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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	74
a61855	Ana Paula Carvalho
aXXXXX	Eduardo Gil
a73831	João Pires Barreira

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.

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

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

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

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.

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

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

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

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 $allTransactions :: Blockchain \rightarrow Transactions$, 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 $ledger :: Blockchain \rightarrow Ledger$, 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 $isValidMagicNr :: Blockchain \rightarrow Bool$, 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 )      Block
( 0 0 0 0 0 0 0 0 )       (Cell 0 4 4) (Block
( 0 0 0 0 1 1 1 0 )        (Cell 0 2 2) (Cell 0 2 2) (Cell 1 2 2) (Block
( 0 0 0 0 1 1 0 0 )         (Cell 1 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1)))
( 1 1 1 1 1 1 0 0 )       (Cell 1 4 4)
( 1 1 1 1 1 1 0 0 )       (Block
( 1 1 1 1 0 0 0 0 )        (Cell 1 2 2) (Cell 0 2 2) (Cell 0 2 2) (Block
( 1 1 1 1 0 0 0 1 )         (Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 1 1 1)))
```

(a) Matriz de exemplo bm.

(b) Quadtree de exemplo qt.

Figura 1: Exemplos de representações de bitmaps.

Uma imagem monocromática em formato bitmap pode ser representada como uma matriz de 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:

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

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

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.

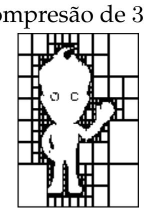
(d) Inversão de cores.

(e) Compresão de 1 nível.

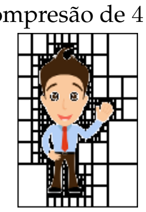
(f) Compresão de 2 níveis.

(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 $compressQTree :: Int \rightarrow QTree\ a \rightarrow QTree\ a$, 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 $outlineQTree :: (a \rightarrow Bool) \rightarrow QTree\ a \rightarrow Matrix\ Bool$, 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,

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n!}{k! * (n-k)!} \tag{1}$$

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,

$$\left(\begin{array}{c} n \\ k \end{array}\right) = h\ k\ (n-k)$$

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

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

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:

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

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:

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

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.

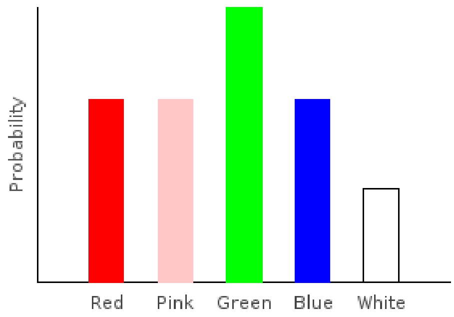


Figura 4: Distribuição de berlindes num saco.

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

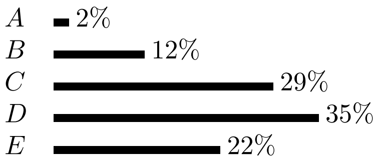
A Mónade para probabilidades e estatística

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

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

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

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

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*:

$\textbf{instance } \mathit{Applicative\ Bag} \textbf{ where}$
$\quad \mathit{pure} = \mathit{return}$
$\quad (<*>) = \mathit{aap}$

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):

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

Outras funções auxiliares:

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

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

Inicialmente chegou-se à codificação (apresentada abaixo) das funções para o tipo *Blockchain*, que são muito parecidas às que já conhecemos para as listas.

$$\begin{aligned}
&\mathit{inBlockchain} = [\mathit{Bc}, \mathit{Bcs}]\\
&\mathit{outBlockchain}\ (\mathit{Bc}\ b) = i_1\ b\\
&\mathit{outBlockchain}\ (\mathit{Bcs}\ (b, \mathit{bc})) = i_2\ (b, \mathit{bc})\\
&\mathit{recBlockchain}\ g = \mathit{id} + (\mathit{id} \times g)\\
&\mathit{cataBlockchain}\ g = g \cdot (\mathit{recBlockchain}\ (\mathit{cataBlockchain}\ g)) \cdot \mathit{outBlockchain}\\
&\mathit{anaBlockchain}\ g = \mathit{inBlockchain} \cdot (\mathit{recBlockchain}\ (\mathit{anaBlockchain}\ g)) \cdot g\\
&\mathit{hyloBlockchain}\ f\ g = \mathit{cataBlockchain}\ f \cdot \mathit{anaBlockchain}\ g
\end{aligned}$$

1. allTransactions

A função ***allTransactions*** foi calculada utilizando um catamorfismo que obedece ao seguinte esquema:

$$\begin{tikzcd}
\mathit{Blockchain} \arrow[r, "\mathit{outBlockchain}"'] \arrow[d, "\mathit{allTransactions} = (\!|g|\!)"'] & \mathit{Block} + \mathit{Block} \times \mathit{Blockchain} \arrow[d, "\mathit{id} + \mathit{id} \times \mathit{allTransactions}"] \\
\mathit{Transactions} & \mathit{Block} + \mathit{Block} \times \mathit{Transactions} \arrow[l, "g = [g1, g2]"]
\end{tikzcd}$$

Em que as funções *g1* e *g2* que compõem o gene *g* são dadas, respetivamente, por:

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

$$Block \times Transactions \xrightarrow[g1 \times id = \pi_2 \cdot \pi_2 \times id]{} Transactions \times Transactions \xrightarrow[\mathsf{conc}]{} Transactions$$

Assim, pudemos chegar à codificação da função *allTransactions*, correspondente ao catamorfismo apresentado acima.

