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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	78
a75008	Filipe Fortunato
a76516	João Vieira
a74036	Manuel Monteiro

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.

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

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.

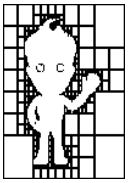
(d) Inversão de cores.

(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,

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

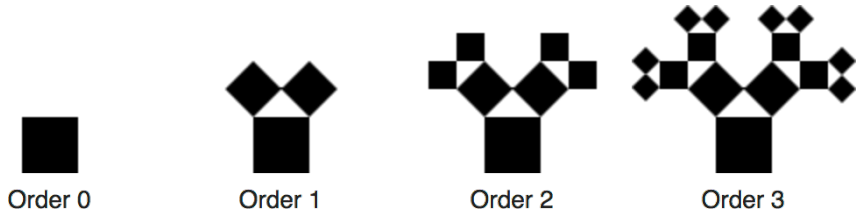


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

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.

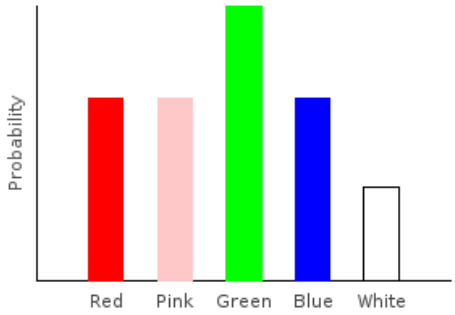


Figura 4: Distribuição de berlindes num saco.

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,

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

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

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

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

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

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

onde $b3$ *é um saco dado em anexo.*

Referências

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

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

Anexos

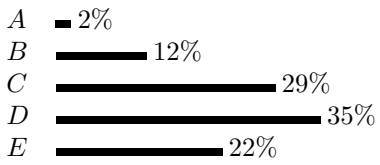
A Mónade para probabilidades e estatística

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

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

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

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



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

B Definições auxiliares

Funções para mostrar *bags*:

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

Igualdade de *bags*:

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

Ainda sobre o mónade *Bag*:

```
instance Applicative Bag where
  pure = return
  (<*>) = aap
```

O exemplo do texto:

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

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

```
b3 :: Bag (Bag (Bag Marble))
b3 = B [(B [(B [(Pink,2),(Green,3),(Red,2),(Blue,2),(White,1)],5)
  ,(B [(Pink,1),(Green,2),(Red,1),(Blue,1)],2)],2)]
```

Outras funções auxiliares:

```
a ↦ b = (a,b)
consol :: (Eq b) ⇒ [(b,Int)] → [(b,Int)]
consol = filter nzero · map (id × sum) · col where nzero (_,x) = x ≢ 0
isempty :: Eq a ⇒ [(a,Int)] → Bool
isempty = all (≡0) · map π2 · consol
col x = nub [k ↦ [d' | (k',d') ← x, k' ≡ k] | (k,d) ← x]
consolidate :: Eq a ⇒ Bag a → Bag a
consolidate = B · consol · unB
```

C Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto e / ou outras funções auxiliares que sejam necessárias.

Problema 1

Neste primeiro problema, é utilizado o tipo de dados *Blockchain*, o que permite criar um isomorfismo entre *inBlockchain* e o *OutBlockchain*, para definirmos o cata-morfismo, anamorfismo e o hylomorfismo da estrutura de dados.

