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

Departamento de Informática
Universidade do Minho

Julho de 2018

Grupo nr.	086
a80789 Rui Azevedo	

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

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

vai ser seleccionado pelo GHCi para ser executado.

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

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.

$$\mathbf{type}\ \mathit{MagicNo} = \mathit{String}$$
$$\mathbf{type}\ \mathit{Time} = \mathit{Int} \quad \text{-- em milisegundos}$$
$$\mathbf{type}\ \mathit{Entity} = \mathit{String}$$
$$\mathbf{type}\ \mathit{Value} = \mathit{Int}$$

Neste contexto, implemente as seguintes funções:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Problema 2

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

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

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:

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

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

```
( 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.

(a) Bitmap de exemplo.

(b) Rotação.

(c) Redimensionamento.

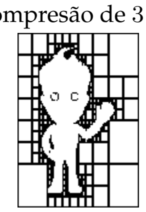
(d) Inversão de cores.

(e) Compresão de 1 nível.

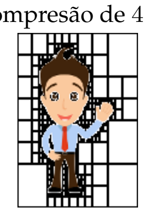
(f) Compresão de 2 níveis.

(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.

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"
```

   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"
```

[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.

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,

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

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

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

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

e

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

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

$$s\ 0 = 1$$
$$s\ (d+1) = s\ d + 1$$

A partir daqui alguém derivou a seguinte implementação:

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

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):*

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

Problema 4

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

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

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

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

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

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

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

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

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

```
> animatePTree 3
```

Problema 5

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

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

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

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

que habita o tipo genérico dos "bags":

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

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

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

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 }.
```

Mais ainda, se quisermos saber o total de berlindes em $bagOfMarbles$ basta calcular $\mathsf{fmap}\ (!)\ bagOfMarbles$ 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,

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

1. Defina a função μ (multiplicação do mónade Bag) e a função auxiliar $singletonbag$.

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

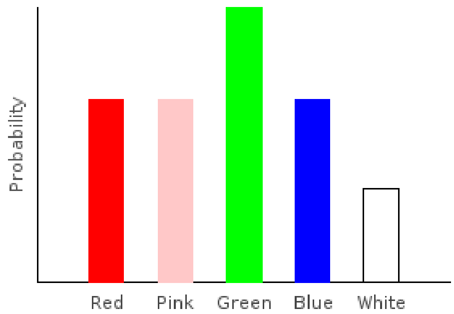


Figura 4: Distribuição de berlindes num saco.

2. Verifique-as com os seguintes testes unitários:

<u>Teste unitário</u> 2 *Lei $\mu \cdot return = id$:*

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

<u>Teste unitário</u> 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

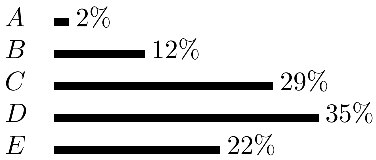
A Mónade para probabilidades e estatística

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

Ainda sobre o mónade Bag:

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

O exemplo do texto:

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

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

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

Outras funções auxiliares:

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

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

As funções definidas para o *BlockChain* foram todas resolvidas usando catamorfismos O seguinte diagrama representa o tipo genérico dos catamorfismos de *BlockChain*

$$\begin{tikzcd} \mathit{BlockChain} \arrow[d, "(\!|g|\!)"'] & \mathit{Block} + \mathit{Block} \times \mathit{BlockChain} \arrow[l, "\mathit{inBlockChain}"'] \arrow[d, "\mathit{id} + \mathit{id} \times (\!|g|\!)"] \\ A & \mathit{Block} + \mathit{Block} \times A \arrow[l, "g"] \end{tikzcd}$$

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

C.0.1 allTransactions

A função *allTransactions* devolve uma lista com todas as transações feitas no BlockChain.