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

2. ledger

A função ***ledger*** foi calculada utilizando um catamorfismo que obedece ao seguinte esquema:

$$\begin{tikzcd}
Blockchain \arrow[r, "outBlockchain"'] \arrow[d, "ledger = (\!|g|\!)"'] & Block + Block \times Blockchain \arrow[d, "id + id \times ledger"] \\
Ledger & Block + Block \times Ledger \arrow[l, "g = [g1, g2]"]
\end{tikzcd}$$

Em que as funções *g1* e *g2* que compõem o gene *g* são dadas, respetivamente, por:

$$\begin{tikzcd}
Block = MagicNo \times (Time \times Transactions) \arrow[r, "\pi_2 \cdot \pi_2"'] & Transactions = ((Entity, (Value, Entity)))* \arrow[d, "\mathsf{map}\ (swap \cdot \pi_2)"'] \\
 & (Entity, Value)* = Ledger
\end{tikzcd}$$

$$\begin{tikzcd}
Block \times Ledger \arrow[r, "g1 \times id = (\mathsf{map}\ (swap \cdot \pi_2) \cdot \pi_2 \cdot \pi_2) \times id"'] & Ledger \times Ledger = (Entity, Value) * \times (Entity, Value)* \arrow[d, "\mathsf{conc}"'] \\
 & (Entity, Value)* = Ledger
\end{tikzcd}$$

Assim, pudemos chegar à codificação da função *ledger*, correspondente ao catamorfismo apresentado acima.

$$ledger = cataBlockchain\ [\mathsf{map}\ (swap \cdot \pi_2) \cdot \pi_2 \cdot \pi_2, \mathsf{conc} \cdot ((\mathsf{map}\ (swap \cdot \pi_2) \cdot \pi_2 \cdot \pi_2) \times id)]$$

3. isValidMagicNr

Para codificar a função ***isValidMagicNr*** utilizámos uma função auxiliar, *magicNrList*, que lista todos os números mágicos de uma *Blockchain*, com recurso a um catamorfismo que obedece ao seguinte esquema:

$$\begin{tikzcd}
Blockchain \arrow[r, "outBlockchain"'] \arrow[d, "magicNrList = (\!|g|\!)"'] & Block + Block \times Blockchain \arrow[d, "id + id \times magicNrList"] \\
(MagicNo)* & Block + Block \times (MagicNo)* \arrow[l, "g = [g1, g2]"]
\end{tikzcd}$$

Em que as funções *g1* e *g2* que compõem o gene *g* são dadas, respetivamente, por:

$$Block = MagicNo \times (Time \times Transactions) \xrightarrow[\pi_1]{} MagicNo \xrightarrow[singl]{} (MagicNo)*$$

$$Block \times (MagicNo)* \xrightarrow[singl \cdot \pi_1 \times id]{} (MagicNo)* \times (MagicNo)* \xrightarrow[\mathsf{conc}]{} (MagicNo)*$$

Finalmente, conseguimos chegar ao código da função *isValidMagicNr* que queríamos inicialmente, aplicando uma função auxiliar *semRepetidos* (que apenas verifica se uma lista tem elementos repetidos) ao resultado do catamorfismo *magicNrList* apresentado anteriormente.

$$\begin{array}{l}
isValidMagicNr = semRepetidos \cdot (cataBlockchain\ [singl \cdot \pi_1, \mathsf{conc} \cdot (singl \cdot \pi_1 \times id)]) \\
\quad \mathbf{where}\ semRepetidos\ [\,] = True \\
\qquad semRepetidos\ [a] = True \\
\qquad semRepetidos\ (h : t) = \mathbf{if}\ (elem\ h\ t \equiv True) \\
\qquad\quad \mathbf{then}\ False \\
\qquad\quad \mathbf{else}\ semRepetidos\ t
\end{array}$$

Problema 2

Inicialmente chegou-se à codificação (apresentada abaixo) das funções para o tipo *QTree*, que são muito parecidas às que já conhecemos para as *LTree* (mas com quatro subárvores em vez de apenas duas).

