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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	18
a80874	João Pimentel
a78352	Bruno Veloso
a80757	Jaime Leite

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" [1], 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.

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

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

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

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.

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

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

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

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.

$$\begin{array}{l}\mathbf{type}\ MagicNo = String\\ \mathbf{type}\ Time = Int\quad \text{-- em milisegundos}\\ \mathbf{type}\ Entity = String\\ \mathbf{type}\ Value = Int\end{array}$$

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.

$$\begin{array}{l}\mathbf{data}\ QTree\ a = Cell\ a\ Int\ Int \mid Block\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)\\ \quad \mathbf{deriving}\ (Eq, Show)\end{array}$$

```
( 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:

$$
\begin{array}{l}
bm2qt :: (Eq\ a) \Rightarrow Matrix\ a \rightarrow QTree\ a \\
bm2qt = anaQTree\ f\ \mathbf{where} \\
\quad f\ m = \mathbf{if}\ one\ \mathbf{then}\ i_1\ u\ \mathbf{else}\ i_2\ (a, (b, (c, d))) \\
\qquad \mathbf{where}\ x = (nub \cdot toList)\ m \\
\qquad\qquad u = (head\ x, (ncols\ m, nrows\ m)) \\
\qquad\qquad one = (ncols\ m \equiv 1 \vee nrows\ m \equiv 1 \vee \mathsf{length}\ \ x \equiv 1) \\
\qquad\qquad (a, b, c, d) = splitBlocks\ (nrows\ m\ `div`\ 2)\ (ncols\ m\ `div`\ 2)\ m
\end{array}
$$

$$
\begin{array}{l}
qt2bm :: (Eq\ a) \Rightarrow QTree\ a \rightarrow Matrix\ a \\
qt2bm = cataQTree\ [f, g]\ \mathbf{where} \\
\quad f\ (k, (i, j)) = matrix\ j\ i\ \underline{k} \\
\quad g\ (a, (b, (c, d))) = (a \updownarrow b) \leftrightarrow (c \updownarrow d)
\end{array}
$$

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 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:

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

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

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

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 $rotateQTree :: QTree\ a \rightarrow QTree\ a$, $scaleQTree :: Int \rightarrow QTree\ a \rightarrow QTree\ a$ e $invertQTree :: QTree\ a \rightarrow QTree\ a$, 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.

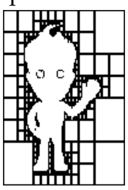
(d) Inversão de cores.

(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,

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

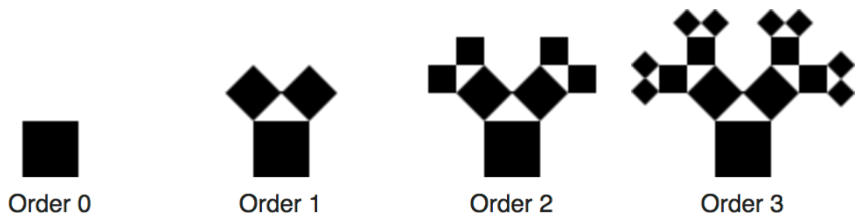


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

$$\begin{aligned}
&h\ k\ d = \frac{f\ k\ d}{g\ d}\\
&f\ k\ d = \frac{(d+k)!}{k!}\\
&g\ d = d!
\end{aligned}$$

assumindo-se $d = n - k \geqslant 0$. É fácil de ver que $f\ k$ e g se desdobram em 4 funções mutuamente recursivas, a saber

$$\begin{aligned}
&f\ k\ 0 = 1\\
&f\ k\ (d+1) = \underbrace{(d+k+1)}_{l\ k\ d} * f\ k\ d\\
&l\ k\ 0 = k+1\\
&l\ k\ (d+1) = l\ k\ d+1
\end{aligned}$$

e

$$\begin{aligned}
&g\ 0 = 1\\
&g\ (d+1) = \underbrace{(d+1)}_{s\ d} * g\ d\\
&s\ 0 = 1\\
&s\ (d+1) = s\ d+1
\end{aligned}$$

A partir daqui alguém derivou a seguinte implementação:

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

Aplicando a lei da recursividade múltipla para $\langle f\ k, l\ k\rangle$ e para $\langle g, s\rangle$ e combinando os resultados com a lei de banana-split, derive as funções $base\ k$ e $loop$ que são usadas como auxiliares acima.

Propriedade QuickCheck 11 *Verificação que* $\binom{n}{k}$ *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:

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

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

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

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":

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

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.

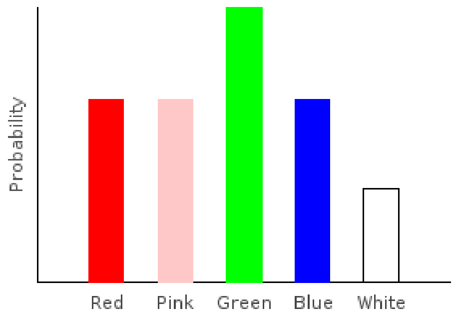


Figura 4: Distribuição de berlindes num saco.

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,

instance $Functor\ Bag$ **where**
  $\mathsf{fmap}\ f = B \cdot \mathsf{map}\ \ (f \times id) \cdot unB$
instance $Monad\ Bag$ **where**
  $x \gg\!\!= f = (\mu \cdot \mathsf{fmap}\ f)\ x$ **where**
  $return = singletonbag$

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

Teste unitário 2 *Lei* $\mu \cdot return = id$*:*

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

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

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

onde $b3$ *é um saco dado em anexo.*

Referências

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

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

Anexos

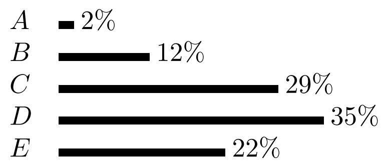
A Mónade para probabilidades e estatística

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca Probability oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

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

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

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



será representada pela distribuição

$$d1 :: \mathsf{Dist}\ Char$$
$$d1 = D\ [(\texttt{'A'}, 0.02), (\texttt{'B'}, 0.12), (\texttt{'C'}, 0.29), (\texttt{'D'}, 0.35), (\texttt{'E'}, 0.22)]$$

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 ⊖ b)
      ominus a b = a ++ neg b
      neg x = [(k, −i) | (k, i) ← x]
```

