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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	99 (preencher)
a66695	Carlos Pereira
a74944	Diana Lopes
a74899	Gabriela Vaz

1 Preâmbulo

A disciplina de Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método à programação funcional em Haskell. Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas e a produzir textos técnico-científicos de qualidade.

2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita "literária" [**?**], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a documentação de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp1718t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp1718t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp1718t.zip` e executando

```
$ lhs2TeX cp1718t.lhs > cp1718t.tex
$ pdflatex cp1718t
```

em que `lhs2tex` é um pre-processador que faz "pretty printing" de código Haskell em LaTeX e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

Por outro lado, o mesmo ficheiro `cp1718t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp1718t.lhs
```

Abra o ficheiro `cp1718t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

[1] O suffixo 'lhs' quer dizer *literate Haskell*.

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo C com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTEX) e o índice remissivo (com `makeindex`),

```
$ bibtex cp1718t.aux
$ makeindex cp1718t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck, que ajuda a validar programas em Haskell, a biblioteca JuicyPixels para processamento de imagens e a biblioteca gloss para geração de gráficos 2D:

```
$ cabal install QuickCheck JuicyPixels gloss
```

Para testar uma propriedade QuickCheck *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

Problema 1

Segundo uma notícia do Jornal de Notícias, referente ao dia 12 de abril, *"apenas numa hora, foram transacionadas 1.2 mil milhões de dólares em bitcoins. Nas últimas 24 horas, foram transacionados 8,5 mil milhões de dólares, num total de 24 mil milhões de dólares referentes às principais criptomoedas"*.

De facto, é inquestionável que as criptomoedas, e em particular as bitcoin, vieram para ficar. Várias moedas digitais, e em particular as bitcoin, usam a tecnologia de block chain para guardar e assegurar todas as transações relacionadas com a moeda. Uma block chain é uma coleção de blocos que registam os movimentos da moeda; a sua definição em Haskell é apresentada de seguida.

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

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

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

Cada transação define a entidade de origem da transferência, o valor a ser transacionado, e a entidade destino (por esta ordem), tal como se define de seguida.

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

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

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

Seguem as restantes definições Haskell para completar o código anterior. Note que $Time$ representa o momento da transação, como o número de milisegundos que passaram desde 1970.

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

Neste contexto, implemente as seguintes funções:

1. Defina a função $allTransactions :: Blockchain \to Transactions$, como um catamorfismo, que calcula a lista com todas as transações numa dada block chain.

   Propriedade QuickCheck 1 *As transações de uma block chain são as mesmas da block chain revertida:*

   $$prop1a = sort \cdot allTransactions \equiv sort \cdot allTransactions \cdot reverseChain$$

   Note que a função $sort$ é usada apenas para facilitar a comparação das listas.

2. Defina a função $ledger :: Blockchain \to Ledger$, utilizando catamorfismos e/ou anamorfismos, que calcula o ledger (i.e., o valor disponível) de cada entidade numa uma dada block chain. Note que as entidades podem ter valores negativos; de facto isso acontecerá para a primeira transação que executarem.

   Propriedade QuickCheck 2 *O tamanho do ledger é inferior ou igual a duas vezes o tamanho de todas as transações:*

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

   Propriedade QuickCheck 3 *O ledger de uma block chain é igual ao ledger da sua inversa:*

   $$prop1c = sort \cdot ledger \equiv sort \cdot ledger \cdot reverseChain$$

3. Defina a função $isValidMagicNr :: Blockchain \to Bool$, utilizando catamorfismos e/ou anamorfismos, que verifica se todos os números mágicos numa dada block chain são únicos.

   Propriedade QuickCheck 4 *A concatenação de uma block chain com ela mesma nunca é válida em termos de números mágicos:*

   $$prop1d = \neg \cdot isValidMagicNr \cdot concChain \cdot \langle id, id \rangle$$

   Propriedade QuickCheck 5 *Se uma block chain é válida em termos de números mágicos, então a sua inversa também o é:*

   $$prop1e = isValidMagicNr \Rightarrow isValidMagicNr \cdot reverseChain$$

Problema 2

Uma estrutura de dados frequentemente utilizada para representação e processamento de imagens de forma eficiente são as denominadas quadtrees. Uma *quadtree* é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

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

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

(a) Matriz de exemplo bm.

```
Block
  (Cell 0 4 4) (Block
    (Cell 0 2 2) (Cell 0 2 2) (Cell 1 2 2) (Block
      (Cell 1 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1)))
  (Cell 1 4 4)
  (Block
    (Cell 1 2 2) (Cell 0 2 2) (Cell 0 2 2) (Block
      (Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 1 1 1)))
```

(b) Quadtree de exemplo qt.

Figura 1: Exemplos de representações de bitmaps.

Uma imagem monocromática em formato bitmap pode ser representada como uma matriz de bits[2], tal como se exemplifica na Figura 1a.

