

Cálculo de Programas

Trabalho Prático

MiEI+LCC — 2017/18

Departamento de Informática
Universidade do Minho

Junho de 2018

| Grupo nr. | 16 |
|-----------|----------------|
| a77730 | Sérgio Jorge |
| a77870 | Vítor Castro |
| a79116 | Marcos Pereira |

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`¹ 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^AT_EX** 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

¹O 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 **C** 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 bits², tal como se exemplifica na Figura 1a.

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

²Cf. 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.

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

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 1b.

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 2a:

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

Esta questão aborda operações de processamento de imagens utilizando quadrees:

1. Defina as funções *rotateQTree* :: *QTree* *a* → *QTree* *a*, *scaleQTree* :: *Int* → *QTree* *a* → *QTree* *a* e *invertQTree* :: *QTree* *a* → *QTree* *a*, como catamorfismos e/ou anamorfismos, que rodam³, redimensionam⁴ e invertem as cores de uma quadtree⁵, respectivamente. Tente produzir imagens similares às Figuras 2b, 2c e 2d:

```

> 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:

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

³Segundo um ângulo de 90° no sentido dos ponteiros do relógio.

⁴Multiplicando o seu tamanho pelo valor recebido.

⁵Um 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.



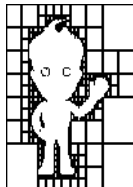
(f) Compressão de 2 níveis.



(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 2e, 2f, 2g e 2h:

```
> 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 2i e 2j:

```
> 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,

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

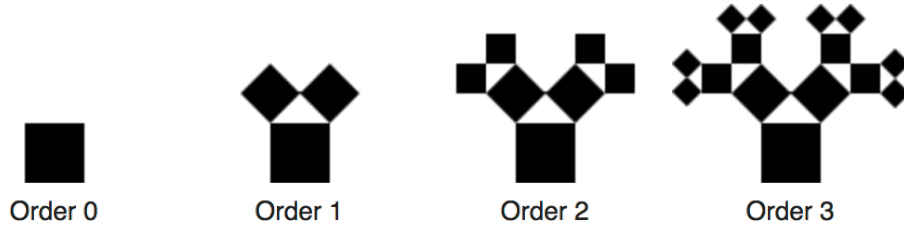


Figure 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

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 (1):

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

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 3).

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.⁶ 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:

⁶“Marble” traduz para “berlinde” em português.

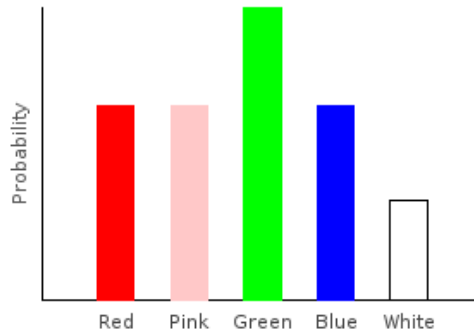


Figure 4: Distribuição de berlindes num saco.

```
{ 2 |-> 3 , 3 |-> 5 , 6 |-> 2 }.
```

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

Finalmente, se quisermos saber a probabilidade da cor de um berlinde 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 4.

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 μ (multiplicação do mónade *Bag*) e a função auxiliar *singletonbag*.
2. Verifique-as com os seguintes testes unitários:

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

```
test5a = bagOfMarbles ≡ μ (return bagOfMarbles)
```

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

```
test5b = (μ · μ) b3 ≡ (μ · fmap μ) b3
```

onde *b3* é um saco dado em anexo.


```
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 π₂ · 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

Problema 1

Antes de proceder à resolução das alíneas, foi necessária a definição das funções relativas à manipulação de Blockchains:

```
inBlockchain = [Bc, Bcs]
```

O out da Blockchain foi calculado da seguinte forma:

$$\begin{aligned}
& out \cdot \mathbf{in} = id \\
\equiv & \quad \{ \text{in} = \text{either Bc Bcs} \} \\
& out \cdot [Bc, Bcs] = id \\
\equiv & \quad \{ \text{Lei 20} \} \\
& [out \cdot Bc, out \cdot Bcs] = id \\
\equiv & \quad \{ \text{Lei 27} \} \\
& \left\{ \begin{array}{l} out \cdot Bc = i_1 \\ out \cdot Bcs = i_2 \end{array} \right. \\
\equiv & \quad \{ \text{Passando para pointwise} \} \\
& \left\{ \begin{array}{l} out (Bc \ a) = i_1 \ a \\ out (Bcs \ (a, b)) = i_2 \ (a, b) \end{array} \right. \\
& \square
\end{aligned}$$