Ainda sobre o mónade Bag:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Applicative}\ \mathit{Bag}\ \mathbf{where} \\
\quad \mathit{pure} = \mathit{return} \\
\quad (<*>) = \mathit{aap}
\end{array}
$$

O exemplo do texto:

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

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

$$
\begin{array}{l}
\mathit{b3} :: \mathit{Bag}\ (\mathit{Bag}\ (\mathit{Bag}\ \mathit{Marble})) \\
\mathit{b3} = B\ [(B\ [(B\ [(\mathit{Pink}, 2), (\mathit{Green}, 3), (\mathit{Red}, 2), (\mathit{Blue}, 2), (\mathit{White}, 1)], 5) \\
\quad , (B\ [(\mathit{Pink}, 1), (\mathit{Green}, 2), (\mathit{Red}, 1), (\mathit{Blue}, 1)], 2)], 2)]
\end{array}
$$

Outras funções auxiliares:

$$
\begin{array}{l}
a \mapsto b = (a, b) \\
\mathit{consol} :: (\mathit{Eq}\ b) \Rightarrow [(b, \mathit{Int})] \rightarrow [(b, \mathit{Int})] \\
\mathit{consol} = \mathit{filter}\ \mathit{nzero} \cdot \mathsf{map}\ \ (\mathit{id} \times \mathit{sum}) \cdot \mathit{col}\ \mathbf{where}\ \mathit{nzero}\ (\_, x) = x \not\equiv 0 \\
\mathit{isempty} :: \mathit{Eq}\ a \Rightarrow [(a, \mathit{Int})] \rightarrow \mathit{Bool} \\
\mathit{isempty} = \mathit{all}\ (\equiv 0) \cdot \mathsf{map}\ \ \pi_2 \cdot \mathit{consol} \\
\mathit{col}\ x = \mathit{nub}\ [k \mapsto [d' \mid (k', d') \leftarrow x, k' \equiv k] \mid (k, d) \leftarrow x] \\
\mathit{consolidate} :: \mathit{Eq}\ a \Rightarrow \mathit{Bag}\ a \rightarrow \mathit{Bag}\ a \\
\mathit{consolidate} = B \cdot \mathit{consol} \cdot \mathit{unB}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{inBlockchain} = [\mathit{Bc}, \mathit{Bcs}] \\
\mathit{outBlockchain}\ (\mathit{Bc}\ b) = i_1\ (b) \\
\mathit{outBlockchain}\ (\mathit{Bcs}\ (a, x)) = i_2\ (a, x) \\
\mathit{cataBlockchain}\ g = g \cdot \mathit{recBlockchain}\ (\mathit{cataBlockchain}\ g) \cdot \mathit{outBlockchain} \\
\mathit{recBlockchain}\ \ f\ \ = \mathit{id} + \mathit{id} \times f \\
\mathit{anaBlockchain}\ g\ \ = \mathit{inBlockchain} \cdot \mathit{recBlockchain}\ (\mathit{anaBlockchain}\ g) \cdot g \\
\mathit{hyloBlockchain}\ h\ g = \mathit{cataBlockchain}\ h \cdot \mathit{anaBlockchain}\ g
\end{array}
$$