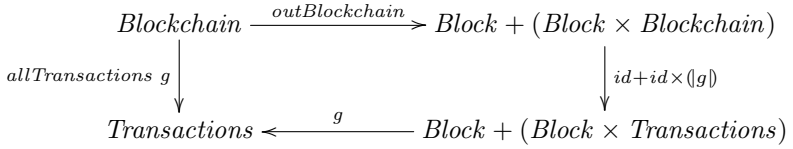
```
inBlockchain = [Bc, Bcs]
outBlockchain (Bc b) = i1 b
outBlockchain (Bcs (b,bc)) = i2 (b,bc)
recBlockchain f = id + (id × f)
cataBlockchain g = g · recBlockchain (cataBlockchain g) · outBlockchain
anaBlockchain g = inBlockchain · recBlockchain (anaBlockchain g) · g
hyloBlockchain g h = cataBlockchain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

C.0.1 1 - allTransactions

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

Para esta função utilizámos um cata-morfismo com um "gene"que nos permite retirar a lista de todas as transações da BlockChain, após a sua utilização. Para isso o "gene"apenas terá de ir a cada bloco, e obter a lista de transações de cada Bloco, e no caso de termos o tuplo compondo um bloco, e após aplicar o cata-morfismo já temos em vez da BlockChain, uma lista de transações, logo apenas temos de concatenar as duas listas presentes no tuplo com a função *conc*.

Com isto chegamos ao seguinte diagrama:

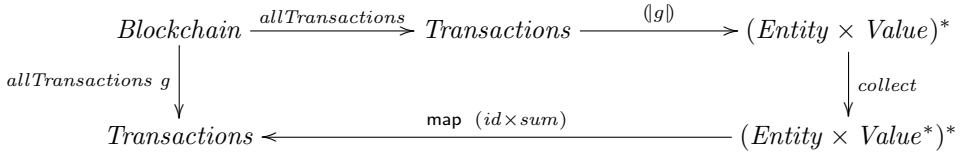


C.0.2 2 - ledger

$$ledger = \mathsf{map}\ (id \times sum) \cdot col \cdot cataList\ [nil, f] \cdot allTransactions$$
$$\textbf{where}\ f = \mathsf{conc} \cdot ((\mathsf{conc} \cdot (singl \times singl) \cdot \langle id \times negate \cdot \pi_2, \pi_2 \rangle \cdot (id \times swap)) \times id)$$

A partir de uma BlockChain, é possível calcular o valor que cada entidade detém, tipicamente designado de ledger. Para isso primeiro aplicamos o cata-morfismo na BlockChain para obter todas as transações da mesma através da função anteriormente definida *allTransactions*. Depois um cata-morfismo para obter a lista de tuplo *(Entity,Value)*, ainda com Entidades repetidas e valores negativos, utilizámos a função *collect* para obter uma lista de valores de cada entidade eliminando as repetidas. Tendo isto basta aplicar um $map(id * sum)$ para somar os valores de cada entidade.

Com isto, podemos desenvolver o seguinte diagrama:

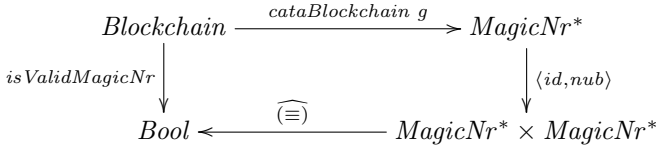


C.0.3 3 - isValidMagicNr

Para verificar se não existem MagicNr repetidos na BlockChain, começámos por aplicar um cata-morfismo para obter a lista de MagicNr da mesma. Depois aplicamos um $\langle id, nub \rangle$ e verificamos a diferença no tuplo criado (a função *nub* remove repetidos da lista) logo se houverem diferenças entre as listas deve-se a haver valores repetidos logo a função retorna False e True no caso de serem as listas iguais.

$$isValidMagicNr = \widehat{(\equiv)} \cdot \langle id, nub \rangle \cdot cataBlockchain\ [singl \cdot \pi_1, cons \cdot (\pi_1 \times id)]$$

Posto isto, desenvolvemos o seguinte diagrama:



Problema 2

Tal como o problema 1 definimos as funções inQTree e outQTree e os respetivos cata, ana e hylomorfismos, tal como um *Functor* que aplica a função apenas às folhas da Tree, neste caso ao pixel.

$$inQTree = [\widehat{Cell} \cdot assocl, \widehat{\widehat{Block}} \cdot assocl \cdot assocl]$$
