

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2017/18

Departamento de Informática  
Universidade do Minho

Junho de 2018

**Grupo nr.** (37)

---

a78679	Diana Ribeiro Barbosa
a78806	José Pedro Ferreira de Oliveira
a77377	Pedro Henrique Moreira Gomes Fernandes

## 1 Preâmbulo

A disciplina de **Cálculo de Programas** tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método à programação funcional em **Haskell**. Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em **Haskell**. Há ainda um outro objectivo: o de ensinar a documentar programas e a produzir textos técnico-científicos de qualidade.

## 2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita “**literária**” [?], cujo princípio base é o seguinte:

*Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a documentação de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp1718t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp1718t.lhs`<sup>1</sup> que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp1718t.zip` e executando

```
$ lhs2TeX cp1718t.lhs > cp1718t.tex
$ pdflatex cp1718t
```

em que **lhs2tex** é um pre-processador que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

Por outro lado, o mesmo ficheiro `cp1718t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp1718t.lhs
```

Abra o ficheiro `cp1718t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

---

<sup>1</sup>O sufixo ‘lhs’ quer dizer *literate Haskell*.

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo **GHCi** para ser executado.

### 3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na **página da disciplina** na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo ?? com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com **BibTeX**) e o índice remissivo (com **makeindex**),

```
$ bibtex cp1718t.aux
$ makeindex cp1718t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário **QuickCheck**, que ajuda a validar programas em **Haskell**, a biblioteca **JuicyPixels** para processamento de imagens e a biblioteca **gloss** para geração de gráficos 2D:

```
$ cabal install QuickCheck JuicyPixels gloss
```

Para testar uma propriedade **QuickCheck** *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

### Problema 1

Segundo uma **notícia do Jornal de Notícias**, referente ao dia 12 de abril, “*apenas numa hora, foram transacionadas 1.2 mil milhões de dólares em bitcoins. Nas últimas 24 horas, foram transacionados 8,5 mil milhões de dólares, num total de 24 mil milhões de dólares referentes às principais criptomoedas*”.

De facto, é inquestionável que as criptomoedas, e em particular as bitcoin, vieram para ficar. Várias moedas digitais, e em particular as bitcoin, usam a tecnologia de block chain para guardar e assegurar todas as transações relacionadas com a moeda. Uma **block chain** é uma coleção de blocos que registam os movimentos da moeda; a sua definição em Haskell é apresentada de seguida.

```
data Blockchain = Bc { bc :: Block } | Bcs { bcs :: (Block, Blockchain) } deriving Show
```

Cada **bloco** numa block chain regista um número (mágico) único, o momento da execução, e uma lista de transações, tal como no código seguinte:

```
type Block = (MagicNo, (Time, Transactions))
```

Cada **transação** define a entidade de origem da transferência, o valor a ser transacionado, e a entidade destino (por esta ordem), tal como se define de seguida.

```
type Transaction = (Entity, (Value, Entity))
type Transactions = [Transaction]
```

A partir de uma block chain, é possível calcular o valor que cada entidade detém, tipicamente designado de ledger:

```
type Ledger = [(Entity, Value)]
```

Seguem as restantes definições Haskell para completar o código anterior. Note que *Time* representa o momento da transação, como o número de **milissegundos** que passaram desde 1970.

```

type MagicNo = String
type Time = Int -- em milisegundos
type Entity = String
type Value = Int

```

Neste contexto, implemente as seguintes funções:

1. Defina a função  $allTransactions :: Blockchain \rightarrow Transactions$ , como um catamorfismo, que calcula a lista com todas as transações numa dada block chain.

**Propriedade QuickCheck 1** *As transações de uma block chain são as mesmas da block chain revertida:*

$$prop1a = sort \cdot allTransactions \equiv sort \cdot allTransactions \cdot reverseChain$$

*Note que a função sort é usada apenas para facilitar a comparação das listas.*

2. Defina a função  $ledger :: Blockchain \rightarrow Ledger$ , utilizando catamorfismos e/ou anamorfismos, que calcula o ledger (i.e., o valor disponível) de cada entidade numa dada block chain. Note que as entidades podem ter valores negativos; de facto isso acontecerá para a primeira transação que executarem.

**Propriedade QuickCheck 2** *O tamanho do ledger é inferior ou igual a duas vezes o tamanho de todas as transações:*

$$prop1b = length \cdot ledger \leq (2*) \cdot length \cdot allTransactions$$

**Propriedade QuickCheck 3** *O ledger de uma block chain é igual ao ledger da sua inversa:*

$$prop1c = sort \cdot ledger \equiv sort \cdot ledger \cdot reverseChain$$

3. Defina a função  $isValidMagicNr :: Blockchain \rightarrow Bool$ , utilizando catamorfismos e/ou anamorfismos, que verifica se todos os números mágicos numa dada block chain são únicos.

**Propriedade QuickCheck 4** *A concatenação de uma block chain com ela mesma nunca é válida em termos de números mágicos:*

$$prop1d = \neg \cdot isValidMagicNr \cdot concChain \cdot \langle id, id \rangle$$

**Propriedade QuickCheck 5** *Se uma block chain é válida em termos de números mágicos, então a sua inversa também o é:*

$$prop1e = isValidMagicNr \Rightarrow isValidMagicNr \cdot reverseChain$$

## Problema 2

Uma estrutura de dados frequentemente utilizada para representação e processamento de imagens de forma eficiente são as denominadas **quadtrees**. Uma *quadtrees* é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

```

data QTree a = Cell a Int Int | Block (QTree a) (QTree a) (QTree a) (QTree a)
deriving (Eq, Show)

```

Uma imagem monocromática em formato bitmap pode ser representada como uma matriz de bits<sup>2</sup>, tal como se exemplifica na Figura ??.

O anamorfismo  $bm2qt$  converte um bitmap em forma matricial na sua codificação eficiente em quadtrees, e o catamorfismo  $qt2bm$  executa a operação inversa:

---

<sup>2</sup>Cf. módulo *Data.Matrix*.

( 0 0 0 0 0 0 0 0 )	Block
( 0 0 0 0 0 0 0 0 )	(Cell 0 4 4) (Block
( 0 0 0 0 1 1 1 0 )	(Cell 0 2 2) (Cell 0 2 2) (Cell 1 2 2) (Block
( 0 0 0 0 1 1 0 0 )	(Cell 1 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1)))
( 1 1 1 1 1 1 0 0 )	(Cell 1 4 4)
( 1 1 1 1 1 1 0 0 )	(Block
( 1 1 1 1 0 0 0 0 )	(Cell 1 2 2) (Cell 0 2 2) (Cell 0 2 2) (Block
( 1 1 1 1 0 0 0 1 )	(Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 1 1 1)))

(a) Matriz de exemplo *bm*.

(b) Quadtree de exemplo *qt*.

Figure 1: Exemplos de representações de bitmaps.

$bm2qt :: (Eq\ a) \Rightarrow Matrix\ a \rightarrow QTree\ a$	$qt2bm :: (Eq\ a) \Rightarrow QTree\ a \rightarrow Matrix\ a$
$bm2qt = anaQTree\ f\ \textbf{where}$	$qt2bm = cataQTree\ [f, g]\ \textbf{where}$
$f\ m = \textbf{if}\ one\ \textbf{then}\ i_1\ u\ \textbf{else}\ i_2\ (a, (b, (c, d)))$	$f\ (k, (i, j)) = matrix\ j\ i\ k$
$\textbf{where}\ x = (nub \cdot toList)\ m$	$g\ (a, (b, (c, d))) = (a \uparrow b) \leftrightarrow (c \uparrow d)$
$u = (head\ x, (ncols\ m, nrows\ m))$	
$one = (ncols\ m \equiv 1 \vee nrows\ m \equiv 1 \vee length\ x \equiv 1)$	
$(a, b, c, d) = splitBlocks\ (nrows\ m \div 2)\ (ncols\ m \div 2)\ m$	

O algoritmo *bm2qt* particiona recursivamente a imagem em 4 blocos e termina produzindo folhas para matrizes unitárias ou quando todos os píxeis de um sub-bloco têm a mesma cor. Para a matriz *bm* de exemplo, a quadtree correspondente *qt* = *bm2qt* *bm* é ilustrada na Figura ??.