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

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

O algoritmo $bm2qt$ particiona recursivamente a imagem em 4 blocos e termina produzindo folhas para matrizes unitárias ou quando todos os píxeis de um sub-bloco têm a mesma côr. Para a matriz bm de exemplo, a quadtree correspondente $qt = bm2qt\ bm$ é ilustrada na Figura 1b.

Imagens a cores podem ser representadas como matrizes de píxeis segundo o código de cores RGBA, codificado no tipo $PixelRGBA8$ em que cada pixel é um quádruplo de valores inteiros $(red, green, blue, alpha)$ contidos entre 0 e 255. Atente em alguns exemplos de cores:

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

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

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

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

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

Esta questão aborda operações de processamento de imagens utilizando quadtrees:

1. Defina as funções $rotateQTree :: QTree\ a \to QTree\ a$, $scaleQTree :: Int \to QTree\ a \to QTree\ a$ e $invertQTree :: QTree\ a \to QTree\ a$, como catamorfismos e/ou anamorfismos, que rodam[3], redimensionam [4] e invertem as cores de uma quadtree[5], respectivamente. Tente produzir imagens similares às Figuras 2b, 2c e 2d:

```
> rotateBMP  "cp1718t_media/person.bmp" "person90.bmp"
> scaleBMP 2 "cp1718t_media/person.bmp" "personx2.bmp"
> invertBMP  "cp1718t_media/person.bmp" "personinv.bmp"
```

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

(a) Bitmap de exemplo.

(b) Rotação.

(c) Redimensionamento.

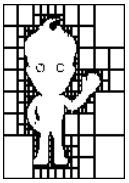
(d) Inversão de cores.

(e) Compresão de 1 nível.

(f) Compresão de 2 níveis.

(g) Compresão de 3 níveis.

(h) Compresão de 4 níveis.



(i) Bitmap de contorno.

(j) Bitmap com contorno.

Figura 2: Manipulação de uma figura bitmap utilizando quadtrees.

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

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

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

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

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

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

2. Defina a função $compressQTree :: Int \to QTree\ a \to QTree\ a$, utilizando catamorfismos e/ou anamorfismos, que comprime uma quadtree cortando folhas da árvore para reduzir a sua profundidade num dado número de níveis. Tente produzir imagens similares (mas não necessariamente iguais) às Figuras 2e, 2f, 2g e 2h:

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

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

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

3. Defina a função $outlineQTree :: (a \to Bool) \to QTree\ a \to Matrix\ Bool$, utilizando catamorfismos e/ou anamorfismos, que recebe uma função que determina quais os píxeis de fundo e converte uma quadtree numa matriz monocromática, de forma a desenhar o contorno de uma malha poligonal contida na imagem. Tente produzir imagens similares (mas não necessariamente iguais) às Figuras 2i e 2j:

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

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

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

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

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

Problema 3

O cálculo das combinações de n k-a-k,

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

envolve três factoriais. Recorrendo à lei de recursividade múltipla do cálculo de programas, é possível escrever o mesmo programa como um simples ciclo-for onde se fazem apenas multiplicações e somas. Para isso, começa-se por estruturar a definição dada da forma seguinte,

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

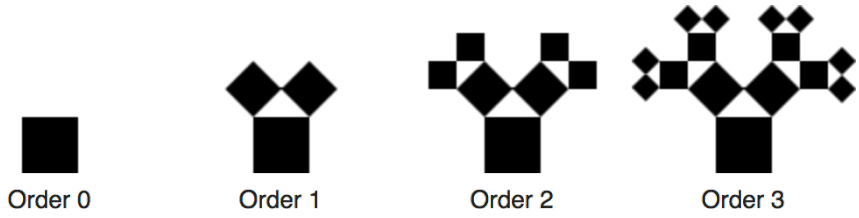


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

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

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

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

e

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

A partir daqui alguém derivou a seguinte implementação:

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

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

Propriedade QuickCheck 11 *Verificação que* $\binom{n}{k}$ *coincide com a sua especificação (1):*

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

Problema 4

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

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

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

1. Defina a função $generatePTree :: Int \to PTree$, como um anamorfismo, que gera uma árvore de Pitágoras para uma dada ordem.

   Propriedade QuickCheck 12 *Uma árvore de Pitágoras tem profundidade igual à sua ordem:*

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

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

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

2. Defina a função $drawPTree :: PTree \to [Picture]$, utilizando catamorfismos e/ou anamorfismos, que anima incrementalmente os passos de construção de uma árvore de Pitágoras recorrendo à biblioteca gloss. Anime a sua solução:

   ```
   > animatePTree 3
   ```

Problema 5

Uma das áreas em maior expansão no campo da informática é a análise de dados e machine learning. Esta questão aborda um *mónade* que ajuda a fazer, de forma simples, as operações básicas dessas técnicas. Esse mónade é conhecido por *bag*, *saco* ou *multi-conjunto*, permitindo que os elementos de um conjunto tenham multiplicidades associadas. Por exemplo, seja