$$outQTree\ (Cell\ a\ b\ c) = i_1\ (a, (b, c))$$
$$outQTree\ (Block\ a\ b\ c\ d) = i_2\ (a, (b, (c, d)))$$
$$baseQTree\ g\ h = (g \times id) + (h \times (h \times (h \times h)))$$
$$recQTree\ g = baseQTree\ id\ g$$
$$cataQTree\ g = g \cdot recQTree\ (cataQTree\ g) \cdot outQTree$$
$$anaQTree\ g = inQTree \cdot recQTree\ (anaQTree\ g) \cdot g$$
$$hyloQTree\ g\ h = cataQTree\ g \cdot anaQTree\ h$$
$$\textbf{instance}\ Functor\ QTree\ \textbf{where}$$
$$\mathsf{fmap}\ g = cataQTree\ (inQTree \cdot baseQTree\ g\ id)$$

C.0.4 1 - rotateQTree

Para rodar uma QTree temos de trocar as linhas e as colunas de cada Cell, e rodar cada QTree do Block, para tal basta trocar as posições das quatro QTrees, para isso aplicamos três splits para obter as quatro QTrees efetuando as trocas ao aplicar apenas projeções nos tuplos. Aplicando um cata-morfismo do nosso *inQTree* após "gene"para obter uma QTree ao aplicar o cata-morfismo.

$$
\begin{aligned}
&rotateQTree = cataQTree\ (inQTree \cdot (f + g))\\
&\quad \textbf{where}\ g = \langle \pi_1 \cdot \pi_2 \cdot \pi_2, \langle \pi_1, \langle \pi_2 \cdot \pi_2 \cdot \pi_2, \pi_1 \cdot \pi_2 \rangle \rangle \rangle\\
&\qquad f = id \times swap
\end{aligned}
$$

Posto isto, podemos desenvolver o seguinte diagrama:

$$
\begin{array}{ccc}
QTree & \xrightarrow{outQTree} & B + QTree^4 \\
{\scriptstyle rotateQTree}\downarrow & & \downarrow{\scriptstyle id + cataQTree\ g} \\
QTree & \xleftarrow[g = inQTree \cdot (f+g)]{} & B + QTree
\end{array}
$$

C.0.5 2 - scaleQTree

Para aumentar o tamanho da QTree apenas temos de multiplicar o valor passado na função pelas linhas e colunas de cada Cell aumentando assim n vezes a QTree. Para tal utilizámos um anamorfismo que ao fazer *outQTree* e aplicar o "gene"obtemos uma QTree aumentada.

$$
\begin{aligned}
&scaleQTree\ i = anaQTree\ ((f + id) \cdot outQTree)\\
&\quad \textbf{where}\ f = id \times ((i*) \times (i*))
\end{aligned}
$$

Posto isto, chegamos ao seguinte diagrama:

$$
\begin{array}{ccc}
QTree & \xleftarrow{inQTree} & B + QTree^4 \\
{\scriptstyle scaleQTree\ i}\uparrow & & \uparrow{\scriptstyle id + [\![h]\!]} \\
QTree & \xrightarrow[h = (f+id) \cdot outQTree]{} & B + QTree
\end{array}
$$

C.0.6 3 - invertQTree

Para inverter as cores de uma QTree, só é necessário trocar as cores dos pixeis da mesma, isto é subtrair a 255 a cada sub-cor (*vermelho, verde e azul*) do pixel. Definindo uma função para tal e utilizar o Functor definido para aplicar a função aos pixeis, obtemos a QTree com as cores invertidas.

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

Posto isto, desenvolvemos o seguinte diagrama:

$$
QTree \xrightarrow{Functor(f)} F\ QTree
$$

$$
\begin{aligned}
&compressQTree = \bot\\
&outlineQTree = \bot
\end{aligned}
$$

Problema 3

Para este problema percebemos que as quatro funções auxiliares definidas no problema seriam o nosso ponto de partida para resolver o mesmo. Então para isso começamos a definir as funções:

$$
