

Cálculo de Programas

Trabalho Prático

MiEI+LCC — 2017/18

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	13
a81047	Catarina Machado
a82339	João Vilaça

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-processor 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 suffixo ‘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 **milisegundos** 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 → 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:*

$$\text{prop1a} = \text{sort} \cdot \text{allTransactions} \equiv \text{sort} \cdot \text{allTransactions} \cdot \text{reverseChain}$$

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

2. Defina a função *ledger :: Blockchain → 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:*

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

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

$$\text{prop1c} = \text{sort} \cdot \text{ledger} \equiv \text{sort} \cdot \text{ledger} \cdot \text{reverseChain}$$

3. Defina a função *isValidMagicNr :: Blockchain → 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:*

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

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

$$\text{prop1e} = \text{isValidMagicNr} \Rightarrow \text{isValidMagicNr} \cdot \text{reverseChain}$$

Problema 2

Uma estrutura de dados frequentemente utilizada para representação e processamento de imagens de forma eficiente são as denominadas **quadrees**. Uma *quadtree* 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)
```

```

( 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*.

Figura 1: Exemplos de representações de bitmaps.

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 quad-trees, e o catamorfismo *qt2bm* executa a operação inversa:

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

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 5.

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, re-dimensionam⁴ 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"

```

²Cf. módulo *Data.Matrix*.

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.

Figura 2: Manipulação de uma figura bitmap utilizando quadrees.

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

$$\text{prop2c} = \text{rotateMatrix} \cdot \text{qt2bm} \equiv \text{qt2bm} \cdot \text{rotateQTree}$$

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,

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



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

onde

$$h \ k \ d = \frac{f \ k \ d}{g \ d}$$

$$f \ k \ d = \frac{(d+k)!}{k!}$$

$$g \ d = d!$$

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

$$f \ k \ 0 = 1$$

$$f \ k \ (d+1) = \underbrace{(d+k+1)}_{l \ k \ d} * f \ k \ d$$

$$l \ k \ 0 = k+1$$

$$l \ k \ (d+1) = l \ k \ d + 1$$

e

$$g \ 0 = 1$$

$$g \ (d+1) = \underbrace{(d+1)}_{s \ d} * g \ d$$

$$s \ 0 = 1$$

$$s \ (d+1) = s \ d + 1$$

A partir daqui alguém derivou a seguinte implementação:

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

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):

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

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:

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

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



Figura 4: Distribuição de berlindes num saco.

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* $\mu \cdot \text{return} = \text{id}$:

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

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

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

onde *b3* é um saco dado em anexo.

Anexos

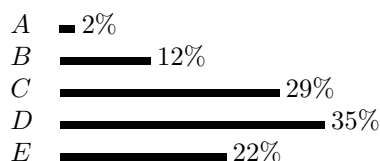
A Mónade para probabilidades e estatística

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

$$\text{newtype Dist } a = D \{ unD :: [(a, ProbRep)] \} \quad (2)$$

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

$$d1 :: \text{Dist Char}$$

$$d1 = D[(\text{'A'}, 0.02), (\text{'B'}, 0.12), (\text{'C'}, 0.29), (\text{'D'}, 0.35), (\text{'E'}, 0.22)]$$

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)  $\Rightarrow$  Show (Bag a) where
  show = showbag  $\cdot$  consol  $\cdot$  unB where
    showbag = concat  $\cdot$ 
      ( $\text{++}[\text{"  "}]$ )  $\cdot$  ( $\text{"{  ":}$ 
      (intersperse " , "
      sort  $\cdot$ 
      (map f) where  $f(a, b) = (\text{show } a) \text{ ++ "  " } \mid \rightarrow \text{" } \text{++ } (\text{show } b)$ 
unB (B x) = x

```

Igualdade de *bags*:

instance ($Eq\ a \Rightarrow Eq\ (Bag\ a)$) **where**
 $b \equiv b' = (unB\ b)\text{'lequal'}(unB\ b')$
where $lequal\ a\ b = isempty\ (a \ominus b)$
 $ominus\ a\ b = a \uplus neg\ b$
 $neg\ x = [(k, -i) \mid (k, i) \leftarrow x]$

Ainda sobre o mónade *Baq*:

```

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

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

O primeiro problema tem como tema uma block chain, ou seja, uma coleção de blocos que registam movimentos da moeda.

De modo a resolvê-lo, antes de procedermos ao desenvolvimento das suas 3 alíneas, tivemos que definir algumas funções que nos ajudarão a implementar as soluções requeridas.