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

um tipo dado.[6] A lista $[Pink, Green, Red, Blue, Green, Red, Green, Pink, Blue, White]$ tem elementos repetidos. Assumindo que a ordem não é importante, essa lista corresponde ao saco

```
{ Red |-> 2 , Pink |-> 2 , Green |-> 3 , Blue |-> 2 , White |-> 1 }
```

que habita o tipo genérico dos "bags":

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

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

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

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

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

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

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

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

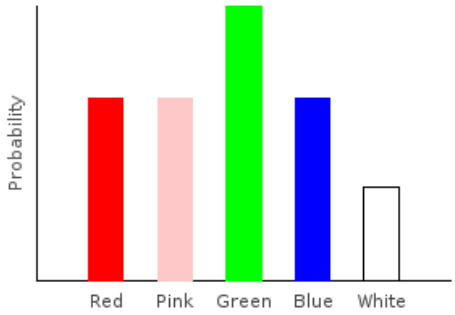


Figura 4: Distribuição de berlindes num saco.

Mais ainda, se quisermos saber o total de berlindes em $bagOfMarbles$ basta calcular fmap $(!)$ $bagOfMarbles$ obtendo-se `{ () |-> 10 }`; isto é, o saco tem 10 berlindes no total.

Finalmente, se quisermos saber a probabilidade da cor de um berlinde que tiremos do saco, basta converter o referido saco numa distribuição correndo:

$$marblesDist = dist\ bagOfMarbles$$

obtendo-se a distribuição (graças ao módulo Probability):

```
Green  30.0%
  Red  20.0%
 Pink  20.0%
 Blue  20.0%
White  10.0%
```

cf. Figura 4.

Partindo da seguinte declaração de Bag como um functor e como um mónade,

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

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

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

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

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

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

onde $b3$ *é um saco dado em anexo.*

Anexos

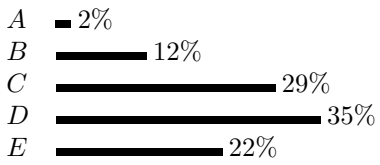
A Mónade para probabilidades e estatística

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

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

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

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



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

B Definições auxiliares

Funções para mostrar *bags*:

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

Igualdade de *bags*:

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

Ainda sobre o mónade *Bag*:

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

O exemplo do texto:

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

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

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

Outras funções auxiliares:

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

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

Pelo enunciado sabemos o tipo de Blockchain, apartir deste conseguimos derivar inBlockchain.

$$inBlockchain = [Bc, Bcs]$$

Para o cálculo de outBlockchain realizamos os seguintes passos:

$$\begin{array}{ll}& outBlockchain \cdot inBlockchain = id\\ \equiv & \quad \{\ \text{Def-in}\ \}\\ & outBlockchain \cdot [Bc, Bcs] = id\\ \equiv & \quad \{\ \text{Fusão-+}\ \}\\ & [outBlockchain \cdot Bc, outBlockchain \cdot Bcs] = id\\ \equiv & \quad \{\ \text{Universal-+}\ \}\\ & \left\{\begin{array}{l} outBlockchain \cdot Bc = id \cdot i_1\\ outBlockchain \cdot Bcs = id \cdot i_2\end{array}\right.\\ \equiv & \quad \{\ \text{Natural-id}\ \}\\ & \left\{\begin{array}{l} outBlockchain \cdot Bc = i_1\\ outBlockchain \cdot Bcs = i_2\end{array}\right.\\ \equiv & \quad \{\ \text{Igualdade extensional ; Def-comp}\ \}\\ & \left\{\begin{array}{l} outBlockchain\ (Bc\ x) = i_1\ x\\ outBlockchain\ (Bcs\ (x, xs)) = i_2\ (x, xs)\end{array}\right.\end{array}$$

□

Deste modo concluimos que:

$$outBlockchain\ (Bc\ x) = i_1\ x$$
$$outBlockchain\ (Bcs\ (x, xs)) = i_2\ (x, xs)$$

Em seguida definimos recBlockchain, cataBlockchain, anaBlockchain e hyloBlockchain com base em todo o raciocinio desenvolvido para os passos anteriores:

$$recBlockchain\ f = id + id \times f$$
$$(\!|g|\!) = g \cdot recBlockchain\ (\!|g|\!) \cdot outBlockchain$$
$$[\!(g)\!] = inBlockchain \cdot recBlockchain\ [\!(g)\!] \cdot g$$
$$hyloBlockchain\ g\ h = (\!|g|\!) \cdot [\!(h)\!]$$

Após a definição de todas as funçoes base, como outBlockchain, inBlockchain, entre outras, partimos para o desenho dos diagramas dos catamorfismos para o calculo de outras funções pedidas. Diagrama de catamorfismo da função allTransactions:

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

Com base no diagrama conseguimos derivar a função:

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

Diagrama de catamorfismo da função ledger:

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

De onde podemos dedosir a seguinte função:

$$ledger = (\!|[aux, \mathsf{conc} \cdot (aux \times id)]|\!)$$
$$\textbf{where}\ aux = (\mathsf{fmap}\ (swap \cdot \pi_2)) \cdot \pi_2 \cdot \pi_2$$