$$\begin{array}{l}
inQTree = [cellAux\ Cell, blockAux\ Block] \\
\quad \mathbf{where}\ cellAux\ f\ (x, (y, z)) = f\ x\ y\ z \\
\qquad blockAux\ g\ (a, (b, (c, d))) = g\ a\ b\ c\ d \\
outQTree\ (Cell\ a\ x\ y) = i_1\ ((a, (x, y))) \\
outQTree\ (Block\ a\ b\ c\ d) = i_2\ ((a, (b, (c, d)))) \\
baseQTree\ g\ f = g \times id + (f \times (f \times (f \times f))) \\
recQTree\ g = id + (g \times (g \times (g \times g))) \\
cataQTree\ g = g \cdot (recQTree\ (cataQTree\ g)) \cdot outQTree \\
anaQTree\ g = inQTree \cdot (recQTree\ (anaQTree\ g)) \cdot g \\
hyloQTree\ f\ g = cataQTree\ f \cdot anaQTree\ g
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ Functor\ QTree\ \mathbf{where} \\
\quad \mathsf{fmap}\ f = cataQTree\ (inQTree \cdot baseQTree\ f\ id)
\end{array}$$

1.1. rotateQTree

A função ***rotateQTree*** consiste em rodar uma *QTree* 90 graus no sentido dos ponteiros do relógio. Esta função foi codificada recorrendo a um catamorfismo que obedece ao seguinte esquema:

$$\begin{tikzcd}
QTree\ A \arrow[r, "outQTree"'] \arrow[d, "rotateQTree = (\!|g|\!)"'] & A + (QTree\ A)^4 \arrow[d, "id + rotateQTree^4"] \\
QTree\ A & A + (QTree\ A)^4 \arrow[l, "g = inQTree \cdot (g1 + g2)"]
\end{tikzcd}$$

Uma *QTree* ou é uma *Cell* ou um *Block*. Assim sendo, para o caso da *Cell* (correspondente ao *g1* da função gene *g*), a única coisa que temos de fazer é inverter as dimensões da submatriz de bits gerada. Por exemplo, se tivermos uma *Cell 1 3 4*, esta irá passar a corresponder a *Cell 1 4 3* após a rotação. Deste modo, *g1* pode ser obtida pelo seguinte esquema:

$$A = Cell\ \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \xrightarrow[id \times swap]{} Cell\ \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 = A$$

Para o caso do *Block* que já contém o resultado recursivo da aplicação desta função, a sua componente *g2* da função gene *g*, corresponde a rodar as suas quatro submatrizes (correspondentes às suas quatro *QTree*) uma posição para direita, de acordo com o seguinte esquema:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} ===> \begin{bmatrix} C & A \\ D & B \end{bmatrix}$$

Esta situação corresponderá a trocar as *QTree* (A, B, C e D) de um *Block* da mesma forma. Ou seja, *Block A B C D* passará a ser *Block C A D B*.

Em último lugar, é preciso ainda juntar estes dois resultados, por forma a ter o tipo final pretendido (*QTree*), o que será facilmente conseguido através da função *inQTree*.

Finalmente, pode-se obter a codificação da função *rotateQTree*, em que *g2* é a função auxiliar que se encarrega da situação descrita acima.

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

1.2. scaleQTree

A função ***scaleQTree*** consiste em aumentar as dimensões de uma *QTree* segundo um escalar dado. Esta função foi codificada recorrendo a um catamorfismo que obedece ao seguinte esquema:

$$\begin{tikzcd} QTree\ A \arrow[r, "outQTree"'] \arrow[d, "scaleQTree = (\![g]\!)"'] & A + (QTree\ A)^4 \arrow[d, "id + scaleQTree^4"] \\ QTree\ A & A + (QTree\ A)^4 \arrow[l, "g = inQTree \cdot (g1 + g2)"] \end{tikzcd}$$

Como já foi dito anteriormente, uma *QTree* ou é uma *Cell* ou um *Block*. Assim sendo, para o caso da *Cell* (correspondente ao *g1* da função gene *g*), a única coisa que temos de fazer é multiplicar os inteiros referentes às dimensões da submatriz de bits gerada. Deste modo, uma *Cell 1 3 4* passará a corresponder a *Cell 1 6 8* após uma escala de fator 2. Assim sendo, a função responsável pela escala de uma *Cell* pode ser facilmente codificada como *g1 (a,(x,y)) = (a, (x*k, y*k))*.

Para o caso do *Block* correspondente às quatro *QTree* do esquema acima – e que contém o resultado recursivo da aplicação desta função às subárvores –, não é preciso fazer mais nada, pelo que o *g2* corresponde apenas a um id.