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

C.0.2 ledger

A função *ledger* devolve uma lista de pares com a correspondência *Entidade - Valor*. Para isso, primeiro foi calculada a lista de todas as transações do BlockChain através do catamorfismo *allTransactions* e, de seguida, é calculado o ledger dessa lista através do catamorfismo *ledgerTrans*. O gene do catamorfismo é um either em que a primeira função é a função que cria a lista vazia(*nil*) e a segunda função mapeia a lista de ledgers já existentes e vai inserindo novos ledgers à lista.

$$\begin{aligned}
&ledger &&= ledgerTrans \cdot allTransactions\\
&ledgerTrans &&= cataList\ \$\ [nil, \widehat{g}]\\
&g\ (e_1, (v, e_2)) &&= \mathsf{map}\ \ (\lambda(entity, value) \rightarrow \mathbf{if}\ e_1 \equiv entity\ \mathbf{then}\ (entity, value - v)\\
& && \mathbf{else\ if}\ e_2 \equiv entity\ \mathbf{then}\ (entity, value + v)\\
& && \mathbf{else}\ (entity, value))
\end{aligned}$$

C.0.3 isValidMagicNr

A função *isValidMagicNr* devolve um valor booleano que representa a unicidade dos números mágicos do BlockChain. É primeiro feito um catamorfismo para criar a lista com todos os números mágicos e, a essa lista, é aplicada a função *allDiferent* que vai verificar se a lista de números tem elementos repetidos.

$$\begin{aligned}
&isValidMagicNr = allDiferent \cdot (cataBlockchain\ \$\ [singl \cdot \pi_1, cons \cdot (\pi_1 \times id)])\\
&allDiferent\ (h : t)\ |\ elem\ h\ t = False\\
&\qquad\qquad\qquad\quad |\ otherwise = allDiferent\ t\\
&allDiferent\ _ \quad\ \ = True
\end{aligned}$$

Problema 2

Os problemas desta secção foram resolvidos através de catamorfismos. Algumas definições são muito parecidas com a definição do fmap, a diferença é o valor que esse catamorfismo atinge. O fmap está definido para aplicar uma função f ao primeiro componente do *Cell* (conteúdo do pixel), e algumas definições das funções abaixo simplesmente aplicam uma função f à segunda componete do *Cell*.

$$\begin{aligned}
&uncurryCell\ f\ (a, (x, y)) = f\ \ a\ \ x\ \ y\\
&uncurryBlock\ f\ (q1, (q2, (q3, q4))) = f\ \ q1\ \ q2\ \ q3\ \ q4\\
&inQTree = [uncurryCell\ (Cell), uncurryBlock\ (Block)]\\
&outQTree\ (Cell\ \ a\ \ x\ \ y) = i_1\ (a, (x, y))\\
&outQTree\ (Block\ \ q1\ \ q2\ \ q3\ \ q4) = i_2\ (q1, (q2, (q3, q4)))\\
&baseQTree\ f\ \ g = (f \times id) + (g \times (g \times (g \times g)))\\
&recQTree\ \ \ g \quad = id + (g \times (g \times (g \times g)))\\
&cataQTree\ g \quad = g \cdot recQTree\ (cataQTree\ g) \cdot outQTree\\
&anaQTree\ g \quad = inQTree \cdot recQTree\ (anaQTree\ g) \cdot g\\
&hyloQTree\ h\ g = cataQTree\ h \cdot anaQTree\ g\\
&\mathbf{instance}\ Functor\ \ QTree\ \mathbf{where}\\
&\quad \mathsf{fmap}\ f = cataQTree\ (inQTree \cdot baseQTree\ f\ \ id)
\end{aligned}$$

C.0.4 rotateQTree

A função *rotateQTree* roda uma imagem 90 graus no sentido dos ponteiros do relógio. O objectivo é trocar a ordem dos ramos da quadtree que representa a imagem mantendo o conteúdo das folhas da àrvore. O processo é muito semelhante ao da função *mirror* das *Leaf Trees*, que espelha uma árvore fazendo swap dos forks da árvore.

$$\begin{aligned}
&rotateQTree = cataQTree\ (inQTree \cdot (id \times swap + swapQ))\\
&\quad \mathbf{where}\ swapQ\ (q1, (q2, (q3, q4))) = (q3, (q1, (q4, q2)))
\end{aligned}$$

C.0.5 scaleQTree

A função *scaleQTree* redimensiona uma determinada imagem. Uma folha da árvore contém informação sobre o conteúdo do pixel assim como a dimensão da região que contém os mesmos pixeis, logo, para redimensionar a árvore, simplesmente temos que multiplicar o valor da região por um dado fator multiplicativo.

$$scaleQTree\ d = cataQTree\ (inQTree \cdot (id \times ((d*) \times (d*)) + id))$$

C.0.6 invertQTree

A função *invertQTree* inverte as cores de uma imagem tornando a imagem monocromática. O conteúdo dos pixeis é representado com o construtor PixelRGBA8 que aceita quatro argumentos: r (red), g(green), b(blue), a(alpha). De modo a inverter as cores, utilizamos o fmap para mapear a árvore substraindo 255 pela primeira componente da $Cell$.