Diagrama de catamorfismo da função auxiliar g para calcular a função isValidMagicNr:

$$\begin{array}{ccc} Blockchain & \xleftarrow{inBlockchain} & Block + Block \times Blockchain \\ {\scriptstyle (\!|isValidMagicNr|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|isValidMagicNr|\!)} \\ (Bool, [MagicNo]) & \xleftarrow[gene]{} & Block + Block \times (Bool, [MagicNo]) \end{array}$$

Por fim, após a construção de algumas funções auxiliares conseguimos construir a função isValidMagicNr:

$$aux1 :: Block \to (Bool, [MagicNo])$$
$$aux1\ (x, (y, z)) = (True, singl\ (x))$$
$$listaR :: (MagicNo, (Bool, [MagicNo])) \to (Bool, [MagicNo])$$
$$listaR\ (b, (bs, xs)) = ((\neg\ ((elem\ b\ xs) \wedge bs)), b : xs)$$
$$isValidMagicNr = (\pi_1 \cdot g)$$
$$\textbf{where}\ g = (\!|[aux1, listaR \cdot (\pi_1 \times id)]|\!)$$

Problema 2

Pelo enunciado sabemos o tipo de QTree, apartir deste conseguimos derivar inQTree.

$$inQTree = [in1, in2]$$
$$in1\ (a,(b,c)) = (Cell\ a\ b\ c)$$
$$in2\ (a,(b,(c,d))) = (Block\ a\ b\ c\ d)$$

Para o cálculo da função outQTree realizamos os seguintes passos:

$$\begin{array}{ll} & outQTree \cdot inQTree = id \\ \equiv & \quad \{\ \text{Def-in}\ \} \\ & outQTree \cdot [in1, in2] = id \\ \equiv & \quad \{\ \text{Fusão-+}\ \} \\ & [outQTree \cdot in1, outQTree \cdot in2] = id \\ \equiv & \quad \{\ \text{Universal-+}\ \} \\ & \begin{cases} outQTree \cdot in1 = id \cdot i_1 \\ outQTree \cdot in2 = id \cdot i_2 \end{cases} \\ \equiv & \quad \{\ \text{Natural-id}\ \} \\ & \begin{cases} outQTree \cdot in1 = i_1 \\ outQTree \cdot in2 = i_2 \end{cases} \\ \equiv & \quad \{\ \text{Igualdade extensional ; Def-comp}\ \} \\ & \begin{cases} outQTree\ (Cell\ a\ b\ c) = i_1\ (a,(b,c)) \\ outQTree\ (Block\ x\ y\ z\ d) = i_2\ (x,(y,(z,d))) \end{cases} \\ \square & \end{array}$$

Com isto concluimos que:

$$outQTree\ (Cell\ a\ b\ c) = i_1\ (a,(b,c))$$
$$outQTree\ (Block\ x\ y\ z\ d) = i_2\ (x,(y,(z,d)))$$

Em seguida, para podermos iniciar a definição das funções pedidas definimos outras funções basicas pedidas.

$$baseQTree\ g\ f = (g \times id) + (f \times (f \times (f \times f)))$$
$$recQTree\ f = baseQTree\ id\ f$$
$$(\!|g|\!) = g \cdot recQTree\ (\!|g|\!) \cdot outQTree$$
$$[\!(g)\!] = inQTree \cdot recQTree\ [\!(g)\!] \cdot g$$
$$hyloQTree\ g\ h = (\!|g|\!) \cdot [\!(h)\!]$$
$$\textbf{instance}\ Functor\ QTree$$
$$\quad \textbf{where}\ \mathsf{fmap}\ f = (\!|inQTree \cdot baseQTree\ f\ id|\!)$$

Após a definição de todas as funçoes base, como outQTree, inQTree, entre outras, partimos para o desenho dos diagramas dos catamorfismos para o calculo de outras funções pedidas. Diagrama de catamorfismo da função rotateQTree:

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & 1 + (QTree\ A) \uparrow 4 \\ {\scriptstyle (\!|rotateQTree|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|rotateQTree|\!) \uparrow 4} \\ QTree\ B & \xleftarrow[gene]{} & 1 + (QTree\ B) \uparrow 4 \end{array}$$

Apartir de onde definimos a função rotateQTree como:

$$rotateQTree = (\!|inQTree \cdot (auxSwap1 + auxSwap2)|\!)$$
$$auxSwap1\ (a,(b,c)) = (a,(c,b))$$
$$auxSwap2\ (a,(b,(c,d))) = (c,(a,(d,b)))$$

Diagrama de catamorfismo scalQAux que usamos para a definição da função scaleQTree:

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & 1 + (QTree\ A) \uparrow 4 \\ {\scriptstyle (\!|scaleQAux|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|scaleQAux|\!) \uparrow 4} \\ QTree\ B & \xleftarrow[gene]{} & 1 + (QTree\ B) \uparrow 4 \end{array}$$