Imagens a cores podem ser representadas como matrizes de píxeis segundo o código de cores **RGBA**, codificado no tipo *PixelRGBA8* em que cada pixel é um quádruplo de valores inteiros (*red, green, blue, alpha*) contidos entre 0 e 255. Atente em alguns exemplos de cores:

```
whitePx = PixelRGBA8 255 255 255 255
blackPx = PixelRGBA8 0 0 0 255
redPx   = PixelRGBA8 255 0 0 255
```

O módulo *BMP*, disponibilizado juntamente com o enunciado, fornece funções para processar ficheiros de imagem bitmap como matrizes:

```
readBMP :: FilePath → IO (Matrix PixelRGBA8)
writeBMP :: FilePath → Matrix PixelRGBA8 → IO ()
```

Teste, por exemplo, no *GHCi*, carregar a Figura ??:

```
> readBMP "cp1718t_media/person.bmp"
```

Esta questão aborda operações de processamento de imagens utilizando quadrees:

1. Defina as funções  $rotateQTree :: QTree\ a \rightarrow QTree\ a$ ,  $scaleQTree :: Int \rightarrow QTree\ a \rightarrow QTree\ a$  e  $invertQTree :: QTree\ a \rightarrow QTree\ a$ , como catamorfismos e/ou anamorfismos, que rodam<sup>3</sup>, redimensionam<sup>4</sup> e invertem as cores de uma quadtree<sup>5</sup>, respectivamente. Tente produzir imagens similares às Figuras ??, ?? e ??:

```
> rotateBMP "cp1718t_media/person.bmp" "person90.bmp"
> scaleBMP 2 "cp1718t_media/person.bmp" "personx2.bmp"
> invertBMP "cp1718t_media/person.bmp" "personinv.bmp"
```

**Propriedade QuickCheck 6** Rodar uma quadtree é equivalente a rodar a matriz correspondente:

$$prop2c = rotateMatrix \cdot qt2bm \equiv qt2bm \cdot rotateQTree$$

<sup>3</sup>Segundo um ângulo de 90° no sentido dos ponteiros do relógio.

<sup>4</sup>Multiplicando o seu tamanho pelo valor recebido.

<sup>5</sup>Um pixel pode ser invertido calculando  $255 - c$  para cada componente *c* de cor RGB, exceptuando o componente alpha.



(a) Bitmap de exemplo.



(b) Rotação.



(c) Redimensionamento.



(d) Inversão de cores.



(e) Compressão de 1 nível.



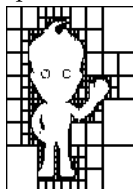
(f) Compressão de 2 níveis.



(g) Compressão de 3 níveis.



(h) Compressão de 4 níveis.



(i) Bitmap de contorno.



(j) Bitmap com contorno.

Figure 2: Manipulação de uma figura bitmap utilizando quadrees.

**Propriedade QuickCheck 7** Redimensionar uma imagem altera o seu tamanho na mesma proporção:

$$\text{prop2d } (\text{Nat } s) = \text{sizeQTree} \cdot \text{scaleQTree } s \equiv ((s*) \times (s*)) \cdot \text{sizeQTree}$$

**Propriedade QuickCheck 8** Inverter as cores de uma quadtree preserva a sua estrutura:

$$\text{prop2e} = \text{shapeQTree} \cdot \text{invertQTree} \equiv \text{shapeQTree}$$

2. Defina a função  $\text{compressQTree} :: \text{Int} \rightarrow \text{QTree } a \rightarrow \text{QTree } a$ , utilizando catamorfismos e/ou anamorfismos, que comprime uma quadtree cortando folhas da árvore para reduzir a sua profundidade num dado número de níveis. Tente produzir imagens similares (mas não necessariamente iguais) às Figuras ??, ??, ?? e ??:

```
> compressBMP 1 "cp1718t_media/person.bmp" "person1.bmp"
> compressBMP 2 "cp1718t_media/person.bmp" "person2.bmp"
> compressBMP 3 "cp1718t_media/person.bmp" "person3.bmp"
> compressBMP 4 "cp1718t_media/person.bmp" "person4.bmp"
```

**Propriedade QuickCheck 9** A quadtree comprimida tem profundidade igual à da quadtree original menos a taxa de compressão:

$$\text{prop2f } (\text{Nat } n) = \text{depthQTree} \cdot \text{compressQTree } n \equiv (-n) \cdot \text{depthQTree}$$

3. Defina a função  $\text{outlineQTree} :: (a \rightarrow \text{Bool}) \rightarrow \text{QTree } a \rightarrow \text{Matrix Bool}$ , utilizando catamorfismos e/ou anamorfismos, que recebe uma função que determina quais os píxeis de fundo e converte uma quadtree numa matriz monocromática, de forma a desenhar o contorno de uma **malha poligonal** contida na imagem. Tente produzir imagens similares (mas não necessariamente iguais) às Figuras ?? e ??:

```
> outlineBMP      "cp1718t_media/person.bmp" "personOut1.bmp"
> addOutlineBMP   "cp1718t_media/person.bmp" "personOut2.bmp"
```

**Propriedade QuickCheck 10** A matriz de contorno tem dimensões iguais às da quadtree:

$$\text{prop2g} = \text{sizeQTree} \equiv \text{sizeMatrix} \cdot \text{outlineQTree } (<0)$$

**Teste unitário 1** Contorno da quadtree de exemplo qt:

$$\text{teste2a} = \text{outlineQTree } (\equiv 0) \text{ qt} \equiv \text{qtOut}$$

## Problema 3

O cálculo das combinações de  $n$   $k$ -a- $k$ ,

$$\binom{n}{k} = \frac{n!}{k! * (n - k)!} \quad (1)$$

envolve três factoriais. Recorrendo à **lei de recursividade múltipla** do cálculo de programas, é possível escrever o mesmo programa como um simples ciclo-for onde se fazem apenas multiplicações e somas. Para isso, começa-se por estruturar a definição dada da forma seguinte,

$$\binom{n}{k} = h \ k \ (n - k)$$

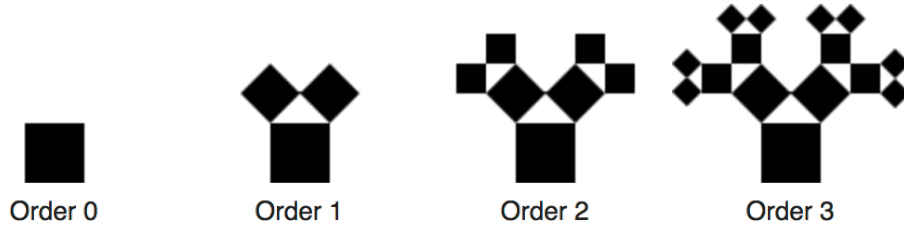


Figure 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

$$\begin{aligned} h \ k \ d &= \frac{f \ k \ d}{g \ d} \\ f \ k \ d &= \frac{(d+k)!}{k!} \\ g \ d &= d! \end{aligned}$$

assumindo-se  $d = n - k \geq 0$ . É fácil de ver que  $f \ k$  e  $g$  se desdobram em 4 funções mutuamente recursivas, a saber

$$\begin{aligned} f \ k \ 0 &= 1 \\ f \ k \ (d+1) &= \underbrace{(d+k+1)}_{l \ k \ d} * f \ k \ d \\ l \ k \ 0 &= k+1 \\ l \ k \ (d+1) &= l \ k \ d + 1 \end{aligned}$$

e

$$\begin{aligned} g \ 0 &= 1 \\ g \ (d+1) &= \underbrace{(d+1)}_{s \ d} * g \ d \\ s \ 0 &= 1 \\ s \ (d+1) &= s \ n + 1 \end{aligned}$$

A partir daqui alguém derivou a seguinte implementação:

$$\binom{n}{k} = h \ k \ (n-k) \text{ where } h \ k \ n = \text{let } (a, -, b, -) = \text{for loop } (base \ k) \ n \text{ in } a / b$$

Aplicando a lei da recursividade múltipla para  $\langle f \ k, l \ k \rangle$  e para  $\langle g, s \rangle$  e combinando os resultados com a **lei de banana-split**, derive as funções *base k* e *loop* que são usadas como auxiliares acima.