$$invertQTree = \mathsf{fmap}\ (\lambda(PixelRGBA8\ r\ g\ b\ a) \rightarrow (PixelRGBA8\ (255 - r)\ (255 - g)\ (255 - b)\ (255 - a)))$$

C.0.7 compressQTree

A função *compressQTree* comprime a árvore num determinado número de níveis. O algoritmo usado foi desenvolvido a pensar no número de níveis que a árvore fica e não em quantas níveis foram cortados. Quando chegamos ao ponto de querer fazer o corte da árvore, essa parte da árvore é convertida para a matriz correspondente para obtermos informação sobre a dimensão que a $Cell$ vai ter.

$$compressQTree\ n\ q = compress\ ((depthQTree\ q) - n)\ q$$
$$compress\ \_\ (Cell\ a\ x\ y) = Cell\ a\ x\ y$$
$$compress\ n\ (Block\ q1\ q2\ q3\ q4) \mid n \leqslant 0 = createCell\ \$\ qt2bm\ (Block\ q1\ q2\ q3\ q4)$$
$$\quad \mid otherwise = Block\ (compress\ (n-1)\ q1)\ (compress\ (n-1)\ q2)\ (compress\ (n-1)\ q3)\ (compress\ (n-1)\ q4$$
$$\quad\quad \textbf{where}\ createCell\ m = Cell\ (getElem\ 1\ 1\ m)\ (nrows\ m)\ (ncols\ m)$$

C.0.8 outlineQTree

A função *outlineQTree* desenha o contorno de uma determinada imagem. O exercício trabalha com uma árvore do tipo $QuadTree\ Bool$, deste modo, as folhas da árvore contêm apenas dois valores possíveis, $True$ ou $False$. Numa primeira fase, a árvore tem que ser mapeada com uma função f, que determina quais são os píxeis de fundo da imagem. O algoritmo usado para desenhar o contorno da árvore passa por ir a cada folha da árvore e, caso o conteúdo do pixel seja $True$, converte-se essa parte da árvore no bitmap correspondente. A conversão vai resultar numa matriz com dimensão l c cujos elementos são todos True. Os elementos das linhas 1 e l e das colunas 1 c mantêm todos com o mesmo valor, o resto dos elementos passam todos a $False$. Após este processo de alteração da matriz converte-se o bitmap novamente para a QuadTree correspondente.

A primeira função derivada foi a seguinte:

$$outlineQTree\ f = qt2bm \cdot nmap \cdot \mathsf{fmap}\ f\ \textbf{where}\ nmap = cataQTree\ [contorno, inQTree \cdot i_2]$$

□

Contúdo, aplicando a lei absorção-cata das leis do cálculo de programas, podemos deriviar a seguinte função:

$$outlineQTree\ f = qt2bm \cdot nmap \cdot \mathsf{fmap}\ f$$
$$\equiv \quad \{\ \text{Absorção-cata}\ \}$$
$$outlineQTree\ f = qt2bm \cdot cataQTree\ \$\ [contorno, inQTree \cdot i_2] \cdot (baseQTree\ f\ id)$$

□

Deste modo, fazemos o trabalho todo dentro do catamorfismo.

$$\begin{array}{l} \mathit{outlineQTree}\ f = \mathit{qt2bm} \cdot (\mathit{cataQTree}\ \$\ [\mathit{contorno}, \mathit{inQTree} \cdot i_2] \cdot (\mathit{baseQTree}\ f\ \mathit{id})) \\ \mathit{contorno}\ (b, (x, y)) \mid b \equiv \mathit{True} \wedge x \geqslant 2 \wedge y \geqslant 2 = \mathit{bm2qt}\ \$ \\ \quad \mathit{matrix}\ x\ y\ \$ \\ \quad (\lambda(i, j) \rightarrow \textbf{if}\ i \equiv 1 \vee i \equiv x \vee j \equiv 1 \vee j \equiv y\ \textbf{then}\ \mathit{True}\ \textbf{else}\ \mathit{False}) \\ \quad \mid \mathit{otherwise} = \mathit{Cell}\ b\ x\ y \end{array}$$