Para a realização da alínea 1, a ideia de resolução passa por retirar a lista de *transações* a cada *Bloco*, sendo que esta é o equivalente a aplicar *p2 . p2* a cada bloco. No caso de encontrar uma *Blockchain*, é necessário concatenar as listas de transações encontradas. Assim, tem-se:

$$\mathit{allTransactions} = \mathit{cataBlockchain}\ [\pi_2 \cdot \pi_2, \mathsf{conc} \cdot ((\pi_2 \cdot \pi_2) \times \mathit{id})]$$

Já nesta alínea, foi necessário comparar entidades, pois, caso elas já estivessem na lista, era, apenas, necessário atualizar o seu valor total das transações. Caso contrário, tinha que ser adicionada à lista, juntamente com o seu valor. Além disso, como uma transação é do tipo *(vendedor, (valor, comprador))*, o comprador terá uma atualização com o valor negativo da transação, pois aquiriu o bem.

O diagrama da Figura 5 demonstra os passos da resolução.

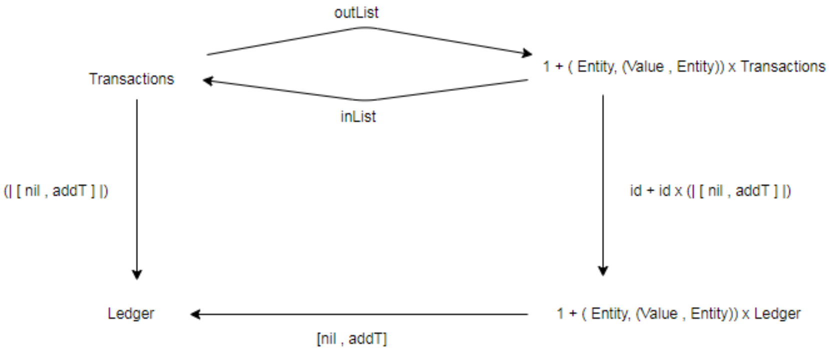


Figura 5: Diagrama de tipos da função Ledger.

$$
\begin{aligned}
&update :: (Entity, Value) \rightarrow Ledger \rightarrow Ledger\\
&update\ a\ [\,] = [a]\\
&update\ (e, v)\ (h : t) \mid e \equiv (\pi_1\ h) = (\pi_1\ h, (\pi_2\ h) + v) : t\\
&\quad \mid otherwise = h : update\ (e, v)\ t\\
&addT :: (Transaction, Ledger) \rightarrow Ledger\\
&addT\ (t, [\,]) = [(\pi_1\ t, \pi_1\ (\pi_2\ t)), (\pi_2\ (\pi_2\ t), -(\pi_1\ (\pi_2\ t)))]\\
&addT\ (t, l) = update\ (\pi_2\ (\pi_2\ t), -(\pi_1\ (\pi_2\ t)))\ (update\ (\pi_1\ t, \pi_1\ (\pi_2\ t))\ l)\\
&ledger = cataList\ [nil, addT] \cdot allTransactions
\end{aligned}
$$

A ideia de resolução passou por gerar uma lista de *(Bc,Bcs)*, em que basicamente se tem um bloco, representada pelo *Bc*, e a zona de comparação pela *Bcs*. Deste modo, é possível verificar se o bloco existe na *Bcs*, com auxílio de uma função tipo *and* sobre os *booleanos* obtidos na verificação, não sendo a *Blockchain* válida, caso aconteça. O diagrama representado na Figura 6 demonstra a linha de pensamento de resolução do problema.

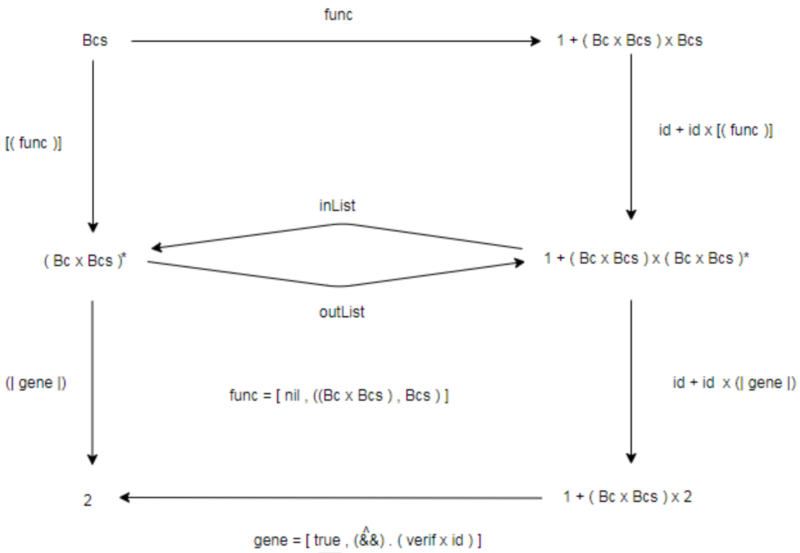


Figura 6: Diagrama de tipos da função isValidMagicNumber.