**Propriedade QuickCheck 11** Verificação que  $\binom{n}{k}$  coincide com a sua especificação (??):

$$prop3 \ (NonNegative \ n) \ (NonNegative \ k) = k \leq n \Rightarrow \binom{n}{k} \equiv n! / (k! * (n-k)!)$$

## Problema 4

**Fractais** são formas geométricas que podem ser construídas recursivamente de acordo com um conjunto de equações matemáticas. Um exemplo clássico de um fractal são as **árvores de Pitágoras**. A construção de uma árvore de Pitágoras começa com um quadrado, ao qual se unem dois quadrados redimensionados pela escala  $\sqrt{2}/2$ , de forma a que os cantos dos 3 quadrados coincidam e formem um triângulo rectângulo isósceles. Este procedimento é repetido recursivamente de acordo com uma dada ordem, definida como um número natural (Figura ??).

Uma árvore de Pitágoras pode ser codificada em Haskell como uma full tree contendo quadrados nos nodos e nas folhas, sendo um quadrado definido simplesmente pelo tamanho do seu lado:

```
data FTree a b = Unit b | Comp a (FTree a b) (FTree a b) deriving (Eq, Show)
type PTree = FTree Square Square
type Square = Float
```

1. Defina a função `generatePTree :: Int → PTree`, como um anamorfismo, que gera uma árvore de Pitágoras para uma dada ordem.

**Propriedade QuickCheck 12** Uma árvore de Pitágoras tem profundidade igual à sua ordem:

$$\text{prop4a } (\text{SmallNat } n) = (\text{depthFTree} \cdot \text{generatePTree}) \, n \equiv n$$

**Propriedade QuickCheck 13** Uma árvore de Pitágoras está sempre balanceada:

$$\text{prop4b } (\text{SmallNat } n) = (\text{isBalancedFTree} \cdot \text{generatePTree}) \, n$$

2. Defina a função `drawPTree :: PTree → [Picture]`, utilizando catamorfismos e/ou anamorfismos, que anima incrementalmente os passos de construção de uma árvore de Pitágoras recorrendo à biblioteca `gloss`. Anime a sua solução:

```
> animatePTree 3
```

## Problema 5

Uma das áreas em maior expansão no campo da informática é a análise de dados e `machine learning`. Esta questão aborda um *mónade* que ajuda a fazer, de forma simples, as operações básicas dessas técnicas. Esse *mónade* é conhecido por *bag*, *saco* ou *multi-conjunto*, permitindo que os elementos de um conjunto tenham multiplicidades associadas. Por exemplo, seja

```
data Marble = Red | Pink | Green | Blue | White deriving (Read, Show, Eq, Ord)
```

um tipo dado.<sup>6</sup> A lista `[Pink, Green, Red, Blue, Green, Red, Green, Pink, Blue, White]` tem elementos repetidos. Assumindo que a ordem não é importante, essa lista corresponde ao saco

```
{ Red |-> 2 , Pink |-> 2 , Green |-> 3 , Blue |-> 2 , White |-> 1 }
```

que habita o tipo genérico dos “bags”:

```
data Bag a = B [(a, Int)] deriving (Ord)
```

O *mónade* que vamos construir sobre este tipo de dados faz a gestão automática das multiplicidades. Por exemplo, seja dada a função que dá o peso de cada berlinde em gramas:

```
marbleWeight :: Marble → Int
marbleWeight Red   = 3
marbleWeight Pink  = 2
marbleWeight Green = 3
marbleWeight Blue  = 6
marbleWeight White = 2
```

Então, se quisermos saber quantos *berlindes* temos, de cada *peso*, não teremos que fazer contas: basta calcular

```
marbleWeights = fmap marbleWeight bagOfMarbles
```

onde `bagOfMarbles` é o saco de berlindes referido acima, obtendo-se:

---

<sup>6</sup>“Marble” traduz para “berlinde” em português.



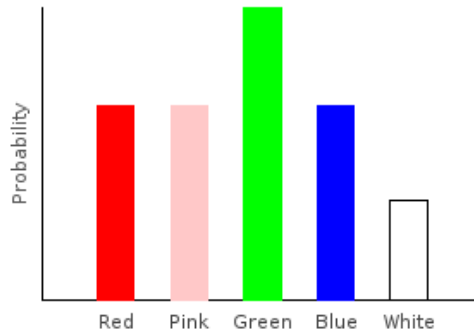


Figure 4: Distribuição de berlines num saco.

$\{ 2 \mapsto 3, 3 \mapsto 5, 6 \mapsto 2 \}$ .

Mais ainda, se quisermos saber o total de berlines em *bagOfMarbles* basta calcular `fmap (!) bagOfMarbles` obtendo-se  $\{ () \mapsto 10 \}$ ; isto é, o saco tem 10 berlines no total.

Finalmente, se quisermos saber a probabilidade da cor de um berline que tiremos do saco, basta converter o referido saco numa distribuição correndo:

```
marblesDist = dist bagOfMarbles
```

obtendo-se a distribuição (graças ao módulo **Probability**):

```
Green  30.0%
Red    20.0%
Pink   20.0%
Blue   20.0%
White  10.0%
```

cf. Figura ??.

Partindo da seguinte declaração de *Bag* como um functor e como um mónade,

```
instance Functor Bag where
  fmap f = B · map (f × id) · unB
instance Monad Bag where
  x >>= f = (μ · fmap f) x where
  return = singletonbag
```

1. Defina a função  $\mu$  (multiplicação do mónade *Bag*) e a função auxiliar *singletonbag*.
2. Verifique-as com os seguintes testes unitários:

**Teste unitário 2** Lei  $\mu \cdot \text{return} = \text{id}$ :

$$\text{test5a} = \text{bagOfMarbles} \equiv \mu (\text{return bagOfMarbles})$$

**Teste unitário 3** Lei  $\mu \cdot \mu = \mu \cdot \text{fmap } \mu$ :

$$\text{test5b} = (\mu \cdot \mu) b3 \equiv (\mu \cdot \text{fmap } \mu) b3$$

onde *b3* é um saco dado em anexo.

# Anexos

## A Mónade para probabilidades e estatística

Mónades são funtores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca **Probability** oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

```
newtype Dist a = D { unD :: [(a, ProbRep)] } (2)
```

em que *ProbRep* é um real de 0 a 1, equivalente a uma escala de 0 a 100/.

Cada par  $(a, p)$  numa distribuição  $d :: \text{Dist } a$  indica que a probabilidade de  $a$  é  $p$ , devendo ser garantida a propriedade de que todas as probabilidades de  $d$  somam 100/. Por exemplo, a seguinte distribuição de classificações por escalões de  $A$  a  $E$ ,



será representada pela distribuição

```
d1 :: Dist Char
d1 = D [ ('A', 0.02), ('B', 0.12), ('C', 0.29), ('D', 0.35), ('E', 0.22) ]
```

que o **GHCI** mostrará assim:

```
'D'  35.0%
'C'  29.0%
'E'  22.0%
'B'  12.0%
'A'   2.0%
```

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular de programação monádica.

## B Definições auxiliares

Funções para mostrar *bags*:

```
instance (Show a, Ord a, Eq a) => Show (Bag a) where
  show = showbag . consol . unB where
    showbag = concat .
      (++) [ " } " ] . ( " { " : ) .
      (intersperse " , " ) .
      sort .
      (map f) where f (a, b) = (show a) ++ " | -> " ++ (show b)
  unB (B x) = x
```

Igualdade de *bags*:

```
instance (Eq a) => Eq (Bag a) where
  b == b' = (unB b) 'lequal' (unB b')
  where lequal a b = isempty (a ⊖ b)
        ominus a b = a ++ neg b
        neg x = [(k, -i) | (k, i) <- x]
```

Ainda sobre o mónade *Bag*:

**instance** *Applicative Bag* **where**

*pure* = *return*  
 (< \* >) = *aap*

O exemplo do texto:

*bagOfMarbles* = *B* [(*Pink*, 2), (*Green*, 3), (*Red*, 2), (*Blue*, 2), (*White*, 1)]

Um valor para teste (bags de bags de bags):

*b3* :: *Bag* (*Bag* (*Bag* *Marble*))  
*b3* = *B* [(*B* [(*B* [(*Pink*, 2), (*Green*, 3), (*Red*, 2), (*Blue*, 2), (*White*, 1)], 5)  
 , (*B* [(*Pink*, 1), (*Green*, 2), (*Red*, 1), (*Blue*, 1)], 2)], 2)]

Outras funções auxiliares:

*a* ↦ *b* = (*a*, *b*)  
*consol* :: (*Eq* *b*) ⇒ [(*b*, *Int*)] → [(*b*, *Int*)]  
*consol* = *filter* *nzero* · *map* (*id* × *sum*) · *col* **where** *nzero* (*\_, x*) = *x* ≠ 0  
*isempty* :: *Eq* *a* ⇒ [(*a*, *Int*)] → *Bool*  
*isempty* = *all* (≡ 0) · *map* *π*<sub>2</sub> · *consol*  
*col* *x* = *nub* [*k* ↦ [*d'* | (*k'*, *d'*) ← *x*, *k'* ≡ *k*] | (*k*, *d*) ← *x*]  
*consolidate* :: *Eq* *a* ⇒ *Bag* *a* → *Bag* *a*  
*consolidate* = *B* · *consol* · *unB*

## C Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o “layout” que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto e / ou outras funções auxiliares que sejam necessárias.