Em último lugar, é preciso ainda juntar estes dois resultados, por forma a ter o tipo final pretendido (*QTree*), o que será facilmente conseguido através da função *inQTree*.

Finalmente, pode-se obter a codificação da função *scaleQTree*:

$$\begin{aligned} &scaleQTree\ k = cataQTree\ (inQTree \cdot (g1 + id)) \\ &\quad \textbf{where}\ g1\ (a, (x, y)) = (a, (x * k, y * k)) \end{aligned}$$

1.3. invertQTree

A função ***invertQTree*** consiste em inverter as cores de uma *QTree*. Esta função foi codificada recorrendo a um catamorfismo que obedece ao seguinte esquema:

$$\begin{tikzcd} QTree\ A \arrow[r, "outQTree"'] \arrow[d, "invertQTree = (\![g]\!)"'] & A + (QTree\ A)^4 \arrow[d, "id + invertQTree^4"] \\ QTree\ A & A + (QTree\ A)^4 \arrow[l, "g = inQTree \cdot (g1 + g2)"] \end{tikzcd}$$

Como já foi dito anteriormente, uma *QTree* ou é uma *Cell* ou um *Block*. Assim sendo, para o caso da *Cell* (correspondente ao *g1* da função gene *g*), a única coisa que temos de fazer é inverter as cores da primeira componente da *Cell* (do tipo *PixelRGBA8*), o que corresponde a subtrair o valor de cada um dos campos do RGB do pixel ao valor 255. Para isso, utilizámos a seguinte função auxiliar:

$$\begin{array}{l} invertColor :: PixelRGBA8 \rightarrow PixelRGBA8 \\ invertColor\ (PixelRGBA8\ r\ g\ b\ a) = PixelRGBA8\ (255 - r)\ (255 - g)\ (255 - b)\ a \end{array}$$

Assim sendo, a função *g1* pode ser codificada à custa da aplicação desta função auxiliar para o primeiro componente da *Cell* (e deixando intactos os restantes dois componentes). Logo, a função pode ser dada por: *g1 (p,(x,y)) = (invertColor p, (x,y))*.

Para o caso do *Block* correspondente às quatro *QTree* do esquema acima – e que contém o resultado recursivo da aplicação desta função às subárvores –, não é preciso fazer mais nada, pelo que o *g2* corresponde apenas a um id.

Em último lugar, é preciso ainda juntar estes dois resultados, por forma a ter o tipo final pretendido (*QTree*), o que será facilmente conseguido através da função *inQTree*.

Finalmente, pode-se obter a codificação da função *invertQTree*:

$$\begin{array}{l} invertQTree = cataQTree\ (inQTree \cdot (g1 + id)) \\ \quad \mathbf{where}\ g1\ (p, (x, y)) = (invertColor\ p, (x, y)) \end{array}$$

2. compressQTree

$$compressQTree = \bot$$

3. outlineQTree

$$\begin{array}{l} outlineQTree\ p = qt2bm \cdot cataQTree\ (inQTree \cdot (aux + (id \times (id \times (id \times id))))) \\ \quad \mathbf{where}\ aux\ (a, (x, y)) = \mathbf{if}\ (p\ a) \equiv True \\ \quad\quad \mathbf{then}\ (True, (x, y)) \\ \quad\quad \mathbf{else}\ (False, (x, y)) \end{array}$$

Problema 3

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

$$\begin{array}{ll} & \langle f\ k, l\ k \rangle \\ \equiv & \quad \{\ \text{Lei de Fokkinga (50)} \times 2\ \} \\ & \begin{cases} f\ k \cdot \mathsf{in} = u \cdot (id + \langle f\ k, l\ k \rangle) \\ l\ k \cdot \mathsf{in} = v \cdot (id + \langle f\ k, l\ k \rangle) \end{cases} \\ \equiv & \quad \{\ \text{Def-+ (21)} \times 2\text{, Natural-id (1)} \times 2\ \} \\ & \begin{cases} f\ k \cdot \mathsf{in} = u \cdot [i_1, i_2 \cdot \langle f\ k, l\ k \rangle] \\ l\ k \cdot \mathsf{in} = v \cdot [i_1, i_2 \cdot \langle f\ k, l\ k \rangle] \end{cases} \\ \equiv & \quad \{\ \text{Fusão-+ (20)} \times 2\ \} \\ & \begin{cases} f\ k \cdot \mathsf{in} = [u \cdot i_1, u \cdot i_2 \cdot \langle f\ k, l\ k \rangle] \\ l\ k \cdot \mathsf{in} = [v \cdot i_1, v \cdot i_2 \cdot \langle f\ k, l\ k \rangle] \end{cases} \\ \equiv & \quad \{\ \text{Def-in}\mathbb{N}_0 \times 2\text{, Fusão-+ (20)} \times 2\ \} \\ & \begin{cases} [f\ k \cdot \underline{0}, f\ k \cdot \mathsf{succ}\,] = [u \cdot i_1, u \cdot i_2 \cdot \langle f\ k, l\ k \rangle] \\ [l\ k \cdot \underline{0}, l\ k \cdot \mathsf{succ}\,] = [v \cdot i_1, v \cdot i_2 \cdot \langle f\ k, l\ k \rangle] \end{cases} \\ \equiv & \quad \{\ \text{Eq-+ (27)} \times 2\text{, explicitando } u \text{ e } v \text{ como } [u1, u2] \text{ e } [v1, v2]\ \} \end{array}$$

