Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2017/18

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	84
a74207	Bruno Dantas
a74745	Carlos Campos
a74166	Pedro Fonseca

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 LaTeX 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

[1] O 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.

$$\mathbf{data}\ Blockchain = Bc\ \{bc :: Block\} \mid Bcs\ \{bcs :: (Block, Blockchain)\}\ \mathbf{deriving}\ Show$$

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

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

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.

$$\mathbf{type}\ Transaction = (Entity, (Value, Entity))$$
$$\mathbf{type}\ Transactions = [Transaction]$$

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

$$\mathbf{type}\ Ledger = [(Entity, Value)]$$

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.

$$\begin{aligned}&\textbf{type } MagicNo = String\\&\textbf{type } Time = Int \quad \text{-- em milisegundos}\\&\textbf{type } Entity = String\\&\textbf{type } Value = Int\end{aligned}$$

Neste contexto, implemente as seguintes funções:

1. Defina a função $allTransactions :: Blockchain \to Transactions$, 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 $ledger :: Blockchain \to Ledger$, utilizando catamorfismos e/ou anamorfismos, que calcula o ledger (i.e., o valor disponível) de cada entidade numa uma 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:*

   $$prop1b = \mathsf{length} \cdot ledger \leqslant (2*) \cdot \mathsf{length} \cdot allTransactions$$

   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 $isValidMagicNr :: Blockchain \to Bool$, 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 *quadtree* é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

$$\begin{aligned}&\textbf{data } QTree\ a = Cell\ a\ Int\ Int \mid Block\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)\\&\quad \textbf{deriving } (Eq, Show)\end{aligned}$$

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

(a) Matriz de exemplo bm.

```
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)))
```

(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 bits[2], 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:

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

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 côr. 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:

$$
\begin{aligned}
whitePx &= PixelRGBA8\ 255\ 255\ 255\ 255 \\
blackPx &= PixelRGBA8\ 0\ 0\ 0\ 255 \\
redPx &= PixelRGBA8\ 255\ 0\ 0\ 255
\end{aligned}
$$

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

$$
\begin{aligned}
&readBMP :: FilePath \to \mathsf{IO}\ (Matrix\ PixelRGBA8) \\
&writeBMP :: FilePath \to Matrix\ PixelRGBA8 \to \mathsf{IO}\ ()
\end{aligned}
$$

Teste, por exemplo, no $GHCi$, carregar a Figura 2a:

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

Esta questão aborda operações de processamento de imagens utilizando quadtrees:

