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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	28
a82441	Alexandre Mendonça Pinho
a82202	Joel Filipe Esteves Gama
a82491	Tiago Martins Pinheiro

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.

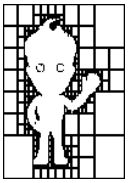
(d) Inversão de cores.

(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,

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

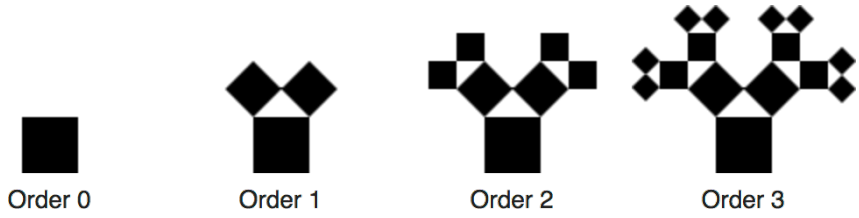


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

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:

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

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* $\binom{n}{k}$ *coincide com a sua especificação (1):*

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

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:

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

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.

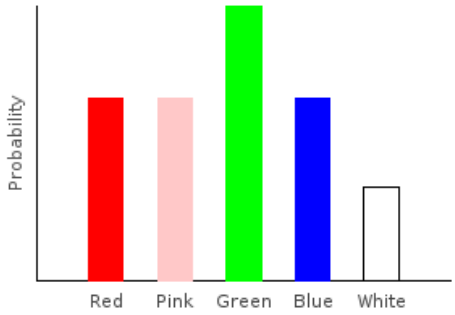


Figura 4: Distribuição de berlindes num saco.

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,

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

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.*

Anexos

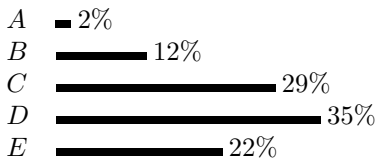
A Mónade para probabilidades e estatística

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

A definição dos combinadores para o Blockchain.

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

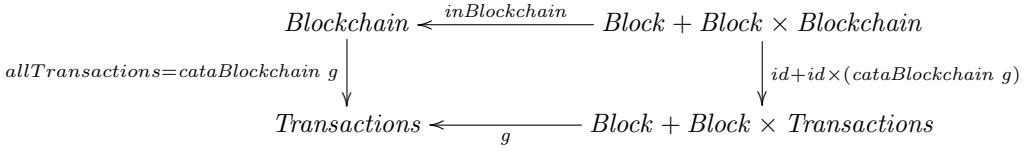
1 - allTransactions

A partir de um bloco, podemos obter a sua lista de transações.

$$MagicNo \times (Time \times Transactions) \xrightarrow[\pi_2 \cdot \pi_2]{} Transactions$$

```
getTransactions :: Block → Transactions
getTransactions = π2 · π2
```

A função *allTransactions* é um catamorfismo de Blockchains que tem como resultado uma lista de transações.



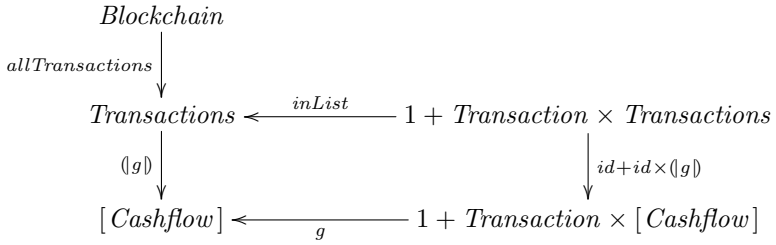
$$allTransactions = cataBlockchain\ g$$
$$\textbf{where}\ g = [getTransactions, \mathsf{conc} \cdot (getTransactions \times id)]$$

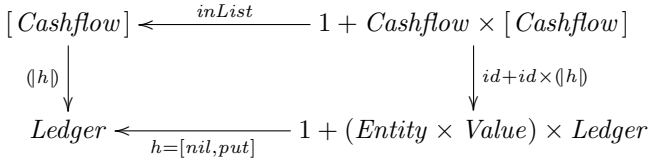
2 - ledger

É definido um tipo auxiliar, `Cashflow`, que representa um depósito de um certo valor na conta de uma entidade.

$$\textbf{type}\ Cashflow = (Entity, Value)$$

Para construir o ledger, primeiro extrai-se todas as transações do blockchain. Depois, através de um catamorfismo, é necessário transformar a lista de transações numa lista de cashflows. Finalmente, são agregados os resultados somando todos os cashflows pertencentes à mesma entidade, também através de um catamorfismo.