$$
\begin{aligned}
&isValidMagicNr = hyloList\ [\underline{True}, \widehat{(\wedge)} \cdot (verif \times id)]\ func\\
&verif :: (Block, Blockchain) \rightarrow Bool\\
&verif\ (b, bs) = cataBlockchain\ [comp\ b, \widehat{(\wedge)} \cdot (comp\ b \times id)]\ bs
\end{aligned}
$$

$$
\begin{aligned}
&func :: Blockchain \rightarrow () + ((Block, Blockchain), Blockchain)\\
&func = ((!) + \langle id, \pi_2 \rangle) \cdot outBlockchain\\
&comp :: Block \rightarrow Block \rightarrow Bool\\
&comp\ (a, b)\ (c, d) = a \not\equiv c
\end{aligned}
$$

Problema 2

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

Sabendo que uma rotação de 90º sobre um retângulo, implica que os pontos das suas arestas fiquem alterados sobre uma ordem de rotação (ver Figura 7). Quer isto dizer que no caso de uma *Cell*, as suas coordenadas *x y* são trocadas para *y x* e, no caso de um *Block*, passa de *Block a b c d* para *Block c a d b*, sendo a função de rotação aplicada recursivamente às sub-árvores do mesmo, já pela nova ordem.

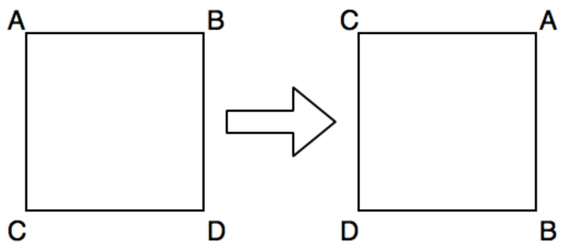


Figura 7: Rotação de 90º, no sentido horário, esquematizada.

$$
\begin{aligned}
&qSwap :: (a, (b, (c, d))) \rightarrow (c, (a, (d, b)))\\
&qSwap = \langle \pi_1 \cdot \pi_2 \cdot \pi_2, \langle \pi_1, \langle \pi_2 \cdot \pi_2 \cdot \pi_2, \pi_1 \cdot \pi_2 \rangle \rangle \rangle\\
&rotateQTree = cataQTree\ (inQTree \cdot (f + qSwap))\\
&\qquad\qquad \textbf{where}\ f = \langle \pi_1, swap \cdot \pi_2 \rangle
\end{aligned}
$$

No caso da função *scaleQTree*, basta multiplicar cada *x e y* de cada *Cell* pelo valor recebido como argumento da referida função. Dito isto, é apenas necessário percorrer todas as *Cells* da árvore, aplicando uma função de multiplicação das dimensões pelo valor desejado.

$$
\begin{aligned}
&scaleQTree\ k = cataQTree\ (inQTree \cdot ((f\ k) + id))\\
&\quad \textbf{where}\ f\ k\ (a, (x, y)) = (a, (x * k, y * k))
\end{aligned}
$$

A função *invertQTree* altera os valores de todos os parâmetros de um *PixelRGBA8*, exceto o *alpha*, sendo que a inversão de *c* passa a *255 - c*. Por conseguinte, é, simplesmente, necessário aplicar uma função (*colorize*) que faça esta alteração a todas as *Cells*

$$
\begin{aligned}
&invertQTree = \mathsf{fmap}\ colorize\\
&\quad \textbf{where}\ colorize\ (PixelRGBA8\ a\ b\ c\ d) = (PixelRGBA8\ (255 - a)\ (255 - b)\ (255 - c)\ d)
\end{aligned}
$$

Como esta função retira um dado número de níveis a uma *Qtree*, o algoritmo pensado para executar este processo passou por executar várias vezes uma função de compressão de um nível, daí a função *compress* ser definida como um ciclo *for*. Chegando ao valor zero, é devolvida a imagem no seu estado atual. É de destacar o comportamento da função g, onde, caso receba uma *Cell*, nada deve ser efetuado, uma vez que não é possível comprimir a mesma; se recebesse um *Block* com quatro *Cells*, estas iriram passar a representar apenas uma única, diminuindo ao número de cores existentes. A escolha do elemento a da célula mencionada foi arbitrária, visto que não é a escolha deste elemento, representante de uma cor, que é relevante para a compressão de um bloco. Para calcular a largura e o comprimento da célula resultante, optou-se pelo método visível na Figura 8, onde $x = x1 + x3$ e $y = y1 + y2$.

