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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	49 (preencher)
a81736	Marco Dantas
a82467	José Fernandes
a81644	César Borges

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 **??** 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 **??**.

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

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

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 **??**.

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:

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

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

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

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

```
> 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 **??**, **??** e **??**:

```
> 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.

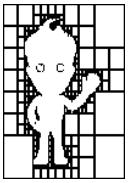
(d) Inversão de cores.

(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 **??**, **??**, **??** e **??**:

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

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

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

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 **??** e **??**:

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

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

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

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,

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

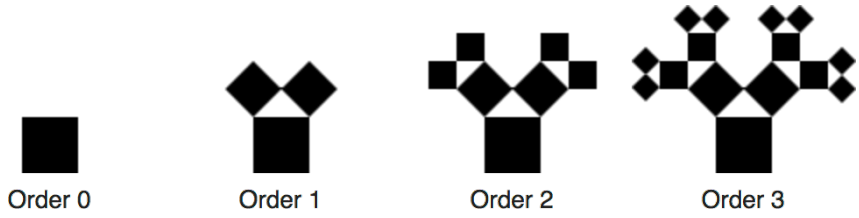


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

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

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

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 (??):*

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

Problema 4

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

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

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

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.

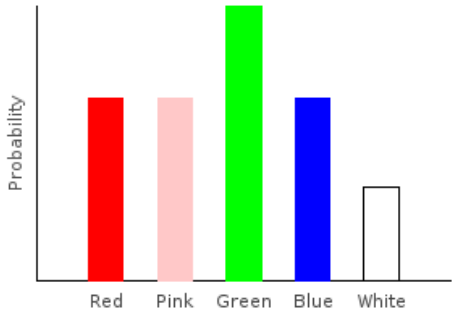


Figura 4: Distribuição de berlindes num saco.

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 **??**.

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.*

Anexos

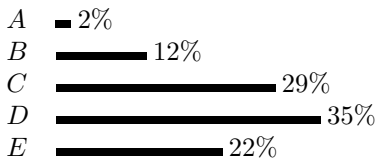
A Mónade para probabilidades e estatística

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

Funções Auxiliares

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

allTransactions

```
allTransactions = cataBlockchain [g, conc · ((g) × id)]
  where g = π2 · π2
```

Utilizamos um catamorfismo para aplicar recursividade à blockChain, quando temos um Bc aplicamos g ao Block:

$$A \times (C \times D) \xrightarrow{\quad g \quad} D$$

Se tivermos um Bcs:

$$(Block \times Transactions) \xrightarrow{conc \cdot (g \times id)} Transactions$$

ledger

$$
\begin{aligned}
&ledger = joinledges \cdot cataBlockchain\ [block2LedgeList, \mathsf{conc} \cdot ((block2LedgeList) \times id)]\\
&\quad \textbf{where}\ joinledges = \mathsf{map}\ \ pairList2Pair \cdot groupBy\ (groupByFst) \cdot sort\\
&\qquad pairList2Pair\ (x) = ((\pi_1 \cdot head)\ x, (sum\ ([\pi_2\ (y) \mid y \leftarrow x])))\\
&\qquad groupByFst\ x\ y = (\pi_1\ x) \equiv (\pi_1\ y)\\
&\qquad block2LedgeList = concat \cdot (\mathsf{map}\ \ (\mathsf{conc} \cdot ledgeSinglePair)) \cdot \pi_2 \cdot \pi_2\\
&\qquad ledgeSinglePair = \langle singl \cdot (id \times \pi_1), singl \cdot (id \times negate) \cdot swap \cdot \pi_2 \rangle
\end{aligned}
$$

Utilizando um catamorfismo obtemos uma lista de pares de entidades e o valor da transação, as funções para obter este resultado encontram-se a seguir: block2LedgeList:

$$
\begin{array}{c}
Block\\
\downarrow{\scriptstyle \pi_2 \cdot \pi_2}\\
Transactions\\
\downarrow{\scriptstyle \mathsf{map}\ \ (\mathsf{conc} \cdot ledgeSinglePair)}\\
[[Entity \times Value]]\\
\downarrow{\scriptstyle concat}\\
[Entity \times Value]
\end{array}
$$

ledgeSinglePair:

$$
\begin{array}{c}
(Entity \times (Value \times Entity))\\
\downarrow{\scriptstyle ledgeSinglePair}\\
[Entity \times Value]
\end{array}
$$