$$ledger = (cataList\ h) \cdot (cataList\ g) \cdot allTransactions$$
$$\textbf{where}\ g = [nil, \mathsf{conc} \cdot (f \times id)]$$
$$h = [nil, put]$$
$$f :: Transaction \to [Cashflow]$$
$$f\ (e_1, (v, e_2)) = (e_1, -v) : (e_2, v) : [\,]$$
$$put :: (Cashflow, Ledger) \to Ledger$$
$$put\ (e, [\,]) = [e]$$
$$put\ (e, (h : t)) = \textbf{if}\ \pi_1\ h \equiv \pi_1\ e\ \textbf{then}\ (\pi_1\ h, \pi_2\ e + \pi_2\ h) : t\ \textbf{else}\ h : put\ (e, t)$$

A função auxiliar `f` converte uma transação numa lista com dois cashflows.

A função auxiliar `put` trata de atualizar o ledger com um cashflow, adicionando ao valor da entidade se já consta no ledger, ou adicionando um novo elemento à lista (uma nova entidade).

3 - isValidMagicNr

Cada bloco tem o seu número mágico.

$$MagicNo \times (Time \times Transactions) \xrightarrow[\pi_1]{} MagicNo$$

$$\begin{aligned}&getMagicNo :: Block \to MagicNo\\&getMagicNo = \pi_1\end{aligned}$$

Para verificar se os número mágicos são válidos, é preciso primeiro obter a lista de todos os números mágicos, o que é feito através de um catamorfismo:

$$\begin{array}{ccc} Blockchain & \xleftarrow{inBlockchain} & Block + Block \times Blockchain \\ {\scriptstyle allMagicNos = cataBlockchain\ g}\downarrow & & \downarrow{\scriptstyle id+id\times(cataBlockchain\ g)} \\ [MagicNo] & \xleftarrow[g]{} & Block + Block \times [MagicNo] \end{array}$$

$$\begin{aligned}&allMagicNos :: Blockchain \to [MagicNo]\\&allMagicNos = cataBlockchain\ g\\&\quad \textbf{where}\ g = [singl \cdot getMagicNo, cons \cdot (getMagicNo \times id)]\end{aligned}$$

Com a lista de todos os números mágicos, podemos construir, através de um anamorfismo, a lista dos pares cabeça-cauda, onde cada elemento é emparelhado com a lista dos elementos que lhe seguem. Os elementos da lista de números mágicos são todos únicos se nenhum dos elementos pertencer à cauda associada.

$$\begin{array}{ccc} [MagicNo] & \xrightarrow[h]{} & 1 + (MagicNo \times [MagicNo]) \times [MagicNo] \\ {\scriptstyle anaList\ h}\downarrow & & \downarrow{\scriptstyle id+id\times(anaList\ h)} \\ [MagicNo \times [MagicNo]] & \xleftarrow[inList]{} & 1 + (MagicNo \times [MagicNo]) \times [MagicNo \times [MagicNo]] \end{array}$$

$$\begin{array}{ccc} [MagicNo] & \xrightarrow{h} & 1 + (MagicNo \times [MagicNo]) \times [MagicNo \times [MagicNo]] \\ & {\scriptstyle outList}\searrow \quad \nearrow{\scriptstyle id+\langle id,\pi_2\rangle} & \\ & 1 + MagicNo \times [MagicNo] & \end{array}$$

$$\begin{aligned}&belongs :: Eq\ a \Rightarrow (a, [a]) \to Bool\\&belongs = \widehat{elem}\\&isValidMagicNr = \neg \cdot (any\ belongs) \cdot (anaList\ h) \cdot allMagicNos\\&\quad \textbf{where}\ h = (id + \langle id, \pi_2 \rangle) \cdot outList\end{aligned}$$

Problema 2

A definição dos combinadores para o tipo QTree.