```

inBlockchain = [Bc, Bcs]
outBlockchain (Bc a) = i₁ (a)
outBlockchain (Bcs (a, b)) = i₂ (a, b)
recBlockchain g = id + (id × g)
cataBlockchain g = g · (recBlockchain (cataBlockchain g)) · outBlockchain
anaBlockchain g = inBlockchain · (recBlockchain (anaBlockchain g)) · g
hyloBlockchain h g = cataBlockchain h · anaBlockchain g

```

Estas funções, nomeadamente *inBlockchain*, *outBlockchain*, *recBlockchain*, *cataBlockchain*, *anaBlockchain* e *hyloBlockchain*, podem ser deduzidas tendo em consideração o Tipo de Dados do problema, a matéria de Cálculo de Programas e com a ajuda de alguns diagramas.

Uma vez que o tipo de Blockchain é *Bc* { *bc* :: *Block* } ou *Bcs* { *bcs* :: (*Block*, *Blockchain*) }, sabemos que o *inBlockchain* e o *outBlockchain* deverão “fechar” e “abrir” a Blockchain, respetivamente, logo, conseguimos representar os diagramas:

$$Blockchain \xleftarrow{\text{inBlockchain}} Block + (Block \times Blockchain)$$

$$Blockchain \xrightarrow{\text{outBlockchain}} Block + (Block \times Blockchain)$$

Assim, conseguimos perceber de imediato a definição de ambas as funções:

$$inBlockchain = [Bc, Bcs]$$

$$outBlockchain (Bc\ a) = i_1\ (a)$$

$$outBlockchain (Bcs\ (a, b)) = i_2\ (a, b)$$

Relativamente às funções *recBlockchain*, *cataBlockchain*, *anaBlockchain* e *hyloBlockchain*, os seus tipos já estavam presentes no enunciado e o significado e intuito de cada uma delas também já era sabido.

A título de exemplo, através do diagrama em seguida conseguimos ter uma melhor perceção de qual deverá ser a definição de cada uma destas funções.

Assumimos que as funções *g* e *h* mencionadas são funções que devolvem a identidade.

$$\begin{array}{ccc}
 Blockchain & \xrightarrow{g} & Block + (Block \times Blockchain) \\
 \downarrow \text{anaBlockchain } g & & \downarrow \text{recBlockchain (anaBlockchain } g) \\
 Blockchain & \xleftarrow[\text{inBlockchain}]{\text{outBlockchain}} & Block + (Block \times Blockchain) \\
 \downarrow \text{cataBlockchain } h & & \downarrow \text{recBlockchain (cataBlockchain } h) \\
 Blockchain & \xleftarrow{h} & Block + (Block \times Blockchain)
 \end{array}$$

Assim, intuitivamente conseguimos perceber a definição de cada uma delas.

$$recBlockchain\ g = id + (id \times g)$$

$$cataBlockchain\ h = h \cdot (recBlockchain\ (cataBlockchain\ h)) \cdot outBlockchain$$

$$anaBlockchain\ g = inBlockchain \cdot (recBlockchain\ (anaBlockchain\ g)) \cdot g$$

$$hyloBlockchain\ h\ g = cataBlockchain\ h \cdot anaBlockchain\ g$$

Na resolução das alíneas recorreremos a alguns diagramas onde também fica implícito o porquê da definição de cada uma destas funções.

1. Função *allTransactions*

O objetivo da função *allTransactions* é calcular a lista com todas as transações numa dada block chain, utilizando um catamorfismo.

O diagrama desta função será:

$$\begin{array}{ccc}
 Blockchain & \xleftarrow{inBlockchain} & Block + (Block \times Blockchain) \\
 \downarrow \text{cataBlockchain } g & & \downarrow \text{recBlockchain (cataBlockchain } g) \\
 Transactions & \xleftarrow{g} & Block + (Block \times Transactions)
 \end{array}$$

O objetivo é descobrir o gene *g*, para assim termos a definição final de como algo do género *allTransactions* = *cataBlockchain* *g*

Assim, tendo em atenção o tipo de *Block*:

$$\text{type } Block = (MagicNo, (Time, Transactions))$$

E o tipo de *Blockchain* já apresentado, deduzimos que o *g* terá que ser um “either”, $g = [b0, joint]$, onde de um lado irá tratar o *Block* e do outro o $Block \times Transactions$. Isso deve-se ao facto de *g* receber como parâmetro: $Block + (Block \times Transactions)$, ou seja, uma “soma”, e devolver: *Transactions*, logo, obrigatoriamente a função terá que ser “either” para abolir o +.

(a) Descobrir *b0*

Para tratar o lado em que o domínio é *Block*, e sabendo que o resultado terá que ser *Transactions*, o objetivo desta função será "retirar" do *Block* as *Transactions*.

Assim, teremos que definir *b0* com projeções π_2 como podemos verificar no diagrama em seguida:

$$\begin{array}{c} \text{MagicNo} \times (\text{Time} \times \text{Transactions}) \\ \downarrow \pi_2 \cdot \pi_2 \\ \text{Transactions} \end{array}$$

Deste modo, fica definido *b0* como:

$$b0 = \pi_2 \cdot \pi_2$$

(b) Descobrir *joint*

Tendo em conta o domínio da função *joint*, ou seja, $\text{Block} \times \text{Transactions}$, percebemos que o objetivo desta função será retirar de *Block* as suas *Transactions* e juntá-las às *Transactions* já acumuladas (passadas como parâmetro).

Assim, uma definição de *joint pointwise* é:

$$\text{joint}(\text{block}, \text{transac}) = (\pi_2(\pi_2 \text{ block})) \uparrow \text{transac}$$

Esta função cumpre os requisitos a que a propusemos, uma vez que retira de *Block* as *Transactions*, e concatena-as às *Transactions* passadas como parâmetro.

Temos então a definição de *b0* e *joint*, pelo que ficamos a saber qual é o gene *g*:

$$\begin{aligned} g &= [b0, \text{joint}] \\ &\equiv \{ \text{Definição de } b0 ; \text{Definição de } \text{joint} \} \\ g &= [\pi_2 \cdot \pi_2, \text{joint}] \\ &\textbf{where } \text{joint}(x, y) = (\pi_2(\pi_2 x)) \uparrow y \end{aligned}$$

Deste modo, está definida a função *allTransactions* pedida:

$$\begin{aligned} \text{allTransactions } a &= \text{cataBlockchain } [\pi_2 \cdot \pi_2, \text{joint}] a \\ &\textbf{where } \text{joint}(x, y) = (\pi_2(\pi_2 x)) \uparrow y \end{aligned}$$

2. Função *ledger*

O objetivo desta função é calcular o valor disponível de cada entidade numa dada block chain.

O tipo de retorno de *ledger* deverá ser o seguinte:

$$[(\text{Entity}, \text{Value})]$$

Para o desenvolvimento desta função aproveitamos a função *allTransactions* definida na alínea anterior, que nos dá a lista de todas as transações efetuadas.

O tipo de uma Transação é o seguinte:

$$(\text{Entity}, (\text{Value}, \text{Entity}))$$

Sendo o primeiro *Entity* a entidade que envia o valor a ser transacionado e o segundo o que diz respeito à entidade que recebe.

Assim, após termos a $[\text{Transaction}]$ aplicamos-lhe um *cataList*. Este catamorfismo tem um gene $g = [\text{nil}, \text{insert}]$ que é responsável por olhar para cada *Transaction* e construir $[(\text{Entity}, \text{Value})]$. Para cada *Transaction* são adicionados dois novos pares à lista: o primeiro com (Entidade que enviou, - Valor que enviou) e o segundo com (Entidade que recebeu, Valor que recebeu).

Esse catamorfismo pode ser representado pelo seguinte diagrama:

$$\begin{array}{ccc}
 [Transaction] & \xleftarrow{[nil, cons]} & 1 + (Transaction \times [Transaction]) \\
 \downarrow cataList\ g & & \downarrow id + (id \times (cataList\ g)) \\
 [(Entity, Value)] & \xleftarrow{g=[nil, insert]} & 1 + (Transaction \times [(Entity, Value)])
 \end{array}$$

Depois de termos a lista com todos os valores transacionados (tanto recebidos como enviados) e respectivas entidades aplicamos-lhe uma função chamada *groupL*.

A função *groupL* ordena a $[(Entity, Value)]$, agrupa devidamente os seus elementos e por fim devolve $[(Entity, Value)]$ mas sem repetições no que diz respeito às entidades, ou seja, a função soma todas as transações da entidade (quando se trata de um valor enviado o mesmo é negativo), ficando assim com os pares (Entidade, Valor Disponível).

Consequentemente, temos a função *ledger* definida:

```

ledger a = groupL (cataList [nil, insert] (allTransactions a))
  where insert (x, y) = (π1 x, -π1 (π2 x)) : (π2 (π2 x), π1 (π2 x)) : y
        groupL t = (sums · map (mapFst head · unzip) · groupBy (λx y → π1 x ≡ π1 y) · sort) t
        mapFst f (a, b) = (f a, b)
        sums [] = []
        sums ((a, b) : t) = (a, sum b) : sums t

```

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

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

3. Função *isValidMagicNr*

```

isValidMagicNr a = all ((≡) 1 · length) · group · sort $ cataBlockchain [list, insert] a
  where list x = [π1 x]
        insert (x, y) = (π1 x) : y

```

Problema 2

O segundo problema tem como tema uma estrutura de dados que é muito utilizada para representação e processamento de imagens- quadtrees. Tal como é referido no enunciado do problema, uma quadtree é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

Antes de procedermos ao desenvolvimento das funções propostas neste problema definimos algumas funções que nos serão muito úteis.

Uma *QTree* poderá ser:

Cell a Int Int

Ou

Block (QTree a) (QTree a) (QTree a) (QTree a)

Em consequência, as definições de *inQTree* e *outQTree* terão que ser as seguintes:

```

inQTree = [uncurryCell, uncurryBlock]
  where uncurryCell (e, (n1, n2)) = Cell e n1 n2
        uncurryBlock :: (QTree a, (QTree a, (QTree a, QTree a))) → QTree a
        uncurryBlock (q1, (q2, (q3, q4))) = Block q1 q2 q3 q4
        outQTree (Cell e n1 n2) = i1 (e, (n1, n2))
        outQTree (Block q1 q2 q3 q4) = i2 (q1, (q2, (q3, q4)))

```

O diagrama da função $inQTree$ é o seguinte:

$$QTree\ a \xleftarrow{inQTree} (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))$$

E o diagrama da função $outQTree$ é o seguinte:

$$QTree\ a \xrightarrow{outQTree} (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))$$

No caso da função $inQTree$, que “fecha” a $QTree$, o retorno deverá ser uma $QTree$, logo, no caso da $Cell$ (lado esquerdo do $+$) temos que ajustar o parâmetro recebido $(a, (Int, Int))$ devolvendo $Cell\ a\ Int\ Int$, para assim a função devolver a informação no tipo de dados correto. No caso do $Block$, recebendo $(q1, (q2, (q3, q4)))$ teremos que retornar $Block\ q1\ q2\ q3\ q4$. Esta função é definida como um “either” porque temos estas duas “hipóteses” de tipo de dados dentro do tipo de dados $QTree$.

No caso da função $outQTree$ o raciocínio é o inverso. Uma vez que esta função recebe uma $QTree$ podemos definir a função com dois casos diferentes, tal como se pode ver na solução por nós proposta.

Os dois diagramas em seguida ajudam-nos a perceber melhor como tratar os dois casos de $QTree$ na função $outQTree$:

$$\begin{array}{c} (a, (Int, Int)) \\ \downarrow i_1 \\ (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \end{array}$$

e

$$\begin{array}{c} (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\ \downarrow i_2 \\ (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \end{array}$$

Assim, quando recebemos uma $Cell$ e $n1\ n2$ o objetivo será injetá-la à esquerda de modo a respeitar o tipo de dados, devolvendo então: $i_1\ (e, (n1, n2))$. No caso de $Block\ q1\ q2\ q3\ q4$, injetamos à direita retornando: $i_2\ (q1, (q2, (q3, q4)))$.

Outras funções cruciais para a resolução deste problema são as seguintes:

$$\begin{aligned} baseQTree\ f\ g &= (f \times id) + (g \times (g \times (g \times g))) \\ recQTree\ g &= baseQTree\ id\ g \\ cataQTree\ g &= g \cdot (recQTree\ (cataQTree\ g)) \cdot outQTree \\ anaQTree\ g &= inQTree \cdot (recQTree\ (anaQTree\ g)) \cdot g \\ hyloQTree\ h\ g &= cataQTree\ h \cdot anaQTree\ g \end{aligned}$$

Para melhor compreensão do intuito de cada uma delas desenhamos dois diagramas:

O primeiro, mostra a função $baseQTree$. Atendendo ao seu tipo, já contido no enunciado, conseguimos perceber qual é o objetivo desta função. A título de exemplo, vamos considerar que f tem um tipo: $f :: A \rightarrow E$ e que g tem um tipo: $g :: C \rightarrow D$:

$$\begin{array}{c} (a, b) + (c, (c, (c, c))) \\ \downarrow baseQTree\ f\ g \\ (e, b) + (d, (d, (d, d))) \end{array}$$

Assim, percebemos de imediata que a função $baseQTree$ terá que ser definida como $(f \times id) + (g \times (g \times (g \times g)))$:

$$\begin{array}{c}
(a, b) + (c, (c, (c, c))) \\
\downarrow (f \times id) + (g \times (g \times (g \times g))) \\
(e, b) + (d, (d, (d, d)))
\end{array}$$

O segundo, que é um pouco mais complexo, é apenas um exemplo do que se pode fazer com a combinação destas funções, nomeadamente *inQTree*, *outQTree*, *recQTree*, *cataQTree*, *anaQTree* e *hyloQTree*. Vamos assumir que neste nosso diagrama as funções *g* e *h* mencionadas são funções que devolvem a identidade, ou seja, não alteram o conteúdo da *QTree*, mas respeitam os tipos de dados ideais:

$$\begin{array}{ccc}
QTree\ a & \xrightarrow{g} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\
\downarrow anaQTree\ g & & \downarrow recQTree\ (anaQTree\ g) \\
QTree\ a & \xleftarrow[inQTree]{outQTree} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\
\downarrow cataQTree\ h & & \downarrow recQTree\ (cataQTree\ h) \\
QTree\ a & \xleftarrow{h} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))
\end{array}$$

A função *hyloQTree* é definida como sendo *hyloQTree h g = cataQTree h · anaQTree g*, ou seja, no diagrama anterior pode ser identificado por uma seta vertical que vai desde o argumento da função *anaQTree* até ao retorno da função *cataQTree*.

Assim, com a ajuda destes diagramas, encontramos as definições procuradas.

Por fim, temos *fmap*, que tem como objetivo aplicar a função *f* a todas as *Cell* da *QTree*, mais especificamente ao conteúdo da *Cell* que diz respeito ao valor/ objeto da matriz (não à dimensão). A função simplesmente aplica a todos esses elementos a função *f*.

Assim, decidimos definir a nossa função *fmap* da seguinte forma:

```
instance Functor QTree where
  fmap f = cataQTree (inQTree · baseQTree f id)
```

No seguinte diagrama conseguimos perceber o que é que as funções *inQTree · baseQTree f id* fazem, considerando que *f* é uma função do tipo: *f :: A → E*:

$$\begin{array}{c}
(a, b) + (c, (c, (c, c))) \\
\downarrow baseQTree\ f\ id \\
(e, b) + (c, (c, (c, c))) \\
\downarrow inQTree \\
QTree\ e
\end{array}$$

Ou seja, numa primeira fase a função *baseQTree f id* aplica a função *f* ao conteúdo da *Cell* e numa segunda fase a função *inQTree* junta o resultado da aplicação da função *baseQTree f id* numa *QTree*.

Assim, aplicando um *cataQTree* a esta composição de funções construímos:

$$\begin{array}{ccc}
QTree\ a & \xleftarrow{inQTree} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\
\downarrow cataQTree\ g & & \downarrow recQTree\ (cataQTree\ g) \\
QTree\ e & \xleftarrow{g} & (a, (Int, Int)) + (QTree\ e, (QTree\ e, (QTree\ e, QTree\ e)))
\end{array}$$

Sendo *g f = inQTree · baseQTree f id*.

Em suma, tal como nos diz a própria definição de catamorfismo, na seta vertical mais à direita o mesmo é aplicado recursivamente à parte direita do + (o *Functor*, ou seja, *recQTree* encarrega-se disso) e, depois disso, temos então a "cauda" processada, tal como podemos ver no diagrama. O nosso gene *g* responsabiliza-se pelo último passo de transformar na *Cell* (lado esquerdo do + inferior) o seu conteúdo (através da função *f*) e de juntar tudo numa só *QTree*.

Agora reunimos todas as condições para nos concentrarmos no desenvolvimento das alíneas deste problema.

1. Função *rotateQTree*

O objetivo desta função é rodar uma *QTree*.

Optamos por utilizar um catamorfismo para definir esta função. Assim, temos que $rotateQTree = cataQTree\ g$. Tendo em conta a definição de *cataQTree* podemos definir *g* como um “either”, onde um dos lados irá tratar a *Cell* e o outro o *Block*.

Consequentemente, para perceber que impacto esta função *rotateQTree* terá na *QTree* analisamos a matriz de bits (Figura 1a) e a respetiva codificação em quadrees (Figura 5).

Vamos agora exemplificar o nosso raciocínio com alguns exemplos:

Numa primeira fase iremos analisar o que acontecerá numa *Cell*. Por exemplo, se tivermos a *Cell*:

```
( 0 0 0 0 )
( 0 0 0 0 )
```

Que é representada por *Cell* 0 4 2, rodando-a 90° fica:

```
( 0 0 )
( 0 0 )
( 0 0 )
( 0 0 )
```

Que é representada por *Cell* 0 2 4.

Logo, fica implícito que no que diz respeito a rodar uma *Cell* o que acontece é que as suas dimensões verticais e horizontais trocam.

Definimos então a função *rotateCell* que roda uma *Cell*:

$$rotateCell\ (e, (n1, n2)) = Cell\ e\ n2\ n1$$

O tipo desta função terá que ser o acima apresentado tendo em consideração os tipos da função *cataQTree* e *inQTree*.

Numa segunda fase analisamos o que acontece num *Block*.

Procedendo da mesma forma, definimos um *Block* de exemplo:

```
( 0 0 0 0 )
( 0 0 0 0 )
( 1 1 1 1 )
( 1 1 1 1 )
```

Que é definida por *Block* (*Cell* 0 2 2) (*Cell* 0 2 2) (*Cell* 1 2 2) (*Cell* 1 2 2).

Rodando o *Block* 90°:

```
( 1 1 0 0 )
( 1 1 0 0 )
( 1 1 0 0 )
( 1 1 0 0 )
```

Ficamos com uma *Block* (*Cell* 1 2 2) (*Cell* 0 2 2) (*Cell* 1 2 2) (*Cell* 0 2 2).

Assim, percebemos que os quatro parâmetros de *Block* rodam entre si. O primeiro parâmetro passará a ser o segundo parâmetro, o segundo passará a ser o último, o terceiro fica em primeiro e, por fim, o último parâmetro fica a ser o terceiro.

Conseguimos então perceber a definição que trata o *Block*:

$$rotateBlock\ (q1, (q2, (q3, q4))) = Block\ q3\ q1\ q4\ q2$$

Assim, temos todas as condições necessárias para definir a função pedida, *rotateQTree*:

```
rotateQTree = cataQTree [rotateCell, rotateBlock]
  where rotateCell (e, (n1, n2)) = Cell e n2 n1
        rotateBlock (q1, (q2, (q3, q4))) = Block q3 q1 q4 q2
```

O diagrama desta função é o seguinte:

$$\begin{array}{ccc}
 QTree\ a & \xleftarrow{inQTree} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\
 \downarrow cataQTree\ g & & \downarrow recQTree\ (cataQTree\ g) \\
 QTree\ a & \xleftarrow{g} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))
 \end{array} \quad (3)$$

Onde:

```
g = [rotateCell, rotateBlock]
  where rotateCell (e, (n1, n2)) = Cell e n2 n1
        rotateBlock (q1, (q2, (q3, q4))) = Block q3 q1 q4 q2
```

2. Função *scaleQTree*

Esta função redimensiona uma *Qtree* tendo em consideração o *Int* passado como parâmetro.

Assim, intuitivamente percebemos que teremos que multiplicar o fator passado como parâmetro pela dimensão da matriz.

Mais uma vez recorreremos a um *cataQTree* para definir a função *scaleQTree*. Deste modo, utilizando o mesmo raciocínio da função anterior, precisamos de definir o gene *g* como um “either” onde a *Cell* e o *Block* serão tratados individualmente.

O gene *g* terá a seguinte definição:

```
g n = [scaleCell n, uncurryBlock]
```

Uma vez que as dimensões da matriz são somente responsabilidade da *Cell* não teremos que alterar nada no *Block*.

De forma a respeitar os tipos, no que diz respeito a tratar o *Block*, utilizamos a função *uncurryBlock* já definida por nós, que apenas altera a forma como o *Block* é mostrado, não alterando o seu conteúdo.

Para tratar a *Cell* vamos recorrer a uma função auxiliar, *scaleCell*, cujo único objetivo será multiplicar as dimensões da *Cell* pelo fator e devolvê-la no tipo de dados correto:

```
scaleCell n (e, (n1, n2)) = Cell e (n1 * n) (n2 * n)
```

O diagrama desta função é o mesmo da função anterior, diagrama (3), variando somente o gene *g*, que neste caso é o anteriormente referido.

Assim, temos definida a função *scaleQTree*:

```
scaleQTree n = cataQTree [scaleCell n, uncurryBlock]
  where scaleCell n (e, (n1, n2)) = Cell e (n1 * n) (n2 * n)
```

3. Função *invertQTree*

O intuito da função *invertQTree* é inverter as cores de uma quadtree. Assim, terá obrigatoriamente de se tratar de uma matriz de píxeis, neste caso de *PixelRGBA8*. É nos também dito que o pixel pode ser invertido calculando (255 - w), sendo w a componente de cor RGB.

Logo, utilizando o mesmo raciocínio da função *scaleQTree*, vamos definir a função *invertQTree* como um *cataQTree* onde também somente a *Cell* precisa de ser modificada, tendo em conta que é a *Cell* que possui o conteúdo da *QTree*, ou seja, o parâmetro que nos interessa alterar.

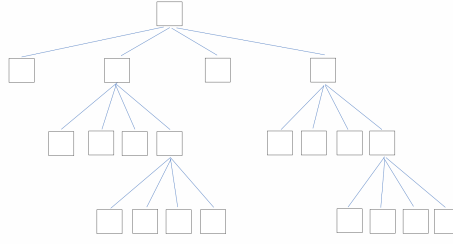


Figura 5: Esquema da *QTree qt*.

Logo, teremos um gene g com a seguinte definição:

$$g = [invertCell, uncurryBlock]$$

Precisamos então somente de definir a função *invertCell*, que poderá ser definida como:

$$\begin{aligned} invertCell ((PixelRGBA8 \ a \ b \ c \ d), (n1, n2)) = \\ Cell (PixelRGBA8 \ (255 - a) \ (255 - b) \ (255 - c) \ (255 - d)) \ n1 \ n2 \end{aligned}$$

Esta função pode também ser ilustrada através do diagrama (3), mas com um gene g definido como o mencionado anteriormente.

Consequentemente, temos todas as condições necessárias para definir a função *invertQTree*:

$$\begin{aligned} invertQTree = cataQTree [invertCell, uncurryBlock] \\ \textbf{where } invertCell ((PixelRGBA8 \ a \ b \ c \ d), (n1, n2)) = \\ Cell (PixelRGBA8 \ (255 - a) \ (255 - b) \ (255 - c) \ (255 - d)) \ n1 \ n2 \end{aligned}$$

4. Função *compressQTree*

O objetivo desta função é comprimir a *QTree* cortando folhas da árvore de modo a reduzir a sua profundidade num dado número de níveis, passado como parâmetro na função. Basicamente, o que irá acontecer à imagem é perder informação e por isso “desfocar”.

Para o desenvolvimento desta função recorreremos a uma *anaQTree* uma vez que consideramos que seria mais simples de tratar o problema usando esse conceito/ definição.

De uma forma mais concreta, pensamos em utilizar uma *anaQTree* uma vez que sabíamos que ela se iria concentrar em cada nível da árvore gradualmente, começando da raiz até às folhas. Assim, de uma forma geral, pensamos que a nossa *anaQTree* deveria ter um gene que averigua-se se estamos no nível desejado e em caso afirmativa eliminasse toda a *QTree* a partir desse nível.

Assim, passamos para o chamado desenvolvimento do problema:

Temos por exemplo *QTree* da Figura 5, que é representada pela *QTree qt*, que já vinha também no enunciado:

$$\begin{aligned} qt = Block \\ (Cell \ 0 \ 4 \ 4) \\ (Block \ (Cell \ 0 \ 2 \ 2) \ (Cell \ 0 \ 2 \ 2) \ (Cell \ 1 \ 2 \ 2) \ (Block \ (Cell \ 1 \ 1 \ 1) \ (Cell \ 0 \ 1 \ 1) \ (Cell \ 0 \ 1 \ 1) \ (Cell \ 0 \ 1 \ 1))) \\ (Cell \ 1 \ 4 \ 4) \\ (Block \ (Cell \ 1 \ 2 \ 2) \ (Cell \ 0 \ 2 \ 2) \ (Cell \ 0 \ 2 \ 2) \ (Block \ (Cell \ 0 \ 1 \ 1) \ (Cell \ 0 \ 1 \ 1) \ (Cell \ 0 \ 1 \ 1) \ (Cell \ 1 \ 1 \ 1))) \end{aligned}$$

Se quisermos aplicar a função *compressQTree 2*, ou seja, comprimir dois níveis, o esquema da *QTree* de retorno deverá ser o seguinte o esquema presenta na Figura 6, representado por:

$$\begin{aligned} qt_compress2 = Block \\ (Cell \ 0 \ 4 \ 4) \\ (Cell \ 0 \ 4 \ 4) \\ (Cell \ 1 \ 4 \ 4) \\ (Cell \ 1 \ 4 \ 4) \end{aligned}$$

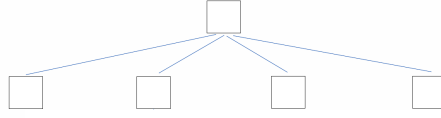


Figura 6: Esquema da *QTree qt* comprimida em 2 níveis.

Deste modo, apercebemo-nos que teríamos que ter uma função que nos “cortasse” todos os ramos da *QTree* a abolir. Percebemos através da Figura 5 e da Figura 6 que se essa função, que poderá chamar-se *corta*, receber como parâmetro a parte da *Qtree* a eliminar deveremos transforma-la numa única *Cell*.

Nesta fase, tivemos que perceber quais as dimensões da *Cell* resultado pelo que, por exemplo, se tivermos:

Block (*Cell* 0 3 2) (*Cell* 1 2 2) (*Cell* 1 4 2) (*Cell* 0 1 2)

Representado por:

```
( 0 0 0 1 1 )
( 0 0 0 1 1 )
( 1 1 1 1 0 )
( 1 1 1 1 0 )
```

A *Cell* resultado deverá ser:

Cell 0 5 4

E em modo matriz: Representado por:

```
( 0 0 0 0 0 )
( 0 0 0 0 0 )
( 0 0 0 0 0 )
( 0 0 0 0 0 )
```

Assim, percebemos que se o *Block a b c d* for então constituído apenas com *Cell*, ou seja, *a, b, c, d*:: *Cell*, as dimensões resultado deverão ser calculadas da seguinte forma:

$$\text{corta} (\text{Block} (\text{Cell } c1 \ n1 \ n2) (\text{Cell } c2 \ m1 \ m2) (\text{Cell } c3 \ k1 \ k2) (\text{Cell } c4 \ o1 \ o2)) = \\ \text{Cell } (c1) (n1 + m1) (n2 + k2)$$

Optamos por dar como resultado o valor da primeira *Cell*, que no nosso caso é *c1*.

No caso de o *Block* ainda não ser constituído por apenas *Cell*, tal como referimos anteriormente, aplicamos recursivamente a função *corta* a cada um dos “argumentos” de *Block*, e ao próprio *Block*, até conseguirmos chegar a um *Block* só com *Cell* e o convertermos para *Cell* da forma já referida. A recursividade irá tratar de nos fornecer a solução que desejamos. Para esta hipótese a função *corta* será então:

$$\text{corta} (\text{Block } q1 \ q2 \ q3 \ q4) = \text{corta} (\text{Block} (\text{corta } q1) (\text{corta } q2) (\text{corta } q3) (\text{corta } q4))$$

Pensando no caso de quando esta função é aplicada a somente uma *Cell* isolada a função deverá retornar a *Cell* argumento exatamente igual:

$$\text{corta} (\text{Cell } a \ b \ c) = \text{Cell } a \ b \ c$$

Juntando estas 3 hipóteses, temos então a função *corta* definida:

$$\begin{aligned}
corta (Cell\ a\ b\ c) &= Cell\ a\ b\ c \\
corta (Block\ (Cell\ c1\ n1\ n2)\ (Cell\ c2\ m1\ m2)\ (Cell\ c3\ k1\ k2)\ (Cell\ c4\ o1\ o2)) &= \\
Cell\ (c1)\ (n1 + m1)\ (n2 + k2) \\
corta (Block\ q1\ q2\ q3\ q4) &= corta (Block\ (corta\ q1)\ (corta\ q2)\ (corta\ q3)\ (corta\ q4))
\end{aligned}$$

Assim, precisamos apenas de definir concretamente, e através de código, em que momento esta função será aplicada.

Aproveitando a função *depthQTree*, que calcula a profundidade de uma *QTree*, podemos então contruir uma função auxiliar, por exemplo chamada *compress*, que irá receber o número de níveis a eliminar. Como sabemos que o nosso *anaQTree* irá olhar individualmente para cada nível, a nossa *compress n* averiguará se a profundidade do ramo atual é igual ao número de níveis a cortar.

Podemos utilizar esta técnica pois o anamorfismo percorre cada ramo de cima (raiz) para baixo e não volta para cima, por isso, quando o sub-ramo tiver uma profundidade igual ao número de níveis a cortar sabemos que é esse o nível que procuramos, para podermos aplicar a nossa função *corta*. Se ainda não tivermos chegado ao nível que procuramos, a função *compress* deverá devolver a *QTree*, ou seja, o ramo, intacto.

Tivemos que ter em consideração o caso em que o número de níveis a cortar é maior que toda a profundidade da árvore, sendo que neste caso toda a *QTree* deverá ser cortada (é aplicada à função *cortar*) ficando somente a *Cell* comprimida.

É ainda mencionar que, tendo em conta o tipo de *anaQTree*, a nossa *compress* deverá retornar sempre uma *QTree* “aberta” e, assim, aplicamos a função *outQTree*.

Finalmente, juntando todos estes passos, temos a função *compressQTree* definida:

$$\begin{aligned}
corta &:: QTree\ a \rightarrow QTree\ a \\
corta (Cell\ a\ b\ c) &= Cell\ a\ b\ c \\
corta (Block\ (Cell\ c1\ n1\ n2)\ (Cell\ c2\ m1\ m2)\ (Cell\ c3\ k1\ k2)\ (Cell\ c4\ o1\ o2)) &= \\
Cell\ (c1)\ (n1 + m1)\ (n2 + k2) \\
corta (Block\ q1\ q2\ q3\ q4) &= corta (Block\ (corta\ q1)\ (corta\ q2)\ (corta\ q3)\ (corta\ q4)) \\
compressQTree\ n &= anaQTree\ (compress\ n) \\
\textbf{where}\ compress\ n\ a \mid (n \equiv (tam\ a) \vee n > (tam\ a)) &= outQTree\ (corta\ a) \\
\mid otherwise &= outQTree\ a \\
tam\ a &= depthQTree\ a
\end{aligned}$$

O anamorfismo que representa esta função é o seguinte:

$$\begin{array}{ccc}
QTree\ a & \xrightarrow{g} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\
\downarrow anaQTree\ g & & \downarrow recQTree\ (anaQTree\ g) \\
QTree\ a & \xleftarrow{inQTree} & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))
\end{array} \quad (4)$$

Tal como dizia no enunciado, contruímos as *QTrees* com compressões 1, 2, 3 e 4, obtendo então, respetivamente, a Figura 7a, a Figura 7b, a Figura 8a e a Figura 8b.

5. Função *outlineQTree*

A função *outlineQTree* 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.

Após analisar o problema e as funções que o enunciado já fornece percebemos que podemos aproveitar uma função já existente e adaptá-la ao nosso problema. Esta função é a função *qt2bm*, que converte uma *QTree a* em *Matrix a*.



(a) Figura Person comprimida em 1 nível.



(b) Figura Person comprimida em 2 níveis.



(a) Figura Person comprimida em 3 níveis.



(b) Figura Person comprimida em 4 níveis.

```

qt2bm :: (Eq a) => QTree a -> Matrix a
qt2bm = cataQTree [f, g] where
f (k, (i, j)) = matrix j i k
g (a, (b, (c, d))) = (a < > b) < - > (c < > d)

```

Assim, olhamos para o gene do *cataQTree [f, g]* e pensamos no que teremos que alterar para a nossa função *outlineQTree* retornar uma *Matrix Bool*. Sabemos que teremos que aplicar a função passada como parâmetro às *Cell* de modo a saber se a mesma se trata de um píxel (conjunto de píxeis) de fundo.

Uma vez que é a *Cell* que contém os píxeis, no que diz respeito ao *Block* nada precisará de ser feito. Logo, a função *g* será exatamente a mesma.

No caso das *Cell*, teremos então que dividir a função *f* em dois casos, um deles quando após aplicada a função dada como parâmetro ao conteúdo da *Cell* dá *True* e o segundo quando dá *False*:

(a) *True*- conteúdo da *Cell* ser um píxel de fundo:

Neste caso, sabemos que teremos que devolver uma *Matrix Bool* onde o interior será *False* e o contorno *True*, por exemplo, para uma Matriz de 4x4:

```

( True True  True  True )
( True False False True )
( True False False True )
( True True  True  True )

```

Logo, aproveitando a função *matrix*, usamos uma expressão lambda onde se as coordenadas (x, y) da matriz pertencerem à borda da mesma, ou seja, isso acontece quando $x \equiv 1$ ou $y \equiv 1$ ou $x \equiv largura\ maxima$ ou $y \equiv altura\ maxima$, o valor é *True*, caso contrário o valor será *False*.

(b) *False*- O conteúdo da *Cell* não ser um píxel de fundo:

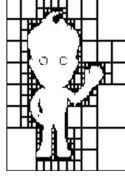
Neste caso, basta aplicar a função *matrix* onde o conteúdo dos elementos é *False*.

Consequentemente, temos todos os dados para definir a nossa função *outlineQTree*:

```

outlineQTree magic a = cataQTree [f magic, g] a
  where f magic (k, (i, j))
        | (magic k) = matrix j i (\(x, y) -> if (x == 1 || y == 1 || x == j || y == i) then True else False)

```



(a) Figura Person produzida com a função *outlineBMP*. (b) Figura Person produzida com a função *addOutlineBMP*.

$$| \text{otherwise} = \text{matrix } j \text{ } i \text{ } \underline{\text{False}} \\ g(a, (b, (c, d))) = (a \uparrow b) \leftrightarrow (c \uparrow d)$$

No enunciado desta função era ainda sugerido que produzíssemos imagens com o auxílio de duas funções que utilizam a *outlineQTree*. As duas imagens que obtivemos estão presentes na Figura 9a e a Figura 9b.

É ainda de salientar que todas as funções do problema passaram em todos os testes do QuickCheck e nos Testes Unitários.

Problema 3

O objetivo deste problema é derivar as funções *base k* e *loop* de modo a podermos calcular as combinações de n k -a- k ,

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

recorrendo a um ciclo-for onde apenas se fazem multiplicações e somas:

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

Tendo em conta o **where** $h \ k \ n$ e o **in** a / b da função acima apresentada e comparando com a definição de $h \ k$ do enunciado,

$$h \ k \ n = \frac{f \ k \ n}{g \ n} \\ f \ k \ n = \frac{(n + k)!}{k!} \\ g \ n = n!$$

constatamos que o nosso **a** em **in** a / b terá que ser a função $f \ k$ e o **b** será a função g .

Assim, para podermos descobrir a definição das funções pedidas (*base k* e *loop*) teremos que, a partir da definição de $f \ k$ (e consequentemente de $l \ k$) e da definição de g (e consequentemente de s), descobriremos a definição de $\langle f \ k, l \ k \rangle$ e de $\langle g, s \rangle$ (com recurso à lei da recursividade múltipla) e posteriormente combinarmos os seus resultados com a lei de banana-split.

Para tal, dividimos o nosso problema em diferentes partes:

1. Determinar $\langle f \ k, l \ k \rangle$

Para isso, tirando partido da indicação do enunciado, ou seja, com o intuito de aplicar a lei da recursividade múltipla, temos:

$$\begin{cases} f \ k \cdot \text{in} = o \cdot F \langle f \ k, l \ k \rangle \\ l \ k \cdot \text{in} = p \cdot F \langle f \ k, l \ k \rangle \end{cases} \\ \equiv \{ \text{Fokkinga: (50)} \} \\ \langle f \ k, l \ k \rangle = \langle \langle o, p \rangle \rangle$$

O objetivo será então determinar o e p e, para isso, teremos que olhar individualmente para cada uma das funções $f \ k$ e $l \ k$, apresentadas no enunciado:

(a) Descobrir o (com a ajuda da função $f \ k$)

$$\begin{aligned}
& \left\{ \begin{array}{l} f \ k \ 0 = 1 \\ f \ k \ (d + 1) = (l \ k \ d) * (f \ k \ d) \end{array} \right. \\
\equiv & \quad \{ \text{Def comp: (74) ; (*) escrita como função prefixo} \} \\
& \left\{ \begin{array}{l} f \ k \cdot 0 = 1 \\ f \ k \cdot (d + 1) = (*) \ (l \ k \ d) \ (f \ k \ d) \end{array} \right. \\
\equiv & \quad \{ \text{Igualdade extencional: (73) ; Def const: (76)} \} \\
& \left\{ \begin{array}{l} f \ k \cdot \underline{0} = \underline{1} \\ f \ k \cdot (d + 1) = (*) \ (l \ k \ d) \ (f \ k \ d) \end{array} \right. \\
\equiv & \quad \{ \text{Definição de mul: mul (a, b) = (*) a b ; } \} \\
& \left\{ \begin{array}{l} f \ k \cdot \underline{0} = \underline{1} \\ f \ k \cdot (d + 1) = mul \ (f \ k \ d, l \ k \ d) \end{array} \right. \\
\equiv & \quad \{ \text{Def split: (78) ; Def comp: (74) ; Definição de succ: succ d = d + 1 ; Igualdade extencional: (73)} \} \\
& \left\{ \begin{array}{l} f \ k \cdot \underline{0} = \underline{1} \\ f \ k \cdot (\text{succ}) = mul \cdot \langle f \ k, l \ k \rangle \end{array} \right. \\
\equiv & \quad \{ \text{Eq + : (27) ; Natural id : (1)} \} \\
& [f \ k \cdot \underline{0}, f \ k \cdot (\text{succ})] = [\underline{1} \cdot id, mul \cdot \langle f \ k, l \ k \rangle] \\
\equiv & \quad \{ \text{Fusão + : (20) ; Absorção x : (11)} \} \\
& f \ k \cdot [0, (\text{succ})] = [\underline{1}, mul] \cdot (id + \langle f \ k, l \ k \rangle) \\
\equiv & \quad \{ \text{Definição de in e functor (dos naturais): } \mathbf{in} = [0, (\text{succ})], F \langle f \ k, l \ k \rangle = (id + \langle f \ k, l \ k \rangle) \} \\
& f \ k \cdot \mathbf{in} = [\underline{1}, mul] \cdot F \langle f \ k, l \ k \rangle
\end{aligned}$$

Logo, $o = [\underline{1}, mul]$.

(b) Descobrir p (com a ajuda da função $l \ k$)

$$\begin{aligned}
& \left\{ \begin{array}{l} l \ k \ 0 = k + 1 \\ l \ k \ (d + 1) = l \ k \ d + 1 \end{array} \right. \\
\equiv & \quad \{ \text{Def comp: (74)} \} \\
& \left\{ \begin{array}{l} l \ k \cdot 0 = k + 1 \\ l \ k \cdot (d + 1) = l \ k \ d + 1 \end{array} \right. \\
\equiv & \quad \{ \text{Definição de succ: succ d = d + 1 ; Def const: (76) ; Igualdade extencional: (73)} \} \\
& \left\{ \begin{array}{l} l \ k \cdot \underline{0} = (\text{succ } k) \\ l \ k \cdot (\text{succ}) = succ \cdot (l \ k) \end{array} \right. \\
\equiv & \quad \{ \text{Eq + : (27) ; Natural id : (1)} \} \\
& [l \ k \cdot \underline{0}, l \ k \cdot (\text{succ})] = [(\text{succ } k), succ \cdot l \ k] \\
\equiv & \quad \{ \text{Natural id : (1) ; Cancelamento x : (7)} \} \\
& [l \ k \cdot \underline{0}, l \ k \cdot (\text{succ})] = [(\text{succ } k) \cdot id, succ \cdot \pi_2 \cdot \langle f \ k, l \ k \rangle] \\
\equiv & \quad \{ \text{Fusão + : (20) ; Absorção x : (11)} \} \\
& l \ k \cdot [0, (\text{succ})] = [(\text{succ } k), succ \cdot \pi_2] \cdot (id + \langle f \ k, l \ k \rangle) \\
\equiv & \quad \{ \text{Definição de in e functor (dos naturais): } \mathbf{in} = [0, (\text{succ})], F \langle f \ k, l \ k \rangle = (id + \langle f \ k, l \ k \rangle) \} \\
& l \ k \cdot \mathbf{in} = [(\text{succ } k), succ \cdot \pi_2] \cdot F \langle f \ k, l \ k \rangle
\end{aligned}$$

Logo, $p = [(\text{succ } k), succ \cdot \pi_2]$.

Deste modo, após encontrarmos a definição de o e de p conseguimos determinar a definição de $\langle f\ k, l\ k \rangle$ uma vez que já havíamos concluído que $\langle f\ k, l\ k \rangle = \langle \langle o, p \rangle \rangle$.

Logo, $\langle f\ k, l\ k \rangle = \langle \langle [1, mul], [(\text{succ } k), \text{succ} \cdot \pi_2] \rangle \rangle$.

2. Determinar $\langle g, s \rangle$

Para descobrir $\langle g, s \rangle$ seguimos o mesmo raciocínio, isto é, tentamos descobrir um v e um j de modo a podermos aplicar a lei da recursividade múltipla:

$$\begin{aligned} & \begin{cases} g \cdot \mathbf{in} = v \cdot F\ \langle g, s \rangle \\ s \cdot \mathbf{in} = j \cdot F\ \langle g, s \rangle \end{cases} \\ \equiv & \quad \{ \text{Fokkinga: (50)} \} \\ & \langle g, s \rangle = \langle \langle v, j \rangle \rangle \end{aligned}$$

(a) Descobrir v (com a ajuda da função g)

$$\begin{aligned} & \begin{cases} g\ 0 = 1 \\ g\ (d + 1) = (g\ d) * (s\ d) \end{cases} \\ \equiv & \quad \{ \text{Def comp: (74)} ; (*) \text{ escrita como função prefixo} \} \\ & \begin{cases} g \cdot 0 = 1 \\ g \cdot (d + 1) = (*)\ (g\ d)\ (s\ d) \end{cases} \\ \equiv & \quad \{ \text{Igualdade extencional: (73)} ; \text{Def const: (76)} \} \\ & \begin{cases} g \cdot \underline{0} = \underline{1} \\ g \cdot (d + 1) = (*)\ (g\ d)\ (s\ d) \end{cases} \\ \equiv & \quad \{ \text{Definição de mul: mul (a, b) = (*) a b ; } \} \\ & \begin{cases} g \cdot \underline{0} = \underline{1} \\ g \cdot (d + 1) = \text{mul}\ (g\ d, s\ d) \end{cases} \\ \equiv & \quad \{ \text{Def split: (78)} ; \text{Def comp: (74)} ; \text{Definição de succ: succ d = d + 1 ; Igualdade extencional: (73)} \} \\ & \begin{cases} g \cdot \underline{0} = \underline{1} \\ g \cdot (\text{succ}) = \text{mul} \cdot \langle g, s \rangle \end{cases} \\ \equiv & \quad \{ \text{Eq + : (27)} ; \text{Natural id : (1)} \} \\ & [g \cdot \underline{0}, g \cdot (\text{succ})] = [\underline{1} \cdot id, \text{mul} \cdot \langle g, s \rangle] \\ \equiv & \quad \{ \text{Fusão + : (20)} ; \text{Absorção x : (11)} \} \\ & g \cdot [\underline{0}, (\text{succ})] = [\underline{1}, \text{mul}] \cdot (id + \langle g, s \rangle) \\ \equiv & \quad \{ \text{Definição de in e functor (dos naturais): } \mathbf{in} = [\underline{0}, (\text{succ})], F\ \langle g, s \rangle = (id + \langle g, s \rangle) \} \\ & g \cdot \mathbf{in} = [\underline{1}, \text{mul}] \cdot F\ \langle g, s \rangle \end{aligned}$$

Logo, $v = [\underline{1}, \text{mul}]$.

(b) Descobrir j (com a ajuda da função s)

$$\begin{aligned} & \begin{cases} s\ 0 = 1 \\ s\ (d + 1) = s\ d + 1 \end{cases} \\ \equiv & \quad \{ \text{Def comp: (74)} \} \\ & \begin{cases} s \cdot 0 = 1 \\ s \cdot (d + 1) = s\ d + 1 \end{cases} \\ \equiv & \quad \{ \text{Definição de succ: succ d = d + 1 ; Def const: (76)} ; \text{Igualdade extencional: (73)} \} \end{aligned}$$

$$\begin{aligned}
& \left\{ \begin{array}{l} s \cdot \underline{0} = \underline{1} \\ s \cdot (\text{succ}) = \text{succ} \cdot s \end{array} \right. \\
& \equiv \{ \text{Eq} + : (27) ; \text{Natural id} : (1) \} \\
& [s \cdot \underline{0}, s \cdot (\text{succ})] = [\underline{1}, \text{succ} \cdot s] \\
& \equiv \{ \text{Natural id} : (1) ; \text{Cancelamento } \times : (7) \} \\