Resultando no seguinte código Haskell:

```
outBlockchain (Bc a) = i₁ a
outBlockchain (Bcs (a, b)) = i₂ (a, b)
```

As restantes funções são:

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

allTransactions

Para definir allTransactions, que calcula uma lista de todas as transações de uma Blockchain, foi usado um catamorfismo de Blockchains:

$$\begin{array}{ccc} \text{Blockchain} & \xrightarrow{\text{out}} & \text{Block} + \text{Block} \times \text{Blockchain} \\ f \downarrow & & \downarrow \text{id} + \text{id} \times f \\ C & \xleftarrow{g} & \text{Block} + \text{Block} \times C \end{array}$$

Este catamorfismo simplesmente usa a função π_2 para extrair as listas de transações de cada bloco:

$$\begin{aligned} \text{allTransactions} &= \text{cataBlockchain } [\text{getTransaction}, \text{getTransactions}] \\ \text{getTransaction} &= \pi_2 \cdot \pi_2 \\ \text{getTransactions} &= \text{conc} \cdot ((\pi_2 \cdot \pi_2) \times \text{id}) \end{aligned}$$

É importante notar que *conc* gera uma lista concatenando um par que contém duas listas: $([a], [a]) \rightarrow [a]$

ledger

Para calcular o ledger de uma blockchain vamos fazer 4 passos:

1. Obter a lista das transações de uma blockchain (usando *allTransactions*);
2. Obter uma lista de entidades a partir da lista de transações (usando o catamorfismo *getEntities*);
3. Eliminar repetições da lista de entidades;
4. Calcular o saldo de cada entidade fazendo map das entidades sobre uma função *getBalance*.

O diagrama seguinte mostra a composição de funções utilizada:

$$\begin{aligned} &\text{Blockchain} \\ &\downarrow \text{allTransactions} \\ &[\text{transaction}] \\ &\downarrow \langle \text{getUniqueEntities}, \text{id} \rangle \\ &([\text{entity}], [\text{transaction}]) \\ &\downarrow \text{id} \times \text{getBalance} \\ &([\text{entity}], (\text{getBalance } [\text{transaction}])) \\ &\downarrow \langle \pi_1, \text{mapPair} \cdot \text{swap} \rangle \\ &([\text{entity}], [\text{saldo}]) \\ &\downarrow \text{zipPair} \\ &[(\text{entity}, \text{saldo})] \end{aligned}$$

A lista $[(entity, saldo)]$ no fim do diagrama anterior é o ledger que pretendemos calcular.

O código Haskell correspondente ao diagrama será:

$$ledger = zipPair \cdot \langle \pi_1, mapPair \cdot swap \rangle \cdot (id \times getBalance) \cdot \langle getUniqueEntities, id \rangle \cdot allTransactions$$

getEntities será um catamorfismo de listas que calcula uma lista de entidades a partir de uma de transações.

$$\begin{aligned} getEntities &:: [Transaction] \rightarrow [String] \\ getEntities &= cataList [nil, conc \cdot ((pairToList \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle) \times id)] \end{aligned}$$

Para transformar uma transação em uma lista com as duas entidades, usamos $pairToList \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle$.

No caso final, usamos *nil* para introduzir uma lista vazia.

É útil recordar que uma transação corresponde ao par $(origem, (valor, destino))$.

A função *getUniqueEntities* remove repetições de uma lista de entidades:

$$\begin{aligned} getUniqueEntities &:: [Transaction] \rightarrow [String] \\ getUniqueEntities &= remDup \cdot getEntities \end{aligned}$$

getBalance será um catamorfismo de listas de transações que calcula o saldo de uma dada entidade.

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

Esta função foi definida em pointwise uma vez que tem de receber a lista de transações antes da entidade, o que nos é útil para o map que vamos querer executar.

A função *delta* retorna a diferença de saldo resultante de uma transação para uma dada entidade (se a entidade não aparece na transação, *delta* será 0):

$$\begin{aligned} delta &:: String \rightarrow Transaction \rightarrow Int \\ delta \text{ entity } (a, (v, b)) & \\ & \quad | \text{ entity } \equiv a = -v \\ & \quad | \text{ entity } \equiv b = v \\ & \quad | \text{ otherwise } = 0 \end{aligned}$$

Foi necessário definir as funções auxiliares *mapPair* e *zipPair*:

$$\begin{aligned} mapPair &:: (a \rightarrow b, [a]) \rightarrow [b] \\ mapPair (f, l) &= \text{map } f \ l \\ zipPair &:: ([a], [b]) \rightarrow [(a, b)] \\ zipPair (x, y) &= \text{zip } x \ y \end{aligned}$$

