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

Departamento de Informática
Universidade do Minho

Julho de 2018

Grupo nr.	109
a82088	Luís Tiago Machado Braga
a82300	João Filipe da Costa Nunes
a82298	Luís Guilherme Gonçalves Macedo da Silva Martins

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.

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

Neste contexto, implemente as seguintes funções:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Problema 2

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

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

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

(a) Matriz de exemplo bm.

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

(b) Quadtree de exemplo qt.

Figura 1: Exemplos de representações de bitmaps.

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

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

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

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

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

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

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

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

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

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

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

Esta questão aborda operações de processamento de imagens utilizando quadtrees:

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

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

[2]Cf. módulo $Data.Matrix$.

[3]Segundo um ângulo de 90º no sentido dos ponteiros do relógio.

[4]Multiplicando o seu tamanho pelo valor recebido.

[5]Um pixel pode ser invertido calculando $255 - c$ para cada componente c de cor RGB, exceptuando o componente alpha.

(a) Bitmap de exemplo.

(b) Rotação.

(c) Redimensionamento.

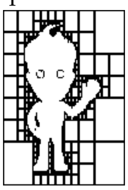
(d) Inversão de cores.

(e) Compresão de 1 nível.

(f) Compresão de 2 níveis.

(g) Compresão de 3 níveis.

(h) Compresão de 4 níveis.



(i) Bitmap de contorno.

(j) Bitmap com contorno.

Figura 2: Manipulação de uma figura bitmap utilizando quadtrees.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Problema 3

O cálculo das combinações de n k-a-k,

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

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

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

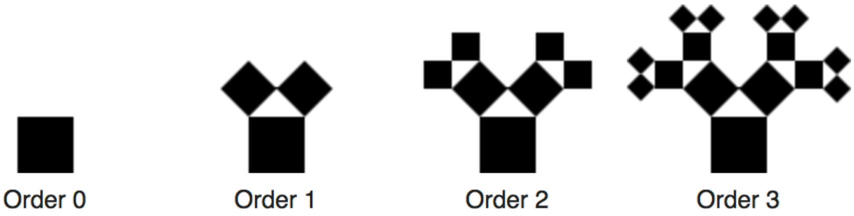


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

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

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

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

e

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

A partir daqui alguém derivou a seguinte implementação:

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

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

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

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

Problema 4

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

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

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

1. Defina a função $generatePTree :: Int \rightarrow PTree$, 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 $drawPTree :: PTree \rightarrow [Picture]$, 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:

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

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 }.
```

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

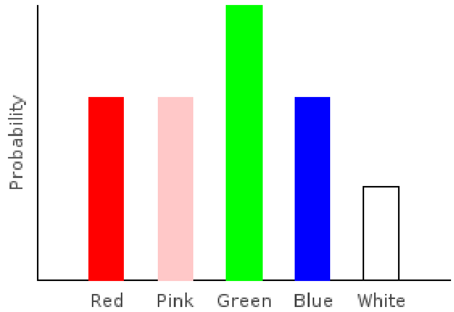


Figura 4: Distribuição de berlindes num saco.

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

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

$$marblesDist = dist\ bagOfMarbles$$

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

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

cf. Figura 4.

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

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

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

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

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

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

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

   onde $b3$ é um saco dado em anexo.

Anexos

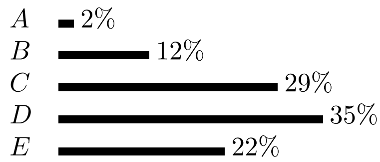
A Mónade para probabilidades e estatística

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

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

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

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



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

B Definições auxiliares

Funções para mostrar *bags*:

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

Igualdade de *bags*:

```
instance (Eq a) ⇒ Eq (Bag a) where
  b ≡ b' = (unB b) `lequal` (unB b')
    where lequal a b = isempty (a ⊖ b)
      ominus a b = a ++ neg b
      neg x = [(k, −i) | (k, i) ← x]
```

Ainda sobre o mónade Bag:

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

O exemplo do texto:

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

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

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

Outras funções auxiliares:

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

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

In e Out de Blockchain

$$\mathit{inBlockchain} = [\mathit{Bc}, \mathit{Bcs}]$$
$$\mathit{outBlockchain}\ (\mathit{Bc}\ a) = i_1\ a$$
$$\mathit{outBlockchain}\ (\mathit{Bcs}\ b) = i_2\ b$$

Anamorfismo, Catamorfismo e Hilomorfismo de uma Blockchain

$$\mathit{recBlockchain}\ f = \mathit{id} + \mathit{id} \times f$$
-- Catamorfismo
$$(\!| g |\!) = g \cdot \mathit{recBlockchain}\ (\!| g |\!) \cdot \mathit{outBlockchain}$$
-- Anamorfismo
$$[\!(g)\!] = \mathit{inBlockchain} \cdot \mathit{recBlockchain}\ [\!(g)\!] \cdot g$$
$$\mathit{hyloBlockchain}\ h\ g = (\!| h |\!) \cdot [\!(g)\!]$$

Resoluções

Resolução da função *allTransactions*

Representando a função através do seguinte catamorfismo:

$$\begin{tikzcd}
Blockchain \arrow[rr, bend left, "outBlockchain"] \arrow[d, "(\!|allTransactions|\!)"'] \arrow[r, phantom, "\cong"] & & Block + Block \times Blockchain \arrow[ll, bend left, "inBlockchain"] \arrow[d, "id+id\times(\!|allTransactions|\!)"] \\
Transactions & & Block + Block \times Transactions \arrow[ll, "{[f, h \cdot (f \times id)]}"]
\end{tikzcd}$$

Pode-se concluir que apenas é necessária uma função f que retire de cada $Block$ a $Transactions$ e uma h que junte a $Transactions$ já retirada às $Transactions$ da $Blockchain$ já processada. Tendo em conta a construção de $Block = (MagicNr, (Time, (Transactions)))$}, a primeira função será a composição:

$$f = \pi_2 \cdot \pi_2$$

e a segunda:

$$h = \mathsf{conc}$$
$$relembrando\ que : \mathsf{conc} :: (a, [\,a\,]) \rightarrow [\,a\,]$$

Desta forma a função $allTransactions$ será definida da seguinte maneira:

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

Resolução da função *ledger*

Tendo em vista o objectivo da função, que como o nome indica, é o de calcular o ledger de cada entidade, podemos usar o catamorfismo definido na função *allTransactions*.

Relembrando a definição da $Transactions$

$$Transactions = [\,(Entity, (Value, Entity))\,]$$

podemos deduzir que a solução poderá passar por fazer a fissão de cada elemento desta lista, ou seja, cada $Transaction :: (Entity\ (Value, Entity))$ passa a ser do tipo $[\,(Entity, Value), (Entity, Value)\,]$, que por si já possui o mesmo tipo do conjunto de chegada da função *ledger* ($Ledger = [\,(Entity, Value)\,]$).

Desenhando o diagrama do isomorfismo do tipo $Transactions$

$$\begin{tikzcd}
Transactions \arrow[r, phantom, "\cong"] \arrow[r, bend left, "out"] & 1 + Transation \times Transactions \arrow[l, bend left, "\mathbf{in}"]
\end{tikzcd}$$

Podemos inferir que a separação dos elementos é um catamorfismo que se denominou por *transSeparator* que pode ser representado da seguinte forma