$$\begin{aligned}&inQTree\ (i_1\ (a, (x, y))) = Cell\ a\ x\ y\\&inQTree\ (i_2\ (a, (b, (c, d)))) = Block\ a\ b\ c\ d\\&outQTree\ (Cell\ a\ x\ y) = i_1\ (a, (x, y))\\&outQTree\ (Block\ a\ b\ c\ d) = i_2\ (a, (b, (c, d)))\\&recQTree\ f\ \ = baseQTree\ id\ f\end{aligned}$$

$$cataQTree\ g = g \cdot recQTree\ (cataQTree\ g) \cdot outQTree$$
$$anaQTree\ h = inQTree \cdot (recQTree\ (anaQTree\ h)) \cdot h$$
$$hyloQTree\ g\ h = cataQTree\ g \cdot anaQTree\ h$$
$$baseQTree\ f\ g = (f \times id) + (g \times (g \times (g \times g)))$$

instance *Functor QTree* **where**
  fmap $f = cataQTree\ (inQTree \cdot (baseQTree\ f\ id))$

$$B\ (X, Y) = A \times (Int \times Int) + Y \times (Y \times (Y \times Y))$$

1 a) - rotateQTree

Para rodar uma QTree 90 graus, é preciso rodar cada célula 90 graus, e recursivamente reposicionar os ramos da árvore. Esta operação é um anamorfismo do próprio tipo.

A operação de rotação de uma célula apenas troca as suas coordenadas, já que a célula representa um conjunto de pixeis (neste caso) todos com o mesmo valor.

$$A \times (B \times C) \xrightarrow[id \times swap]{} A \times (C \times B)$$

$$rotateCell = id \times swap$$

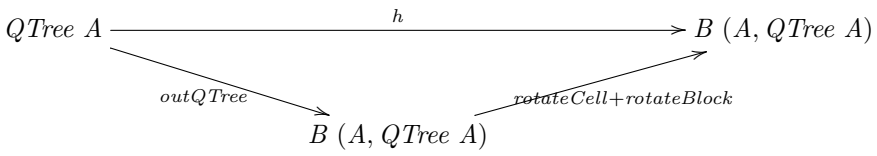
A operação de rotação de um conjunto de ramos implica a troca de posição entre eles.

$$A \times (B \times (C \times D)) \xrightarrow[rotateBlock]{} C \times (A \times (D \times B))$$

$$rotateBlock = \langle \pi_1 \cdot \pi_2 \cdot \pi_2, \langle \pi_1, \langle \pi_2 \cdot \pi_2 \cdot \pi_2, \pi_1 \cdot \pi_2 \rangle \rangle \rangle$$

Assim, temos o seguinte anamorfismo de QTrees:

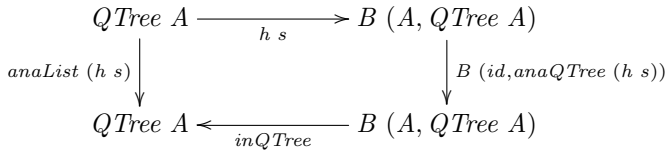
$$\begin{array}{ccc} QTree\ A & \xrightarrow[h]{} & B\ (A, QTree\ A) \\ {\scriptstyle anaList\ h}\downarrow & & \downarrow{\scriptstyle B\ (id, anaQTree\ h)} \\ QTree\ A & \xleftarrow[inQTree]{} & B\ (A, QTree\ A) \end{array}$$

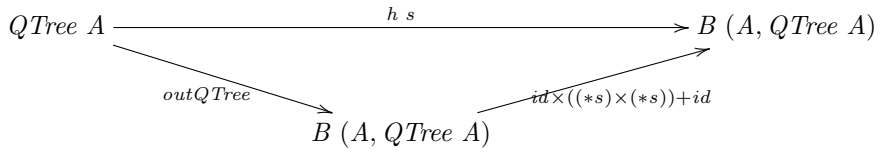


$$rotateQTree = anaQTree\ h$$
  where $h = (rotateCell + rotateBlock) \cdot outQTree$

1 b) - scaleQTree

A operação de redimensionar uma imagem por um certo fator implica apenas o redimensionamente de cada célula por esse mesmo fator. Trata-se de um anamorfismo, pois o resultado necessita de mais informação do que o conteúdo de cada célula (o seu tamanho atual).





$$scaleQTree\ s = anaQTree\ (h\ s)$$
$$\textbf{where}\ h\ s = ((id \times ((*s) \times (*s))) + id) \cdot outQTree$$

1 c) - invertQTree

Inverter as cores de uma quadtree é uma operação que é feita invertendo a cor de cada pixel.

$$invertPx\ (PixelRGBA8\ r\ g\ b\ a) = PixelRGBA8\ (255 - r)\ (255 - g)\ (255 - b)\ a$$

Assim, podemos simplesmente utilizar o padrão funtor para implementar a função desejada.

$$invertQTree = \mathsf{fmap}\ invertPx$$

2 - compressQTree

Comprimir uma quadtree ao reduzir a sua profundidade em `n` níveis é equivalente a excluir todos os ramos depois de um certo nível (a profundidade atual menos `n`).