Depois de termos [(Entity,Value)] vamos chamar a função ”joinledges”que essencialmente ordena todos os pares da lista, para ficarem os pares com a mesma ”Entity”seguidos na lista. Depois agrupamos todos esses pares numa lista, sendo o critério a ”Entity”. Por fim fazemos um map onde por cada lista contida da [[(Entity,Value)]] formamos um par através da ”pairList2Pair”esse par será (Entity, soma de todos os ”Values”na lista), obtendo assim o resultado final.

isValidMagicNr

$$
\begin{aligned}
&isValidMagicNr = \widehat{(\equiv)} \cdot \langle \mathsf{length}\ \cdot n, \mathsf{length}\ \cdot nub \cdot n \rangle\\
&\quad \textbf{where}\ n = cataBlockchain\ [singl \cdot \pi_1, cons \cdot (\pi_1 \times id)]
\end{aligned}
$$

Foi utilizado um catamorfismo para criar uma lista com todos os ”MagicNr”, depois foi comparado o comprimento dessa lista com o da mesma, mas sem repetidos, utilizando a função ”nub”. Caso seja igual apenas existem ”MagicNr”únicos.

Problema 2

Funções Auxiliares

$$
\begin{aligned}
&pQ1 = \pi_1\\
&pQ2 = \pi_1 \cdot \pi_2\\
&pQ3 = \pi_1 \cdot \pi_2 \cdot \pi_2\\
&pQ4 = \pi_2 \cdot \pi_2 \cdot \pi_2\\
&q4x\ (x, y, z, t) = (x \times (y \times (z \times t)))
\end{aligned}
$$

$$
\begin{aligned}
&q4xu\ h = q4x\ (h,h,h,h)\\
&cellBuild\ (a,(b,c)) = Cell\ a\ b\ c\\
&q4toList\ ((x,(y,(z,d)))) = x:y:z:d:[\,]
\end{aligned}
$$

$$
\begin{aligned}
&inQTree\ (i_1\ c) = cellBuild\ c\\
&inQTree\ (i_2\ x) = Block\ (pQ1\ x)\ (pQ2\ x)\ (pQ3\ x)\ (pQ4\ x)\\
&outQTree\ (Cell\ a\ b\ c) = i_1\ (a,(b,c))\\
&outQTree\ (Block\ x\ y\ z\ t) = i_2\ (x,(y,(z,t)))\\
&baseQTree\ f\ h = (f \times id) + (q4xu\ h)\\
&recQTree\ f = baseQTree\ id\ f\\
&cataQTree\ g = g \cdot recQTree\ (cataQTree\ g) \cdot outQTree\\
&anaQTree\ g = inQTree \cdot recQTree\ (anaQTree\ g) \cdot g\\
&hyloQTree\ h\ g = cataQTree\ h \cdot anaQTree\ g\\
&\textbf{instance}\ Functor\ QTree\ \textbf{where}\\
&\quad \mathsf{fmap}\ f = cataQTree\ (inQTree \cdot baseQTree\ f\ id)
\end{aligned}
$$

Soluções

rotateQTree

$$
\begin{aligned}
&rotateQTree = cataQTree\ [cellBuild \cdot (id \times swap), inQTree \cdot i_2 \cdot z]\\
&\quad \textbf{where}\ z\ (q1,(q2,(q3,q4))) = (q3,(q1,(q4,q2)))
\end{aligned}
$$

Redimencionamento a partir da alteração dos inteiros contidos em cada célula

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

Troca das células de um bloco para obtermos a rotação

$$A \times (B \times (C \times D) \xrightarrow{(z)} C \times (A \times (D \times B))$$

scaleQTree

$$scaleQTree\ i\ x = cataQTree\ [cellBuild \cdot (\lambda(a,(b,c)) \to (a,(b*i,c*i))), inQTree \cdot i_2]\ x$$