$$\begin{tikzcd}
Transaction \arrow[rr, bend left, "out"] \arrow[d, "(\!|transSeparator|\!)"'] \arrow[r, phantom, "\cong"] & & 1 + Transaction \times Transactions \arrow[ll, bend left, "\mathbf{in}"] \arrow[d, "id+id\times(\!|transSeparator|\!)"] \\
Ledger & & 1 + Transaction \times Ledger \arrow[ll, "g"]
\end{tikzcd}$$

Todavia, falta definir o g (generator). Que terá de ser do tipo

$$g = [f, k]$$

em que

$$f = nil\ e\ k = \mathsf{conc} \cdot (transDivide \times id)$$

na qual $transDivide$ será a função que faz a fissão de uma $Transaction$. Função essa que pode ser definida da seguinte forma:

$$transDivide\ (a,(b,c)) = [(a,-b),(c,b)]$$

Por consequência, a função $transSeparator$ pode ser definida da seguinte maneira:

$$transSeparator = (\![[nil, \mathsf{conc} \cdot (transDivide \times id)]]\!)$$

Depois de aplicada esta função apenas falta somar o valor transacionado inerente a cada entidade, de forma a obter o seu balanço total numa determinada $Blockchain$.

Tal acção pode ser levada a cabo, aplicando o mecanismo:

$$consolLedger = \mathsf{map}\ \ (id \times sum) \cdot col$$

Que nos faz chegar à solução final.

Desta forma a função $ledger$ poderá ser assim definida:

$$ledger = consolLedger \cdot transSeparator \cdot allTransactions$$

Resolução da função *isValidMagicNr*

O principal objetivo desta função é verificar se todos os números da Blockchain são válidos ou seja, se nenhum número mágico *MagicNo* aparece mais que uma vez. Para tal foi usado um anamorfismo que é iniciado com uma lista vazia, onde à medida em que passa pela Blockchain verifica-se se o *MagicNo* está nesta lista em caso afirmativo, o Block é substituido por:

$$Block = (\texttt{"True"},(0,[\,]))$$

Onde, neste caso o *MagicNo* é substituido por True sendo este adicionado à lista, o número mágico é portanto válido. Em caso negativo o *MagicNo* já se encontra na lista, o Block é portanto substituido por:

$$Block = (\texttt{"False"},(0,[\,]))$$

Desta forma obtemos uma BlockChain que está populado com True e False, no passo seguinte é verificado se ao longo do BlockChain todos os *MagiNo* são True.

O raciocínio do grupo encontra-se expresso nos seguintes diagramas, anamorfismo e catamorfismo respetivamente:

$$\begin{tikzcd}
Blockchain \arrow[r, "magic2BoolGen"'] \arrow[d, "{[\!(magic2BoolGen)\!]}"'] & Block + Block \times Blockchain \arrow[d, "{id+id\times[\!(magi2BoolGen)\!]}"] \\
Blockchain & Block + Block \times Blockchain \arrow[l, "inBlockchain"]
\end{tikzcd}$$

$$\begin{tikzcd}
Blockchain \arrow[r, "outBlockchain"'] \arrow[d, "{(\!|validCheckGen|\!)}"'] & Block + Block \times Blockchain \arrow[d, "{id+id\times(\!|validCheckGen|\!)}"] \\
Bool & Block + Block \times Bool \arrow[l, "validCheckGen"]
\end{tikzcd}$$

No fim foram traduzidos num hilomorfismo cujos genes são os seguintes:

$$\begin{array}{l}
magic2BoolGen :: ([MagicNo], Blockchain) \rightarrow Block + (Block, ([MagicNo], Blockchain)) \\
magic2BoolGen\ (l, Bc\ a) = \mathbf{if}\ (elem\ (\pi_1\ a)\ l)\ \mathbf{then}\ i_1\ (\texttt{"False"},(0,[\,]))\ \mathbf{else}\ i_1\ (\texttt{"True"},(0,[\,])) \\