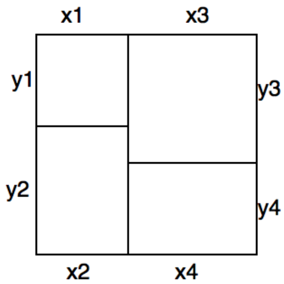


Figura 8: Método de escolha dos valores dos lados da célula.

Se nenhum dos casos anteriormente mencionados se verificarem, a função g é aplicada aos quatro ramos da árvore

$$
\begin{array}{l}
g\ b@(Cell\ a\ x\ y) = b \\
g\ b@(Block\ (Cell\ a\ x1\ y1)\ (Cell\ \_\ x2\ \_)\ (Cell\ \_\ \_\ y3)\ (Cell\ \_\ \_\ \_)) = \mathbf{let}\ x = x1 + x2 \\
\quad y = y1 + y3 \\
\quad \mathbf{in}\ (Cell\ a\ x\ y) \\
g\ h@(Block\ a\ b\ c\ d) = Block\ (g\ a)\ (g\ b)\ (g\ c)\ (g\ d) \\
compressQTree\ k\ y = \mathsf{for}\ g\ y\ k
\end{array}
$$

Para a realização desta função, o algoritmo passou por aplicar a função *f*, recebida no *input*, a todos os elementos da *QuadTree*. Este metodo foi adotado, já que preenche todo o exterior da imagem em si com píxeis negros, retornando uma imagem completamente contornada. No entanto, era possível obter uma imagem muito semelhante à apresentada no enunciado com o uso de casos específicos relativos à matriz de *booleanos* da *QTree*.

$$outlineQTree\ f = qt2bm \cdot \mathsf{fmap}\ f$$

Problema 3

No problema 3, a ideia inicial de resolução passou por obter as versões pointfree das funções dadas. Assim, têm-se as demonstrações seguintes:

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

$$\begin{cases} \textbf{Em cima :}\ (76) \times 2,\ (74)\ ,\ (73) \\ \textbf{Em baixo :}\ succ\ x = x + 1,\ mul\ uncurry\ (\times),\ d + 1 = s\ d,\ (78),\ (74)\ ,\ (73) \end{cases}$$

$$\begin{cases} g\ .\ \underline{0} = \underline{1} \\ g\ .\ succ = mul\ .\ <s\ ,\ g> \end{cases}$$

$\{<s\ ,\ g> = swap\ .\ <g\ ,\ s>,\ (27)\ e\ (20)\}$

$g\ .\ [\underline{0}\ ,\ succ] = [\underline{1}\ ,\ mul\ .\ swap\ .\ <s\ ,\ g>]$

$\{[\underline{0}\ ,\ succ\ ] = in,\ (1),\ (22)\ e\ id + <g\ ,\ s> = F\ <g\ ,\ s>\}$

$g\ .\ in = [\underline{1}\ ,\ mul\ .\ swap]\ .\ F < g\ ,\ s>$

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

$$\begin{cases} \textbf{Em cima:}\ (76) \times 2\ ,\ (74)\ e\ (73) \\ \textbf{Em baixo:}\ succ\ x = x + 1 \end{cases}$$

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

$\{s = \pi 2\ .\ <s\ ,\ d>,\ (27)\ ,\ (20)\ e\ [\underline{0}, succ] = in\}$

$s\ .\ in = [\underline{1}\ ,\ succ\ .\ \pi 2\ .\ <s\ ,\ d>]$

$\{(1),\ (22)\ e\ id + <g,\ s> = F\ <g\ ,\ s>\}$

$s\ .\ in = [\underline{1}\ ,\ succ\ .\ \pi 2]\ .\ F < g\ ,\ s>$

f k 0 =1

f k (d+1) = (d+k+1) x f k d

Em cima: (76) x 2 , 74 e 73

Em baixo: succ x = x +1 , mul = uncurry (x),d+k+1 = l k d,(78), (74)e (73)

(f k) 0 = 1

(f k) succ = mul . <l k , f k >

{ (27) e (20) }

(f k) . [0 ,succ] = [1, mul . <l k , f k>]

{ <l k , f k> = swap . <f k, l k>}

(f k) . [0 , succ] = [1 , mul . swap . <f k, l k>]

{ (1) e (22)}

(f k) . [0 , succ] = [1 , mul . swap] . (id + <f k ,l k>)