& [s \cdot \underline{0}, s \cdot (\text{succ})] = [\underline{1} \cdot id, \text{succ} \cdot \pi_2 \cdot \langle g, s \rangle] \\
& \equiv \{ \text{Fusão } + : (20) ; \text{Absorção } \times : (11) \} \\
& s \cdot [\underline{0}, (\text{succ})] = [\underline{1}, \text{succ} \cdot \pi_2] \cdot (id + \langle s, g \rangle) \\
& \equiv \{ \text{Definição de in e functor (dos naturais): } \mathbf{in} = [\underline{0}, (\text{succ})], F \langle \langle s, g \rangle = (id + \langle s, g \rangle) \} \\
& s \cdot \mathbf{in} = [\underline{1}, \text{succ} \cdot \pi_2] \cdot F \langle s, g \rangle
\end{aligned}$$

Logo, $j = [\underline{1}, \text{succ} \cdot \pi_2]$.

Deste modo, após encontrarmos a definição de v e de j conseguimos determinar a definição de $\langle g, s \rangle$ uma vez que já havíamos constatado que $\langle g, s \rangle = \langle \langle v, j \rangle \rangle$.

Logo, $\langle g, s \rangle = \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle$.

3. Aplicar a lei de banana split à definição de $\langle f \ k, l \ k \rangle$ e $\langle g, s \rangle$

A lei de banana split (51) é a seguinte:

$$\begin{aligned}
& \langle \langle i \rangle, \langle j \rangle \rangle \\
& \equiv \{ \text{Banana-split} : (51) \} \\
& \langle (i \times j) \cdot \langle F \pi_1, F \pi_2 \rangle \rangle
\end{aligned}$$

Assim, podemos constatar que, no nosso caso:

$$\langle i \rangle = \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle$$

e

$$\langle j \rangle = \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle$$

Assim, retomando o resultado anterior e aplicando a lei temos:

$$\begin{aligned}
& \langle \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle, \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle \rangle \\
& \equiv \{ \text{Banana-split} : (51) \} \\
& \langle \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle \times \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle \cdot \langle F \pi_1, F \pi_2 \rangle \rangle
\end{aligned}$$

Agora podemos então continuar a resolver o problema, sempre com o intuito de chegar a um $\langle [b, i] \rangle$ para podermos aplicar a definição de ciclo for:

$$\text{for } b \ i = \langle [i, b] \rangle$$

Retomando o resultado anterior e continuando a aplicar as leis do cálculo de programas, temos:

$$\begin{aligned}
& \langle \langle \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle \times \langle [\underline{1}, mul], [\underline{1}, \text{succ} \cdot \pi_2] \rangle \rangle \cdot \langle F \pi_1, F \pi_2 \rangle \rangle \\
& \equiv \{ \text{Absorção } \times : (11) ; \text{Lei da troca} : (28) \} \\
& \langle \langle \langle [\underline{1}, \text{succ} \cdot \pi_2] \rangle, \langle mul, \text{succ} \cdot \pi_2 \rangle \rangle \cdot F \pi_1, [\langle \underline{1}, \underline{1} \rangle, \langle mul, \text{succ} \cdot \pi_2 \rangle] \cdot F \pi_2 \rangle \rangle
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{Definição do Functor (dos naturais): } F \pi_1 = id + \pi_1, F \pi_2 = id + \pi_2 \} \\
&\quad \langle \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \text{mul}, \text{succ} \cdot \pi_2 \rangle \rangle \cdot (id + \pi_1), [\langle \underline{1}, \underline{1} \rangle, \langle \text{mul}, \text{succ} \cdot \pi_2 \rangle] \cdot (id + \pi_2) \rangle \rangle \\
&\equiv \{ \text{Absorção } + : (22) \times 2 ; \text{ Fusão } \times : (10) \times 2 ; \text{ Natural id} : (1) \times 2 \} \\
&\quad \langle \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle \rangle, [\langle \underline{1}, \underline{1} \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle] \rangle \rangle \\
&\equiv \{ \text{Lei da troca : (28)} \} \\
&\quad \langle \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle, \langle \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle \rangle \\
&\equiv \{ \text{Definição de for: for } b \ \underline{i} = \langle [i, b] \rangle \} \\
&\quad \text{for } \langle \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle
\end{aligned}$$

Assim, tendo em conta a definição do enunciado:

$$\text{for loop } (base \ k)$$

E comparando com o nosso resultado:

$$\text{for } \langle \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle$$

Temos que:

$$\begin{aligned}
\text{loop} &= \langle \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle \\
\text{base } k &= \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle
\end{aligned}$$

4. Derivar a definição de *base k*

Focando em *base k*:

$$\text{base } k = \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle$$

Vamos agora introduzir variáveis à definição *pointfree* para podermos ter o resultado da função *base k* em haskell.

Uma vez que o tipo de *split*, por exemplo, $\langle id, id \rangle$ é:

$$\begin{array}{c}
A \\
\downarrow \langle id, id \rangle \\
A \times A
\end{array}$$

O tipo de $\langle \langle id, id \rangle, \langle id, id \rangle \rangle$ será:

$$\begin{array}{c}
A \\
\downarrow \langle \langle id, id \rangle, \langle id, id \rangle \rangle \\
(A \times A) \times (A \times A)
\end{array}$$

Assim, conseguimos perceber qual é o tipo da variável que temos que adicionar uma vez que no nosso caso também se trata de um *split* de *split*.

Logo, continuando a derivação:

$$\begin{aligned}
&\text{base } k = \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle \\
&\equiv \{ \text{Igualdade extensional : (73)} \} \\
&\text{base } k = \langle \langle \underline{1}, (\underline{\text{succ}} \ k) \rangle, \langle \underline{1}, \underline{1} \rangle \rangle a \\
&\equiv \{ \text{Def split : (78)} \}
\end{aligned}$$

$$\begin{aligned}
& base\ k = (\langle \underline{1}, (\underline{succ\ k}) \rangle\ a, \langle \underline{1}, \underline{1} \rangle\ a) \\
\equiv & \{ \text{Def split} : (78) \times 2 \} \\
& base\ k = (\langle \underline{1}\ a, (\underline{succ\ k})\ a \rangle, \langle \underline{1}\ a, \underline{1}\ a \rangle) \\
\equiv & \{ \text{Def const} : (76) \times 4 ; \text{Definição de succ: succ } k = k + 1 \} \\
& base\ k = ((1, k + 1), (1, 1))
\end{aligned}$$

Porém, tendo em consideração a implementação desejada:

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

Vemos que em $\textbf{let } (a, -, b, -)$ o tipo de dados é um quádruplo o que implica que os tipos de \textbf{loop} e $\textbf{base\ } k$, mais especificamente o tipo de retorno, terão que ser também um quádruplo (o $\textbf{for loop } (base\ k)\ n$, por sua vez, também deverá retornar um quádruplo).

Deste modo, através da aplicação de uma simples função que nos altere o tipo de dados, por exemplo:

$$\begin{aligned}
& altera :: ((a, b), (c, d)) \rightarrow (a, b, c, d) \\
& altera ((a, b), (c, d)) = (a, b, c, d)
\end{aligned}$$

Conseguimos ter a derivação desejada da função $\textbf{base\ } k$:

$$base\ k = (1, k + 1, 1, 1)$$

5. Derivar a definição de \textbf{loop}

Por último, para derivar a definição de \textbf{loop} teremos que pensar da mesma forma, ou seja, inserir variáveis. Porém, ao contrário da função $\textbf{base\ } k$, as variáveis a inserir terão que ser do tipo $((a, b), (c, d))$ uma vez que temos um \textbf{split} com projeções π_1 e π_2 . Nos diagramas abaixo pode-se verificar o motivo desta diferença de domínio:

Tendo em conta o primeiro \textbf{split} do \textbf{split} maior, temos:

$$\begin{array}{c}
(A \times B) \times (C \times D) \\
\downarrow \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle \\
Z \times B
\end{array}$$

É de salientar que o tipo $\text{mul } (a, b) = a * b$, ou seja, o tipo desta função é:

$$\begin{array}{c}
(A \times B) \\
\downarrow \text{mul} \\
Z
\end{array}$$

Tendo agora em consideração o segundo \textbf{split} do \textbf{split} maior, temos:

$$\begin{array}{c}
(A \times B) \times (C \times D) \\
\downarrow \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \\
(Z \times D)
\end{array}$$

Temos então,

$$\begin{aligned}
& loop = \langle \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle \\
\equiv & \{ \text{Igualdade extensional} : (73) \} \\
& loop = \langle \langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle ((a, b), (c, d))
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{Def split : (78)} \} \\
&\quad \text{loop} = (\langle \text{mul} \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle ((a, b), (c, d)), \langle \text{mul} \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle ((a, b), (c, d))) \\
&\equiv \{ \text{Def split : (78) x2 ; Def comp : (74) x6 ; Def proj : (81) x6} \} \\
&\quad \text{loop} = ((\text{mul} (a, b), \text{succ } b), (\text{mul} (c, d), \text{succ } d)) \\
&\equiv \{ \text{Definição de succ : succ } k = k + 1 \text{ x2 ; Definição de mul : mul (m,n) = m * n x2} \} \\
&\quad \text{loop} = ((a * b, b + 1), (c * d, d + 1))
\end{aligned}$$

Deparámo-nos com o mesmo problema da função *base k* e por isso vamos aplicar novamente a função *altera* para reparar o tipo de dados de retorno. Temos então:

$$\text{loop} (a, b, c, d) = (a * b, b + 1, c * d, d + 1)$$

Deste modo, obtemos os dois resultados pretendidos:

$$\text{base } k = (1, k + 1, 1, 1)$$

$$\text{loop} (a, b, c, d) = (a * b, b + 1, c * d, d + 1)$$

Problema 4