magic2BoolGen\ (l, Bcs\ b) = \mathbf{if}\ (elem\ p\ l)\ \mathbf{then}\ i_2\ i2fals\ \mathbf{else}\ i_2\ i2verd \\
\quad \mathbf{where}\ p = (\pi_1 \cdot \pi_1)\ b \\
\quad\quad pross \ \ = [p] +\!\!+\ l \\
\quad\quad verd \ \ = (\texttt{"True"},(0,[\,])) \\
\quad\quad fals \ \ = (\texttt{"False"},(0,[\,])) \\
\quad\quad i2verd = (verd, (pross, \pi_2\ b)) \\
\quad\quad i2fals = (fals, (l, \pi_2\ b))
\end{array}$$

$$\begin{array}{l}
validCheckGen = [bc, bcs]\ \mathbf{where} \\
\quad bc\ a = (\pi_1\ a \equiv \texttt{"True"}) \\
\quad bcs\ b = ((\pi_1 \cdot \pi_1)\ b \equiv \texttt{"True"} \wedge \pi_2\ b)
\end{array}$$

$$isValidMagicNr = curry\ (hyloBlockchain\ validCheckGen\ magic2BoolGen)\ [\,]$$

Problema 2

In e Out de QuadTree

$$
\begin{array}{l}
inQTree = [cell, block] \ \textbf{where} \ cell \ (a,(b,c)) = Cell \ a \ b \ c \\
\quad block \ (a,(b,(c,d))) = Block \ a \ b \ c \ d \\
outQTree \ (Cell \ a \ b \ c) = i_1 \ (a,(b,c)) \\
outQTree \ (Block \ d \ e \ f \ g) = i_2 \ (d,(e,(f,g)))
\end{array}
$$

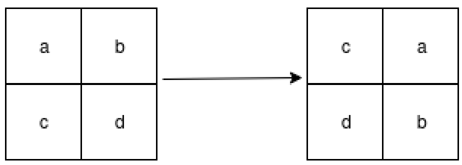
Anamorfismo, Catamorfismo, Hilomorfismo, functor base e map de uma QuadTree

$$
\begin{array}{l}
baseQTree \ f \ g = f \times id + g \times (g \times (g \times g)) \\
recQTree \ f = id + f \times (f \times (f \times f)) \\
\quad \text{-- Catamorfismo} \\
(\!| g |\!) = g \cdot recQTree \ (\!| g |\!) \cdot outQTree \\
\quad \text{-- Anamorfismo} \\
[\!(g)\!] = inQTree \cdot recQTree \ [\!(g)\!] \cdot g \\
hyloQTree \ h \ g = (\!| h |\!) \cdot [\!(g)\!] \\
\textbf{instance} \ Functor \ QTree \ \textbf{where} \\
\quad \mathsf{fmap} \ f = (\!| (inQTree \cdot (baseQTree \ f \ id)) |\!)
\end{array}
$$

Resoluções

Resolução da função *rotateQTree*

Pretende-se com esta função aplicar uma rotação a uma $QTree$, de modo que, aplicando tal função numa imagem traduzida por uma $QTree$ esta rode no sentido dos ponteiros do relógio da seguinte maneira:



Para efeito basta usar um Catamorfismo que aplica a troca. Pode-se assim deduzir que a função $rotateQTree$ é um Catamorfismo, cujo seu *generator* e a própria função podem ser definidos da seguinte forma:

$$
\begin{array}{l}
qSwapperGen :: (a,(Int,Int)) + (QTree \ a,(QTree \ a,(QTree \ a, QTree \ a))) \to QTree \ a \\
qSwapperGen = [cell, rigthrotation] \ \textbf{where} \ cell \ (a,(b,c)) = Cell \ a \ c \ b \quad \text{-- as dimensões trocam} \\
\quad rigthrotation \ (a,(b,(c,d))) = Block \ c \ a \ d \ b \quad \text{-- rotação no sentido dos ponteiros} \\
rotateQTree = (\!| qSwapperGen |\!)
\end{array}
$$

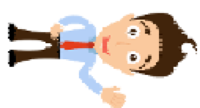


Figura 5: Resultado do grupo do rotateBMP

Resolução da função *scaleQTree*

O objectivo desta função é alterar o tamanho de uma $Cell$ segundo um factor dado. Para tal, definiu-se o Anaformismo:

$$\begin{tikzcd}
QTree \arrow[r, "scaleGen"] \arrow[d, "{[\![scaleQTree]\!]}"'] & Cell + QTree \times ((Int, QTree) \times ((Int, QTree), (Int, QTree))) \arrow[d, "{id + [\![scaleQTree]\!] \times ([\![scaleQTree]\!] \times ([\![scaleQTree]\!] \times [\![scaleQTree]\!]))}"] \\
QTree \arrow[r, bend left, "outQTree"] \arrow[r, phantom, "\cong"] & Cell + QTree \times (QTree \times (QTree \times QTree)) \arrow[l, bend left, "inQTree"]