Através de um catamorfismo vamos em todas as células da árvore multiplicar os seus inteiros por um fator.

invertQTree

$$
\begin{aligned}
&invertQTree = \mathsf{fmap}\ (f)\\
&\quad \textbf{where}\ f\ (PixelRGBA8\ r\ g\ b\ a) = PixelRGBA8\ (255-r)\ (255-g)\ (255-b)\ (a)
\end{aligned}
$$

Utilizando um fmap modificamos o pixel de cada célula

compressQTree

$$
\begin{aligned}
&compressAux :: QTree\ a \to QTree\ a\\
&compressAux\ (Block\ (Cell\ a\ b1\ c1)\ (Cell\ \_\ b2\ c2)\ (Cell\ \_\ b3\ c3)\ (Cell\ \_\ b4\ c4)) = (Cell\ a\ (b1+b2)\ (c1+c3)\\
&compressAux\ x = x
\end{aligned}
$$

Em principio a segunda linha de código nunca chega a ser executada e apenas está aí para o Haskell não dar erro.

No compressQTree, o anaQTree usa um par com o tamanho da árvore e ela própria, deste modo, o elemento do par com o tamanho é decrementado e a árvore é inalterada até que, o valor anterior seja

menor ou igual ao do recebido pelo compressQTree, neste caso o Block é convertido para uma Cell, ou não existe mais sub-árvores. Foi utilizado um catamorfismo para transformar um Block num Cell.

$$
\begin{aligned}
&compAux :: Int \to (Int, QTree\ a) \to (a, (Int, Int)) + (((Int, QTree\ a), ((Int, QTree\ a), ((Int, QTree\ a), (Int, QTre\\
&compAux\ i\ (\_, Cell\ x\ b\ c) = i_1\ (x, (b, c))\\
&compAux\ i\ (r, Block\ a\ b\ c\ d) = \mathbf{if}\ (r > i)\ \mathbf{then}\ i_2\ ((r-1, a), ((r-1, b), ((r-1, c), (r-1, d))))\ \mathbf{else}\ i_1\ \$\ (a1, (b\\
&\quad \mathbf{where}\ (Cell\ a1\ b1\ c1) = fullCompression\ (Block\ a\ b\ c\ d)\\
&fullCompression = cataQTree\ [cellBuild, compressAux \cdot inQTree \cdot i_2]\\
&compressQTree\ i\ x = anaQTree\ (compAux\ i)\ (depthQTree\ x, x)
\end{aligned}
$$

O catamorfismo utilizado para esta solução devolve um par (Int, QTree a), sendo o primeiro elemento a altura total e o segundo a árvore.Chamamos altura visto que o catamorfismo começa por baixo. Com a função auxiliar f vamos etiquetando cada subÁrvore com a sua altura, (Altura, Árvore).

$$(A \times B) \times (A \times B) \times (A \times B) \times (A \times B) \xrightarrow{q4xu\ (\pi_1)} A \times (A \times (A \times A))$$

$$A \times (A \times (A \times A)) \xrightarrow{(+1) \cdot maximum \cdot q4toList} A$$

Com a função auxiliar v verificamos se estamos a uma altura inferior ou igual à profundidade dada, em caso afimativo comprimimos utilizando a "compressAux"para cortar folhas.

outlineQTree

$$
\begin{aligned}
&outlineQTree\ f = cataQTree\ [mat, joinmats]\\
&\quad \mathbf{where}\ mat\ (x, (i, j)) = matrix\ j\ i\ (\lambda(a, b) \to \mathbf{if}\ ((a \equiv 1 \vee a \equiv j) \vee (b \equiv 1 \vee b \equiv i))\ \mathbf{then}\ (f\ x)\ \mathbf{else}\ False)\\
&\qquad joinmats\ (a, (b, (c, d))) = (a \updownarrow b) \leftrightarrow (c \updownarrow d)
\end{aligned}
$$

O catamorfismo permite construir a matriz sempre temos uma célula, neste caso verificamos se estámos nas bordas da matriz e aí chamamos a função "f", caso contrário será Falso. Quando temos um Block juntamos as suas matrizes.