{id + <f k , l k > = F < f k , l k >} e { [0 , succ] = in}

(f k) . in = [1 , mul . swap] . F <f k, l k>

l k 0 = k +1

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

Em cima: (76) x 2, succ x = x + 1, (74) , (73)

Em baixo: (succ x = x + 1), (74), (73)

(l k) . 0 = succ k

(l k) . succ = succ (l k)

{(27) e (20)}

(l k) .[0 , succ] = [succ k , succ . (l k)]

{l k = π2 . <f k, l k> (7)}

(l k) . [0 , succ] = [succ k , succ . π2 . <f k, l k>]

{(1) e (22)}

(l k). [0 , succ] = [succ k, succ . π2] (id + <f k, l k>)

{id + <f k, l k> = F <f k, l k> e [0, succ] = in}

(l k) . in = [succ k , succ . π2] . (F <f k , l k>)

Pela regra *Fokkinga*, aplicada aos pares de funções f e l, e g com s, tem-se:

Pela regra 50 (Fokkinga)

<g , s> = (| < [1 , mul . swap] , [1, succ . π2] > |)

{(50) Fokkinga}

<f k , l k> = (| < [1, mul . swap] , [succ k, succ . π2]>|)

Sendo assim, utilizando a lei de *Banana-Split*, obtém-se:

{Banana-split (51)}

<<f k, l k> , <g , s>>

{Resultado dos Fokkingas}

< (| < [1 , mul . swap] , [succ k , succ . π2]> |) , (| < [1 , mul . swap], [1 , succ . π2] > |) >

{(51) e F π1 = id + π1 e F π2 = id + π2}

(| (< [1 , mul . swap] , [succ k , succ . π2]> x <[1 , mul . swap], [1 , succ .π2]>) . <(id +π1) ,(id + π2)> |)

{(11)}

(| < < [1 , mul . swap] , [succ k , succ π2]> .(id +π1), <[1 , mul . swap],[1 , succ .π2]> . (id +π2)>|)

{(9) x 2, (22) x 4, (3) x 2}

(| <<[1 , mul . swap . π1], [succ k , succ . π2 . π1]>, < [1, mul . swap .π2],[1 , succ . π2 . π2]>>|)

{(28) x 2}

(| < [<1 , succ k> , <mul . swap .π1, succ .π2 .π1 >],[<1,1>,<mul . swap .π2, succ . π2 . π2>] > |)

{(28) e <a , b> = (a , b) x 3}

(| ((1 , succ k) , (1 , 1)), << mul . swap . π1, succ . π2 . π1> , <mul . swap . π2 , succ . π2 . π2>>] |)

{ for b i = (| [i , b] |) e (9) x 2}

for << mul . swap , succ .π2 > . π1, <mul . swap , succ . π2> . π2 > ((1 , succ k) , (1, 1))

Como se tem *for loop (base k) n*, então:

$$
\begin{aligned}
&parentisa\ ((a,b),(c,d)) = (a,b,c,d)\\
&desparentisa\ (a,b,c,d) = ((a,b),(c,d))\\
&base\ \ k = parentisa\ ((1, \mathsf{succ}\ \ k),(1,1))\\
&loop = parentisa \cdot \langle\langle mul \cdot swap, \mathsf{succ}\ \cdot \pi_2\rangle \cdot \pi_1, \langle mul \cdot swap, \mathsf{succ}\ \cdot \pi_2\rangle \cdot \pi_2\rangle \cdot desparentisa
\end{aligned}
$$

Problema 4

$$
\begin{aligned}
&inFTree = [Unit, (\lambda a \rightarrow \widehat{(\widehat{Comp}\ a)})]\\
&outFTree\ (Unit\ c) = i_1\ c\\
&outFTree\ (Comp\ a\ t1\ t2) = i_2\ (a,(t1,t2))\\
&cataFTree\ a = a \cdot (recFTree\ (cataFTree\ a)) \cdot outFTree\\
&anaFTree\ f = inFTree \cdot (recFTree\ (anaFTree\ f)) \cdot f\\
&hyloFTree\ a\ c = cataFTree\ a \cdot anaFTree\ c\\
&baseFTree\ f\ g\ h = g + (f \times (h \times h))\\
&recFTree\ f = baseFTree\ id\ id\ f\\
&\textbf{instance}\ BiFunctor\ FTree\\
&\quad \textbf{where}\ bmap\ f\ g = cataFTree\ (inFTree \cdot baseFTree\ f\ g\ id)
\end{aligned}
$$