A função *addInt* soma um par de Ints:

$$\begin{aligned} addInt &:: (Int, Int) \rightarrow Int \\ addInt (a, b) &= a + b \end{aligned}$$

A função *remDup* Remove elementos duplicados de uma lista:

$$\begin{aligned} remDup &:: (Ord a) \Rightarrow [a] \rightarrow [a] \\ remDup &= \text{map } head \cdot group \cdot sort \end{aligned}$$

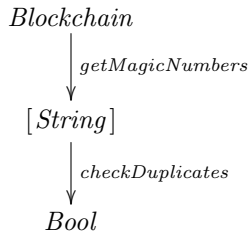
Esta função foi uma descoberta interessante, e tem um desempenho melhor que a equivalente nativa - *nub* é $O(N^2)$ enquanto que *remDup* é $O(N \log(N))$.

Por fim, a função *pairToList* cria um par a partir de uma lista com dois elementos:

$$\begin{aligned} pairToList &:: (a, a) \rightarrow [a] \\ pairToList (x, y) &= [x, y] \end{aligned}$$

isValidMagicNr

Para definir *isValidMagicNr* foi definido um catamorfismo *getMagicNumbers*:



Ao que corresponde o seguinte código Haskell:

$$isValidMagicNr = checkDuplicates \cdot getMagicNumbers$$

O catamorfismo *getMagicNumbers* retorna a lista de números mágicos de uma Blockchain:

$$\begin{aligned} getMagicNumbers &:: Blockchain \rightarrow [String] \\ getMagicNumbers &= cataBlockchain [cons \cdot \langle \pi_1, nil \rangle, cons \cdot (\pi_1 \times id)] \end{aligned}$$

O caso final, $cons \cdot \langle \pi_1, nil \rangle$, retorna uma lista que contém apenas o número mágico do último bloco.

A função *checkDuplicates* retorna *true* se uma lista não tem elementos duplicados, usando a função *remDup* definida anteriormente.

$$\begin{aligned} checkDuplicates &:: (Ord\ a) \Rightarrow [a] \rightarrow Bool \\ checkDuplicates\ x &= (remDup\ x) \equiv x \end{aligned}$$

Foram também definidas duas Blockchains de teste de maneira a verificar as funcionalidades criadas para a resolução deste primeiro problema:

```
block1 = ("1234",
  (177777, [
    ("Marcos", (200, "Tarracho")),
    ("Antonio", (200, "Joao")),
    ("Tarracho", (200, "Marcos")),
    ("Marcos", (200, "Tarracho"))
  ]))
block2 = ("6789",
  (177888, [
    ("Marcos", (200, "Tarracho")),
    ("Antonio", (200, "Joao"))
  ]))
block3 = ("4444",
  (177888, [
    ("Maria", (200, "Matilde")),
    ("Matilde", (200, "Maria"))
  ]))
testBlockchain1 = Bcs (block1, Bc block2)
testBlockchain2 = Bcs (block3, Bcs (block1, Bc block2))
```

Problema 2

Antes de proceder à resolução das alíneas, foi necessária a definição das funções relativas à manipulação de QTrees.

$$\begin{aligned} toCell (a, (b, c)) &= Cell\ a\ b\ c \\ toBlock (a, (b, (c, d))) &= Block\ a\ b\ c\ d \end{aligned}$$

Aplicando a definição de toBlock e toCell temos:

$$inQTree = [toCell, toBlock]$$

O out da QTree foi calculado da seguinte forma:

$$\begin{aligned} outQTree \cdot inQTree &= id \\ \equiv \quad \{ \text{Definição de inQTree} \} \\ outQTree \cdot [toCell, toBlock] &= id \\ \equiv \quad \{ \text{Lei da Fusão} \} \\ [outQTree \cdot toCell, outQTree \cdot toBlock] &= id \\ \equiv \quad \{ \text{Universal-+} \} \\ \left\{ \begin{array}{l} id \cdot i_1 = outQTree \cdot toCell \\ id \cdot i_2 = outQTree \cdot toBlock \end{array} \right. \\ \equiv \quad \{ \text{Lei 1, 73 e 74} \} \\ \left\{ \begin{array}{l} outQTree\ toCell\ (a, (b, c)) = i_1\ (a, (b, c)) \\ outQTree\ toBlock\ (a, (b, (c, d))) = i_2\ (a, (b, (c, d))) \end{array} \right. \end{aligned}$$

□

Resultando na seguinte definição em Haskell:

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

As restantes funções são:

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

rotateQTree

Para uma rotação de 90 graus da QTree, temos de reposicionar as células e os blocos. Usou-se um catamorfismo de QTree.