$$\left\{\begin{array}{l} f\ k \cdot \underline{0} = [u1, u2] \cdot i_1 \\ f\ k \cdot \mathsf{succ} = [u1, u2] \cdot i_2 \cdot \langle f\ k, l\ k\rangle \end{array}\right. \left\{\begin{array}{l} l\ k \cdot \underline{0} = [v1, v2] \cdot i_1 \\ l\ k \cdot \mathsf{succ} = [v1, v2] \cdot i_2 \cdot \langle f\ k, l\ k\rangle \end{array}\right.$$

$\equiv$ { Cancelamento-+ (18) $\times 4$ }

$$\left\{\begin{array}{l} f\ k \cdot \underline{0} = u1 \\ f\ k \cdot \mathsf{succ} = u2 \cdot \langle f\ k, l\ k\rangle \end{array}\right. \left\{\begin{array}{l} l\ k \cdot \underline{0} = v1 \\ l\ k \cdot \mathsf{succ} = v2 \cdot \langle f\ k, l\ k\rangle \end{array}\right.$$

$\equiv$ { Igualdade extensional (73) $\times 4$, Def-comp (74) $\times ?$, Natural-const (3) $\times 2$, Def-succ $\times 2$, Def-split (78) $\times 2$ }

$$\left\{\begin{array}{l} f\ k\ 0 = u1\ \ d \\ f\ k\ (d+1) = u2 \cdot \langle f\ k\ d, l\ k\ d\rangle \end{array}\right. \left\{\begin{array}{l} l\ k\ 0 = v1\ \ d \\ l\ k\ (d+1) = v2 \cdot \langle f\ k\ d, l\ k\ d\rangle \end{array}\right.$$

Analisando os resultados obtidos e as definições *pointwise* das funções $f\,k$ e $l\,k$ presentes no enunciado, podemos deduzir quais as funções correspondentes a $u1$, $u2$, $v1$ e $v2$ na solução apresentada acima:

$$\left\{\begin{array}{l} u1 = \underline{1} \\ u2 = mul \end{array}\right. \left\{\begin{array}{l} v1 = \mathsf{succ} \\ v2 = \mathsf{succ} \cdot \pi_2 \end{array}\right.$$

Logo $\langle f\ k, l\ k\rangle = (\!|\langle [u1, u2], [v1, v2]\rangle|\!) = (\!|\langle [\underline{1}, mul], [\mathsf{succ}, \mathsf{succ} \cdot \pi_2]\rangle|\!)$.

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

$\langle g, s\rangle$

$\equiv$ { Lei de Fokkinga (50) $\times 2$ }

$$\left\{\begin{array}{l} g \cdot \mathsf{in} = u \cdot (id + \langle g, s\rangle) \\ s \cdot \mathsf{in} = v \cdot (id + \langle g, s\rangle) \end{array}\right.$$

$\equiv$ { Def-+ (21) $\times 2$, Natural-id (1) $\times 2$ }

$$\left\{\begin{array}{l} g \cdot \mathsf{in} = u \cdot [i_1, i_2 \cdot \langle g, s\rangle] \\ s \cdot \mathsf{in} = v \cdot [i_1, i_2 \cdot \langle g, s\rangle] \end{array}\right.$$

$\equiv$ { Fusão-+ (20) $\times 2$ }

$$\left\{\begin{array}{l} g \cdot \mathsf{in} = [u \cdot i_1, u \cdot i_2 \cdot \langle g, s\rangle] \\ s \cdot \mathsf{in} = [v \cdot i_1, v \cdot i_2 \cdot \langle g, s\rangle] \end{array}\right.$$

$\equiv$ { Def-in$\mathbb{N}_0$ $\times 2$, Fusão-+ (20) $\times 2$ }

$$\left\{\begin{array}{l} [g \cdot \underline{0}, g \cdot \mathsf{succ}] = [u \cdot i_1, u \cdot i_2 \cdot \langle g, s\rangle] \\ [s \cdot \underline{0}, s \cdot \mathsf{succ}] = [v \cdot i_1, v \cdot i_2 \cdot \langle g, s\rangle] \end{array}\right.$$

$\equiv$ { Eq-+ (27) $\times 2$, explicitando u e v como $[u1, u2]$ e $[v1, v2]$ }

$$\left\{\begin{array}{l} g \cdot \underline{0} = [u1, u2] \cdot i_1 \\ g \cdot \mathsf{succ} = [u1, u2] \cdot i_2 \cdot \langle g, s\rangle \end{array}\right. \left\{\begin{array}{l} s \cdot \underline{0} = [v1, v2] \cdot i_1 \\ s \cdot \mathsf{succ} = [v1, v2] \cdot i_2 \cdot \langle g, s\rangle \end{array}\right.$$

$\equiv$ { Cancelamento-+ (18) $\times 4$ }