Problema 3

O objectivo do exercício é escrever num único ciclo *for* o cálculo das combinações de n k-a-k. Começamos por derivar através da recursividade múltipla as funções *f k* e *l k* e as funções *g* e *s* como catamorfismos. Chegamos à definição das seguintes funções:

$$\langle f\ k, l\ k\rangle = (\![[\langle one, k+1\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_1\rangle]]\!)$$

$$\langle g, s\rangle = (\![[\langle one, one\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle]]\!)$$

$\square$

Com as seguintes derivações conseguimos usar as leis do cálculo de programas para derivar as funções *base k* e *loop*.

$$\begin{array}{l} \mathsf{for}\ loop\ (base\ k) = \langle (\![[\langle one, one\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle]]\!), (\![[\langle one, k+1\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_1\rangle]]\!)\rangle \\ \equiv \qquad \{\ \text{banana-split}\ \} \\ \mathsf{for}\ loop\ (base\ k) = (\![([\langle one, one\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle] \times [\langle one, \underline{(k+1)}\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle]) \cdot \langle F\ \pi_1, F\ \pi_2\rangle]\!) \\ \equiv \qquad \{\ \text{absorção-x}\ \} \\ \mathsf{for}\ loop\ (base\ k) = (\![\langle [\langle one, one\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle] \cdot F\ \pi_1, [\langle one, \underline{(k+1)}\rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2\rangle] \cdot F\ \pi_2\rangle]\!) \\ \equiv \qquad \{\ \text{absorção-+ , natural-id , fusão-x}\ \} \\ \mathsf{for}\ loop\ (base\ k) = (\![\langle [\langle one, one\rangle, \langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle], [\langle one, \underline{(k+1)}\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle]\rangle]\!) \\ \equiv \qquad \{\ \text{Lei da troca}\ \} \\ \mathsf{for}\ loop\ (base\ k) = (\![[\langle\langle one, one\rangle, \langle one, \underline{(k+1)}\rangle\rangle, \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle]]\!) \\ \equiv \qquad \{\ \text{Pela definição do catamorfismo do ciclo for}\ \} \\ \left\{ \begin{array}{l} base\ k = \langle\langle one, one\rangle, \langle one, \underline{(k+1)}\rangle\rangle \\ loop = \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle \end{array} \right. \end{array}$$

$\square$

A função derivada trabalha com duplos de duplos mas o pedido é trabalhar com 4-uplos. De modo a não alterar a função derivada, usamos as funções *untuple* e *tuple* para a função continuar a trabalhar com duplos de duplos e o resultado ser entregue num 4-uplo.

$$\begin{array}{l} base\ k = (1, k+1, 1, 1) \\ loop = untuple \cdot \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle \cdot tuple \\ \quad \textbf{where}\ untuple\ ((a, b), (c, d)) = (a, b, c, d) \\ \qquad tuple\ (a, b, c, d) = ((a, b), (c, d)) \end{array}$$

Problema 4

$$
\begin{array}{l}
\mathit{uncurryComp}\ f\ (a,(f1,f2)) = f\ a\ f1\ f2 \\
\mathit{inFTree} = [\mathit{Unit},\mathit{uncurryComp}\ (\mathit{Comp})] \\
\mathit{outFTree}\ (\mathit{Unit}\ b) = i_1\ b \\
\mathit{outFTree}\ (\mathit{Comp}\ a\ f1\ f2) = i_2\ (a,(f1,f2)) \\
\mathit{baseFTree}\ f\ g\ h = g + f \times (h \times h) \\
\mathit{recFTree}\ f \quad = id + id \times (f \times f) \\
\mathit{cataFTree}\ g \quad = g \cdot \mathit{recFTree}\ (\mathit{cataFTree}\ g) \cdot \mathit{outFTree} \\
\mathit{anaFTree}\ g \quad = \mathit{inFTree} \cdot \mathit{recFTree}\ (\mathit{anaFTree}\ g) \cdot g \\
\mathit{hyloFTree}\ h\ g = \mathit{cataFTree}\ h \cdot \mathit{anaFTree}\ g \\
\textbf{instance}\ \mathit{Bifunctor}\ \mathit{FTree}\ \textbf{where} \\
\quad \mathit{bimap}\ f\ g = \mathit{cataFTree}\ (\mathit{inFTree} \cdot (\mathit{baseFTree}\ f\ g\ id))
\end{array}
$$