1. Defina as funções $rotateQTree :: QTree\ a \to QTree\ a$, $scaleQTree :: Int \to QTree\ a \to QTree\ a$ e $invertQTree :: QTree\ a \to QTree\ a$, como catamorfismos e/ou anamorfismos, que rodam[3], redimensionam [4] e invertem as cores de uma quadtree[5], 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"
```

[2] Cf. módulo $Data.Matrix$.
[3] Segundo um ângulo de 90º no sentido dos ponteiros do relógio.
[4] Multiplicando o seu tamanho pelo valor recebido.
[5] Um pixel pode ser invertido calculando $255 - c$ para cada componente c de cor RGB, exceptuando o componente alpha.

(a) Bitmap de exemplo.

(b) Rotação.

(c) Redimensionamento.

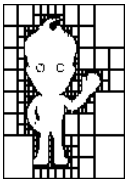
(d) Inversão de cores.

(e) Compresão de 1 nível.

(f) Compresão de 2 níveis.

(g) Compresão de 3 níveis.

(h) Compresão de 4 níveis.



(i) Bitmap de contorno.

(j) Bitmap com contorno.

Figura 2: Manipulação de uma figura bitmap utilizando quadtrees.

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

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

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

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

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

$$prop2e = shapeQTree \cdot invertQTree \equiv shapeQTree$$

2. Defina a função $compressQTree :: Int \to QTree\ a \to QTree\ a$, 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:*

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

3. Defina a função $outlineQTree :: (a \to Bool) \to QTree\ a \to Matrix\ Bool$, 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:*

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

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

$$teste2a = outlineQTree\ (\equiv 0)\ qt \equiv qtOut$$

Problema 3

O cálculo das combinações de n k-a-k,

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

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,

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

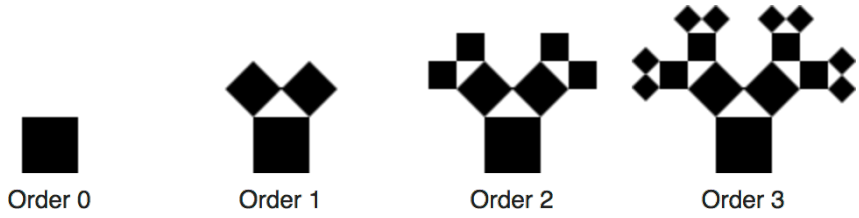


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

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

assumindo-se $d = n - k \geqslant 0$. 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:

$$\left(\begin{array}{c} n \\ k \end{array} \right) = h\ k\ (n-k)\ \textbf{where}\ h\ k\ n = \textbf{let}\ (a, \_, b, \_) = \mathsf{for}\ (loop)\ (base\ k)\ n\ \textbf{in}\ a\ /\ b$$

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* $\left(\begin{array}{c} n \\ k \end{array} \right)$ *coincide com a sua especificação (1):*

$$prop3\ (NonNegative\ n)\ (NonNegative\ k) = k \leqslant n \Rightarrow \left(\begin{array}{c} n \\ k \end{array} \right) \equiv n!\ /\ (k! * (n-k)!)$$

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:

$$\textbf{data}\ FTree\ a\ b = Unit\ b \mid Comp\ a\ (FTree\ a\ b)\ (FTree\ a\ b)\ \textbf{deriving}\ (Eq, Show)$$
$$\textbf{type}\ PTree = FTree\ Square\ Square$$
$$\textbf{type}\ Square = Float$$

1. Defina a função $generatePTree :: Int \to PTree$, 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:*

   $$prop4a\ (SmallNat\ n) = (depthFTree \cdot generatePTree)\ n \equiv n$$

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

   $$prop4b\ (SmallNat\ n) = (isBalancedFTree \cdot generatePTree)\ n$$

2. Defina a função $drawPTree :: PTree \to [Picture]$, 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

$$\textbf{data}\ Marble = Red \mid Pink \mid Green \mid Blue \mid White\ \textbf{deriving}\ (Read, Show, Eq, Ord)$$

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":

$$\textbf{data}\ Bag\ a = B\ [(a, Int)]\ \textbf{deriving}\ (Ord)$$

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

$$\begin{aligned}
&marbleWeight :: Marble \to Int\\
&marbleWeight\ Red\ \ = 3\\
&marbleWeight\ Pink\ = 2\\
&marbleWeight\ Green = 3\\
&marbleWeight\ Blue\ = 6\\
&marbleWeight\ White = 2
\end{aligned}$$

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

$$marbleWeights = \mathsf{fmap}\ marbleWeight\ bagOfMarbles$$

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

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

[6] "Marble" traduz para "berlinde" em português.

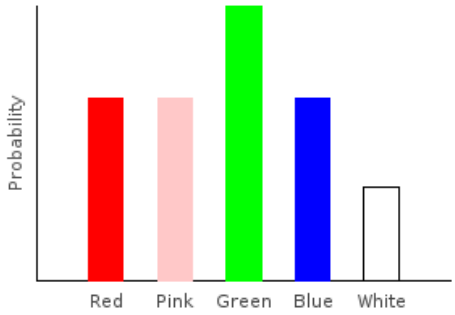


Figura 4: Distribuição de berlindes num saco.

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,

$$\begin{aligned}&\textbf{instance}\ Functor\ Bag\ \textbf{where}\\&\quad \mathsf{fmap}\ f = B \cdot \mathsf{map}\ (f \times id) \cdot unB\\&\textbf{instance}\ Monad\ Bag\ \textbf{where}\\&\quad x \gg\!= f = (\mu \cdot \mathsf{fmap}\ f)\ x\ \textbf{where}\\&\quad return = singletonbag\end{aligned}$$

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* $\mu \cdot return = id$*:*

$$test5a = bagOfMarbles \equiv \mu\ (return\ bagOfMarbles)$$

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

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

onde $b3$ *é um saco dado em anexo.*

Referências

[1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[2] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.

Anexos

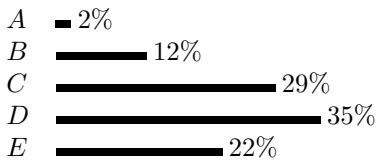
A Mónade para probabilidades e estatística

Mónades são functores 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

$$\textbf{newtype}\ \mathsf{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{2}$$

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

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



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

B Definições auxiliares

Funções para mostrar *bags*:

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

Igualdade de *bags*:

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

Ainda sobre o mónade *Bag*:

```
instance Applicative Bag where
  pure = return
  (<*>) = aap
```

O exemplo do texto:

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

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

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

Outras funções auxiliares:

```
a ↦ b = (a, b)
consol :: (Eq b) ⇒ [(b, Int)] → [(b, Int)]
consol = filter nzero · map (id × sum) · col where nzero (_, x) = x ≢ 0
isempty :: Eq a ⇒ [(a, Int)] → Bool
isempty = all (≡ 0) · map 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

```
inBlockchain = [Bc, Bcs]
outBlockchain (Bc b) = i1 b
outBlockchain (Bcs (b1, b2)) = i2 (b1, b2)
recBlockchain f = id + id × f
cataBlockchain g = g · recBlockchain (cataBlockchain g) · outBlockchain
anaBlockchain g = inBlockchain · recBlockchain (anaBlockchain g) · g
hyloBlockchain f g = (cataBlockchain f) · (anaBlockchain g)
```

Definiu-se a função *allTransactions* como um catamorfismo sobre a estrutura *Blockchain*. O diagrama correspondente é ilustrado de seguida.

$$\begin{array}{ccc} Blockchain & \xleftarrow{inBlockchain} & Block + Block \times Blockchain \\ {\scriptstyle (\!|allTransactions|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|allTransactions|\!)} \\ Transactions & \xleftarrow[allTransactions]{} & Block + Block \times Transactions \end{array}$$

Para definir o gene deste catamorfimo (*allTransactions*), precisamos de definir as duas funções a executar na alternativa. Dito de outro modo, precisamos de uma função do tipo $Block \to Transactions$ e de outra do tipo $Block * Transactions \to Transactions$. A primeira está definida como $\pi_2 \cdot \pi_2$, uma vez que pretendemos obter do tipo *Block* a lista *Transactions* (duas projeções consecutivas à direita). Para a segunda, visto que se assume que a chamada recursiva já foi efetuada, é necessário processar o *Block* da mesma forma e combinar as duas listas *Transactions*. A definição fica $\widehat{(+\!\!+)} \cdot ((pi2 \cdot pi2) \times id)$. Juntando as duas funções na alternativa (*Either*) e no catamorfismo, obtemos a definição final que se segue.

$$allTransactions = cataBlockchain\ [\pi_2 \cdot \pi_2, \widehat{(+\!\!+)} \cdot ((\pi_2 \cdot \pi_2) \times id)]$$

Na resolução da segunda questão deste problema, conseguir obter o *ledger* de uma *Blockchain* levou a que fossem precisas várias funções auxiliares, definidas sobre catamorfismos de listas. O primeiro passo consiste na reutilzação da função anterior para obter uma lista de *Transaction*, ou simplesmente *Transactions*. Depois, difiniram-se três funções - catamorfismos sobre listas - para concluir todo o processo de construção do *ledger*. A primeira função auxiliar é a função $divideTransactions :: [(Entity, (Value, Entity))] \rightarrow [(Entity, Value)]$. O objetivo é pegar em cada transação e expandir em duas partes. A primeira é composta pela entidade que enviou a transação e pelo valor negativo. A segunda é composta pela entidade que recebeu a transação e o valor (positivo). O resultado deste catamorfismo é processado pela próxima função - $collectTransactions :: [(Entity, Value)] \rightarrow [(Entity, [Value])]$ - responsável por associar os valores das mesmas entidades, ou seja, criar uma lista dos movimentos para cada entidade. Finalmente, a função $sumValues :: [(Entity, [Value])] \rightarrow [(Entity, Value)]$ efetua a soma dos movimentos de cada entidade, obtendo-se assim uma lista com o saldo atual de todas as entidades (*ledger*).