De modo a gerar uma *PTree* a partir de um determinado número de níveis, é necessário, antes de mais tem que ser definido o valor do lado do quadrado de nível zero, daí o uso de uma função auxiliar que tenha como parâmetro o mesmo. A partir do diagrama na Figura 9, é possível perceber o raciocíneo para aplicar a equação de formação dos nodos da árvore, sendo esta $c \times (\frac{\sqrt{2}}{2})^n$, onde c o lado do nível zero e n o nível em questão. Assim, depois de gerar um nível, caso ainda existam níveis por criar, tem que ser aplicada a função aos nodos filhos.

$$
\begin{tikzcd}
\mathit{Int} \arrow[rr, "h"] \arrow[d, "[\!(h)\!]"'] & & \mathit{Float} + (\mathit{Float} \times (\mathit{Int} \times \mathit{Int})) \arrow[d, "id + (id \times ([\!(h)\!] \times [\!(h)\!]))"] \\
\mathit{PTree} \arrow[rr, bend left, "outPtree"] & & \mathit{Float} + (\mathit{Float} \times (\mathit{PTree} \times \mathit{PTree})) \arrow[ll, bend left, "inPTree"]
\end{tikzcd}
$$

Figura 9: Diagrama para gerar uma *PTree*.

$$
\begin{aligned}
& generateAux\ x0\ n = anaFTree\ (h \cdot outNat) \\
& \quad \textbf{where}\ h = \underline{(lado\ 0)} + \langle lado \cdot \mathsf{succ}\ , \langle id, id \rangle \rangle \\
& \qquad lado = tamanho \cdot (n-) \\
& \qquad tamanho = (x0 *) \cdot (((sqrt\ 2)\ /\ 2) \uparrow) \\
& generatePTree\ n = generateAux\ 100\ n\ n
\end{aligned}
$$

De modo a desenhar uma árvore de Pitágoras com *n* níveis, consoante os pretendidos, foi necessário descobrir o algoritmo relativo às alterações nas coordenadas e nos ângulos, de cada nodo "pai"para nodos "filhos". Como este problema envolvia o uso da biblioteca *Gloss*, foi ainda necessário implementar o algoritmo acima mencionado com funções da mesma. Pelas imagens destas árvores, é possível ver que ambos os "filhos"fazem um ângulo de 45 graus com a aresta do pai, visível na Figura 10, onde *alpha* e *beta* representam o ângulo mencionado. Estes têm que possuir, ambos, este valor para a árvore ser simétrica.

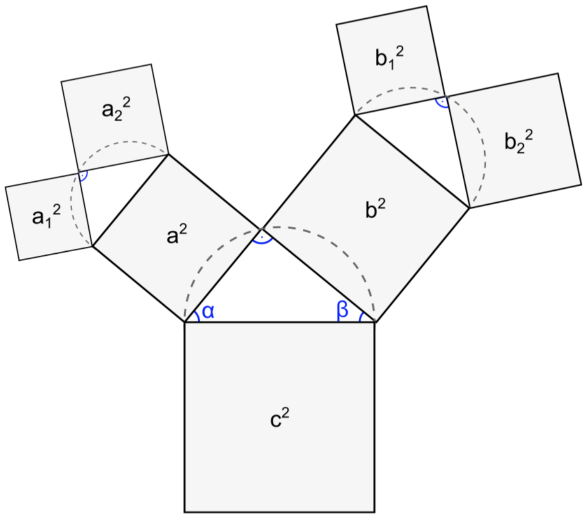


Figura 10: Parcela de uma *Árvore de Pitágoras*.

$$
\begin{aligned}
& drawPTree\ p = aux\ p\ (0,0)\ 0\ \textbf{where} \\
& \quad aux :: PTree \rightarrow (Float, Float) \rightarrow Float \rightarrow [Picture] \\
& \quad aux\ (Unit\ c)\ (x,y)\ ang = [Translate\ x\ y\ (Rotate\ ang\ (square\ c))] \\
& \quad aux\ (Comp\ c\ l\ r)\ (x,y)\ ang = [Translate\ x\ y\ (Rotate\ ang\ (square\ c))] + \!\! + (aux\ l\ (x + somaXLeft, y + somaYL \\
& \qquad \textbf{where}\ somaX = c\ /\ 2 \\
& \qquad\quad somaY = c \\
& \qquad\quad angRads = ang * pi\ /\ 180 \\
& \qquad\quad branchToGlobal\ angle\ (dx, dy) = (dx * cos\ angle + dy * sin\ angle, dy * cos\ angle - dx * sin\ angle) \\
& \qquad\quad (somaXLeft, somaYLeft) = branchToGlobal\ angRads\ (-somaX, somaY) \\
& \qquad\quad (somaXRight, somaYRight) = branchToGlobal\ angRads\ (somaX, somaY)
\end{aligned}
$$

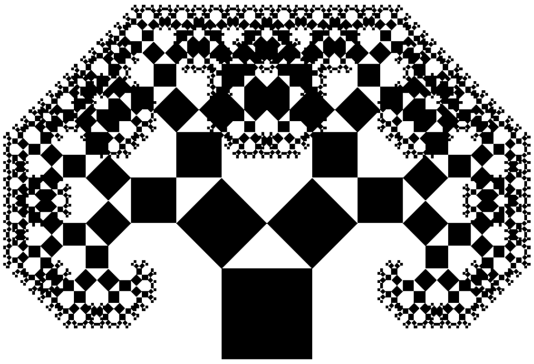


Figura 11: Exemplo de execução da função para $n = 10$.