Problema 3

$$
\begin{aligned}
&base\ k = (1, \mathsf{succ}\ \ k, 1, 1)\\
&loop = unmakeP \cdot \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle \cdot makeP\\
&\quad \mathbf{where}\ makeP\ (a, b, c, d) = ((a, b), (c, d))\\
&\qquad unmakeP\ ((a, b), (c, d)) = (a, b, c, d)
\end{aligned}
$$

Explicação

$$
\begin{aligned}
&\quad fk\\
\equiv &\quad \{\ \text{Pointfree}\ \}\\
&\quad \begin{cases} f\ k \cdot \underline{0} = \underline{1} \\ f\ k \cdot \mathsf{succ}\ = mul \cdot \langle f\ k, l\ k\rangle \end{cases}\\
\equiv &\quad \{\ \text{Eq-+}\ \}\\
&\quad [f\ k \cdot \underline{0}, fk \cdot \mathsf{succ}\ ] = [\underline{1}, mul \cdot \langle f\ k, l\ k\rangle]\\
\equiv &\quad \{\ \text{Fusão-+ e Absorção-+}\ \}\\
&\quad f\ k \cdot inN = [\underline{1}, mul] \cdot (id + \langle f\ k, l\ k\rangle)\\
\square &
\end{aligned}
$$

lk

$\equiv$ { Pointfree }

$$\begin{cases} l\ k \cdot \underline{0} = \mathsf{succ}\ \cdot k \\ l\ k \cdot \mathsf{succ}\ = \mathsf{succ}\ \cdot l\ k \end{cases}$$

$\equiv$ { Eq-+ }

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

$\equiv$ { Fusão-+, Absorção-+ e Cancelamento-x }

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

$\square$

Juntando as duas funções para usar "Fokkinga": $\begin{cases} f\ k \cdot inN = [\underline{1}, mul] \cdot (id + \langle f\ k, l\ k \rangle) \\ l\ k \cdot inN = [\mathsf{succ}\ \cdot k, \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle f\ k, l\ k \rangle) \end{cases}$

$\equiv$ { Fokkinga }

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

$\square$

g

$\equiv$ { Pointfree }

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

$\equiv$ { Eq-+, Fusão-+ e Absorção-+ }

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

$\square$

s

$\equiv$ { Pointfree }

$$\begin{cases} s \cdot \underline{0} = \underline{1} \\ s \cdot \mathsf{succ}\ = \mathsf{succ}\ \cdot s \end{cases}$$

$\equiv$ { Eq-+, Fusão-+, Absorção-+ e Cancelamento-x }

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

$\square to$ Juntando as duas funções para usar "Fokkinga": $\begin{cases} g \cdot inN = [\underline{1}, mul] \cdot (id + \langle g, s \rangle) \\ s \cdot inN = [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle g, s \rangle) \end{cases}$

$\equiv$ { Fokkinga }

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

$\square$

$Seja \langle f\ k, l\ k \rangle = (\!|i|\!) e \langle g, s \rangle = (\!|j|\!)$

$\equiv$ { recorrendo aos resultados de usar Fokkinga, lei Banana-split, Absorção-x }

$\langle (\!|i|\!), (\!|j|\!) \rangle = (\!|\langle \langle [\underline{1}, mul], [\mathsf{succ}\ \cdot k, \mathsf{succ}\ \cdot \pi_2] \rangle \cdot F\ \pi_1, \langle [\underline{1}, mul], [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \rangle \cdot F\ \pi_2 \rangle|\!)$