Assim concluimos que a função pode-se definir da seguinte forma:

$$\begin{aligned} &scaleQTree\ x = (\!|inQTree \cdot (mult\ x + id)|\!) \\ &\quad \textbf{where}\ mult\ x\ (a,(b,c)) = (a,(b * x, c * x)) \end{aligned}$$

Diagrama de catamorfismo da função invertQTree:

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & 1 + (QTree\ A) \uparrow 4 \\ {\scriptstyle (\!|invertQTree|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|invertQTree|\!) \uparrow 4} \\ QTree\ B & \xleftarrow[gene]{} & 1 + (QTree\ B) \uparrow 4 \end{array}$$

onde as variáveis A e B são do tipo PixelRGBA8. Este diagrama serviu como base para definição da função:

$$\begin{aligned} &invertQTree = (\!|[in1 \cdot (auxC \times id), in2 \cdot (id \times (id \times (id \times id)))]|\!) \\ &\quad \textbf{where}\ auxC\ (PixelRGBA8\ a\ b\ c\ d) = (PixelRGBA8\ (255 - a)\ (255 - b)\ (255 - c)\ d) \end{aligned}$$

Diagrama de catamorfismo da função compressQTree:

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & 1 + (QTree\ A) \uparrow 4 \\ {\scriptstyle (\!|compressQTree|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|compressQTree|\!) \uparrow 4} \\ QTree\ B & \xleftarrow[gene]{} & 1 + (QTree\ B) \uparrow 4 \end{array}$$

Logo verificamos que a função compressQTree tem a seguinte definição:

$$\begin{aligned} &veCorte\ (\_,(Cell\ n\ x\ y)) = i_1\ ((n,(x,y))) \\ &veCorte\ (0,(Block\ a\ b\ c\ d)) = i_1\ (block\_Cell\ (Block\ a\ b\ c\ d)) \\ &veCorte\ (alt,(Block\ a\ b\ c\ d)) = i_2\ ((n,a),((n,b),((n,c),((n,d))))) \\ &\quad \textbf{where}\ n = pred\ alt \\ &block\_Cell\ (Block\ a\ b\ c\ d) = calculaMatrix\ (tratar\ a)\ b\ c\ d\ (depthQTree\ (Block\ a\ b\ c\ d)) \\ &tratar\ (Cell\ a\ b\ c) = (a,(b,c)) \\ &tratar\ (Block\ a\ b\ c\ d) = tratar\ a \\ &calculaMatrix\ (a,(b,c))\ \_\ \_\ \_\ t = (a,(t,t)) \\ &compressQTree\ c\ q = [\![(veCorte)]\!]\ ((depthQTree\ q) - c, q) \end{aligned}$$

Diagrama de catamorfismo outlineQAux que usamos para a definição da função outlineQTree:

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & 1 + (QTree\ A) \uparrow 4 \\ {\scriptstyle (\!|outlineQAux|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|outlineQAux|\!) \uparrow 4} \\ Matrix\ Bool\ B & \xleftarrow[gene]{} & 1 + Matrix\ Bool\ B \end{array}$$

Logo a função outlineQTree é definida da seguinte forma:

$$outlineQTree\ p = qt2bm \cdot (\!|[in1 \cdot (p \times id), in2 \cdot (id \times (id \times (id \times id)))]|\!)$$

Problema 3

Inicializamos a resolução deste problema por aplicar a lei da recursividade múltipla a ¡ f k, l k¿. Para isto realizamos os seguintes passos:

$$\left\{\begin{array}{l} f\ k\cdot \mathsf{in} = h\cdot F<f\ k, l\ k> \\ l\ k\cdot \mathsf{in} = x\cdot F<f\ k, l\ k> \end{array}\right.$$

$\equiv$ { Def-inNat; Def-h; Def-x }

$$\left\{\begin{array}{l} f\ k\cdot [\underline{0}, (\mathsf{succ}\ )] = [h1, h2]\cdot F<f\ k, l\ k> \\ l\ k\cdot [\underline{0}, (\mathsf{succ}\ )] = [x1, x2]\cdot F<f\ k, l\ k> \end{array}\right.$$

$\equiv$ { Def-functor }

$$\left\{\begin{array}{l} f\ k\cdot [\underline{0}, (\mathsf{succ}\ )] = [h1, h2]\cdot (id + <f\ k, l\ k>) \\ l\ k\cdot [\underline{0}, (\mathsf{succ}\ )] = [x1, x2]\cdot (id + <f\ k, l\ k>) \end{array}\right.$$

$\equiv$ { absorção-+; fusão-+ }

$$\left\{\begin{array}{l} [(f\ k\cdot 0), f\ k\cdot (\mathsf{succ}\ )] = [h1, h2\cdot <f\ k, l\ k>] \\ [\underline{(l\ k\cdot 0)}, l\ k\cdot (\mathsf{succ}\ )] = [x1, x2\cdot <f\ k, l\ k>] \end{array}\right.$$