$$compressQTree\ n\ t = compressQTreeAux\ ((depthQTree\ t) - n)\ t$$

A função auxiliar é uma função recursiva, e o seu primeiro argumento é decrementado a cada nível abaixo do atual. Se o argumento não for positivo e quadtree for um bloco de quadtrees, esse bloco é substituido por uma célula de igual tamanho à soma dos tamanhos dos quatro ramos.

$$compressQTreeAux\ n\ (Block\ nw\ ne\ sw\ se) =$$
$$\textbf{if}\ (n > 0)$$
$$\textbf{then}\ Block\ (f\ (n-1)\ nw)\ (f\ (n-1)\ ne)\ (f\ (n-1)\ sw)\ (f\ (n-1)\ se)$$
$$\textbf{else}\ nwCell\ (Block\ nw\ ne\ sw\ se)\ 0\ 0$$
$$\textbf{where}\ f = compressQTreeAux$$
$$compressQTreeAux\ n\ (Cell\ a\ x\ y) = Cell\ a\ x\ y$$

O algoritmo de determinação da célula que substitui o bloco caso seja necessário é também recursivo. A função mantém um acumulador do tamanho das células já consideradas, e no caso de base, retorna uma célula com a cor da célula no canto superior esquerdo do bloco original, e com o tamanho total deste. Caso contrário, soma a largura e altura da quadtree do canto inferior direito aos respetivos acumuladores, e invoca a função com os acumuladores atualizados na quadtree do canto superior esquerdo.

$$nwCell\ (Cell\ a\ x\ y)\ i\ j = Cell\ a\ (x+i)\ (y+j)$$
$$nwCell\ (Block\ nw\ ne\ sw\ se)\ i\ j = nwCell\ nw\ (i+x)\ (j+y)$$
$$\textbf{where}\ (Cell\ \_\ x\ y) = nwCell\ se\ 0\ 0$$

3 - outlineQTree

Para transformar uma quadtree numa matrix com o seu contorno, transforma-se primeiro esta quadtree numa quadtree com o seu contorno, e em torno produz-se a matriz pretendida.

$$
\begin{array}{l}
QTree\ A \\
\Big\downarrow{\scriptstyle outlineQTreeAux\ p} \\
QTree\ Bool \\
\Big\downarrow{\scriptstyle qt2bm} \\
Matrix\ Bool
\end{array}
$$

$$outlineQTree\ p = qt2bm \cdot (outlineQTreeAux\ p)$$

A função auxiliar é um catamorfismo:

$$
\begin{array}{ccc}
QTree\ A & \xleftarrow{inQTree} & B\ (A, QTree\ A) \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle B\ (id, (\!|g|\!))} \\
QTree\ Bool & \xleftarrow[g]{} & B\ (A, QTree\ Bool
\end{array}
$$

$$
\begin{array}{l}
outlineQTreeAux\ p = cataQTree\ g \\
\quad \textbf{where}\ g = [outlineCell\ p, inQTree \cdot i_2]
\end{array}
$$

A função outlineCell testa se uma célula faz parte do fundo com a função dada. Se sim, retorna um bloco de contorno (valor de True nas bordas e False no interior) com tamanho igual. Se não, retorna a mesma célula mas com o valor de False.

$$outlineCell\ p\ (a, (x, y)) = cond\ p\ \underline{(outlinedBlock\ x\ y)}\ \underline{(Cell\ False\ x\ y)}\ a$$

$$
\begin{array}{l}
outlinedBlock\ x\ y = Block \\
\quad (Block\ (Cell\ True\ 1\ 1) \\
\qquad (Cell\ True\ (x-2)\ 1) \\
\qquad (Cell\ True\ 1\ (y-2)) \\
\qquad (Cell\ False\ (x-2)\ (y-2))) \\
\quad (Cell\ True\ 1\ (y-1)) \\
\quad (Cell\ True\ (x-1)\ 1) \\
\quad (Cell\ True\ 1\ 1)
\end{array}
$$

Problema 3

Conversão de f k:

$$
\begin{array}{ll}
 & \left\{ \begin{array}{l} f\ k\ 0 = 1 \\ f\ k\ (d+1) = (d+k+1) * f\ k\ d \end{array} \right. \\
\equiv & \quad \{\ \text{Igualdade extensional; (d + k + 1) = l k d}\ \} \\
 & \left\{ \begin{array}{l} f\ k \cdot \underline{0} = \underline{1} \\ f\ k \cdot succ = mul\ \langle f\ k, l\ k \rangle \end{array} \right. \\
\equiv & \quad \{\ \text{Eq-+}\ \} \\
 & [f\ k \cdot \underline{0}, f\ k \cdot succ] = [\underline{1}, mul\ \langle f\ k, l\ k \rangle]
\end{array}
$$

$$\equiv \quad \{ \text{ Natural-id; Fusão-+; in = } [\underline{0}, succ] \text{ ; Absorção-+ } \}$$