$$ledger = sumValues \cdot collectTransactions \cdot divideTransactions \cdot allTransactions$$

$$\begin{array}{ccc} Transactions & \xrightarrow{outList} & 1 + Transaction \times Transactions \\ {\scriptstyle (\!|divideTransactions|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|divideTransactions|\!)} \\ (Entity \times Value) & \xleftarrow[divideTransactions]{} & 1 + Transaction \times (Entity \times Value) \end{array}$$

$$\begin{array}{l} divideTransactions = cataList\ [nil, aux] \\ \quad \textbf{where}\ aux = \widehat{(+\!\!+)} \cdot (\widehat{(+\!\!+)} \times id) \cdot ((singl \times singl) \times id) \cdot (\langle id \times (mul \cdot \langle id, \underline{(-1)} \rangle \cdot \pi_1), swap \cdot \pi_2 \rangle \times id) \end{array}$$

$$\begin{array}{ccc} (Entity \times Value)^* & \xrightarrow{outList} & 1 + (Entity \times Value) \times (Entity \times Value)^* \\ {\scriptstyle (\!|collectTransactions|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|collectTransactions|\!)} \\ (Entity \times Value^*)^* & \xleftarrow[collectTransactions]{} & 1 + (Entity \times Value) \times (Entity \times Value^*)^* \end{array}$$

$$collectTransactions = cataList\ [nil, \widehat{(append)}]$$

$$\begin{array}{ccc} (Entity \times Value^*)^* & \xrightarrow{outList} & 1 + (Entity \times Value^*) \times (Entity \times Value^*)^* \\ {\scriptstyle (\!|sumValues|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|sumValues|\!)} \\ (Entity \times Value)^* & \xleftarrow[sumValues]{} & 1 + (Entity \times Value^*) \times (Entity \times Value)^* \end{array}$$

$$sumValues = cataList\ [nil, cons \cdot ((id \times sum) \times id)]$$

No sentido de verificar a ocorrência de números repetidos (*MagicNo*), optou-se por efetuar um catamorfismo inicial que retorna a lista de todos os números mágicos.

$$\begin{array}{l} isValidMagicNr = aux \cdot cataBlockchain\ [singl \cdot \pi_1, cons \cdot (\pi_1 \times id)] \\ \quad \textbf{where}\ aux\ [\,] = True \\ \qquad aux\ (h : t) = \textbf{if}\ (elem\ h\ t)\ \textbf{then}\ False \\ \qquad\quad \textbf{else}\ aux\ t \end{array}$$

Problema 2

$$
\begin{aligned}
&inQTree = [uncurriedCell, uncurriedBlock]\\
&outQTree\ (Cell\ a\ b\ c) = i_1\ (a, (b, c))\\
&outQTree\ (Block\ a\ b\ c\ d) = i_2\ (a, (b, (c, d)))\\
&baseQTree\ g\ f = (g \times id) + (f \times (f \times (f \times f)))\\
&recQTree\ f = baseQTree\ id\ f\\
&cataQTree\ g = g \cdot recQTree\ (cataQTree\ g) \cdot outQTree\\
&anaQTree\ g = inQTree \cdot recQTree\ (anaQTree\ g) \cdot g\\
&hyloQTree\ f\ g = (cataQTree\ f) \cdot (anaQTree\ g)\\
&uncurriedCell\ (a, (b, c)) = Cell\ a\ b\ c\\
&uncurriedBlock\ (a, (b, (c, d))) = Block\ a\ b\ c\ d\\
&\textbf{instance}\ Functor\ QTree\ \textbf{where}\\
&\quad \mathsf{fmap}\ f = cataQTree\ (inQTree \cdot (baseQTree\ f\ id))
\end{aligned}
$$

Após a definição de todas as funções de manipulação sobre a estrutura $QTree$, procedemos à definição das três primeiras funções do problema. A função $rotateQTree$ recorre a um catamorfismo para trocar a dimensão de cada $Cell$, assim como a ordem das chamadas recursivas do catamorfismo (rodar os quadrantes).

$$
\begin{array}{ccc}
QTree\ A & \xrightarrow{outQTree} & A \times (Int \times Int) + (QTree\ A)^4 \\
{\scriptstyle (\!|rotateQTree|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|rotateQTree|\!)^4} \\
QTree\ A & \xleftarrow[rotateQTree]{} & A \times (Int \times Int) + (QTree\ A)^4
\end{array}
$$

$$
\begin{aligned}
&rotateQTree = cataQTree\ [uncurriedCell \cdot (id \times swap), g2]\\
&\quad \textbf{where}\ g2\ (a, (b, (c, d))) = Block\ c\ a\ d\ b
\end{aligned}
$$