C.0.9 generatePTree

A função *generatePTree* vai criar uma árvore de pitágoras de uma certa ordem. O anamorfismo definido abaixo, numa primeira fase é criado um para cujo primeiro componente é a ordem dada e o segundo componente vai ser a dimensão do primeiro quadrado que vai ser gerado. De seguida, dentro do anamorfismo, o inteiro que a função recebe é decrementado sucessivamente até o valor de 0 e a constante inicial é sucessivamente multiplicada pela escala $\sqrt{2}/2$. Quando o anamorfismo termina é criada a árvore de pitágoras, uma árvore binária balanceada.

$$
\begin{array}{l}
\mathit{generatePTree} = \mathit{anaFTree}\ (\mathit{create} \cdot (\mathit{outNat} \times id)) \cdot \langle id, \underline{50} \rangle \\
\mathit{create}\ (i_1\ (), x) = i_1\ x \\
\mathit{create}\ (i_2\ n, x) = i_2\ (x, (\mathit{calculate}, \mathit{calculate})) \\
\quad \textbf{where}\ \mathit{calculate} = (n, ((\mathit{sqrt}\ 2)\ /\ 2) * x)
\end{array}
$$

C.0.10 drawPTree

A função *drawPTree* desenha, através da biblioteca *Gloss*, a árvore de pitágoras. Isto é feito desenhando sucessivamente o quadrado base e os outros dois seguintes que correspondem a uma rotação de 315 graus (quadrado da esquerda) e 45 graus (quadrado da direita), e uma translação de *a / 2* em relação ao eixo dos xx e *a* em relação aos eixos dos yy.

$$
\begin{array}{l}
\mathit{drawPTree} = \mathit{cataFTree}\ [\mathit{singl} \cdot \mathit{square}, \mathit{multiSquare}] \\
\mathit{multiSquare}\ (a, ([\,],[\,])) = [\,] \\
\mathit{multiSquare}\ (a, ((\mathit{sl} : \mathit{sls}), (\mathit{sr} : \mathit{srs}))) = (\mathit{Pictures}\ ([\mathit{Line}\ [(-a\ /\ 2, -a\ /\ 2), (-a\ /\ 2, a\ /\ 2), (a\ /\ 2, a\ /\ 2), (a\ /\ 2, -a \\
\quad [\mathit{translate}\ (-a\ /\ 2)\ a\ \$\ \mathit{rotate}\ 315\ \mathit{sl}] + \\
\quad [\mathit{translate}\ (a\ /\ 2)\ a\ \$\ \mathit{rotate}\ 45\ \mathit{sr}])) : \mathit{multiSquare}\ (a, (\mathit{sls}, \mathit{srs}))
\end{array}
$$

Problema 5

C.0.11 singletonBag

A função *singletonBag* recebe um tipo A e coloca-o dentro de um bag unitário. A função pode ser equiparada ao conjunto elementar da Teoria de Conjuntos do ramo da Matemática.

$$\mathit{singletonbag}\ a = B\ [(a, 1)]$$

C.0.12 muB

A função *muB* converte um bag de um bag num bag apenas. Um bag é representado pelo seguinte tipo: **data** $\mathit{Bag}\ a = [(a, \mathit{Int})]$ e, por consequência, um bag de uma bag é representado da seguinte maneira: $\mathit{Bag}\ (\mathit{Bag}\ a) = [(\mathit{Bag}\ a, \mathit{Int})]$. Os pares da lista representam a cardinalidade dos bags que estão contidas na lista. Para sumariarmos tudo num só bag, mapeamos os bags de cada par e multiplicamos os respetivos segundos componentes pelo número de bags que existem desse tipo. Por fim concatenamos as listas numa só e associamos o construtor dos bags.

$$\mu = B \cdot concat \cdot joinBag \cdot unB$$

$$\begin{array}{l} joinBag\ [\,] = [\,] \\ joinBag\ ((bag, no) : bags) = (\mathsf{map}\ \ (id \times (*no))\ (unB\ \ bag)) : joinBag\ \ bags \end{array}$$

C.0.13 dist

A função *dist* cálcula a distribuição probabilística de uma bag. Cada segundo componente de um par é dividido pelo número total de elementos de um bag.

$$\begin{array}{l} dist\ (B\ \ a) = D\ \$\ \mathsf{map}\ \ (\lambda(a, i) \rightarrow (a, (/)\ (toFloat\ \ i)\ (toFloat\ \ x)))\ a \\ \quad \mathbf{where}\ x = sum\ \$\ \mathsf{map}\ \ \pi_2\ \ a \end{array}$$

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

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

[7]Exemplos tirados de [?].