$\equiv$ { Lei da troca – 2 vezes }

$$
\begin{aligned}
& (\!|\langle[\langle \underline{1}, \mathsf{succ}\ \cdot k\rangle, \langle mul, \mathsf{succ}\ \rangle \cdot \pi_2], \cdot\rangle \cdot F\ \pi_1\ [\langle \underline{1}, \underline{1}\rangle, \langle mul, \mathsf{succ}\ \rangle \cdot \pi_2] \cdot F\ \pi_2|\!) \\
\equiv \quad & \{\ \text{F f = (id + f), Absorção-+, Fusão-x}\ \} \\
& (\!|\langle[\langle \underline{1}, \mathsf{succ}\ \cdot k\rangle, \langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle], [\langle \underline{1}, \underline{1}\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle]\rangle|\!) \\
\equiv \quad & \{\ \text{Lei da Troca}\ \} \\
& (\!|[\langle\langle \underline{1}, \mathsf{succ}\ \cdot k\rangle, \langle \underline{1}, \underline{1}\rangle\rangle, \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle]|\!)
\end{aligned}
$$

$$
\equiv \quad \{\ \mathsf{for}\ b\ i = (\!|[\underline{i}, b]|\!)\ \}
$$

$$
\begin{cases}
base\ k = \langle\langle \underline{1}, \mathsf{succ}\ \cdot k\rangle, \langle \underline{1}, \underline{1}\rangle\rangle \\
loop = \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle
\end{cases}
$$

□

Problema 4

```
branchBuild (a, (b, c)) = Comp a b c
inFTree = [Unit, branchBuild]
outFTree (Unit u) = i1 u
outFTree (Comp a b c) = i2 (a, (b, c))
baseFTree f g h = g + (f × (h × h))
recFTree f = baseFTree id id f
cataFTree g = g · recFTree (cataFTree g) · outFTree
anaFTree g = inFTree · recFTree (anaFTree g) · g
hyloFTree h g = cataFTree h · anaFTree g
instance Bifunctor FTree where
  bimap f g = cataFTree (inFTree · baseFTree f g id)
```

Para resolver o problema de gerar uma árvore de Pitágoras decidimos que a raiz desa seria sempre 1, e que, as sucessivas raizes das subárvores seriam $\sqrt{2}/2$ maiores que a anterior, desta forma garantimos uma relação (percentagem) direta de um nodo à raiz da árvore. Esta caracteristica foi utilizada no segundo problema dasta questão para redimensionar os quadrados.

$$
\begin{array}{ccc}
1 + \mathbb{N}_0 \times Float & \xleftarrow{outNat \times id} & \mathbb{N}_0 \times Float \\
\Big\downarrow{\scriptstyle anaFtree\ generateSquare} & & \Big\downarrow{\scriptstyle id + (anaFtree\ generateSquare)} \\
1 + PTree & \xleftarrow[generateSquare]{} & PTree
\end{array}
$$

```
generateSquare :: (Int, Square) → Square + (Square, ((Int, Square), (Int, Square)))
generateSquare (n, f) = if (0 ⩾ n) then i1 f else i2 (f, ((predNat n, fator f), (predNat n, fator f)))
  where fator f = f ∗ (sqrt (2) / 2)
generatePTree = anaFTree (generateSquare) · ⟨id, 1.0⟩
```

```
type Vect = (Float, Float)
poligono = Polygon [(0.0, 0.0), (−100.0, 0.0), (−100.0, 100.0), (0.0, 100.0)]
```

Poligono de base. Nos poligonos do gloss o primeiro ponto da lista conta como o ponto central. Por causa disto sempre que aplicamos uma transformação a um poligono temos garantia que o primeiro ponto se encontra sempre no centro.

```
transformPoligono :: Vect → Float → Float → Picture
transformPoligono (x, y) ang escala = translate x y $ rotate ang $ Graphics.Gloss.scale escala escala $ poligono
  -- Aplica a translação, rotação e scale a um poligono
subractPair (x1, y1) (x2, y2) = (x2 − x1, y2 − y1)
addPair (x1, y1) (x2, y2) = (x2 + x1, y2 + y1)
```

```
middlepoint (x1, y1) (x2, y2) = ((x2 + x1) / 2, (y2 + y1) / 2)
resizeVect i (x, y) = (i * x, i * y)
nextVects (Polygon [x, y, z, d]) = (vectAddicional, vecVert)
  where vecVert = subractPair x d
    vectHorizontal = subractPair x (middlepoint x y)
    vectAddicional = addPair (addPair vectHorizontal vecVert) littleVec
    littleVec = resizeVect (cos (pi / 4) * sqrt 2 / 2) vecVert
```

Baseia-se na ordem que escolhemos para os pontos dos poligonos.