### Problema 1

De maneira a iniciar a resolução do problema foram definidas as seguintes funções:

*inBlockchain* = [*Bc*, *Bcs*]

Através do isomorfismo **in.out = id** foi possível obter o out para Blockchain de acordo com a prova seguinte:

$$\begin{aligned}
 & out \cdot \mathbf{in} = id \\
 \equiv & \quad \{ \text{Definição de in para Blockchain} \} \\
 & out \cdot [Bc, Bcs] = id \\
 \equiv & \quad \{ \text{Lei 20 Fusão-+} \} \\
 & [out \cdot Bc, out \cdot Bcs] = id \\
 \equiv & \quad \{ \text{Lei 27 Eq-+} \} \\
 & \left\{ \begin{array}{l} out \cdot Bc = i_1 \\ out \cdot Bcs = i_2 \end{array} \right. \\
 \equiv & \quad \{ \text{Lei 73, Lei 74} \} \\
 & \left\{ \begin{array}{l} out (Bc \ a) = i_1 \ a \\ out (Bcs \ (a, b)) = i_2 \ (a, b) \end{array} \right. \\
 & \square
 \end{aligned}$$

*outBlockchain* (*Bc* *a*) = *i*<sub>1</sub> *a*  
*outBlockchain* (*Bcs* (*a*, *b*)) = *i*<sub>2</sub> (*a*, *b*)

O diagrama do catamorfismo de Blockchain apresenta-se de seguida, e através deste foi possível deduzir as restantes funções.

$$\begin{array}{ccc}
 Blockchain & \xrightarrow{\text{out}} & Block + Block \times Blockchain \\
 \downarrow k = (\llbracket g \cdot \rrbracket)_A & & \downarrow id + id \times k \\
 A & \xleftarrow{g} & Block + Block \times A
 \end{array}$$

$$\begin{aligned}
 recBlockchain \, f &= id + (id \times f) \\
 cataBlockchain \, g &= g \cdot (recBlockchain \, (cataBlockchain \, g)) \cdot outBlockchain \\
 anaBlockchain \, g &= inBlockchain \cdot (recBlockchain \, (anaBlockchain \, g)) \cdot g \\
 hyloBlockchain \, f \, g &= cataBlockchain \, f \cdot anaBlockchain \, g
 \end{aligned}$$

De seguida são apresentadas as soluções obtidas pelo grupo para cada uma das alíneas do problema 1.

## C.1 allTransactions

A função **allTransactions** calcula a lista de todas as Transactions presentes numa Blockchain, deste modo foi definido um catamorfismo de Blockchain que para o caso base de um Block - (MagicNo,(Time,Transactions)) utiliza a função  $\pi_2$  para extrair a lista de Transactions. Para o caso de um Blockchain - (Block,Blockchain) utiliza de novo a função  $\pi_2$  para extrair a lista de Transactions do Block e a função *conc* para concatenar a lista de transações ao resultado recursivo do resto da Blockchain. A solução é apresentada a seguir.

$$allTransactions = cataBlockchain \, [\pi_2 \cdot \pi_2, conc \cdot ((\pi_2 \cdot \pi_2) \times id)]$$

## C.2 Ledger

A função *Ledger* calcula o valor que cada entidade detém numa Blockchain, para defini-la utilizamos três catamorfismos e um conjunto de funções auxiliares que serão descritas mais abaixo. O processo é seguinte:

- Obter a lista de transações através do catamorfismo **allTransactions**
- Obter um par com a lista das entidades (obtida através da função **getEntities**) e a lista das transações
- Aplicar um catamorfismo **cataSaldo** à lista das transações, obtendo um par da lista das entidades e do catamorfismo aplicado às transações
- Obter um par com a lista das entidades e uma lista com o saldo de cada entidade
- Obter uma lista de pares (Entidade,Saldo).

A solução é apresentada de seguida:

$$ledger = \widehat{zip} \cdot \langle \widehat{\pi_1}, \widehat{\cdot swap} \rangle \cdot (id \times cataSaldo) \cdot \langle getEntities, id \rangle \cdot allTransactions$$

O catamorfismo de listas **cataSaldo** dada uma lista de transações e uma entidade calcula o saldo dessa entidade.

$$\begin{aligned}
 cataSaldo &:: [Transaction] \rightarrow String \rightarrow Int \\
 cataSaldo \, transactions \, entity &= (cataList \, [0, addInt \cdot ((getSaldo \, entity) \times id)]) \, transactions
 \end{aligned}$$

$$\begin{aligned}
 getSaldo &:: String \rightarrow Transaction \rightarrow Int \\
 getSaldo \, e \, (a, (b, c)) &| e \equiv a = -b \\
 &| e \equiv c = b \\
 &| otherwise = 0
 \end{aligned}$$

Foi utilizada a função *addInt* porque a *add* pré-definida trabalha com Integer

```

addInt :: (Int, Int) → Int
addInt (a, b) = a + b

```

O catamorfismo de listas **cataEntities** que dá a lista de todas as entidades presentes em todas as transações

```

cataEntities = cataList [nil, conc · ((pairToList · ⟨π1, π2 · π2⟩) × id)]

```

Esta função elimina as entidades repetidas da lista produzida pelo catamorfismo anterior através da função **rmDuplicates**.

```

getEntities :: (Ord a) ⇒ [(a, (b, a))] → [a]
getEntities = rmDuplicates · cataEntities
rmDuplicates :: (Ord a) ⇒ [a] → [a]
rmDuplicates = map head · group · sort
pairToList :: (a, a) → [a]
pairToList (x, y) = [x, y]

```

### C.3 isValidMagicNr

A função **isValidMagicNr** verifica se todos os números mágicos numa blockchain são únicos. Para definir esta função utilizamos um catamorfismo **cataNrMagico** que para o caso base de um Block, aplica um  $\langle \pi_1, nil \rangle$  de maneira a obter o número mágico desse Block e aplica a função nil ao lado direito do par para obter uma lista vazia, o resultado de  $\pi_1$  (número mágico) vai ser posteriormente inserido na lista através da função *cons*. Para o caso de um Blockchain, é usada novamente a função *cons* para inserir o número mágico à cabeça da lista resultado recursivo para o resto da Blockchain. Neste momento temos uma lista com todos os números mágicos da Blockchain, é altura de verificar se existem alguns repetidos utilizamos para isso uma função **checkDuplicates** que dada uma lista verifica se existem ou não elementos repetidos. A solução é apresentada de seguida.

```

isValidMagicNr = checkDuplicates · cataNrMagico
-- catamorfismo que transforma um blockchain
cataNrMagico :: Blockchain → [String]
cataNrMagico = cataBlockchain [cons · ⟨π1, nil⟩, cons · (π1 × id)]
-- verifica se existem número mágicos repetidos
checkDuplicates :: (Ord a) ⇒ [a] → Bool
checkDuplicates x = ((rmDuplicates x) ≡ x)

```

### Problema 2

De maneira a iniciar a resolução do problema 2, foi necessário primeiro definir funções que nos permitam criar catamorfismos e anamorfismos para a estrutura de dados do problema em causa.

```

pairToCell (a, (b, c)) = Cell a b c
pairToBlock (a, (b, (c, d))) = Block a b c d

```

```

inQTree = [pairToCell, pairToBlock]