Problema 5

Como a função *singletonbag* é do tipo *singletonbag :: a -¿ Bag a*, tem que ser criado um par, que possua o valor recebido e a quantidade um. Para isto, é formado um *split* de *id* (parâmetro de *input*) e *(const 1)*, ficando algo do tipo (a,1). Em seguida, este tem que ser colocado numa lista, que possui apenas um elemento, ele mesmo, daí o uso de *singl*. Por fim, tem que ser gerado o *Bag* com esta lista.

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

Olhando para os tipos das funções que eram necessárias de definir, é notório que o *muB* vai de um *Bag (Bag a)* para um *Bag a*. Como um *Bag a* é do tipo *B [(a, Int)]*, um *Bag (Bag a)* será *B [(B a, Int)]*, ou seja, *B [(B [(a,Int)], Int)]*. Sendo assim, é necessário utilizar a propriedade distributiva da multiplicação do segundo valor no par do *Bag* mais externo, por todos os elementos da lista lá existente (*função ff após getList*). Além disso, como existem várias listas, que têm de ser concatenadas numa só, é aplicado um *foldr (++)* para que se obtenha a lista final pretendida. Por fim, basta aplicar a formação de um *Bag* à lista já obtida. Os passos de obtenção de *muB* podem ser, facilmente, observados na Figura 12:

$$\text{Bag (Bag a)} \xrightarrow[\text{getList}]{} \text{[(Bag a, Int)]} \xrightarrow[\text{fmap (getList} \times \text{id)}]{} \text{[([(a, Int)], Int)]} \xrightarrow[\text{fmap ff}]{} \text{[[(a, Int)]]} \xrightarrow[\text{foldr(++)}]{} \text{[(a, Int)]} \xrightarrow[\text{B}]{} \text{Bag a}$$

Figura 12: Passos de obtenção da função muB.

$$
\begin{aligned}
&getList :: Bag\ a \rightarrow [(a, Int)] \\
&getList\ (B\ l) = l \\
&ff :: ([(a, Int)], Int) \rightarrow ([(a, Int)]) \\
&ff\ ([\,], a) = [\,] \\
&ff\ (((a, b) : t), c) = (a, b * c) : ff\ (t, c) \\
&\mu = B \cdot foldr\ (+\!\!+)\ [\,] \cdot \mathsf{fmap}\ ff \cdot \mathsf{fmap}\ (getList \times id) \cdot getList
\end{aligned}
$$

Finalmente, para realizar a função *dist*, também foi observado o seu tipo para ajudar no processo, tendo-se *dist :: Bag Marble -¿ Dist Marble*. Deste modo, como cada par de *Marble* possui o número de ocorrências no saco associado a si, este tem de ser convertido à sua probabilidade de ocorrência. Para isto, basta dividir este parâmetro de número de ocorrências pelo núemro total de berlindes no saco, dado pela função *size*. Tendo já uma lista de pares de *Marble* e *ProbRep*, é necessário criar um objeto do tipo *Dist* e aplicar-lhe a função *norm*, definida no módulo *Probability.hs*.

$$
\begin{aligned}
&probsB :: Eq\ a \Rightarrow [(a, Int)] \rightarrow [(a, ProbRep)] \\
&probsB\ l = \mathsf{fmap}\ f\ l \\
&\quad \mathbf{where}\ f\ (a, b) = (a, (fromIntegral\ b)\ /\ (fromIntegral\ m))
\end{aligned}
$$

$$
\begin{aligned}
&\quad m = \mathit{size}\ l \\
&\mathit{size} :: [(a, \mathit{Int})] \rightarrow \mathit{Int} \\
&\mathit{size} = \mathit{cataList}\ [\underline{0}, (\widehat{(+) \cdot \pi_2})] \\
&\mathit{dist} = \mathit{norm} \cdot D \cdot \mathit{probsB} \cdot \mathit{getList}
\end{aligned}
$$

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!|g|\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!|g|\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}
$$

[7] Exemplos tirados de [2].

Índice