Utilizamos a ordem inferior direito, inferior esquerdo, superior esquerdo, superior direito.

A transformação a aplicar para posicionar o quadrado direito seguinte é apenas um vetor com a mesma dimensão direção e sentido que o lado vertical (assumindo que este é o primeiro quadrado).

A transformação a aplicar ao quadrado direito é o resultado de somar à transformação aplicada ao quadrado direito uma transformação com a mesma dimensão, direção e sentido da base do quadrado direito.

```
rotateVector angulo (x, y) = (nx, ny)
  where ang = angulo * pi / 180
    nx = (x * (cos ang)) + (y * (sin ang))
    ny = ((-x) * (sin ang)) + (y * (cos ang))
type Transforms = ((Vect, Float), (FTree Float Float))
nextCall :: Transforms → Picture + (Picture, (Transforms, Transforms))
nextCall ((vecAcum, angAcum), Unit a) = i1 $ transformPoligono vecAcum angAcum a
nextCall ((vecAcum, angAcum), Comp a b c) = i2 (transformPoligono vecAcum angAcum a, (left, right))
  where left = ((addPair vecAcum vleft, angAcum − 45), b)
    right = ((addPair vecAcum vright, angAcum + 45), c)
    (vleft, vright) = vectTransforms $ nextVects poligono
    vectTransforms = rotateVector angAcum · resizeVect a × rotateVector angAcum · resizeVect a
```

Função do anamorfismo.

O caso de paragem é o segundo elemento do par ser uma unit, neste caso transformamos o float no Poligono de base é lhe aplicado o scale ao fator do float e são aplicadas as transformações até agora acumuladas.

No caso de o segundo elemento ser um Comp transformamos o float numa Poligono como no exemplo anterior, passamos para a chamada recursiva os ramos da arvore e as novas transformações a ser aplicadas aos floats consecutivos.

Para a chamada da esquerda subtraimos 45 ao angulo acumulado, para a chamada da direita adicionamos 45.

Os vetores adicionados aos vetores acumulados são rotações e redimensionamento dos vetores das transformações executadas no poligono base.

Temos de aplicar a transformação ao vetor base aqui devido à maneira como as transformações são aplicadas às pictures.

```
joinPics = cataFTree [singl, aux]
  where aux (a, (b, c)) = a : (zipWith (λx b → pictures [a, x, b]) b c)
```

Catamorfismo que transforma a FTree Picture Picture em [Picture].

Função aux junta duas listas de pictures juntando os elementos que se encontram em posições com o mesmo indice numa picture e adicionando a essa picture a picture que será adicionada à cabeça da lista.

```
drawPTree = joinPics · (anaFTree nextCall) · setup
  where setup x = (((0, 0), 0), x)
      -- split (split (split (const 0) (const 0)) (const 0)) id
main :: IO ()
main = animatePTree 6
```

Problema 5

Cria-se uma Bag apenas com um

$$a \xrightarrow{singletonbag} Bag\ a$$

$$singletonbag\ x = B\ [(x, 1)]$$

Retira as bags interiores a uma bag comum, deixando os elementos nelas contidas

$$(Bag\ (Bag\ a)) \xrightarrow{\mu} Bag\ a$$

$$\mu\ (B\ d) = B\ (concat\ ([concat\ (replicate\ a\ x) \mid ((B\ x), a) \leftarrow d]))$$

$$dist\ (B\ x) = uniform\ (concat\ [replicate\ a\ d \mid (d, a) \leftarrow x])$$

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:

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

[7] Exemplos tirados de [?].