```

Através do isomorfismo **in.out = id** foi possível obter o out para Blockchain de acordo com a prova seguinte:

$$\begin{aligned}
& outQTree \cdot inQTree = id \\
\equiv & \quad \{ \text{Definição de inQTree} \} \\
& outQTree \cdot [toCell, toBlock] = id \\
\equiv & \quad \{ \text{Lei 20 Fusão-+} \} \\
& [outQTree \cdot toCell, outQTree \cdot toBlock] = id \\
\equiv & \quad \{ \text{Lei 17 Universal-+} \} \\
& \begin{cases} id \cdot i_1 = outQTree \cdot toCell \\ id \cdot i_2 = outQTree \cdot toBlock \end{cases} \\
\equiv & \quad \{ \text{Lei 73, Lei 74} \} \\
& \begin{cases} outQTree \cdot toCell (a, (b, c)) = i_1 (a, (b, c)) \\ outQTree \cdot toBlock (a, (b, (c, d))) = i_2 (a, (b, (c, d))) \end{cases} \\
& \square
\end{aligned}$$

$$\begin{aligned}
outQTree (Cell \ a \ b \ c) &= i_1 (a, (b, c)) \\
outQTree (Block \ a \ b \ c \ d) &= i_2 (a, (b, (c, d)))
\end{aligned}$$

O diagrama do catamorfismo de QTree apresenta-se de seguida, e através deste foi possível deduzir as restantes funções.

$$\begin{array}{ccc}
QTree \ A & \xrightarrow{out} & (A, (Int, Int)) + (QTree \ A, (QTree \ A, (QTree \ A, QTree \ A))) \\
\downarrow k=cataQTree \ g & & \downarrow recQTree(k) \\
QTree \ A & \xleftarrow{g} & (A, (Int, Int)) + (QTree \ A, (QTree \ A, (QTree \ A, QTree \ A)))
\end{array}$$

$$\begin{aligned}
baseQTree \ g \ f &= (g \times id) + (f \times (f \times (f \times f))) \\
recQTree \ f &= baseQTree \ id \ f \\
cataQTree \ g &= g \cdot (recQTree (cataQTree \ g)) \cdot outQTree \\
anaQTree \ g &= inQTree \cdot (recQTree (anaQTree \ g)) \cdot g \\
hyloQTree \ g1 \ g2 &= cataQTree \ g1 \cdot anaQTree \ g2 \\
\textbf{instance Functor QTree where} \\
\quad fmap \ f &= cataQTree (inQTree \cdot (baseQTree \ f \ id))
\end{aligned}$$

## C.4 rotateQTree

Esta função roda 90 graus uma QTree, para tal é necessário alterar a posição dos blocos e o formato das células. Foi feito um catamorfismo de QTree, em que no caso base de uma Cell é usada a função **rotateCell** que roda 90 graus uma Cell, ou seja, troca o elementos do par que define a dimensão da matriz.

Para o caso de um Block troca-se a posição das QTree de maneira a ser feita uma rotação de 90 graus, através da função **rotateBlock**, A solução encontrada está apresentada de seguida.

$$\begin{aligned}
rotateQTree &= cataQTree [rotateCell, rotateBlock] \\
rotateCell \ (a, (b, c)) &= Cell \ a \ c \ b \\
rotateBlock \ (a, (b, (c, d))) &= Block \ c \ a \ d \ b
\end{aligned}$$

Assumindo que A, B, C e D são QTree que pertencem a um bloco a rotação de 90 graus leva a um reposicionamento das QTree de acordo com o diagrama seguinte.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} C & A \\ D & B \end{bmatrix}$$

## C.5 scaleQTree

A função `scaleQTree` recalcula a dimensão de uma QTree de acordo com um fator, para tal é necessário multiplicar os elementos que definem o tamanho de cada célula por esse fator. A solução é apresentada de seguida.

```
scaleQTree a = cataQTree [scaleCell a, pairToBlock]
scaleCell mult (x, (y, z)) = Cell x (mult * y) (mult * z)
```

O seguinte diagrama demonstra o catamorfismo de QTree utilizado, em que  $g = [scaleCell, pairToBlock]$

$$\begin{array}{ccc} Int \times QTree A & \xrightarrow{out} & Int \times (A, (Int, Int)) + (QTree A, (QTree A, (QTree A, QTree A))) \\ \downarrow k=cataQTree\ g & & \downarrow recQTree\ (k) \\ QTree A & \xleftarrow{g} & Int \times (A, (Int, Int)) + (QTree A, (QTree A, (QTree A, QTree A))) \end{array}$$

## C.6 invertQTree

A função `invertQTree` inverte as cores de uma QTree

```
invertQTree = cataQTree [invertCell, pairToBlock]
invertCell ((PixelRGBA8 r g b a), (n, m)) = Cell (PixelRGBA8 (255 - r) (255 - g) (255 - b) a) n m
```

O seguinte diagrama demonstra o catamorfismo de QTree utilizado, em que  $g = [invertCell, pairToBlock]$ .

$$\begin{array}{ccc} QTree A & \xrightarrow{out} & (A, (Int, Int)) + (QTree A, (QTree A, (QTree A, QTree A))) \\ \downarrow k=cataQTree\ g & & \downarrow recQTree(k) \\ QTree A & \xleftarrow{g} & (A, (Int, Int)) + (QTree A, (QTree A, (QTree A, QTree A))) \end{array}$$

## C.7 compressQTree

A função **compressQTree** corta as folhas da árvore de maneira a reduzir a sua profundidade dado um determinado nível, para a definição desta função foi feito um anamorfismo de QTree. A solução obtida é apresentada a seguir.

```
compressQTree a b = (anaQTree geneForCompression) (a, b)
geneForCompression (x, (Cell a b c)) = i1 (a, (b, c))
geneForCompression (x, block@(Block a b c d))
  | x ≥ (depthQTree block) = i1 ((attributeValue block), ((π1 (sizeQTree block)), (π2 (sizeQTree block))))
  | otherwise = i2 (((x, a), ((x, b), ((x, c), (x, d)))))
```

A função **attributeValue** é usada para obter um valor qualquer de uma QTree, foi criada para contornar a necessidade de atribuir um valor a uma Cell que foi criada a partir de um Block.

```
attributeValue :: QTree a → a
attributeValue (Cell x y z) = x
attributeValue (Block x y z k) = attributeValue x
```

O gene do anamorfismo é explicado a seguir:

- Caso receba uma Cell, coloca-a do lado esquerdo usando  $i_1$ , pois chegamos a uma folha da árvore que não deve ser cortada.

- Caso receba um Block, e a sua profundidade é menor ou igual ao nível de compressão dado como argumento, elimina esse bloco convertendo-o para uma Cell e por consequência elimina os seus filhos. De seguida, executa aplica  $i_1$  para colocar a nova célula do lado esquerdo do par.
- Caso receba um Block, e a sua profundidade é maior que o nível de compressão dado como argumento, coloca-o do lado direito usando  $i_2$ , sem efetuar qualquer alteração.

O seguinte diagrama demonstra o anamorfismo de QTree utilizado:

$$\begin{array}{ccc}
 QTree\ A & \xleftarrow{inQTree} & (A, (Int, Int)) + (QTree\ A, (QTree\ A, (QTree\ A, QTree\ A))) \\
 \uparrow f & & \uparrow id+f \times f \times f \times f \\
 Int \times QTree\ A & \xrightarrow{g} & (A, (Int, Int)) + (Int \times (QTree\ A, (Int \times QTree\ A, (Int \times QTree\ A, Int \times QTree\ A))))
 \end{array}$$

## C.8 outlineQTree

A função **outlineQTree** apresenta o contorno de uma malha poligonal explicada no enunciado. Inicialmente é preciso verificar se a célula, após aplicada a função dada, é de valor *True*. Em caso positivo, utiliza-se a função **outlineBlock** que, dado um tamanho de bloco, procede ao contorno do mesmo.

O catamorfismo de QTree definido transforma uma QTree na respetiva QTree de Booleanos. Após esta conversão, basta utilizar a função **qt2bm** para converter para uma Matrix, como pedido no enunciado.