\begin{aligned}
&f\ k\ 0 = one\\
&f\ k\ (+1) = mul \cdot \langle lk, fk \rangle\\
&l\ k\ 0 = (+1)\\
&l\ k\ (+1) = (+1) \cdot lk\\
\equiv\ &\quad \{\ \text{Igualdade extensional} \times 2, \text{Def-Comp, Eq-+}\ \}
\end{aligned}
$$

$$
\begin{cases}
[\mathit{fk} \cdot \mathit{zero}, \mathit{fk} \cdot (+1)] = [\mathit{one}, \mathit{mul} \cdot \langle \mathit{lk}, \mathit{fk} \rangle] \\
[\mathit{lk} \cdot \mathit{zero}, \mathit{lk} \cdot (+1)] = [+1, (+1) \cdot \mathit{lk}]
\end{cases}
$$

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

$$
\begin{cases}
\mathit{fk} \cdot [\mathit{zero}, +1] = [\mathit{one}, \mathit{mul}] \cdot (\mathit{id} + \langle \mathit{lk}, \mathit{fk} \rangle) \\
\mathit{lk} \cdot [\mathit{zero}, +1] = [+1, +1 \cdot \pi_2] \cdot (\mathit{id} + \langle \mathit{fk}, \mathit{lk} \rangle
\end{cases}
$$

$\equiv$ { Fokkinga }

$$\langle \mathit{fk}, \mathit{lk} \rangle = (\!| \langle [\mathit{one}, \mathit{mul}], [+1, +1 \cdot \pi_2] \rangle |\!)$$

□

Após obtermos o $\langle \mathit{fk}, \mathit{lk} \rangle$, passamos à demonstração do segundo $\langle g, s \rangle$:

$$
\begin{cases}
g\ \mathit{zero} = \mathit{one} \\
g\ (+1) = \mathit{mul} \cdot \langle s, g \rangle \\
s\ \mathit{zero} = \mathit{one} \\
s\ (+1) = (+1) \cdot s
\end{cases}
$$

$\equiv$ { Igualdade extensional × 2, Def-Comp, Eq-+ }

$$
\begin{cases}
[g \cdot \mathit{zero}, g \cdot (+1)] = [\mathit{one}, \mathit{mul} \cdot \langle s, g \rangle] \\
[s \cdot \mathit{zero}, s \cdot (+1)] = [\mathit{one}, \mathsf{succ} \cdot \pi_2 \cdot \langle g, s \rangle]
\end{cases}
$$

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

$$
\begin{cases}
g \cdot [\mathit{zero}, +1] = [\mathit{one}, \mathit{mul}] \cdot (\mathit{id} + \langle s, g \rangle) \\
s \cdot [\mathit{zero}, +1] = [\mathit{one}, +1 \cdot \pi_2] \cdot (\mathit{id} + \langle g, s \rangle)
\end{cases}
$$

$\equiv$ { Fokkinga }

$$\langle g, s \rangle = (\!| \langle [\mathit{one}, \mathit{mul}], [\mathit{one}, +1] \rangle |\!)$$

□

Finalmente, ao termos o resultado das duas demonstrações, combinamos os resultados com a lei de banana-split para derivar o loop e a base:

$$\langle (\!| \langle [\mathit{one}, \mathit{mul}], [+1, +1 \cdot \pi_2] \rangle |\!), (\!| \langle [\mathit{one}, \mathit{mul}], [\mathit{one}, +1 \cdot \pi_2] \rangle |\!) \rangle$$

$\equiv$ { Banana-split }

$$(\!| \langle [\mathit{one}, \mathit{mul}], [+1, +1 \cdot \pi_2] \rangle \times \langle [\mathit{one}, \mathit{mul}], [\mathit{one}, +1 \cdot \pi_2] \rangle \cdot \langle F\ \pi_1, F\ \pi_2 \rangle |\!)$$

$\equiv$ { Absorção-x, Fusão-x }

$$(\!| \langle \langle [\mathit{one}, \mathit{mul}] \cdot F\ \pi_1, [+1, +1 \cdot \pi_2] \cdot F\ \pi_1 \rangle, \langle [\mathit{one}, \mathit{mul}] \cdot F\ \pi_2, [\mathit{one}, +1 \cdot \pi_2] \cdot F\ \pi_2 \rangle \rangle |\!)$$

$\equiv$ { Def $Ff = \mathit{id} + f$ }