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \\ \downarrow cataQTree\ r & & \downarrow recQTree\ (cataQTree\ r) \\ QTree\ A & \xleftarrow{r} & (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \end{array}$$

Por questões de legibilidade, colocou-se r ao invés do *either* de *rotateCell* e *rotateBlock*.

Por isso, rotateQTree vem:

```
rotateQTree = cataQTree [rotateCell, rotateBlock]
rotateCell (a, (b, c)) = Cell a c b
rotateBlock (a, (b, (c, d))) = Block c a d b
```

scaleQTree

Para redimensionar, temos de multiplicar cada célula pelo fator de multiplicação. Usámos um catamorfismo de QTree. De notar que, devido à definição da *baseQTree*, é possível aplicação do *outQTree* apesar de termos um *Int* de entrada.

$$\begin{array}{ccc}
 \text{Int} \times \text{QTree } A & \xleftarrow{\text{inQTree}} & \text{Int} \times (A, (\text{Int}, \text{Int})) + ((\text{QTree } A) \uparrow 4) \\
 \text{cataQTree } s \downarrow & & \downarrow \text{recQTree (cataQTree } s) \\
 \text{QTree } A & \xleftarrow{s} & \text{Int} \times (A, (\text{Int}, \text{Int})) + ((\text{QTree } A) \uparrow 4)
 \end{array}$$

Por questões de legibilidade, colocou-se *s* ao invés do *either* de *scaleCell* e *toBlock*. Por isso, scaleQTree vem:

```
scaleQTree a = cataQTree [scaleCell a, toBlock]
scaleCell mult (a, (b, c)) = Cell a (mult * b) (mult * c)
```

invertQTree

Para inverter as cores de uma QTree, temos de inverter a cor de cada pixel. Usámos um catamorfismo de QTree.

$$\begin{array}{ccc}
 \text{QTree } A & \xleftarrow{\text{inQTree}} & (A, (\text{Int}, \text{Int})) + ((\text{QTree } A) \uparrow 4) \\
 \text{cataQTree } i \downarrow & & \downarrow \text{recQTree (cataQTree } i) \\
 \text{QTree } A & \xleftarrow{i} & (A, (\text{Int}, \text{Int})) + ((\text{QTree } A) \uparrow 4)
 \end{array}$$

Por questões de legibilidade, colocou-se *i* ao invés do *either* de *invertCell* e *toBlock*. Por isso, invertQTree vem:

```
invertQTree = cataQTree [invertCell, toBlock]
invertCell ((PixelRGBA8 r g b a), (x, y)) = Cell (PixelRGBA8 (255 - r) (255 - g) (255 - b) a) x y
```

compressQTree

Para a realização da *compressQTree*, foi criado um anamorfismo que recebe um par (*Int*, *QTree*). Neste par, o inteiro representa o nível de compressão desejado.

$$\begin{array}{ccc}
 \text{QTree } A & \xleftarrow{\text{in}} & (A, (\text{Int}, \text{Int})) + ((\text{QTree } A) \uparrow 4) \\
 \uparrow f & & \uparrow \text{id} + f \times f \times f \times f \\
 \text{Int} \times \text{QTree } A & \xrightarrow{g} & (A, (\text{Int}, \text{Int})) + ((\text{Int} \times \text{QTree } A) \uparrow 4)
 \end{array}$$

```
compressQTree a b = (anaQTree geneCompress) (a, b)
geneCompress :: (Int, QTree a) -> (a, (Int, Int)) + ((Int, QTree a), ((Int, QTree a),
((Int, QTree a), (Int, QTree a))))
```



```

geneCompress (x, (Cell a b c)) = i1 (a, (b, c))
geneCompress (x, t@(Block a b c d))
  | x ≥ (depthQTree t) = i1 ((anyValue t), ((π1 (sizeQTree t)), (π2 (sizeQTree t))))
  | otherwise = i2 (((x, a), ((x, b), ((x, c), (x, d)))))

```

O gene do anamorfismo faz uma de três coisas:

- Se recebe uma *Cell*, simplesmente coloca-a do lado esquerdo usando i_1 , uma vez que chegamos a uma folha da árvore que não deve ser cortada.
- Se recebe um *Block*, e a sua profundidade é menor ou igual ao nível de compressão desejado, converte esse bloco para uma célula, efetivamente cortando os filhos desse bloco. De seguida, executa i_1 para colocar a nova célula do lado esquerdo.
- Se recebe um *Block* e a sua profundidade é maior que o nível de compressão desejado, simplesmente coloca-o do lado direito usando i_2 , sem alterar o seu conteúdo.