Relativamente à função $scaleQTree$, a ideia passou por recorrer a um catamorfismo. A função do lado esquerdo da alternativa - g1 - multiplica a dimensão pelo argumento fornecido, enquanto a segunda - g2 - apenas volta a construir valores de $QTree$. Não é possível aplicar o fmap definido, uma vez que a $baseQTree$ está definida para aplicar funções apenas ao primeiro parâmetro de $Cell$.

$$
\begin{array}{ccc}
QTree\ A & \xrightarrow{outQTree} & A \times (Int \times Int) + (QTree\ A)^4 \\
{\scriptstyle (\!|scaleQTree|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|scaleQTree|\!)^4} \\
QTree\ A & \xleftarrow[scaleQTree]{} & A \times (Int \times Int) + (QTree\ A)^4
\end{array}
$$

$$
\begin{aligned}
&scaleQTree\ i = cataQTree\ [g1, g2]\\
&\quad \textbf{where}\ g1\ (a, (b, c)) = Cell\ a\ (i * b)\ (i * c)\\
&\qquad g2\ (a, (b, (c, d))) = Block\ a\ b\ c\ d
\end{aligned}
$$

A função de inversão de cores $invertQTree$ foi definida através da utilização do fmap definido no início do problema. O objetivo é aplicar a mesma função a todas as $Cell$, por isso estamos em condições ideias para o aplicar. A função é muito simples, recebe um PixelRGB8 e subtraí a 255 o valor das suas componentes (exceto alpha).

$$
\begin{array}{ccc}
QTree\ PixelRGBA8 & \xrightarrow{outQTree} & PixelRGBA8 \times (Int \times Int) + (QTree\ PixelRGBA8)^4 \\
{\scriptstyle (\!|invertQTree|\!)}\downarrow & & \downarrow{\scriptstyle f + id} \\
QTree\ PixelRGBA8 & \xleftarrow[invertQTree]{} & PixelRGBA8 \times (Int \times Int) + (QTree\ PixelRGBA8)^4
\end{array}
$$

$$
\begin{lstlisting}
invertQTree = fmap f
  where f (PixelRGBA8 r g b a) = PixelRGBA8 (255 - r) (255 - g) (255 - b) a
\end{lstlisting}
$$

Através de um anamorfismo é possível definir a função $compressQTree$. Antes de qualquer aplicar qualquer tipo de recursividade, testa-se se o parâmetro é maior que a profundidade da árvore. Caso isto se verifique, toda a árvore é substituída por uma única $Cell$. Caso contrário, recorre-se a um anamorfismo vai construindo a estrutura em função de um inteiro - diferença entre a profundidade e o nível de compressão - calculado antes da execução do anamorfismo. Assumem-se três comportamentos diferentes durante a execução.

1. Sempre que se atinge uma $Cell$ aplica-se uma injeção à esquerda
2. Se a função receber o valor inteiro com o valor zero no caso de um $Block$, altera-o para uma $Cell$ e injeta à esquerda
3. Se o valor inteiro for superior a 0 no caso de um $Block$, injeta à direita as sub-árvores com o valor inteiro decrementado

$$
\begin{array}{ccc}
QTree\ A & \xleftarrow{inQTree} & A \times (Int \times Int) + (QTree\ A)^4 \\
{\scriptstyle anaNat\ compressQTree}\uparrow & & \uparrow{\scriptstyle id+(anaNat\ compressQTree)^4} \\
Int \times QTree\ A & \xrightarrow[compressQTree]{} & A \times (Int \times Int) + (Int \times QTree\ A)^4
\end{array}
$$

$$
\begin{lstlisting}
compressQTree i t = if (((depthQTree t) - i) < 0)
  then let (x, y) = sizeQTree t in (Cell (searchCell t) x y)
    else anaQTree f ((depthQTree t) - i, t)
  where f (_, (Cell a b c)) = i1 (a, (b, c))
    f (0, s@(Block a b c d)) = let (x, y) = sizeQTree s in i1 ((searchCell a), (x, y))
    f (n, (Block a b c d)) = i2 ((n - 1, a), ((n - 1, b), ((n - 1, c), (n - 1, d))))
  -- Retorna o parâmetro 'a' do Cell mais à esquerda
searchCell = cataQTree [g1, g2]
  where g1 (a, (b, c)) = a
    g2 (a, (b, (c, d))) = a

outlineQTree f = qt2bm . (fmap f)
\end{lstlisting}
$$

Problema 3

Para resolver este problema foram efetuados os seguintes passos:

1. Converter as quatro funções de *pointwise* para *pointfree*
2. Aplicar a lei de Fokkinga para combinar os pares *fk*,*lk* e *s*,*g*
3. Aplicar a lei de banana-split para combinar as duas soluções
4. Aplicar a lei da troca para converter num catamorfismo de alternativas (for)
5. Definir as funções polimórficas tuple/untuple para uma tipagem correta

A prova dos cálculos é mostrada de seguida:

$$
\begin{array}{ll}
 & \left\{ \begin{array}{l} g \cdot 0 = 1 \\ g\ (d+1) = (d+1) * g\ d \end{array} \right. \\
\equiv & \quad \{\ (73), (76), (20), (27), \text{DEF-mul, DEF-succ}\ \} \\
 & g \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, mul \cdot \langle g, s \rangle] \\
\equiv & \quad \{\ (1), (22)\ \} \\
 & g \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, mul] \cdot (id + \langle g, s \rangle)
\end{array}
$$

□

$$
\begin{cases} s \cdot 0 = 1 \\ s\ (d+1) = s\ d + 1 \end{cases}
$$

$\equiv$ { (73), (76), (20), (27), DEF-succ }

$$
s \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, \mathsf{succ}\ \cdot s]
$$

$\equiv$ { (1), (7), (22) }

$$
s \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle g, s \rangle)
$$

□

$$
\begin{cases} g \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, mul] \cdot (id + \langle g, s \rangle) \\ s \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle g, s \rangle) \end{cases}
$$

$\equiv$ { (50) }

$$
\langle g, s \rangle = (\!|\langle [\underline{1}, \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle|\!)
$$

□

$$
\begin{cases} f\ k\ 0 = 1 \\ f\ k\ (d+1) = (d + k + 1) * f\ k\ d \end{cases}
$$

$\equiv$ { (73), (76), (20), (27), DEF-mul, DEF-succ }

$$
f\ k \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, mul \cdot \langle f\ k, l\ k \rangle]
$$

$\equiv$ { (1), (22) }

$$
f\ k \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, mul] \cdot (id + \langle f\ k, l\ k \rangle)
$$

□

$$
\begin{cases} l\ k\ 0 = k + 1 \\ l\ k\ (d+1) = l\ k\ d + 1 \end{cases}
$$

$\equiv$ { (73), (76), (20), (27), DEF-succ }

$$
l\ k \cdot [\underline{0}, \mathsf{succ}\ ] = [\mathsf{succ}\ , \mathsf{succ}\ \cdot l\ k]
$$

$\equiv$ { (1), (7), (22) }

$$
l\ k \cdot [\underline{0}, \mathsf{succ}\ ] = [\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle fk, lk \rangle)
$$

□

$$
\begin{cases} f\ k \cdot [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, mul] \cdot (id + \langle f\ k, l\ k \rangle) \\ l\ k \cdot [\underline{0}, \mathsf{succ}\ ] = [\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle fk, lk \rangle) \end{cases}
$$

$\equiv$ { (50) }

$$
\langle f\ k, l\ k \rangle = (\!|\langle [\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle|\!)
$$

□

$$
\begin{cases} \langle f\ k, l\ k \rangle = (\!|\langle [\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle|\!) \\ \langle g, s \rangle = (\!|\langle [\underline{1}, \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle|\!) \end{cases}
$$

$\equiv$ { (51) }

$$
h\ k = \langle (\!|\langle [\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle|\!), (\!|\langle [\underline{1}, \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle|\!) \rangle
$$