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

$$\equiv \quad \{ \text{ F f = (id + f) } \}$$

$$f\ k \cdot \mathbf{in} = [\underline{1}, mul] \cdot F\ \langle f\ k, l\ k \rangle$$

Conversão de l k:

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

$$\equiv \quad \{ \text{ Igualdade extensional; Cancelamento-x } \}$$

$$\begin{cases} l\ k \cdot \underline{0} = succ \cdot k \\ l\ k \cdot succ = succ \cdot \pi_2 \cdot \langle f\ k, l\ k \rangle \end{cases}$$

$$\equiv \quad \{ \text{ Eq-+ } \}$$

$$[l\ k \cdot \underline{0}, l\ k \cdot succ] = [succ \cdot k, succ \cdot \pi_2 \cdot \langle f\ k, l\ k \rangle]$$

$$\equiv \quad \{ \text{ Natural-id; Fusão-+; in = } [\underline{0}, succ] \text{ ; Absorção-+ } \}$$

$$l\ k \cdot \mathbf{in} = [succ \cdot k, succ \cdot \pi_2] \cdot (id + \langle f\ k, l\ k \rangle)$$

$$\equiv \quad \{ \text{ F f = (id + f) } \}$$

$$l\ k \cdot \mathbf{in} = [succ \cdot k, succ \cdot \pi_2] \cdot F\ \langle f\ k, l\ k \rangle$$

Conversão de g:

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

$$\equiv \quad \{ \text{ Igualdade extensional; (d + 1) = s d } \}$$

$$\begin{cases} g \cdot \underline{0} = \underline{1} \\ g \cdot succ = mul\ \langle g, s \rangle \end{cases}$$

$$\equiv \quad \{ \text{ Eq-+ } \}$$

$$[g \cdot \underline{0}, g \cdot succ] = [\underline{1}, mul\ \langle g, s \rangle]$$

$$\equiv \quad \{ \text{ Natural-id; Fusão-+; in = } [\underline{0}, succ] \text{ ; Absorção-+ } \}$$

$$g \cdot \mathbf{in} = [\underline{1}, mul] \cdot (id + \langle g, s \rangle$$

$$\equiv \quad \{ \text{ F f = (id + f) } \}$$

$$g \cdot \mathbf{in} = [\underline{1}, mul] \cdot F\ \langle g, s \rangle$$

Conversão de s:

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

$$\equiv \quad \{ \text{ Igualdade extensional; Cancelamento-x } \}$$

$$\begin{cases} s \cdot \underline{0} = \underline{1} \\ s \cdot succ = succ \cdot \pi_2 \cdot \langle g, s \rangle \end{cases}$$

$$\equiv \quad \{ \text{ Eq-+ } \}$$

$$[s \cdot \underline{0}, s \cdot succ] = [\underline{1}, succ \cdot \pi_2 \cdot \langle g, s \rangle]$$

$$\equiv \quad \{ \text{ Natural-id; Fusão-+; in = } [\underline{0}, succ] \text{ ; Absorção-+ } \}$$

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

$$
\begin{aligned}
&\equiv \quad \{\ \text{F f = (id + f)}\ \} \\
&\quad s \cdot \mathbf{in} = [\underline{1}, succ \cdot \pi_2] \cdot F\ \langle g, s\rangle
\end{aligned}
$$

Aplicando a lei da recursividade múltipla para $\langle f\ k, l\ k\rangle$:

$$
\begin{aligned}
&\quad \begin{cases} f\ k \cdot \mathbf{in} = [\underline{1}, mul] \cdot F\ \langle f\ k, l\ k\rangle \\ l\ k \cdot \mathbf{in} = [succ \cdot k, succ \cdot \pi_2] \cdot F\ \langle f\ k, l\ k\rangle \end{cases} \\
&\equiv \quad \{\ \text{Fokkinga}\ \} \\
&\quad \langle f\ k, l\ k\rangle = (\!|\langle [\underline{1}, mul], [succ \cdot k, succ \cdot \pi_2]\rangle|\!)
\end{aligned}
$$