A função *anyValue* é usada para obter um valor qualquer de uma *QTree*. Esta funcionalidade é útil para quando queremos atribuir um valor a uma nova *Cell* criada a partir de um *Block*.

```

anyValue :: QTree a → a
anyValue (Cell a b c) = a
anyValue (Block a b c d) = anyValue a

```

No futuro, poderia-se usar uma função que calcula a média do valor dos filhos de maneira a realizar uma compressão mais elegante.

outlineQTree

Para fazer o *outline* da figura é preciso verificar se a célula, após aplicada a função dada, é de valor *True*. Se sim, utiliza-se a função *outlineBlock* que, dado um tamanho de bloco, procede ao contorno do mesmo. O cata apresentado é o que transforma uma *QTree* em a respetiva *QTree* de *Bool*. Após esta conversão, basta utilizar a função *qt2bm* para converter para *Matrix*, tal como pedido.

$$\begin{array}{ccc}
 QTree\ A & \xleftarrow{\text{inQTree}} & f \times (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \\
 \text{cataQTree } o \downarrow & & \downarrow \text{recQTree (cataQTree } o) \\
 QTree\ Bool & \xleftarrow{o} & f \times (A, (Int, Int)) + ((QTree\ A) \uparrow 4)
 \end{array}$$

Por questões de legibilidade, colocou-se *o* ao invés do *either* de *outlineCell fun* e *toBlock*. Por isso, *outlineQTree* vem:

```

outlineQTree fun = qt2bm · (cataQTree [outlineCell fun, toBlock])
outlineCell fun (a, (b, c)) = if (fun a) then (outlineBlock b c) else (Cell (fun 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)

```

Problema 3

Tendo em conta o enunciado do problema 3, decidimos que a estratégia a adotar seria fazer o *split* de *f* e *l* e o *split* de *g* e *s*. Faz-se, então, a seguinte demonstração:

$$\begin{aligned}
& \left\{ \begin{array}{l} fk \cdot 0 = 1 \\ fk \cdot (d+1) = (d+k+1) * fk \cdot d \end{array} \right\} \quad \left\{ \begin{array}{l} lk \cdot 0 = 1 \\ lk \cdot (d+1) = lk \cdot d + 1 \end{array} \right\} \\
\equiv & \quad \{ \text{lei 73 (x2), lei 74 (x4), definição de (d+k+1), lei 76 (x2), lei 78} \} \\
& \left\{ \begin{array}{l} fk \cdot \underline{0} = \underline{1} \\ fk \cdot \text{succ} = \text{mul} \cdot \langle lk, fk \rangle \end{array} \right\} \quad \left\{ \begin{array}{l} lk \cdot \underline{0} = \underline{(k+1)} \\ lk \cdot \text{succ} = \text{succ} \cdot lk \end{array} \right\} \\
\equiv & \quad \{ \text{lei eq-+} \} \\
& \left\{ \begin{array}{l} [fk \cdot \underline{0}, fk \cdot \text{succ}] = [\underline{1}, \text{mul} \cdot \langle lk, fk \rangle] \\ [lk \cdot \underline{0}, lk \cdot \text{succ}] = [\underline{(k+1)}, \text{succ} \cdot lk] \end{array} \right\} \\
\equiv & \quad \{ \text{definição de in dos naturais, lei da fusão (x2) e lei da absorção (x2)} \} \\
& \left\{ \begin{array}{l} fk \cdot \text{in} = [\underline{1}, \text{mul}] \cdot (id + \langle lk, fk \rangle) \\ lk \cdot \text{in} = [\underline{(k+1)}, \text{succ}] \cdot (id + lk) \end{array} \right\} \\
\equiv & \quad \{ \text{definição de swap e lei do cancelamento-x} \} \\
& \left\{ \begin{array}{l} fk \cdot \text{in} = ([\underline{1}, \text{mul}] \cdot \text{swap}) \cdot (id + \langle fk, lk \rangle) \\ lk \cdot \text{in} = ([\underline{(k+1)}, \text{succ} \cdot \pi_2] \cdot (id + \langle fk, lk \rangle)) \end{array} \right\} \\
\equiv & \quad \{ \text{fokkinga} \} \\
& \langle fk, lk \rangle = (\langle [\underline{1}, \text{mul} \cdot \text{swap}], [\underline{(k+1)}, \text{succ} \cdot \pi_2] \rangle \cdot \cdot)_A
\end{aligned}$$