A solução obtido é apresentada a seguir.

```

outlineQTree f = qt2bm · (cataQTree [outlineCell f, pairToBlock])
outlineCell f (a, (b, c)) = if (f a) then (outlineBlock b c) else (Cell (f a) b c)
outlineBlock a b = Block
  (Block (Cell True 1 1)
   (Cell True (a - 2) 1)
   (Cell True 1 (b - 2))
   (Cell False (a - 2) (b - 2)))
  (Cell True 1 (b - 1))
  (Cell True (a - 1) 1)
  (Cell True 1 1)

```

O seguinte diagrama demonstra o catamorfismo de QTree utilizado, em que  $g = [outlineCell\ f, pairToBlock]$

$$\begin{array}{ccc}
 QTree\ A & \xrightarrow{out} & f \times (A, (Int, Int)) + (QTree\ A, (QTree\ A, (QTree\ A, QTree\ A))) \\
 \downarrow k=cataQTree\ g & & \downarrow recQTree(k) \\
 QTree\ A & \xleftarrow{g} & f \times (A, (Int, Int)) + (QTree\ A, (QTree\ A, (QTree\ A, QTree\ A)))
 \end{array}$$

## Problema 3

Como sugerido no enunciado a abordagem seguida foi fazer o  $\langle f, l \rangle$  e  $\langle g, s \rangle$ , para isso deduzimos através das definições em pointwise as funções. Finalmente aplicamos a lei da recursividade múltipla para os dois splits obtidos e obtemos as definições da base e loop para o for.



$$\begin{aligned}
& \left\{ \begin{array}{l} fk \cdot 0 = 1 \\ fk \cdot (d+1) = (d+k+1) * fk \cdot d \end{array} \right\} \quad \left\{ \begin{array}{l} lk \cdot 0 = 1 \\ lk \cdot (d+1) = lk \cdot d + 1 \end{array} \right\} \\
\equiv & \quad \{ \text{Lei 73 (x2), Lei 74 (x4), Definição de (d+k+1), Lei 76 (x2), Lei 78} \} \\
& \left\{ \begin{array}{l} fk \cdot \underline{0} = \underline{1} \\ fk \cdot \text{succ} = \text{mul} \cdot \langle lk, fk \rangle \end{array} \right\} \quad \left\{ \begin{array}{l} lk \cdot \underline{0} = \underline{(k+1)} \\ lk \cdot \text{succ} = \text{succ} \cdot lk \end{array} \right\} \\
\equiv & \quad \{ \text{Lei 27 eq+} \} \\
& \left\{ \begin{array}{l} [fk \cdot \underline{0}, fk \cdot \text{succ}] = [\underline{1}, \text{mul} \cdot \langle lk, fk \rangle] \\ [lk \cdot \underline{0}, lk \cdot \text{succ}] = [\underline{(k+1)}, \text{succ} \cdot lk] \end{array} \right\} \\
\equiv & \quad \{ \text{Definição de in dos naturais, Lei Fusão-+ (x2), Lei Absorção-+ (x2)} \} \\
& \left\{ \begin{array}{l} fk \cdot \mathbf{in} = [\underline{1}, \text{mul}] \cdot (id + \langle lk, fk \rangle) \\ lk \cdot \mathbf{in} = [\underline{(k+1)}, \text{succ}] \cdot (id + lk) \end{array} \right\} \\
\equiv & \quad \{ \text{Definição de swap e Lei 7 Cancelamento-x} \} \\
& \left\{ \begin{array}{l} fk \cdot \mathbf{in} = ([\underline{1}, \text{mul}] \cdot \text{swap}) \cdot (id + \langle fk, lk \rangle) \\ lk \cdot \mathbf{in} = ([\underline{(k+1)}, \text{succ} \cdot \pi_2] \cdot (id + \langle fk, lk \rangle)) \end{array} \right\} \\
\equiv & \quad \{ \text{Lei 50 Fokkinga} \} \\
& \langle fk, lk \rangle = (\langle [\underline{1}, \text{mul} \cdot \text{swap}], [\underline{(k+1)}, \text{succ} \cdot \pi_2] \rangle \cdot \cdot)_A \\
& \square
\end{aligned}$$

$$\begin{aligned}
& \left\{ \begin{array}{l} g \cdot 0 = 1 \\ g \cdot (d+1) = (d+1) * g \cdot d \end{array} \right\} \quad \left\{ \begin{array}{l} s \cdot 0 = 1 \\ s \cdot (d+1) = s \cdot d + 1 \end{array} \right\} \\
\equiv & \quad \{ \text{Lei 73 (x2), Lei 74 (x4), Lei 76 (x2), Definição de (d+1) e Lei 78} \} \\
& \left\{ \begin{array}{l} g \cdot \underline{0} = \underline{1} \\ g \cdot \text{succ} = \text{mul} \cdot \langle s, g \rangle \end{array} \right\} \quad \left\{ \begin{array}{l} s \cdot \underline{0} = \underline{1} \\ s \cdot \text{succ} = \text{succ} \cdot s \end{array} \right\} \\
\equiv & \quad \{ \text{Lei 27 eq+} \} \\
& \left\{ \begin{array}{l} [g \cdot \underline{0}, g \cdot \text{succ}] = [\underline{1}, \text{mul} \cdot \langle s, g \rangle] \\ [s \cdot \underline{0}, s \cdot \text{succ}] = [\underline{1}, \text{succ} \cdot s] \end{array} \right\} \\
\equiv & \quad \{ \text{Definição de in dos naturais, Lei Fusão (x2), Lei Absorção (x2)} \} \\
& \left\{ \begin{array}{l} g \cdot \mathbf{in} = [\underline{1}, \text{mul}] \cdot (id + \langle s, g \rangle) \\ s \cdot \mathbf{in} = ([\underline{(1)}, \text{succ}] \cdot \pi_1) \cdot (id + \langle s, g \rangle) \end{array} \right\} \\
\equiv & \quad \{ \text{Propriedade do swap (x2) e Lei 7 Cancelamento-x} \} \\
& \left\{ \begin{array}{l} g \cdot \mathbf{in} = [\underline{1}, \text{mul} \cdot \text{swap}] \cdot (id + \langle g, s \rangle) \\ s \cdot \mathbf{in} = ([\underline{1}, \text{succ} \cdot \pi_1 \cdot \text{swap}] \cdot (id + \langle g, s \rangle)) \end{array} \right\} \\
\equiv & \quad \{ \text{Lei 50 Fokkinga} \} \\
& \langle g, s \rangle = (\langle [\underline{1}, \text{mul} \cdot \text{swap}], [\underline{1}, \text{succ} \cdot \pi_1 \cdot \text{swap}] \rangle \cdot \cdot)_A \\
& \square
\end{aligned}$$

$$\begin{aligned}
& \left\{ \begin{array}{l} \langle i \cdot \rangle_A = \langle \langle \underline{1}, mul \cdot swap \rangle, [(k+1), succ \cdot \pi_2] \rangle \cdot \rangle_A \\ \langle j \cdot \rangle_A = \langle \langle \underline{1}, mul \cdot swap \rangle, [\underline{1}, succ \cdot \pi_1 \cdot swap] \rangle \cdot \rangle_A \end{array} \right. \\
& \equiv \quad \{ \text{Lei 51 Banana-split} \} \\
& \langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \underline{1}, mul \cdot swap \rangle, [(k+1), succ \cdot \pi_2] \rangle \times \langle \langle \underline{1}, mul \cdot swap \rangle, [\underline{1}, succ \cdot \pi_1 \cdot swap] \rangle \rangle \cdot \langle F \pi_1, F \pi_2 \rangle \cdot \rangle_A \\
& \equiv \quad \{ \text{Lei da troca} \} \\
& \langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \underline{1}, (k+1) \rangle, \langle mul \cdot swap, succ \cdot \pi_2 \rangle \rangle \times \langle \langle \underline{1}, \underline{1} \rangle, \langle mul \cdot swap, succ \cdot \pi_1 \cdot swap \rangle \rangle \rangle \cdot \langle F \pi_1, F \pi_2 \rangle \cdot \rangle_A \\
& \equiv \quad \{ \text{Lei da troca (x2), Def de funtor (F p1) e (F p2), Lei 75 Definição Constante, 3.90 apontamentos} \} \\
& \langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \underline{1}, (k+1) \rangle, \langle mul \cdot swap, succ \cdot \pi_2 \rangle \rangle \cdot F \pi_1, [\langle \underline{1}, (k+1) \rangle, \langle mul \cdot swap, succ \cdot \pi_2 \rangle] \cdot F \pi_2 \rangle \cdot \rangle_A \\
& \equiv \quad \{ \text{Lei da troca} \} \\
& \langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \underline{1}, (k+1) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle, \langle \langle mul \cdot swap, succ \cdot \pi_2 \rangle \cdot \pi_1, \langle mul \cdot swap, succ \cdot \pi_1 \cdot swap \rangle \cdot \pi_2 \rangle \rangle \cdot \rangle_A \\
& \equiv \quad \{ \text{Definição for b i} \} \\
& \left\{ \begin{array}{l} b = \langle \langle mul \cdot swap, succ \cdot \pi_2 \rangle \cdot \pi_1, \langle mul \cdot swap, succ \cdot \pi_1 \cdot swap \rangle \cdot \pi_2 \rangle \\ i = \langle \langle \underline{1}, (k+1) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle \end{array} \right. \\
& \square
\end{aligned}$$