```

inFTree = [Unit, uncurryB Comp]
  where uncurryB f (a, (t1, t2)) = f a t1 t2
outFTree (Unit c) = i1 c
outFTree (Comp a t1 t2) = i2 (a, (t1, t2))
baseFTree f g h = g + (f × (h × h))
recFTree f = baseFTree id id f
cataFTree a = a · (recFTree (cataFTree a)) · outFTree
anaFTree f = inFTree · (recFTree (anaFTree f)) · f
hyloFTree a c = cataFTree a · anaFTree c

instance Bifunctor FTree where
  bimap f g = cataFTree (inFTree · baseFTree f g id)
generatePTree a = anaFTree ((0 + ⟨m, ⟨id, id⟩⟩) · outNat) a
  where
    m x = 50 * (sqrt (2) / 2) ↑ abs (x - a)
drawPTree a = anaList ((nil + ⟨list, id⟩) · outNat) (depthFTree a)
  where
    list n = (foldMap lineLoop (squares n))
    squares n = concat $ take (depthFTree a - n) $ iterateM mkBranches start
    start = [(-100, 0), (0, 0), (0, -100), (-100, -100)]
    iterateM f x = iterate (≫=f) (pure x)
    mkBranches [a, b, c, d] = let d = 0.5 < * > (b < + > ((-1) < * > a))
                                l1 = d < + > orth d
                                l2 = orth l1
                                in
                                [[a < + > l2, b < + > (2 < * > l2), a < + > l1, a]
                                 , [a < + > (2 < * > l1), b < + > l1, b, b < + > l2]]
    (a, b) < + > (c, d) = (a + c, b + d)
    n < * > (a, b) = (a * n, b * n)
    orth (a, b) = (-b, a)

```

Problema 5

O quinto e último problema diz respeito a *monades*. O mónade do problema é conhecido por *bag*, *saco* ou *multi-conjunto*, permitindo que os elementos de um conjunto tenham multiplicidades associadas.

Para este problema o objetivo é que completemos a definição de *Monad Bag*, nomeadamente no que diz respeito ao μ (multiplicação do mónade *Bag*) e a respetiva função auxiliar *singletonbag*. É também necessário definir a função *dist*.

Começando pela função *singletonbag*, tal como o próprio nome indica, o objetivo dela é ter um saco com apenas um elemento do valor passado como parâmetro, para ser o *return* do *Monad*.

Assim, a sua definição é intuitiva e é a seguinte:

$$\text{singletonbag } a = B [(a, 1)]$$

É o construtor *B* que se encarrega de fazer com que o tipo de retorno de *singletonbag* seja o desejado.

Quanto ao μ , sabemos que o seu tipo de dados deverá ser o seguinte:

$$\mu :: \text{Bag } (\text{Bag } a) \rightarrow \text{Bag } a$$

Deste modo, tendo ainda em consideração a definição de *Monad* sabemos que esta função recebe um *Bag* de *Bags*, como por exemplo:

$$\begin{aligned} b2 &:: \text{Bag } (\text{Bag } \text{Marble}) \\ b2 &= B [(B [(Pink, 2), (Green, 3), (Red, 2), (Blue, 2), (White, 1)], 5), \\ &\quad (B [(Pink, 1), (Green, 2), (Red, 1), (Blue, 1)], 2)] \end{aligned}$$

E que o objetivo é que a função retorne um *Bag* somente, que, no caso de correr μ *b2* deverá ser:

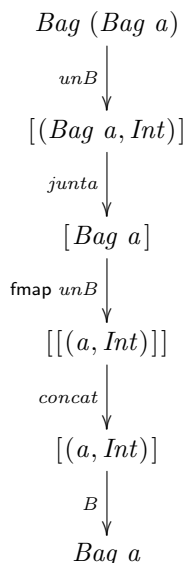
$$\begin{aligned} b1 &:: \text{Bag } \text{Marble} \\ b1 &= B [(Pink, 12), (Green, 19), (Red, 12), (Blue, 12), (White, 5)] \end{aligned}$$

Sucintamente, o objetivo do μ é que dado um *Bag* com *Bags* lá dentro todos os seus elementos se juntem num só *Bag*. Para tal, é necessário ter em atenção quantos *Bags* estão dentro do *Bag* maior e agrupar convenientemente os seus conteúdos, tal como vimos no exemplo anterior.

Assim, com recurso a algumas funções auxiliares produzimos o código em seguida:

$$\begin{aligned} \mu b &= B (\text{concat } (\text{fmap } \text{unB } (\text{junta } (\text{unB } b)))) \\ \text{where } \text{junta } ((ba, int) : bas) &= (\text{fmapSpecial } (*int) ba) : (\text{junta } bas) \\ \text{junta } [] &= [] \\ \text{fmapSpecial } f &= B \cdot \text{map } (id \times f) \cdot \text{unB} \end{aligned}$$

O diagrama seguinte mostra as alterações nos tipos de dados que vão acontecendo no decorrer da definição de μ por nós proposta:



Passando a explicar por palavras passo a passo o que acontece na função μ , temos que:

1. *un B*

Numa primeira fase “desembrulhamos” o *Bag* de *Bags*, passando agora a ter uma lista com $(Bag\ a, Int)$ onde poderemos iterar mais facilmente sobre os elementos.

2. *junta*

Esta função, com a ajuda de uma função auxiliar chamada *fmapSpecial*, transforma a lista de $(Bag\ a) \times Int$, sendo que este *Int* representa o número de “sacos” que existem de *Bag a*, numa só lista de *Bag a*. A função multiplica o número de sacos que existem daquele tipo pelo conteúdo dentro do *Bag a*. Ou seja, se temos 3 sacos com 2 berlindes azuis dentro de cada um sabemos que no total temos 6 berlindes azuis, e é exatamente isso que a função faz dentro de cada *Bag a* da lista.

3. *fmap unB*

Este passo “abre” todos os *Bag a* dentro de $[Bag\ a]$, ficando agora com o tipo $[(a, Int)]$. Mais uma vez, este passo é feito para facilitar o manuseamento dos elementos.

4. *concat*

Esta função, já predefinida, é responsável por concatenar a $[(a, Int)]$ em $[(a, Int)]$. O que esta função faz é somente juntar os pares (a, Int) numa só lista.

5. *B*

Neste último passo é aplicado o construtor de *Bag* à lista de (a, Int) e com isso fica garantido que os (a, Int) repetidos se juntam, somando os respetivos *Int*, e que o tipo de dados de retorno é o tipo desejado, nomeadamente *Bag a*.

Para a função *dist*, tendo em consideração o exemplo dado no enunciado, onde para o “saco”:

$B [(Pink, 2), (Green, 3), (Red, 2), (Blue, 2), (White, 1)]$

O resultado da aplicação desta função deverá ser:

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

Percebemos que *dist* divide o número de elementos de cada elemento pelo número total de elementos.

Assim, definimos a função como:

```
dist (B a) = D ((map (\(x, y) → (x, (/) (toFloat y) (toFloat (number a)))) a)
  where number [] = 0
        number ((_, int) : cs) = int + number cs
```

A função percorre todos os elementos de $B\ a\ [(a, Int)]$, mais especificamente, uma vez que é este o parâmetro que passamos ao *map* e divide cada *Int* pelo número total de elementos do conjunto, que é calculado com a ajuda da função auxiliar *number*.

De modo a retornarmos o resultado da forma correta, nomeadamente *Dist a*, tivemos que fazer duas pequenas alterações no código: a primeira foi converter o *y* e o *number a* (numero total de berlindes) de cada par para *Float* antes de os dividirmos. A segunda é aplicar o construtor *D*, do módulo *Probability*, ao resultado do *map* para assim conseguirmos obter o tipo de dados desejado.

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:⁷

$$id = \langle f, g \rangle$$

⁷Exemplos tirados de [?].

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

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

$$\begin{array}{ccc}
\mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
\downarrow \langle g \rangle & & \downarrow id + \langle g \rangle \\
B & \xleftarrow{g} & 1 + B
\end{array}$$