Aplicando a lei da recursividade múltipla para $\langle g, s\rangle$:

$$
\begin{aligned}
&\quad \begin{cases} g \cdot \mathbf{in} = [\underline{1}, mul] \cdot F\ \langle g, s\rangle \\ s \cdot \mathbf{in} = [\underline{1}, succ \cdot \pi_2] \cdot F\ \langle g, s\rangle \end{cases} \\
&\equiv \quad \{\ \text{Fokkinga}\ \} \\
&\quad \langle g, s\rangle = (\!|\langle [\underline{1}, mul], [\underline{1}, succ \cdot \pi_2]\rangle|\!)
\end{aligned}
$$

Combinação dos resultados:

$$
\begin{aligned}
&\quad \langle (\!|[\underline{1}, mul], [succ \cdot k, succ \cdot \pi_2]|\!), (\!|[\underline{1}, mul], [\underline{1}, succ \cdot \pi_2]|\!)\rangle \\
&\equiv \quad \{\ \text{"Banana-split"}\} \\
&\quad (\!|([\underline{1}, mul], [succ \cdot k, succ \cdot \pi_2]) \times ([\underline{1}, mul], [\underline{1}, succ \cdot \pi_2])) \cdot \langle F\ \pi_1, F\ \pi_2\rangle|\!) \\
&\equiv \quad \{\ \text{F f = (id + f); Absorção-x}\ \} \\
&\quad (\!|\langle([\underline{1}, mul], [succ \cdot k, succ \cdot \pi_2]) \cdot (id + \pi_1), ([\underline{1}, mul], [\underline{1}, succ \cdot \pi_2]) \cdot (id + \pi_2)\rangle|\!) \\
&\equiv \quad \{\ \text{Fusão-x}\ \} \\
&\quad (\!|\langle([\underline{1}, mul] \cdot (id + \pi_1), [succ \cdot k, succ \cdot \pi_2]) \cdot (id + \pi_1), ([\underline{1}, mul] \cdot (id + \pi_2), [\underline{1}, succ \cdot \pi_2]) \cdot (id + \pi_2)\rangle|\!) \\
&\equiv \quad \{\ \text{Absorção-+ (x4); Natural-id (x2)}\ \} \\
&\quad (\!|\langle([\underline{1}, mul \cdot \pi_1], [succ \cdot k, succ \cdot \pi_2 \cdot \pi_1]), ([\underline{1}, mul \cdot \pi_2], [\underline{1}, succ \cdot \pi_2 \cdot \pi_2])\rangle|\!)
\end{aligned}
$$

Dedução de base e loop:

$$
\begin{aligned}
&\quad \text{for } loop\ base = (\!|\langle\langle [\underline{1}, mul \cdot \pi_1], [succ, succ \cdot \pi_2 \cdot \pi_1]\rangle, \langle [\underline{1}, mul \cdot \pi_2], [\underline{1}, succ \cdot \pi_2 \cdot \pi_2]\rangle\rangle|\!) \\
&\equiv \quad \{\ \text{for b i = } (\!|[\underline{i}, b]|\!)\text{; Lei da troca (x3)}\ \} \\
&\quad (\!|[base, loop]|\!) = (\!|[\langle\langle \underline{1}, succ\rangle, \langle \underline{1}, \underline{1}\rangle\rangle, \langle\langle mul \cdot \pi_1, succ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, succ \cdot \pi_2 \cdot \pi_2\rangle\rangle]|\!) \\
&\equiv \quad \{\ (\!|f|\!) = (\!|g|\!) \equiv \text{f = g ; Eq-+}\ \} \\
&\quad \begin{cases} base = \langle\langle \underline{1}, +1\rangle, \langle \underline{1}, \underline{1}\rangle\rangle \\ loop = \langle\langle mul \cdot \pi_1, succ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, succ \cdot \pi_2 \cdot \pi_2\rangle\rangle \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&base = tuploaux \cdot \langle\langle \underline{1}, succ\rangle, \langle \underline{1}, \underline{1}\rangle\rangle \\
&loop = tuploaux \cdot \langle\langle mul \cdot \pi_1, succ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, succ \cdot \pi_2 \cdot \pi_2\rangle\rangle \cdot paresaux \\
&tuploaux :: ((integer, integer), (integer, integer)) \to (integer, integer, integer, integer) \\
&tuploaux\ ((x, y), (w, z)) = (x, y, w, z) \\
&paresaux :: (integer, integer, integer, integer) \to ((integer, integer), (integer, integer)) \\
&paresaux\ (x, y, w, z) = ((x, y), (w, z))
\end{aligned}
$$

Problema 4

A definição dos combinadores para o tipo FTree.