$$(\!| \langle \langle [\mathit{one}, \mathit{mul}] \cdot \mathit{id} + \pi_1, [+1, +1 \cdot \pi_2] \cdot \mathit{id} + \pi_1 \rangle, \langle [\mathit{one}, \mathit{mul}] \cdot \mathit{id} + \pi_2, [\mathit{one}, +1 \cdot \pi_2] \cdot \mathit{id} + \pi_2 \rangle \rangle |\!)$$

$\equiv$ { Absorção-+, Nat-id }

$$(\!| \langle \langle [\mathit{one}, \mathit{mul} \cdot \pi_1], [+1, +1 \cdot \pi_2 \cdot \pi_1] \rangle, \langle [\mathit{one}, \mathit{mul} \cdot \pi_2], [\mathit{one}, +1 \cdot \pi_2 \cdot \pi_2] \rangle \rangle |\!)$$

$\equiv$ { Lei da Troca × 2 }

$$(\!| [\langle \langle \mathit{one}, +1 \rangle, \langle \mathit{one}, \mathit{one} \rangle \rangle, \langle \langle \mathit{mul} \cdot \pi_1, +1 \cdot \pi_2 \cdot \pi_1 \rangle, \langle \mathit{mul} \cdot \pi_2, +1 \cdot \pi_2 \cdot \pi_2 \rangle \rangle] |\!)$$

$\equiv$ { for $\mathit{loop}\ \mathit{base} = (\!| [\mathit{base}, \mathit{loop}] |\!)$, Universal-+ }

$$
\begin{cases}
\mathit{base} = \langle \langle \mathit{one}, +1 \rangle, \langle \mathit{one}, \mathit{one} \rangle \rangle \\
\mathit{loop} = \langle \langle \mathit{mul} \cdot \pi_1, +1 \cdot \pi_2 \cdot \pi_1 \rangle, \langle \mathit{mul} \cdot \pi_2, +1 \cdot \pi_2 \cdot \pi_2 \rangle \rangle
\end{cases}
$$

□

Para podermos aplicar a nossa base e o nosso loop, temos de ter um par de tuplos, como a função recebe 4 inteiros sem serem tuplos, temos de utilizar uma função auxiliar *flatq* ou $unflaq$ de maneira a retornar os tipos corretos, após aplicar a função demonstrada acima.

$$
\begin{aligned}
&base = flatq \cdot \langle f, g\rangle \\
&\quad \textbf{where } f = \langle one, +1\rangle \\
&\qquad g = \langle one, one\rangle \\
&\qquad flatq\ ((a,b),(c,d)) = (a,b,c,d) \\
&loop = flatq \cdot \langle f, g\rangle \cdot unflatq \\
&\quad \textbf{where } f = \langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle \\
&\qquad g = \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle \\
&\qquad flatq\ ((a,b),(c,d)) = (a,b,c,d) \\
&\qquad unflatq\ (a,b,c,d) = ((a,b),(c,d))
\end{aligned}
$$

Problema 4

Tal como nos problemas 1 e 2 definimos agora para a estrutura FTree, o isomorfismo *in/outFTree* e o cata, ana e hylomorfismos da mesma e ainda um bifuntor que aplica duas funções, uma para as folhas da árvore e outra para o elemento nos nodos.

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

C.0.7 1 - generatePTree

Para gerar uma PTree através de um inteiro temos de aplicar um anamorfismo ao inteiro passado como argumento pela função. Aplicando o nosso "gene"após aplicar o *outNat* ao inteiro obtemos as unidades básicas da nossa árvore e o anamorfismo trata de criar o resto dos nodos da árvore. O último quadrado terá como tamanho de lado 1, os restantes seguem o cálculo $(\sqrt{2})^x * \frac{\sqrt{2}}{2} = (\sqrt{2})^{x-1}$.

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

Posto isto, desenvolvemos o seguinte diagrama:

$$
\begin{array}{ccc}
PTree & \xleftarrow[inFTree]{} & U + C \times (FTree \times FTree) \\