$$\left\{\begin{array}{l} g \cdot \underline{0} = u1 \\ g \cdot \mathsf{succ} = u2 \cdot \langle g, s\rangle \end{array}\right. \left\{\begin{array}{l} s \cdot \underline{0} = v1 \\ s \cdot \mathsf{succ} = v2 \cdot \langle g, s\rangle \end{array}\right.$$

$\equiv$ { Igualdade extensional (73) $\times 4$, Def-comp (74) $\times ?$, Natural-const (3) $\times 2$, Def-succ $\times 2$, Def-split (78) $\times 2$ }

$$\left\{\begin{array}{l} g\ 0 = u1\ \ d \\ g\ (d+1) = u2 \cdot \langle g\ d, s\ d\rangle \end{array}\right. \left\{\begin{array}{l} s\ 0 = v1\ \ d \\ s\ (d+1) = v2 \cdot \langle g\ d, s\ d\rangle \end{array}\right.$$

Analisando os resultados obtidos e as definições *pointwise* das funções g e s presentes no enunciado, podemos deduzir quais as funções correspondentes a $u1$, $u2$, $v1$ e $v2$ na solução apresentada acima:

$$\left\{\begin{array}{l} u1 = \underline{1} \\ u2 = mul \end{array}\right. \left\{\begin{array}{l} v1 = \underline{1} \\ v2 = \mathsf{succ} \cdot \pi_2 \end{array}\right.$$

Logo $\langle g, s\rangle = (\!|\langle [u1, u2], [v1, v2]\rangle|\!) = (\!|\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ} \cdot \pi_2]\rangle|\!)$.

Como sabemos que o ciclo *for* ao qual queremos chegar pode ser escrito como um catamorfismo do tipo $\mathsf{for}\ b\ i = (\!|[\underline{i}, b]|\!)$, vamos aplicar a lei de *"Banana-split"* por forma a, combinando os resultados anteriores, obtermos um resultado que esteja sob esta forma.

$$
\begin{array}{ll}
& \langle (\!|\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ} \cdot \pi_2]\rangle|\!), (\!|\langle [\underline{1}, mul], [\mathsf{succ}, \mathsf{succ} \cdot \pi_2]\rangle|\!)\rangle \\
\equiv & \quad \{ \text{"Banana-split"(51) , Def-Functor-}\mathbb{N}_0 \times 2 \} \\
& (\!|(\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ} \cdot \pi_2]\rangle \times \langle [\underline{1}, mul], [\mathsf{succ}, \mathsf{succ} \cdot \pi_2]\rangle) \cdot \langle id + \pi_1, id + \pi_2\rangle|\!) \\
\equiv & \quad \{ \text{Lei da Troca (28)} \times 2 \} \\
& (\!|([\langle \underline{1}, \underline{1}\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle] \times [\langle \underline{1}, \mathsf{succ}\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle]) \cdot \langle id + \pi_1, id + \pi_2\rangle|\!) \\
\equiv & \quad \{ \text{Absorção-x (11)} \} \\
& (\!|\langle [\langle \underline{1}, \underline{1}\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle] \cdot (id + \pi_1), [\langle \underline{1}, \mathsf{succ}\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle] \cdot (id + \pi_2)\rangle|\!) \\
\equiv & \quad \{ \text{Absorção-+ (22)} \times 2 \text{ , Natural-id (1)} \times 2 \} \\
& (\!|\langle [\langle \underline{1}, \underline{1}\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle \cdot \pi_1], [\langle \underline{1}, \mathsf{succ}\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle \cdot \pi_2]\rangle|\!) \\
\equiv & \quad \{ \text{Lei da Troca (28)} \} \\
& (\!|[\langle \langle \underline{1}, \underline{1}\rangle, \langle \underline{1}, \mathsf{succ}\rangle\rangle, \langle \langle mul, \mathsf{succ} \cdot \pi_2\rangle \cdot \pi_1, \langle mul, \mathsf{succ} \cdot \pi_2\rangle \cdot \pi_2\rangle]|\!) \\
\equiv & \quad \{ \text{Def-x (10)} \} \\
& (\!|[\langle \langle \underline{1}, \underline{1}\rangle, \langle \underline{1}, \mathsf{succ}\rangle\rangle, \langle mul, \mathsf{succ} \cdot \pi_2\rangle \times \langle mul, \mathsf{succ} \cdot \pi_2\rangle]|\!) \\
\square &
\end{array}
$$

Tal como queríamos inicialmente, a solução encontra-se sob a forma $\mathsf{for}\ b\ i = (\!|[\underline{i}, b]|\!)$, pelo que podemos retirar *base* e *loop* (*i* e *b*).