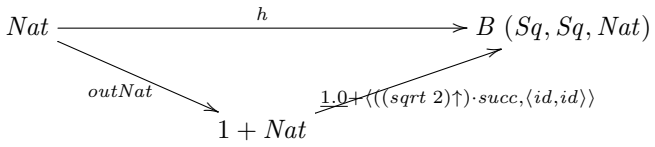
$$
\begin{aligned}
&inFTree = [Unit, (\lambda a \rightarrow \overbrace{(Comp\ a)}^{\frown})] \\
&outFTree\ (Unit\ a) = i_1\ a \\
&outFTree\ (Comp\ a\ b\ c) = i_2\ (a, (b, c)) \\
&baseFTree\ f\ g\ h = g + f \times (h \times h) \\
&recFTree\ f = baseFTree\ id\ id\ f \\
&cataFTree\ g = g \cdot recFTree\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

$$B\ (X, Y, Z) = Y + X \times (Z \times Z)$$

1 - generatePTree

A função `generatePTree` constroi uma PTree dado um número natural, pelo que é um anamorfismo. A sua folhas mais exteriores tem o valor de 1.0, e a cada nível mais perto da raiz da árvore o tamanho aumenta por um fator de $sqrt\ 2$.

$$
\begin{array}{ccc}
Nat & \xrightarrow{\quad h \quad} & B\ (Sq, Sq, Nat) \\
{\scriptstyle anaNat\ h}\downarrow & & \downarrow{\scriptstyle B\ (id,id,anaNat\ h)} \\
FTree\ Sq\ Sq & \xleftarrow[\quad inFTree \quad]{} & B\ (Sq, Sq, FTree\ Sq\ Sq)
\end{array}
$$



$$
\begin{aligned}
&generatePTree = anaFTree\ h \\
&\quad \textbf{where}\ h = (\underline{1.0} + \langle (sqrt\ (2)\uparrow) \cdot succ, \langle id, id \rangle \rangle) \cdot outNat
\end{aligned}
$$

2 - drawPTree

Para desenhar a árvore de Pitágoras como uma animação, é preciso construir a lista de imagens em que cada imagem é um frame da animação. Isto é feito através de um ciclo for (um catamorfismo) cujo corpo é uma função que recebe a lista da animação já construida e o número do próximo frame a incluir nessa lista. Este ciclo for tem tantas iterações quanto a profundidade da árvore para desenhar.

$$
\begin{array}{ccc}
FTree\ Sq\ Sq & & \\
{\scriptstyle depthFTree}\downarrow & & \\
Nat & \xleftarrow{\quad \mathsf{in} \quad} & 1 + Nat \\
{\scriptstyle (\!|[\underline{zero}, body]|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|[\underline{zero}, body]|\!)} \\
[Picture] \times Nat & \xleftarrow[\quad g=[\underline{zero}, body] \quad]{} & 1 + [Picture] \times Nat
\end{array}
$$

```
drawPTree t = animation
  where (animation, _) = for body zero (depthFTree t)
    zero = ([f 0], 0)
    f x = drawPTreeAt x t
    body (r, n) = (r ++ [f (n + 1)], n + 1)
```

A função que gera cada frame da animação primeiro emparelha cada folha da árvore com um número que representa a sua posição relativa à raiz.

```
tagPTree n = inFTree · (f + f × (rec × rec)) · outFTree
  where f = ⟨id, n̲⟩
    rec = tagPTree (n + 1)
```

Depois, a função *drawPTreeAt* constroi uma lista de imagens, uma para cada quadrado, através de uma função auxiliar, torna essa lista numa só imagem, e aumenta o seu tamanho para que os resultados sejam visíveis no ecrã.

```
drawPTreeAt x = (Graphics.Gloss.scale 20.0 20.0) · pictures · (drawPTreeAtAux x (0.0, 0.0) (0.0, 1.0)) · tagPTree
```

A função auxiliar é uma função recursiva. Ela mantém como argumentos o atual frame, a posição e direção do próximo quadrado a desenhar. Nas chamadas recursivas estes valores são atualizados conforme previsto pela geometria da árvore de Pitágoras.

```
drawPTreeAtAux x pos dir (Unit (a, n)) = [drawSquare x n a pos dir]
drawPTreeAtAux x pos dir (Comp (a, n) e d) =
  [drawSquare x n a pos dir] ++
  (drawPTreeAtAux x newPos1 newDir1 e) ++
  (drawPTreeAtAux x newPos2 newDir2 d)
    where newPos1 = addV pos (addV (resizeV a dir) (rotateV (−90.0) (resizeV (a / 2.0) dir)))
      newDir1 = rotateV (−45.0) $ scaleV ((sqrt 2.0) / 2.0) dir
      newPos2 = addV pos (addV (resizeV a dir) (rotateV (90.0) (resizeV (a / 2.0) dir)))
      newDir2 = rotateV 45.0 $ scaleV ((sqrt 2.0) / 2.0) dir
```