$\equiv$ { Eq-+ }

$$\left\{\begin{array}{l} \underline{(f\ k\cdot 0)} = h1 \\ f\ k\cdot \mathsf{succ} = h2\cdot <f\ k, l\ k> \end{array}\right. \quad \left\{\begin{array}{l} \underline{(l\ k\cdot 0)} = x1 \\ l\ k\cdot \mathsf{succ} = x2\cdot <f\ k, l\ k> \end{array}\right.$$

$\equiv$ { 73; 74; 76; 78 }

$$\left\{\begin{array}{l} f\ k\ 0 = h1 \\ f\ k\ (\mathsf{succ}\ a) = h2\ (f\ k\ a, l\ k\ a) \end{array}\right. \quad \left\{\begin{array}{l} l\ k\ 0 = x1 \\ l\ k\ (\mathsf{succ}\ a) = x2\ (f\ k\ a, l\ k\ a) \end{array}\right.$$

□

Com este resultado podemos concluir que:

$$\left\{\begin{array}{l} h1 = \underline{1} \\ h2 = mult \end{array}\right. \quad \left\{\begin{array}{l} x1 = \underline{(k+1)} \\ x2 = \mathsf{succ}\cdot \pi_2 \end{array}\right.$$

□

Aplicando agora a lei da recursividade múltipla a ¡ g, s¿. Para isto realizamos os seguintes passos:

$$\left\{\begin{array}{l} g\cdot \mathsf{in} = d\cdot F<g, s> \\ s\cdot \mathsf{in} = w\cdot F<g, s> \end{array}\right.$$

$\equiv$ { Def-inNat; Def-d; Def-w }

$$\left\{\begin{array}{l} g\cdot [\underline{0}, (\mathsf{succ}\ )] = [d1, d2]\cdot F<g, s> \\ s\cdot [\underline{0}, (\mathsf{succ}\ )] = [w1, w2]\cdot F<g, s> \end{array}\right.$$

$\equiv$ { Def-functor }

$$\left\{\begin{array}{l} g\cdot [\underline{0}, (\mathsf{succ}\ )] = [d1, d2]\cdot (id + <g, s>) \\ s\cdot [\underline{0}, (\mathsf{succ}\ )] = [w1, w2]\cdot (id + <g, s>) \end{array}\right.$$

$\equiv$ { absorção-+; fusão-+ }

$$\left\{\begin{array}{l} [(g\cdot 0), g\cdot (\mathsf{succ}\ )] = [d1, d2\cdot <g, s>] \\ [\underline{(s\cdot 0)}, s\cdot (\mathsf{succ}\ )] = [w1, w2\cdot <g, s>] \end{array}\right.$$

$\equiv$ { Eq-+ }

$$\left\{\begin{array}{l} \underline{(g\cdot 0)} = d1 \\ g\cdot \mathsf{succ} = g2\cdot <g, s> \end{array}\right. \quad \left\{\begin{array}{l} \underline{(s\cdot 0)} = w1 \\ s\cdot \mathsf{succ} = w2\cdot <g, s> \end{array}\right.$$

$\equiv$ { 73; 74; 76; 78 }

$$\left\{\begin{array}{l} g\ 0 = d1 \\ g\ (\mathsf{succ}\ a) = d2\ (g\ a, s\ a) \end{array}\right. \quad \left\{\begin{array}{l} s\ 0 = w1 \\ s\ (\mathsf{succ}\ a) = w2\ (g\ a, s\ a) \end{array}\right.$$

□

Com este resultado podemos concluir que:

$$\begin{cases} d1 = \underline{1} \\ d2 = mult \cdot swap \end{cases} \quad \begin{cases} w1 = \underline{1} \\ w2 = \mathsf{succ} \cdot \pi_2 \end{cases}$$

□

Combinando os resultados com a lei de banana-split obtemos o seguinte:

$$\begin{aligned}
& \langle (\!| \langle [h1, h2], [x1, x2] \rangle |\!), (\!| \langle [d1, d2], [w1, w2] \rangle |\!) \rangle \\
\equiv & \quad \{ \text{ Lei da Troca } \} \\
& \langle (\!| [\langle h1, x1 \rangle, \langle h2, x2 \rangle] |\!), (\!| [\langle d1, w1 \rangle, \langle d2, w2 \rangle] |\!) \rangle \\
\equiv & \quad \{ \text{ "Banana-split"} \} \\
& (\!| [\langle h1, x1 \rangle, \langle h2, x2 \rangle] \times [\langle d1, w1 \rangle, \langle d2, w2 \rangle] \cdot \langle F\ \pi_1, F\ \pi_2 \rangle |\!) \\
\equiv & \quad \{ \text{ Def-Funtor; Absorção-x } \} \\
& (\!| \langle [\langle h1, x1 \rangle, \langle h2, x2 \rangle] \cdot (id + \pi_1), [\langle d1, w1 \rangle, \langle d2, w2 \rangle] \cdot (id + \pi_2) \rangle |\!) \\
\equiv & \quad \{ \text{ Absorção-+; Natural-id } \} \\
& (\!| \langle [\langle h1, x1 \rangle, \langle h2, x2 \rangle \cdot \pi_1], [\langle d1, w1 \rangle, \langle d2, w2 \rangle \cdot \pi_2] \rangle |\!) \\
\equiv & \quad \{ \text{ Fusão-x } \} \\
& (\!| \langle [\langle h1, x1 \rangle, \langle h2 \cdot \pi_1, x2 \cdot \pi_1 \rangle], [\langle d1, w1 \rangle, \langle d2 \cdot \pi_2, w2 \cdot \pi_2 \rangle] \rangle |\!) \\
\equiv & \quad \{ \text{ Lei da Troca } \} \\
& (\!| [\langle \langle h1, x1 \rangle, \langle d1, w1 \rangle \rangle, \langle \langle h2 \cdot \pi_1, x2 \cdot \pi_1 \rangle, \langle d2 \cdot \pi_2, w2 \cdot \pi_2 \rangle \rangle] |\!)
\end{aligned}$$