□

Neste passo, faz-se a demonstração respetiva ao g e s:

$$\begin{aligned}
& \left\{ \begin{array}{l} g \cdot 0 = 1 \\ g \cdot (d+1) = (d+1) * g \cdot d \end{array} \right\} \quad \left\{ \begin{array}{l} s \cdot 0 = 1 \\ s \cdot (d+1) = s \cdot d + 1 \end{array} \right\} \\
\equiv & \quad \{ \text{lei 73 (x2), lei 74 (x4), lei 76 (x2), definição de (d+1) e lei 78} \} \\
& \left\{ \begin{array}{l} g \cdot \underline{0} = \underline{1} \\ g \cdot \text{succ} = \text{mul} \cdot \langle s, g \rangle \end{array} \right\} \quad \left\{ \begin{array}{l} s \cdot \underline{0} = \underline{1} \\ s \cdot \text{succ} = \text{succ} \cdot s \end{array} \right\} \\
\equiv & \quad \{ \text{lei eq-+} \} \\
& \left\{ \begin{array}{l} [g \cdot \underline{0}, g \cdot \text{succ}] = [\underline{1}, \text{mul} \cdot \langle s, g \rangle] \\ [s \cdot \underline{0}, s \cdot \text{succ}] = [\underline{1}, \text{succ} \cdot s] \end{array} \right\} \\
\equiv & \quad \{ \text{definição de in dos naturais, lei da fusão (x2) e lei da absorção (x2)} \} \\
& \left\{ \begin{array}{l} g \cdot \text{in} = [\underline{1}, \text{mul}] \cdot (id + \langle s, g \rangle) \\ s \cdot \text{in} = ([\underline{(1)}, \text{succ}] \cdot \pi_1) \cdot (id + \langle s, g \rangle) \end{array} \right\} \\
\equiv & \quad \{ \text{propriedade do swap (x2) e cancelamento-x} \} \\
& \left\{ \begin{array}{l} g \cdot \text{in} = [\underline{1}, \text{mul} \cdot \text{swap}] \cdot (id + \langle g, s \rangle) \\ s \cdot \text{in} = ([\underline{1}, \text{succ} \cdot \pi_1 \cdot \text{swap}] \cdot (id + \langle g, s \rangle)) \end{array} \right\} \\
\equiv & \quad \{ \text{fokkinga} \} \\
& \langle g, s \rangle = (\langle [\underline{1}, \text{mul} \cdot \text{swap}], [\underline{1}, \text{succ} \cdot \pi_1 \cdot \text{swap}] \rangle \cdot \cdot)_A
\end{aligned}$$

□

Por fim, procedeu-se à combinação dos resultados obtidos através de um *banana-split*, tal como sugerido no enunciado.

$$\left\{ \begin{array}{l} \langle i \cdot \cdot \rangle_A = (\langle [\underline{1}, \text{mul} \cdot \text{swap}], [\underline{(k+1)}, \text{succ} \cdot \pi_2] \rangle \cdot \cdot)_A \\ \langle j \cdot \cdot \rangle_A = (\langle [\underline{1}, \text{mul} \cdot \text{swap}], [\underline{1}, \text{succ} \cdot \pi_1 \cdot \text{swap}] \rangle \cdot \cdot)_A \end{array} \right\}$$

$$\begin{aligned}
&\equiv \{ \text{lei banana-split} \} \\
&\langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \underline{1}, \text{mul} \cdot \text{swap} \rangle, \langle \underline{(k+1)}, \text{succ} \cdot \pi_2 \rangle \rangle \times \langle \langle \underline{1}, \text{mul} \cdot \text{swap} \rangle, \langle \underline{1}, \text{succ} \cdot \pi_1 \cdot \text{swap} \rangle \rangle \cdot \langle F \pi_1, F \pi_2 \rangle \cdot \langle \rangle_A \\
&\equiv \{ \text{lei da troca} \} \\
&\langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_2 \rangle \rangle \times \langle \langle \underline{1}, \underline{1} \rangle, \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_1 \cdot \text{swap} \rangle \rangle \cdot \langle F \pi_1, F \pi_2 \rangle \cdot \langle \rangle_A \\
&\equiv \{ \text{conforme 3.90 a 3.95 dos apontamentos / lei 11} \} \\
&\langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_2 \rangle \rangle \cdot F \pi_1, \langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_2 \rangle \rangle \cdot F \pi_2 \rangle \cdot \langle \rangle_A \\
&\equiv \{ \text{lei da troca} \} \\
&\langle \langle i \cdot \rangle_A, \langle j \cdot \rangle_A \rangle = \langle \langle \langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \underline{1}, \underline{1} \rangle \rangle, \langle \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_2 \rangle \cdot \pi_1, \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_1 \cdot \text{swap} \rangle \cdot \pi_2 \rangle \rangle \cdot \langle \rangle_A \\
&\equiv \{ \text{definição de for b i} \} \\
&\begin{cases} b = \langle \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_2 \rangle \cdot \pi_1, \langle \text{mul} \cdot \text{swap}, \text{succ} \cdot \pi_1 \cdot \text{swap} \rangle \cdot \pi_2 \rangle \\ i = \langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \underline{1}, \underline{1} \rangle \rangle \end{cases} \\
&\square
\end{aligned}$$