$\equiv$ { h k = for loop base , for f i = cataNat either (const i) f }

$$
(\!|[base, loop]|\!) = (\!|\langle [\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle \times \langle [\underline{1}, \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul] \rangle \cdot \langle F\ \pi_1, F\ \pi_2 \rangle|\!)
$$

$$
\begin{aligned}
&\equiv \quad \{\ (11)\ \} \\
&\quad (\!|[base, loop]|\!) = (\!|\langle\langle[\mathsf{succ}\ , \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul]\rangle \cdot (F\ \pi_1), \langle[\underline{1}, \mathsf{succ}\ \cdot \pi_2], [\underline{1}, mul]\rangle \cdot (F\ \pi_1)\rangle|\!) \\
&\equiv \quad \{\ (11), (28)\ \} \\
&\quad (\!|[base, loop]|\!) = (\!|\langle[\langle \mathsf{succ}\ , \underline{1}\rangle, \langle \mathsf{succ}\ \cdot \pi_2, mul\rangle \cdot \pi_1], [\langle \underline{1}, \underline{1}\rangle, \langle \mathsf{succ}\ \cdot \pi_2, mul\rangle \cdot \pi_2]\rangle|\!) \\
&\equiv \quad \{\ (28)\ \} \\
&\quad (\!|[base, loop]|\!) = (\!|[\langle\langle \underline{1}, \mathsf{succ}\ \rangle, \langle \underline{1}, \underline{1}\rangle\rangle, \langle\langle mul, \mathsf{succ}\ \cdot \pi_2\rangle \cdot \pi_1, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle \cdot \pi_2\rangle]|\!) \\
&\square
\end{aligned}
$$

$$
\begin{aligned}
&base = untuple \cdot \langle\langle \underline{1}, \mathsf{succ}\ \rangle, \langle \underline{1}, \underline{1}\rangle\rangle \\
&loop = untuple \cdot \langle aux \cdot \pi_1 \cdot tuple, aux \cdot \pi_2 \cdot tuple\rangle \\
&\quad \textbf{where}\ aux = \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle \\
&tuple :: (a, b, c, d) \to ((a, b), (c, d)) \\
&tuple\ (a, b, c, d) = ((a, b), (c, d)) \\
&\quad \text{-- tuple = split (p1 . p1) ((p2 . p1) ¿¡ p2)} \\
&untuple :: ((a, b), (c, d)) \to (a, b, c, d) \\
&untuple\ ((a, b), (c, d)) = (a, b, c, d) \\
&\quad \text{-- untuple = split (p1) (split (p1 . p2) (p2 . p2))}
\end{aligned}
$$

Problema 4

$$
\begin{aligned}
&inFTree = [Unit, g2] \\
&\quad \textbf{where}\ g2\ (a, (b, c)) = Comp\ a\ b\ c \\
&outFTree\ (Unit\ b) = i_1\ b \\
&outFTree\ (Comp\ a\ b\ c) = i_2\ (a, (b, c)) \\
&baseFTree\ f\ g\ h = g + (f \times (h \times h)) \\
&recFTree\ f = baseFTree\ id\ id\ f \\
&cataFTree\ g = g \cdot recFTree\ (cataFTree\ g) \cdot outFTree \\
&anaFTree\ g = inFTree \cdot recFTree\ (anaFTree\ g) \cdot g \\
&hyloFTree\ f\ g = (cataFTree\ f) \cdot (anaFTree\ g) \\
&\textbf{instance}\ Bifunctor\ FTree\ \textbf{where} \\
&\quad bimap\ f\ g = cataFTree\ (inFTree \cdot (baseFTree\ f\ g\ id))
\end{aligned}
$$

$$
\begin{array}{ccc}
PTree & \xleftarrow{\quad inFTree \quad} & Float + Float \times PTree \times PTree \\
{\scriptstyle anaNat\ generatePTree}\uparrow & & \uparrow{\scriptstyle id+id\times(anaNat\ generatePTree)} \\
(Float \times Int) & \xrightarrow[generatePTree]{} & Float + Float \times (Float \times Int) \times (Float \times Int)
\end{array}
$$

A função para gerar a árvore de Pitágoras recorre a um anamorfismo e comporta-se da seguinte forma. Após a receção do argumento, divide-se esse argumento num tuplo e transforma-se o elemento do lado esquerdo no tipo *Float* (dimensão dos lados dos quadrados). É construído assim um tuplo $(Dimensão, Ordem)$. Aquilo que o gene do anamorfismo faz é verificar se a ordem é zero ou maior do que 0, tendo diferente comportamentos consoante este caso. Caso seja zero - atingiu uma folha - injeta esse valor à esquerda para mais tarde ser aplicado o construtor *Unit* a esse valor. Caso seja maior do que zero - deve continuar a recursividade - injeta à esquerda o valor atual e duas vezes o tuplo já com o valor redimensionado. Note-se que o decremento da ordem já é efetuado pelo *outNat*, pelo que não é preciso aplicar *pred* neste último caso.