□

Pelas aulas práticas sabemos que:

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

Posto isto, sabemos que:

$$loop = \langle \langle h2 \cdot \pi_1, x2 \cdot \pi_1 \rangle, \langle d2 \cdot \pi_2, w2 \cdot \pi_2 \rangle \rangle base\ k = \langle \langle h1, x1 \rangle, \langle d1, w1 \rangle \rangle$$

, ou seja

$$loop = \langle \langle mult \cdot \pi_1, \mathsf{succ} \cdot \pi_2 \cdot \pi_1 \rangle, \langle mult \cdot swap \cdot \pi_2, \mathsf{succ} \cdot \pi_2 \cdot \pi_2 \rangle \rangle base\ k = \langle \langle \underline{1}, \underline{k} + 1 \rangle, \langle \underline{1}, \underline{1} \rangle \rangle$$

Então é fácil concluír que, usando a função auxiliar rec4Pares, a definição de base e loop é:

$$\begin{aligned}
& rec4Pares\ ((a, c), (b, d)) = (a, c, b, d) \\
& base\ k = rec4Pares\ ((1, k + 1), (1, 1)) \\
& loop\ (a, b, c, d) = rec4Pares\ ((a * b, b + 1), (c * d, d + 1))
\end{aligned}$$

Problema 4

Pelo enunciado sabemos o tipo de FTree, apartir deste conseguimos derivar inFTree.

$$\begin{aligned}
& inFTree = [Unit, in3] \\
& in3\ (a, (c, d)) = (Comp\ a\ c\ d)
\end{aligned}$$

Para o cálculo da função outFTree realizamos os seguintes passos:

$$\begin{aligned}
& outFTree \cdot inFTree = id \\
\equiv & \quad \{ \text{ Def-in } \} \\
& outFTree \cdot [Unit, in3] = id
\end{aligned}$$

$$
\begin{array}{ll}
\equiv & \{\ \text{Fusão-+}\ \} \\
& [outFTree \cdot Unit, outFTree \cdot in3] = id \\
\equiv & \{\ \text{Universal-+}\ \} \\
& \begin{cases} outFTree \cdot Unit = id \cdot i_1 \\ outFTree \cdot in3 = id \cdot i_2 \end{cases} \\
\equiv & \{\ \text{Natural-id}\ \} \\
& \begin{cases} outFTree \cdot Unit = i_1 \\ outFTree \cdot in3 = i_2 \end{cases} \\
\equiv & \{\ \text{Igualdade extensional ; Def-comp}\ \} \\
& \begin{cases} outFTree\ (Unit\ a) = i_1\ (a) \\ outFTree\ (Comp\ x\ y\ z) = i_2\ (x, (y, z)) \end{cases} \\
\square &
\end{array}
$$

Logo,

$$
\begin{array}{l}
outFTree\ (Unit\ c) = i_1\ c \\
outFTree\ (Comp\ a\ t1\ t2) = i_2\ (a, (t1, t2))
\end{array}
$$

Em seguida definimos as outras funções basicas necessárias.

$$
\begin{array}{l}
baseFTree\ f\ g\ h = g + (f \times (h \times h)) \\
recFTree\ f = baseFTree\ id\ id\ f \\
(\!|a|\!) = a \cdot (recFTree\ (\!|a|\!)) \cdot outFTree \\
[\!(f)\!] = inFTree \cdot (recFTree\ [\!(f)\!]) \cdot f \\
hyloFTree\ a\ c = (\!|a|\!) \cdot [\!(c)\!] \\
\textbf{instance}\ BiFunctor\ FTree \\
\quad \textbf{where}\ bmap\ f\ g = (\!|inFTree \cdot baseFTree\ f\ g\ id|\!)
\end{array}
$$