A base foi extraída diretamente do cálculo anterior, enquanto que o *loop* precisou de uns ajustes de tipos internos, os quais foram realizados através das funções *tuple* e *untuple*.

$$\begin{aligned}
\text{untuple } ((a, b), (c, d)) &= (a, b, c, d) \\
\text{tuple } (a, b, c, d) &= ((a, b), (c, d)) \\
\text{base} &= \text{untuple} \cdot \langle \langle \underline{1}, \text{succ} \rangle, \langle \underline{1}, \underline{1} \rangle \rangle \\
\text{loop} &= \text{untuple} \cdot \langle \langle \text{mul} \cdot \text{swap} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle \cdot \text{tuple}, \langle \text{mul} \cdot \text{swap} \cdot \pi_2, \text{succ} \cdot \pi_1 \cdot \text{swap} \cdot \pi_2 \rangle \cdot \text{tuple} \rangle
\end{aligned}$$

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.

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

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

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

As restantes funções são:

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

A partir da lei 47 (def-map-cata) foi definido:

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

generatePTree

generatePTree deve gerar uma árvore de Pitágoras para uma dada ordem, sendo definida como um anamorfismo.

Para este efeito, o grupo partiu do diagrama do anamorfismo de *FTree*, que pode ser usado em vez de *PTree* simplesmente porque:

type PTree = FTree Square Square

O diagrama é o seguinte:

$$\begin{array}{ccc}
 FTree\ A\ B & \xleftarrow{\text{in}} & B + A \times (FTree\ A\ B \times FTree\ A\ B) \\
 \uparrow f & & \uparrow id + id \times (f \times f) \\
 C & \xrightarrow{g} & B + A \times (C \times C)
 \end{array}$$

Neste diagrama, A e B ambos representam o tipo *Float*, para que a *FTree* seja convertível para uma *PTree*.

Na primeira tentativa, o grupo definiu um anamorfismo que recebe um *Int* igual ao rank da árvore de Pitágoras pretendida. Esse *Int* iria diminuindo, e o anamorfismo pararia quando este fosse igual a zero.

O gene deste primeiro anamorfismo era:

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

O que aconteceu foi que o anamorfismo criado gerava uma árvore invertida, pelo que seria necessário começar com o inteiro a zero e parar de iterar quando este fosse igual ao rank pretendido.

Para isso foi criado um segundo anamorfismo que recebe um par (*Int*, *Int*), onde o primeiro inteiro representa o rank da iteração atual, e o segundo representa o rank final pretendido.

Ao encaixar uma função $\langle \underline{1}, id \rangle$ antes do anamorfismo, consegue-se manter esta mudança invisível para o utilizador. Ficamos então com uma função *generatePTree*:

generatePTree = *anaFTree genePTree* · $\langle \underline{0}, id \rangle$

Em que *genePTree* é o gene do anamorfismo:

genePTree = (*id* + (*id* × $\langle id, id \rangle$)) · (*id* + (*id* × (*succ* × *id*))) · (*id* + $\langle rankToMultiplier \cdot \pi_1, id \rangle$) · ((*rankToMultiplier* · π_1) + *id*) · *checkComplete*

Este gene é representado no seguinte diagrama:

$$\begin{array}{c}
 Int \times Int \\
 \downarrow checkComplete \\
 Int \times Int + Int \times Int \\
 \downarrow (rankToMultiplier \cdot \pi_1) + id \\
 Float + Int \times Int \\
 \downarrow id + \langle rankToMultiplier \cdot \pi_1, id \rangle \\
 Float + Float \times (Int \times Int) \\
 \downarrow id + id \times (succ \times id) \\
 Float + Float \times (Int \times Int) \\
 \downarrow id + id \times \langle id, id \rangle \\
 Float + Float \times ((Int \times Int) \times (Int \times Int))
 \end{array}$$

De seguida são definidas as funções usadas pelo anamorfismo:

rankToMultiplier retorna o multiplicador de uma *PTree* para um dado *Rank*. Por exemplo, o multiplicador de ordem 0 é 1, o de ordem 1 é $\frac{\sqrt{2}}{2}$, e o de ordem 2 é $(\frac{\sqrt{2}}{2})^2$.