Assim sendo:

- *base* corresponderá ao primeiro componente do *either* do resultado: $\langle \langle \underline{1}, \underline{1}\rangle, \langle \underline{1}, \mathsf{succ}\rangle\rangle$
- *loop* corresponderá ao segundo componente do *either* do resultado: $\langle mul, \mathsf{succ} \cdot \pi_2\rangle \times \langle mul, \mathsf{succ} \cdot \pi_2\rangle$

Aplicando as leis de inserção de variáveis necessárias, verificamos que a função *base* quando aplicada a um qualquer *k* resulta em *((1,1), (1,k+1))*.

Finalmente, podemos apresentar as definições finais de *base k* e *loop*:

$$
\begin{array}{l}
base\ k = pairToTuple\ ((1, k + 1), (1, 1)) \\
loop = pairToTuple \cdot (\langle mul, \mathsf{succ} \cdot \pi_2\rangle \times \langle mul, \mathsf{succ} \cdot \pi_2\rangle) \cdot tupleToPair
\end{array}
$$

Em que *pairToTuple* e *tupleToPair* são duas funções auxiliares que foram usadas de forma a tipar corretamente os resultados dos nossos cálculos com os tipos presentes no enunciado deste trabalho. Estas funções apenas convertem um par de pares no 4-tuplo correspondente, para quaisquer tipos de dados.

$$
\begin{array}{l}
pairToTuple :: ((t0, t1), (t2, t3)) \to (t0, t1, t2, t3) \\
pairToTuple\ ((a, b), (c, d)) = (a, b, c, d) \\
tupleToPair :: (t0, t1, t2, t3) \to ((t0, t1), (t2, t3)) \\
tupleToPair\ (a, b, c, d) = ((a, b), (c, d))
\end{array}
$$

Problema 4

Inicialmente chegou-se à codificação (apresentada abaixo) das funções para o tipo *FTree*, que já são conhecidas pela resolução da Ficha 6 das aulas teórico-práticas.

$$
\begin{array}{l}
inFTree = [Unit, compAux\ Comp] \\
\quad \textbf{where}\ compAux\ g\ (a,(b,c)) = g\ a\ b\ c \\
outFTree\ (Unit\ b) = i_1\ b \\
outFTree\ (Comp\ a\ b\ c) = i_2\ (a,(b,c)) \\
baseFTree\ f\ g\ h = g + f \times (h \times h) \\
recFTree\ g = id + (id \times (g \times g)) \\
cataFTree\ g = g \cdot (recFTree\ (cataFTree\ g)) \cdot outFTree \\
anaFTree\ g = inFTree \cdot (recFTree\ (anaFTree\ g)) \cdot g \\
hyloFTree\ f\ g = cataFTree\ f \cdot anaFTree\ g
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Bifunctor\ FTree\ \textbf{where} \\
\quad bimap\ f\ g = cataFTree\ (inFTree \cdot (baseFTree\ f\ g\ id))
\end{array}
$$

1. generatePTree

A função ***generatePTree*** que gera uma PTree (árvore de Pitágoras) a partir de um inteiro que indica a ordem dessa mesma árvore, foi construída a partir de um anamorfismo que obedece ao seguinte esquema:

$$
\begin{tikzcd}
FTree\ Square\ Square & \mathbb{N}_0 + \mathbb{N}_0 \times (FTree\ Square\ Square)^2 \arrow[l, "inFTree"'] \\
\mathbb{N}_0 \arrow[u, "generatePTree = anaFTree\ g"] \arrow[r, "g"] & \mathbb{N}_0 + \mathbb{N}_0 \times (FTree\ Square\ Square)^2 \arrow[u, "id + id \times (generatePTree)^2"']
\end{tikzcd}
$$

Assim sendo, a função *g* que trata da recursividade do anamorfismo é dada por uma função *genSquare* que recebe um par *(o,n)* em que o primeiro valor corresponde à ordem da *PTree* que pretendemos gerar, e o segundo corresponde ao nível atual do cálculo (a meio do passo recursivo).