$$
\begin{aligned}
&generatePTree = anaFTree\ ((\pi_1 + g) \cdot distr \cdot (id \times outNat)) \cdot \langle fromIntegral, id\rangle \\
&\quad \textbf{where}\ g = \langle \pi_1, \langle aux, aux\rangle\rangle \\
&\quad\quad \textbf{where}\ aux = (*(sqrt\ (2)\ /\ 2)) \times id \\
&drawPTree = \bot
\end{aligned}
$$

Problema 5

$$singletonbag = B \cdot singl \cdot \langle id, \underline{1} \rangle$$

A função singletonbag permite a criação de um saco único, ou seja, um saco cujo o seu número de repetições é apenas 1.

$$\mu = B \cdot openBags \cdot (\mathsf{fmap}\ unB)$$

Esta é a função de multiplicação dos sacos, cujo o objetivo é multiplicar as ocorrências dos berlindes, consoante a repetição dos mesmos. Isto quer dizer que se existirem 2 sacos com 2 berlindes do tipo Blue e 1 do tipo White, a multiplicação monádica irá nos indicar que existem no total 4 berlindes do tipo Blue e 2 do tipo White. Esta funcionalidade é executada à custa da aplicação da unidade monádica a cada elemento do saco, seja ele o par (elemento,repetições) que caracterição o berlinde no saco em questão, ou seja ele outro saco também. De seguida é aplicado a função openBags que será explicada a seguir. E no fim é aplicado o construtor B.

$$openBags :: Bag\ [(a, Int)] \to [(a, Int)]$$
$$openBags = concat \cdot (\mathsf{map}\ \ multMarbles) \cdot unB$$

A função openBags, abstratamente, abre o saco, retira o seu conteudo e aplica uma série de transformações a cada elemento, enviando depois para a função de multiplicação, o resultado final, para que esta possa ”colocar”dentro de outro saco, com ajuda do construtor B. A primeira alteração a executar, é aplicar a unidade monádica ao conteúdo, e assim criar o par (elemento,repetições) de novo, para posteriormente executar a função multMarbles a cada elemento, e depois os vários resultados são concatenados.

$$multMarbles :: ([(a, Int)], Int) \to [(a, Int)]$$
$$multMarbles\ (a, b) = \mathsf{map}\ \ (id \times (*b))\ a$$

A função multMarbles, faz apenas a multiplicação do número de repetições do saco, pelos vários elementos que nem estão embutidos, sejam eles outros sacos, ou apenas os berlindes.

$$dist\ a = distAux\ a\ (sum\ (\mathsf{map}\ \ \pi_2\ (unB\ a)))$$

A distribuição vai ser calculada com a ajuda de uma sequencia de funções que faz a contagem de todos os berlindes dentro de um saco, e de outra função que faz o calculo da probabilidade com ajuda desse número total. O resultado final será usado pelo Mónade Dist.

$$distAux :: Bag\ a \to Int \to \mathsf{Dist}\ a$$
$$distAux\ (B\ a)\ n = D\ (getFreq\ a\ n)$$

A função distAux contrói o Mónade Dist, mas primeiro calcula a frequencia relativa de cada berlinde no saco, com o auxílio da função getFreq.

$$getFreq :: [(a, Int)] \to Int \to [(a, ProbRep)]$$
$$getFreq\ [\,]\ \_ = [\,]$$
$$getFreq\ ((tipo, i) : xs)\ n = (tipo, (fromIntegral\ i)\ /\ (fromIntegral\ n)) : getFreq\ xs\ n$$

Por último, esta função, contrói a lista de pares (a,ProbRep), em que ProbRep é apenas a divisão do número de berlindes do tipo ”a”, pelo numero total de berlindes no saco.

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

[7] Exemplos tirados de [?].

$$
\begin{array}{ll}
\equiv & \{ \text{ identity } \} \\
& \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
\Box &
\end{array}
$$

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

$$
\begin{array}{ccc}
\mathbb{N}_0 & \xleftarrow{\mathsf{in}} & 1 + \mathbb{N}_0 \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|g|\!)} \\
B & \xleftarrow[g]{} & 1 + B
\end{array}
$$

Índice