$$\begin{aligned}
untuple ((i, j), (k, z)) &= (i, j, k, z) \\
tuple (i, j, k, z) &= ((i, j), (k, z)) \\
loop &= untuple \cdot \langle \langle mul \cdot swap \cdot \pi_1, succ \cdot \pi_2 \cdot \pi_1 \rangle \cdot tuple, \langle mul \cdot swap \cdot \pi_2, succ \cdot \pi_1 \cdot swap \cdot \pi_2 \rangle \cdot tuple \rangle \\
base &= untuple \cdot \langle \langle one, succ \rangle, \langle one, one \rangle \rangle
\end{aligned}$$

## Problema 4

De maneira a resolver o problema 4, foi necessário definir as funções que facilitam a manipulação do tipo de dados *FTree*:

*inFTree* usa os construtores de *FTree*, usando uma função auxiliar *toComp* de maneira a poder converter um par recebido.

$$\begin{aligned}
inFTree &= [Unit, toComp] \\
toComp (a, (b, c)) &= Comp a b c
\end{aligned}$$

*outFTree* foi derivada de uma maneira semelhante à out das Blockchain (Problema 1):

$$\begin{aligned}
outFTree (Unit a) &= i_1 a \\
outFTree (Comp a b c) &= i_2 (a, (b, c))
\end{aligned}$$

As restantes funções são:

$$\begin{aligned}
baseFTree f g h &= g + (f \times (h \times h)) \\
recFTree f &= baseFTree id id f \\
cataFTree g &= g \cdot (recFTree (cataFTree g)) \cdot outFTree \\
anaFTree g &= inFTree \cdot (recFTree (anaFTree g)) \cdot g \\
hyloFTree f g &= cataFTree f \cdot anaFTree g
\end{aligned}$$

A partir da lei 47 (Def-map-cata) do formulário desta unidade curricular ficou definido o bifunctor:

$$\begin{aligned}
\text{instance Bifunctor FTree where} \\
bimap f g &= cataFTree (inFTree \cdot (baseFTree f g id))
\end{aligned}$$

## generatePTree

A função *generatePTree* deve gerar, para um dado valor inteiro de entrada, a árvore de Pitágoras de ordem correspondente, composta por quadrados com escalas adequadas a cada nível. Esta função será definida como um anamorfismo.

Para este efeito, partimos do diagrama do anamorfismo de *FTree*, uma vez que dada a definição de *PTree* com **type** *PTree* = *FTree* *Square* *Square*, é possível inferir que a estrutura geral será idêntica, sendo apenas definidos os tipos que a *FTree* utiliza. Dada a definição de *Square* com **type** *Square* = *Float*, os tipos A e B abaixo definidos corresponderão, naturalmente, a esse mesmo tipo.

O diagrama é então o seguinte:

$$\begin{array}{ccc}
 FTree\ A\ B / \_out & \longleftarrow & B + A \times (FTree\ A\ B \times FTree\ A\ B) / \_in \\
 \uparrow f & & \uparrow id + id \times (f \times f) \\
 C & \xrightarrow{g} & B + A \times (C \times C)
 \end{array}$$

Numa primeira tentativa, a ideia para o anamorfismo partia de um valor numérico inteiro de entrada que correspondia à ordem pretendida para a árvore de Pitágoras. Esse inteiro seria diminuído a cada iteração, ocorrendo o caso de paragem para esta computação quando esse valor atingisse o 0.

Este anamorfismo inicial servia-se então do seguinte gene:

*genePTree* = (*id* +  $\langle \pi_2, \langle \pi_1, \pi_1 \rangle \rangle$ ) · (*id* + (*pred* × *id*)) · (*id* +  $\langle id, orderMultiplier \rangle$ ) · (*fromIntegral* + *id*) · *oneToLeft*

No entanto, o anamorfismo inicialmente sugerido tinha como resultado uma árvore de Pitágoras de dimensões invertidas, o que obrigou a que se partisse do valor de ordem mínimo para a árvore, 0, e se iterasse consecutivamente até ser atingido o valor de ordem pretendido para a árvore a construir.

Com este intuito, surgiu uma segunda versão para o anamorfismo, que define o tipo dos valores de entrada como o tuplo (*Int*, *Int*). O primeiro elemento deste par representa o valor de ordem da iteração atual e o segundo elemento corresponde ao valor de ordem final, que foi definido para a árvore a ser criada.

Para que seja obtido o par acima descrito é necessário aplicar um *split* ao valor inteiro de entrada. Este *split* será dado por  $\langle 0, id \rangle$  e quando aplicado ao valor de entrada, permite que a seguir seja aplicado o anamorfismo pretendido, ficando assim definida a função *generatePTree*:

$$generatePTree = anaFTree\ genePTree \cdot \langle 0, id \rangle$$

A função *genePTree* será o gene do anamorfismo:

$$\begin{aligned}
 genePTree = & (id + (id \times \langle id, id \rangle)) \cdot (id + (id \times (succ \times id))) \cdot (id + \langle orderMultiplier \cdot \pi_1, id \rangle) \\
 & \cdot ((orderMultiplier \cdot \pi_1) + id) \cdot checkComplete
 \end{aligned}$$

$$\begin{aligned}
 & Int \times Int \\
 & \downarrow checkComplete \\
 & Int \times Int + Int \times Int \\
 & \downarrow (orderMultiplier \cdot \pi_1) + id \\
 & Float + Int \times Int \\
 & \downarrow id + \langle orderMultiplier \cdot \pi_1, id \rangle \\
 & Float + Float \times (Int \times Int) \\
 & \downarrow id + id \times (succ \times id) \\
 & Float + Float \times (Int \times Int) \\
 & \downarrow id + id \times \langle id, id \rangle \\
 & Float + Float \times ((Int \times Int) \times (Int \times Int))
 \end{aligned}$$

Serão aqui apresentadas as funções a que o anamorfismo recorre.

Uma delas, *orderMultiplier*, retorna o multiplicador de uma *PTree* para um dado número de ordem. Dado o valor de escala definido pelo enunciado, de  $\frac{\sqrt{2}}{2}$ , sabe-se então que o valor de escala a aplicar nos quadrados a adicionar numa dada ordem é dado por  $(\frac{\sqrt{2}}{2})^o$ , sendo *o* o número representante da ordem.

```
orderMultiplier :: Int → Float
orderMultiplier a = (((sqrt 2) / 2) ↑ a)
```

Por sua vez, a função *checkComplete* executa  $i_1$  sobre um par de inteiros se estes forem iguais ou  $i_2$  se forem diferentes. Esta função é útil para a determinação da última iteração do anamorfismo.

A primeira guarda, na qual surge  $b < 0 = i_1(a, 0)$ , verifica se o valor em *b* é negativo para evitar um número infinito de iterações quando é pedida uma *BTree* com ordem negativa. Nesse caso, a ordem assumida toma o valor 0. Nas restantes guardas é efetuado o que havia sido definido em cima.

```
checkComplete :: (Int, Int) → (Int, Int) + (Int, Int)
checkComplete (a, b)
  | b < 0 = i1 (a, 0)
  | a ≡ b = i1 (a, b)
  | otherwise = i2 (a, b)
```

## drawPTree

Não foi desenvolvida uma definição para a função *drawPTree*.

```
drawPTree = ⊥
```

Verificam-se com isto as propriedades *QuickCheck* relativas a este problema, como se pode verificar de seguida:

```
*Main> quickCheck (prop4a 14)
+++ OK, passed 1 tests.
*Main> quickCheck (prop4b 14)
+++ OK, passed 1 tests.
```

## Problema 5

As funções que se pretendem ver desenvolvidas para a primeira alínea deste enunciado conferem funcionalidades essenciais aos mónades, evidenciando as suas propriedades de multiplicação, no caso da função  $\mu$  e de unidade, no caso da função *singletonbag* (ou *u*).