Após a definição de todas as funçoes base, como outFTree, inFTree, entre outras, partimos para o desenho dos diagramas do anamorfismo e catamorfismo para o calculo de outras funções pedidas. Diagrama de anamorfismo da função generatePTree:

$$
\begin{array}{ccc}
PTree\ A & \xrightarrow[gene]{} & \mathbb{N}_0 + I\!R_0 \times (PTree\ A) \uparrow 2 \\
{\scriptstyle [\!(generatePTree)\!]}\downarrow & & \downarrow{\scriptstyle id+id \times [\!(generatePTree)\!] \uparrow 2} \\
\mathbb{N}_0 & \xleftarrow[inPTree]{} & I\!R_0 + I\!R_0 \times (\mathbb{N}_0) \uparrow 2
\end{array}
$$

Após elaborar o diagrama definimos em haskell a função generatePTree. Tendo obtido a seguinte definição:

$$
\begin{array}{l}
auxP\ (n, 0) = Unit\ ((sqrt\ (2)\ /\ 2) \uparrow n) \\
auxP\ (n, j) = Comp\ ((sqrt\ (2)\ /\ 2) \uparrow n)\ (auxP\ (n+1, j-1))\ (auxP\ (n+1, j-1))
\end{array}
$$

Assim, depois de verificar que a definição obtida estava correta, passamos a definição da mesmo para o formato de anamorfismo. Obtendo o descrito em seguida como definição de generatePTree.

$$
\begin{array}{l}
generatePTree\ n = [\!(auxPGen)\!]\ (0, n) \\
auxPGen\ (n, 0) = i_1\ ((sqrt\ (2)\ /\ 2) \uparrow n) \\
auxPGen\ (n, j) = i_2\ (((sqrt\ (2)\ /\ 2) \uparrow n), ((n+1, j-1), (n+1, j-1)))
\end{array}
$$

Diagrama de catamorfismo da função drawPTree:

$$
\begin{array}{ccc}
PTree\ A & \xleftarrow{inPTree} & 1 + I\!R_0 \times (PTree\ A) \uparrow 2 \\
{\scriptstyle (\!|drawPTree|\!)}\downarrow & & \downarrow{\scriptstyle id+id \times (\!|drawPTree|\!) \uparrow 2} \\
[Picture] & \xleftarrow[gene]{} & 1 + I\!R_0 \times [Picture]
\end{array}
$$

```
-- drawPTree (Unit a) = singl(Rotate 0 (Translate 0 0 (square a)))
-- drawPTree (Comp a b c) = singl(square a) ++ map ((Rotate (-45).(Translate 0 a)) drawPTree b ++ map ((Rot
```

$$drawPTree\ (Unit\ a) = singl\ (Rotate\ 0\ (Translate\ 0\ 0\ (square\ a)))$$
$$drawPTree\ (Comp\ a\ b\ c) = singl\ (square\ a) +\!\!+ drawPTree\ b +\!\!+ drawPTree\ c$$

Problema 5

Para a resolução desta questão começamos por desenhar um diagrama para melhor compreender a função dist. A imagem do diagrama pode ser consultada no ficheiro em anexo "diagrama.jpg".

Assim, começamos pela definição da função dist:

$$dist\ b = prob\ b\ (nBags\ b)$$

Para transformar o nosso resultado no tipo Dist realizamos a função "prob":

$$prob\ (B\ t)\ i = D\ (intToProb\ t\ i)$$

Para transformar o resultado para uma lista onde os seus elementos são (a, ProbRep), para estar em concordância com o tipo Dist a usamos intToProb:

$$intToProb\ [\,]\ \_ = [\,]$$
$$intToProb\ ((a, n) : t)\ i = cons\ ((a, (fromIntegral\ n)\ /\ (fromIntegral\ i)), intToProb\ t\ i)$$

Por fim, para calcular o número total de elementos de uma Bag realizamos a função nBag:

$$nBags\ (B\ [\,]) = 0$$
$$nBags\ (B\ ((a, i) : t)) = i + nBags\ (B\ t)$$

Após a definição desta tentamos solucionar o resto das funções pedidas:

$$singletonbag = B \cdot singl \cdot \langle id, \underline{1} \rangle$$
$$\mu = B \cdot bagValor \cdot (\mathsf{fmap}\ unB)$$
$$bagValor :: Bag\ [(a, Int)] \to [(a, Int)]$$
$$bagValor = concat \cdot (\mathsf{map}\ \ novoValor) \cdot unB$$
$$novoValor :: ([(a, Int)], Int) \to [(a, Int)]$$
$$novoValor\ (x, y) = \mathsf{map}\ \ (id \times (*y))\ x$$

Para verificar se as funçoes muB e singletonbag estão corretas corremos o seguinte comanto na linhas de comandos:

$$\mathbf{do}\ \{x \leftarrow bagOfMarbles; return\ (marbleWeight\ x)\}$$

de onde obtivemos o seguinte resultado:

$$\{2 \mapsto 3, 3 \mapsto 5, 6 \mapsto 2\}$$

Deste modo, conseguimos concluir que as nossas definições das funções pedidas estão corretas.

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

□

[7] Exemplos tirados de [?].

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