{\scriptstyle generatePTree\ i}\uparrow & & \uparrow{\scriptstyle id+id\times(\llbracket h\rrbracket\times\llbracket h\rrbracket)} \\
Int & \xrightarrow[h=g\cdot outNat]{} & U + C \times (Int \times Int)
\end{array}
$$

$$drawPTree = \bot$$

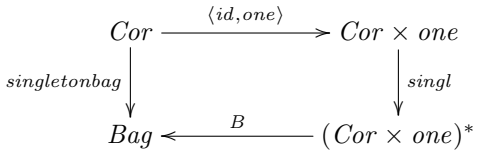
Problema 5

C.0.8 singletonbag

Esta função apenas recebe a cor e cria um Bag com apenas um berlinde dessa cor, daí o nome *singletonbag*. Para isso apenas necessitamos de criar o par com o $\langle id, count\rangle$ aplicando a identidade, ou seja, ficamos com a cor passada na função no lado esquerdo e no lado direito apenas é posto o número um representando apenas o berlinde que o saco contém. Como um Bag é uma lista contendo esses tuplos, temos de aplicar a **singl** antes de aplicar o construtor do Bag.

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

Podemos chegar ao seguinte diagrama da função:

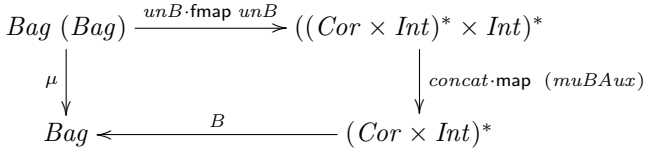


C.0.9 muB

Para multiplicarmos Bag's dentro de Bag's para devolver apenas um Bag, primeiro teremos de aplicar um *fmap* que é o functor de bags, para fazermos *unB* aos Bag's que estão no Bag. Após isto, fazemos o *unB* ao Bag, e de seguida aplicamos um *map* a esse Bag e esse map irá aplicar outro map aos Bag's desse Bag multiplicando o valor do Bag de fora com os de dentro. Esse map retorna-nos uma lista de [(a,Int)] daí fazermos o *concat* a esse tipo , e aplicando o construtor *B* ficando assim apenas com um Bag já multiplicado.

$$\mu = B \cdot concat \cdot \mathsf{map}\ \ (muBAux) \cdot unB \cdot \mathsf{fmap}\ unB$$
$$muBAux :: ([(a, Int)], Int) \to [(a, Int)]$$
$$muBAux\ (x, y) = \mathsf{map}\ \ (id \times (y*))\ x$$

O diagrama desta função demonstra-se da seguinte maneira:

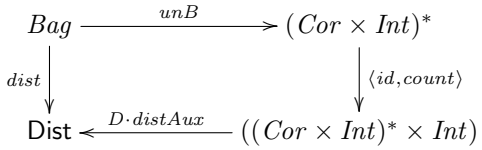


C.0.10 dist

Para saber a distribuição de cada cor num bag, primeiro temos de calcular o número de berlindes do bag, ou seja somar os berlindes de cada cor, ficando com uma lista só do segundo tuplo, e somando-a temos os berlindes todos de um Bag. Se fizermos um $\langle id, count\rangle$ obtemos um tuplo sendo o primeiro elemento a lista original do Bag de berlindes e o segundo a contagem de berlindes desse Bag. Depois basta aplicarmos um map à lista que consiste apenas em dividir o número de berlindes de cada cor pelo total de berlindes calculado. Após aplicado o map , temos uma lista de tuplos *(Cor,Float)* e basta-nos aplicar o construtor *D* para obter uma Distribuição.

$$dist = D \cdot distAux \cdot \langle id, count\rangle \cdot unB$$
$$\textbf{where}\ count = sum \cdot (\mathsf{map}\ \ \pi_2)$$
$$distAux :: ([(a, Int)], Int) \to [(a, Float)]$$
$$distAux\ (x, y) = \mathsf{map}\ \ (\lambda(x1, x2) \to (x1, toFloat\ (x2)\ /\ toFloat\ (y)))\ x$$

Podemos então desenhar um diagrama que traduz a nossa função:

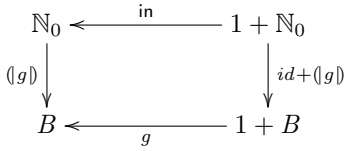


D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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:



[7] Exemplos tirados de [2].

Índice