A primeira será uma função polimórfica que permitirá reduzir em uma unidade o nível de monadificação a uma entrada que esteja num nível de monadificação igual ou superior a 2, ou seja, o seu tipo poderá ser dado por:

$$T\ A \xleftarrow{\mu} T\ (T\ A)$$

Neste caso, está em uso o mónade *Bag* e com as reduções dos níveis de monadificação será necessário ajustar os valores de multiplicidade do conteúdo da *Bag* resultante. A função  $\mu$  fica então definida por:

```
-- mulMults :: [(a, Int)], Int -> [(a, Int)]
-- (unB . (fmap unB)) b3 == map (unB <| id) (unB b3)
mulMults ([], c) = []
mulMults (((a, b) : t), c) = (a, b * c) : (mulMults (t, c))
μ = B · concat · (map mulMults) · map (unB × id) · unB
```

A ação da função *unB* remove a monadificação do seu argumento, o que coloca os pares (Elemento, Multiplicidade) exatamente na forma de tuplo. Desta forma será possível utilizar os valores de multiplicidade dos elementos menos aninhados, os segundos elementos dos pares, que serão relevantes para a redução do nível de monadificação.

$$[(a, Int)] \xleftarrow{unB} Bag\ a$$

De seguida será necessário tornar utilizáveis os valores de multiplicidade dos elementos do nível seguinte de aninhamento. Para isto, ao resultado da aplicação de  $unB$ , uma lista de tuplos (pares (Elemento, Multiplicidade)), será necessário remover a monadificação aos seus elementos (os primeiros elementos dos tuplos da lista que resultado da aplicação de  $unB$ ), “expondo” as suas multiplicidades, por ação de  $unB$ , e deixando intacta a multiplicidade do nível superior (os segundos elementos dos tuplos da lista anteriormente mencionada), por ação de  $id$ . Estas funções serão mapeadas e aplicadas paralelamente a cada elemento da lista e ficam desta forma utilizáveis todos os valores de multiplicidade necessários para o processo de redução do nível de monadificação.

$$[[[(a, Int)], Int]] \xleftarrow{\text{map } (unB \times id)} [(Bag\ a, Int)]$$

Será agora necessário remover a multiplicidade menos aninhada, e no entanto garantir a manutenção da correção das multiplicidades no mónade  $Bag$  resultante. Para isto, por ação de multiplicação das multiplicidades é obtido esse efeito. Para isto é usada a função  $mulMults$ , que será mapeada a cada elemento da lista resultante anterior. Para isto pretendemos que o 2º elemento dos tuplos menos aninhados sejam multiplicados pelo 2º elemento do 1º elemento dos tuplos menos aninhados, que por sua vez, como se pode inferir e verificar pelos exemplos aqui explicitados, será um tuplo também, que se apresenta na forma (Elemento, Multiplicidade). A lista de resultado ignora por completo o 2º elemento do tuplo principal, o que está de acordo com o pretendido.

$$[[[(a, b)]]] \xleftarrow{\text{map } mulMults} [[[(a, b)], b]]$$

À lista de listas resultante será aplicada a função  $concat$ , que permitirá unificar os conteúdos das listas interiores e a partir da lista resultante é construído o mónade resultado, por ação de  $B$ , obtendo-se assim um mónade num grau imediatamente inferior de monadificação.

Segue-se a função que permite exibir a propriedade de unidade. A já mencionada  $singletonbag$  ou  $u$  “encapsula” valores de entrada, conferindo-lhes um grau superior (em uma unidade) de monadificação. Neste caso está em uso o mónade  $Bag$  e, assim sendo, a função  $singletonbag$  poderá tomar a definição que se segue. A um único elemento que se pretenda colocar num  $Bag$ , será necessário colocá-lo na forma adequada para que possa ser monadificado por ação de  $B$ , ou seja, numa lista de tuplos de 2 elementos, já que  $B$  é do tipo:

$$Bag\ a \xleftarrow{B} [(a, Int)]$$

Para qualquer elemento que se pretenda encapsular, o valor da sua multiplicidade será de 1. Dessa forma o tuplo será algo como o par (Elemento, 1). Para este efeito entra em ação a função  $s\_tuple$ , que simplesmente forma o par adequado.

$$(a, 1) \xleftarrow{s\_tuple} a$$

Obtendo o par adequado, será agora apenas necessário colocá-lo numa lista, o que será efetuado pela função  $singl$ .

$$[a] \xleftarrow{singl} a$$

À lista de saída da aplicação da função  $singl$  será apenas necessário conferir-lhe monadificação, pelo que é novamente utilizado  $B$ .

$$\begin{aligned} s\_tuple\ a &= (a, 1) \\ singletonbag &= B \cdot singl \cdot s\_tuple \end{aligned}$$

Resta apenas a função *dist*, que para um qualquer *Bag* apresenta as percentagens de distribuição dos seus conteúdos, recorrendo ao mónade *Dist*.

$$\text{Dist } a \xleftarrow{\text{dist}} \text{Bag } a$$

Com base na composição de funções, a função *dist* apresenta a definição do código que se segue. Por ação da função *unB*, a partir do mónade de entrada obtém-se a lista de pares (Elemento, Int), sendo o valor inteiro a sua multiplicidade do elemento no *Bag* que acabou de ser removido. A cada um desses pares, com recurso ao mapeamento da função *repMarbles*, é obtida uma lista, que representa explicitamente a ideia definida em cada par (p.e. no caso do elemento ser uma *Marble*, seria possível um dos elementos da lista original ser algo como (Blue, 3), que por ação da função *repMarbles*, dá origem a [Blue, Blue, Blue]), utilizando a versão *uncurried* da função pré-definida *replicate* e o par (Marble,Int) com a ordem inversa por ação da função *swap*, ou seja, um par (Int, Marble). As listas aninhadas são de seguida concatenadas e é aplicada a função *uniform*, que obterá a distribuição e os valores de probabilidade adequados.

$$\begin{aligned} \text{repMarbles} &= \widehat{\text{replicate}} \cdot \text{swap} \\ \text{dist} &= \text{uniform} \cdot \text{concat} \cdot \text{map } \text{repMarbles} \cdot \text{unB} \end{aligned}$$

A segunda alínea deste problema pretende apenas demonstrar a correção das funções desenvolvidas, por verificação da validade das propriedades de multiplicação (à esquerda) e unidade (à direita) já mencionadas e que são referidas nas notas teóricas desta unidade curricular.

$$\begin{array}{ccc} T(TA) & \xleftarrow{\mu} & T(T(TA)) \\ \downarrow \mu & & \downarrow T\mu \\ TA & \xleftarrow{\mu} & T(TA) \end{array} \qquad \begin{array}{ccc} T(TA) & \xleftarrow{u} & TA \\ \downarrow \mu & \nearrow id & \downarrow Tu \\ TA & \xleftarrow{\mu} & T(TA) \end{array}$$

Com tudo isto, os resultados dos testes *test5a* e *test5b* são os seguintes:

```
*Main> quickCheck (bagOfMarbles == muB (return bagOfMarbles))
+++ OK, passed 1 tests.
*Main> quickCheck ((muB . muB) b3 == (muB . fmap muB) b3)
+++ OK, passed 1 tests.
```

## D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:<sup>7</sup>

$$\begin{aligned}
 id &= \langle f, g \rangle \\
 &\equiv \{ \text{universal property} \} \\
 &\quad \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
 &\equiv \{ \text{identity} \} \\
 &\quad \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
 &\square
 \end{aligned}$$

Os diagramas podem ser produzidos recorrendo à *package*  $\text{\LaTeX}$  *xymatrix*, por exemplo:

$$\begin{array}{ccc}
 N_0 & \xleftarrow{\text{in}} & 1 + N_0 \\
 \scriptstyle \langle g \rangle \downarrow & & \downarrow \scriptstyle id + \langle g \rangle \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

---

<sup>7</sup>Exemplos tirados de [?].