Caso estejamos no nível final (i.e. o == n), a função gera o tamanho do quadrado (*Square* que é um *Float*) com recurso à função auxiliar *calculateSide* que multiplica o valor da ordem por sqrt(2)/2.

Caso estejamos a meio do caso recursivo (i.e. o != n), a função gera o tamanho do quadrado (*Square* que é um *Float*) correspondente a esse nível (também com recurso à função auxiliar *calculateSide*), devolvendo também dois pares correspondentes às duas subárvores, contendo novamente a ordem da *PTree* e o nível seguinte (n+1) que serão utilizados na próxima execução.

$$
\begin{array}{l}
genSquare :: (Int, Int) \to (Square) + (Square, ((Int, Int), (Int, Int))) \\
genSquare\ (o,n) = \textbf{let}\ calculateSide\ x = fromIntegral\ (x) * (sqrt\ (2)\ /\ 2) \\
\quad \textbf{in if}\ o \equiv n \\
\qquad \textbf{then}\ i_1\ (calculateSide\ o) \\
\qquad \textbf{else}\ i_2\ (calculateSide\ n, ((o, n+1), (o, n+1)))
\end{array}
$$

Desta forma, a função *generatePTree* não será mais do que um anamorfismo que utiliza esta função *genSquare* q, invocando-a inicialmente para o nível 0 (e para a ordem *o* recebida).

$$generatePTree\ o = anaFTree\ genSquare\ (o, 0)$$

2. drawPTree

$$drawPTree = \bot$$

Problema 5

1. singletonbag

A função *singletonbag* é uma função auxiliar que apenas serve para colocar um único elemento do tipo *a* (correspondente, no nosso problema, a um *Marble*) numa única *Bag*.

Ou seja, coloca um par composto pelo elemento *a* recebido (i.e. *marble*) e pelo número 1, o que significa que a *Bag* passou a ter apenas uma ocorrência do elemento *a* (i.e. *marble*).

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

2. muB (multiplicação de *Bag*)

A função μ referente à multiplicação do mónade *Bag* é uma função correspondente ao seguinte esquema:

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

Assim, esta função é responsável por, a partir de uma *Bag* cujos elementos são também eles do tipo *Bag*, construir uma única *Bag* contendo os elementos destas últimas.

Então, podemos começar por codificar a função auxiliar *dist* que obedece ao esquema abaixo, gerando a lista dos elementos de uma *Bag*.

$$Bag\ a \xrightarrow[dist]{} [a]$$

Esta função pode ser facilmente obtida utilizando uma função auxiliar *replicateMarbles* que apenas replica as ocorrências dos elementos do tipo *a* (que, no nosso caso, correspondem a *Marble*s), numa única lista de *a*s:

$$\begin{array}{l}
replicateMarbles :: [(a, Int)] \to [a] \\
replicateMarbles\ [\,] = [\,] \\
replicateMarbles\ ((m, 0) : t) = replicateMarbles\ t \\
replicateMarbles\ ((m, k) : t) = m : replicateMarbles\ ((m, k - 1) : t) \\
dist = replicateMarbles \cdot unB
\end{array}$$

Para o caso particular do nosso problema, as letras *a* do esquema anterior, correspondem, no entanto, também elas a elementos do tipo *Bag*. Assim:

$$Bag\ (Bag\ a) \xrightarrow[dist]{} [Bag\ a]$$

Se na função original μ, começarmos por aplicar esta nova função *dist*, o esquema de μ passa a corresponder ao seguinte:

$$Bag\ (Bag\ a) \xrightarrow[dist]{} [Bag\ a] \xrightarrow[f=?]{} Bag\ a$$

Pelo que nos resta apenas chegar a uma função (representada pela letra *f* no esquema anterior) que, aplicada a uma lista de *Bag*s, devolva uma única *Bag* contendo os elementos presentes nas *Bag*s da lista. Esta função, denominada *concatBags*, apresenta-se no esquema seguinte e no código abaixo deste:

$$[Bag\ a] \xrightarrow[concatBags]{} Bag\ a$$

$$\begin{array}{l}
concatBags :: [Bag\ a] \to Bag\ a \\
concatBags\ [B\ a] = B\ a \\
concatBags\ (B\ a : B\ b : t) = concatBags\ (B\ (a +\!\!+ b) : t)
\end{array}$$

Chegámos, então, ao esquema final da função μ que queríamos obter inicialmente:

$$Bag\ (Bag\ a) \xrightarrow[dist]{} [Bag\ a] \xrightarrow[concatBags]{} Bag\ a$$

Por fim, podemos apresentar a codificação final da função μ, correspondente ao esquema anterior:

$$\mu = concatBags \cdot dist$$

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:

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

[7] Exemplos tirados de [2].

Índice