A função que desenha um dos quadrados recebe a posição em que desenhar e a direção para deduzir a rotação necessária. Se a profundidade atual for maior que a profundidade máxima pretendida para o frame da animação, a função retorna uma imagem vazia.

```
drawSquare max_n n a (x, y) dir =
  if (n ⩽ max_n)
  then Translate x y (Rotate (getAngle dir) (rectangleSolid a a))
  else Blank
```

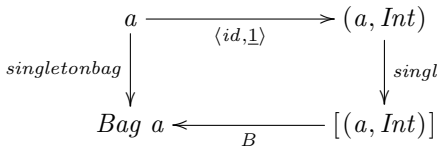
Funções auxiliares para manipulação de vetores 2D.

```
addV (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
scaleV k (x, y) = (x ∗ k, y ∗ k)
lengthV (x, y) = sqrt (x ↑↑ 2 + y ↑↑ 2)
resizeV l v = scaleV (l / (lengthV v)) v
getAngle (x, y) = toDeg $ atan2 y x
rotateV theta (x, y) = (x ∗ cos rad + y ∗ sin rad, y ∗ cos rad − x ∗ sin rad)
  where rad = toRad theta
toRad deg = deg / 360.0 ∗ (2.0 ∗ pi)
toDeg rad = rad / (2.0 ∗ pi) ∗ 360.0 − 90.0
```

Problema 5

1 - Singletonbag

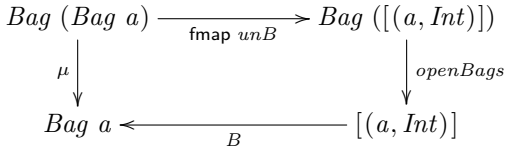
A *singletonbag* começa por aplicar o $\langle id, \underline{1} \rangle$ ao seu argumento *a*, cor do berlinde. Desta forma, obtém um tuplo onde o primeiro elemento é a cor do berlinde e o segundo é a constante 1. De seguida, é aplicada a função singl, que transforma o tuplo numa lista de tuplos. E, por fim, o construtor *B* constrói uma *Bag*.



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

2 - muB

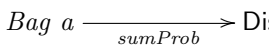
O μ vai aplicar sucessivamente ao seu argumento (Um saco que contém outros sacos) a função unB que, de modo simplista, abre cada saco. Depois, é aplicada a função $openBags$ que volta a chamar a função unB e, de seguida, passa o seu retorno à $multMarbles$ que multiplica o número de berlindes (2º elemento do tuplo *(a,int)*) por um fator, que neste caso é o número de sacos daquele "tipo". Por fim, é feito o *concat*, formando uma lista, e aplicado o construtor B.



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

3 - Dist

A função *Dist* recebe um saco (Bag) e calcula qual é a probabilidade de cada cor de berlinde que esse Bag contém. Para isso, começa por calcular quantos berlindes tem o saco, através da função *numberMarbles*, passando o resultado dessa função e o argumento recebido à função *sumProb*. Essa função calcula a probabilidade para cada cor, depois constrói o tipo *Dist*, que a cada cor associa a sua respetiva probabilidade.



$$dist\ a = sumProb\ a\ (numberMarbles\ a)$$

Funções auxiliares

```
  -- muB
openBags :: Bag [(a, Int)] → [(a, Int)]
openBags = concat · (map multMarbles) · unB
multMarbles :: ([(a, Int)], Int) → [(a, Int)]
multMarbles (a, b) = map (id × (∗b)) a
  -- Dist
sumProb :: Bag a → Int → Dist a
sumProb (B a) nMarbles = D (getProb a nMarbles)
getProb :: [(a, Int)] → Int → [(a, ProbRep)]
getProb [] _ = []
getProb ((tipo, i) : xs) nMarbles = cons ((tipo, (fromIntegral i) / (fromIntegral nMarbles)), getProb xs nMarbles)
numberMarbles :: Bag a → Int
numberMarbles (B []) = 0
numberMarbles (B ((tipo, i) : xs)) = i + numberMarbles (B xs)
```

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

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

[7] Exemplos tirados de [?].