rankToMultiplier :: *Int* → *Float*
rankToMultiplier *a* = (((*sqr*t 2) / 2) ↑ *a*)

checkComplete executa i_1 sobre um par de inteiros se estes forem iguais, ou i_2 se forem diferentes. Esta função é importante para determinar quando chegamos à última iteração do anamorfismo.

checkComplete :: (*Int*, *Int*) → (*Int*, *Int*) + (*Int*, *Int*)
checkComplete (*a*, *b*)
 | *b* < 0 = i_1 (*a*, 0)
 | *a* ≡ *b* = i_1 (*a*, *b*)
 | otherwise = i_2 (*a*, *b*)

O caso

$b < 0 = i_1 (a, 0)$

evita um loop infinito no caso de ser pedida uma *PTree* com rank negativo.

drawPTree

O grupo não definiu a função *drawPTree*.

drawPTree = ⊥

Problema 5

Para a realização deste exercício, foi necessário compreender qual a função de *singletonbag* e *muB*. A primeira permite, tendo um determinado objeto, inseri-lo num dado *Bag*. Quanto à segunda, dado um determinado *Bag* com *Bags* e o respetivo número dos mesmos, deve permitir criar apenas um *Bag* com o conteúdo de todos os interiores.

singletonbag

Para a realização de *singletonbag*, devemos primeiro criar um tuplo com o elemento que recebemos, e o respetivo número de elementos, ou seja, um.

$$A \xrightarrow{\text{toTuple}} A \times 1$$

Depois de ter o tuplo, este deve ser inserido dentro de uma lista, uma vez que o construtor de *Bag* recebe tal. Para isso, usou-se *singl*. Posteriormente, basta aplicar o construtor *B*. Abaixo, mostram-se as definições realizadas.

singletonbag = *B* · *singl* · *toTuple*
toTuple *a* = (*a*, 1)

μ

Para fazer o μ , tendo em conta o seu objetivo, optou-se por, desde logo, realizar um *fmap unB*. Isto vai permitir fazer *unBag* dos *Bags* interiores do *Bag* fornecido. Desta forma, dentro do *Bag* inicial vamos ter tuplos (cujo primeiro elemento é uma lista de elementos depois do *unBag*, e o segundo é o respetivo número de sacos iguais existentes).

$$Bag (Bag A, Int)^* \xrightarrow{\text{fmap } unB} Bag ((A, Int)^*, Int)^*$$

Depois de termos a *Bag* com tuplos, em que o primeiro elemento do mesmo é uma lista de tuplos do *Bag* interior, e em que o segundo é o número respetivo de *Bags* existentes à priori, é necessário retirar este último. Para retirar sem perder informação, devemos multiplicar o número de elementos de cada tipo em cada um dos *bags* interiores, pelo respetivo número de *bags* daquele tipo. Isso foi feito com recurso a um *map multBags*, após um *unBag*.

Faz-se então o *unB*:

$$Bag ((A, Int)^*, Int)^* \xrightarrow{unB} ((A, Int)^*, Int)^*$$

E agora aplicar a função *multBags* a toda a lista, através de um *map*.

$$((A, Int)^*, Int)^* \xrightarrow{\text{map } multBags} ((A, Int)^*)^*$$

Tendo feito todos os passos anteriores, basta concatenar as listas numa só, através da função *concat*. Naturalmente, posteriormente, aplica-se o construtor *B*, obtendo-se a *Bag* final.

$$\begin{aligned} \mu &= B \cdot \text{concat} \cdot (\text{map } multBags) \cdot unB \cdot (\text{fmap } unB) \\ multBags &:: ([(a, Int)], Int) \rightarrow [(a, Int)] \\ multBags ([], c) &= [] \\ multBags (((a, b) : tail), c) &= [(a, b * c)] ++ (multBags (tail, c)) \end{aligned}$$

dist

O *dist*, após recebido um *Bag* tem, naturalmente, que fazer o seu *unBag*. Tendo a lista de tuplos, em que o primeiro elemento é um elemento específico e o segundo o número de existentes desse elemento. Feito isto, é usada a *marbleReplication* (com recurso a *map*), para transformar tuplos de $A \times Int$ em uma lista de *A*, como se demonstra.

$$(A, Int)^* \xrightarrow{\text{map } marbleReplication} A^{**}$$

Tendo agora uma lista com as listas formadas após replicação, resta fazer o *concat* das mesmas. Esta lista fica assim pronta a ser processada pela função *uniform*, obtendo-se a distribuição pedida.

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