptyTree "pulga"
     EmptyTree)
```

seja convertida em

```
"peixe"
  "cao"
    Empty
    "gato"
      Empty
      Empty
  "pulga"
    Empty
    Empty
```

9. Torne o tipo `Tree` instância da class **`Functor`**. Exemplo:

```
*Set> fmap (^2) (Node (Node EmptyTree 4 EmptyTree) 5
    (Node EmptyTree 6 EmptyTree))
25
  16
    Empty
    Empty
  36
    Empty
    Empty
```

10. Torne o tipo `Set` do capítulo anterior instância das classes **`Eq`** e **`Show`**. No caso de **`Show`** represente o conjunto com elementos separados por vírgulas e entre parêntesis, como é habitual em matemática.

```
*Set> fromList [1,5..30]
{1,5,9,13,17,21,25,29}
```

11. Torne o tipo `Set` do capítulo anterior instância da class **`Functor`**. Exemplo:

```
*Set> fmap (*2) (fromList [1,5..30])
{2,10,18,26,34,42,50,58}
```

12. Torne o tipo `Map` do capítulo anterior instância da classe **`Eq`**. Dois mapas são iguais se contiverem as mesmas chaves, e para cada chave apresentem valores iguais.

13. Torne o tipo `Map` do capítulo anterior instância da classe **`Show`**. Ao mapa com duas entradas `("a",1)` e `("b",2)` deve corresponder a *string* `{"a": 1, "b": 2}`.

14. Considere a classe `Visible` definida da seguinte forma:

```
class Visible a where
  toString  :: a -> String
  dimension :: a -> Int
```

Crie instâncias desta classe para os tipos **`Char`**, **`Bool`**, lista de `Visible` e pares de `Visible`.

15. Complete as seguintes declarações.

```
instance (Ord a, Ord b) => Ord (a,b) where ...
instance Ord a => Ord [a] where ...
```