\end{tikzcd}$$

Em que $scaleGen$ e $scaleQTree$ são definidos do seguinte modo:

$$\mathbf{type}\ PairQTInt\ t = (Int, QTree\ t)$$

$$scaleGen :: PairQTInt\ t \to (t, (Int, Int)) + (PairQTInt\ t, (PairQTInt\ t, (PairQTInt\ t, PairQTInt\ t)))$$
$$scaleGen\ (x, Cell\ a\ b\ c) = i_1\ (a, (b * x, c * x))$$
$$scaleGen\ (x, Block\ a\ b\ c\ d) = i_2\ ((x, a), ((x, b), ((x, c), (x, d))))$$

$$scaleQTree = curry\ [\![scaleGen]\!]$$

Figura 6: Resultado do grupo do scaleBMP

Resolução da função *invertQTree*

A função $invertQTree$ tem como propósito a inversão de cores de uma $QTree\ PixelRGBA8$.

$$\begin{array}{l}
Relembrando\ o\ construtor \\
PixelRGBA8 :: Pixel8 \to Pixel8 \to Pixel8 \to Pixel8 \to PixelRGBA8 \\
em\ que \\
Pixel8 = red\ green\ blue\ alpha, respectivamente \cdot
\end{array}$$

Visto que, para inverter um pixel basta subtrair 255-c para cada componente c de cor RGB, excepto para alpha. Desta forma, é possível deduzir o Catamorfismo que faz a inversão. Catamorfismo esse representado no diagrama abaixo.

$$\begin{tikzcd}
QTree\ PixelRGBA8 \arrow[r, bend left, "outQTree"] \arrow[r, phantom, "\cong"] \arrow[d, "{(\!| invertQTree |\!)}"'] & Cell + QTree \times (QTree \times (QTree \times QTree)) \arrow[l, bend left, "inQTree"] \arrow[d, "{id + (\!| invertQTree |\!) \times ((\!| invertQTree |\!) \times ((\!| invertQTree |\!) \times (\!| invertQTree |\!)))}"] \\
QTree\ PixelRGBA8 & Cell + QTree \times (QTree \times (QTree \times QTree)) \arrow[l, "invertGen"]
\end{tikzcd}$$

Cuja definição das funções é dada da seguinte maneira,

$$\mathbf{type}\ QTPixel = QTree\ PixelRGBA8$$

$$invertGen :: (PixelRGBA8, (Int, Int)) + (QTPixel, (QTPixel, (QTPixel, QTPixel))) \to QTPixel$$

$$invertGen = inQTree \cdot (baseQTree\ pixelInvert\ id)$$
$$pixelInvert\ (PixelRGBA8\ red\ green\ blue\ alpha) = PixelRGBA8\ (255 - red)\ (255 - green)\ (255 - blue)\ alpha$$
$$invertQTree = (\!|(invertGen)|\!)$$

Figura 7: Resultado do grupo do invertBMP

Resolução da função *compressQTree*

O objetivo desta função é comprimir uma QTree dado um nível de compressão x, para tal foi elaborado um conjunto de funções. Em primeiro lugar medimos a profundidade total da árvore original sendo depois subtraido o nível de compressão à profundidade, é feita de seguida uma contagem decrescente até chegar à profundidade resultante da subtração. Ao chegar a esta profundidade se for uma *Cell* não se aplica alterações caso contrário é um *Block* onde este é transformado numa *Cell*.

$$
\begin{array}{l}
compressGen :: (Int, QTree\ a) \rightarrow (a, (Int, Int)) + ((Int, QTree\ a), ((Int, QTree\ a), ((Int, QTree\ a), (Int, QTree\ a)\\
compressGen\ (countDown, Cell\ a\ b\ c) = i_1\ (a, (b, c))\\
compressGen\ (countDown, Block\ a\ b\ c\ d) = \textbf{if}\ (countDown \equiv 0)\ \textbf{then}\ i_1\ (wandh\ (Block\ a\ b\ c\ d))\ \textbf{else}\ i_2\ counts\\
\quad \textbf{where}\ count = countDown - 1\\
\qquad counts = ((count, a), ((count, b), ((count, c), (count, d))))
\end{array}
$$

Originalmente estava planeado fazer esta função com o recurso a três catas, sendo depois transformados num só, de modo a facilitar o entendimento desta função esta foi separada em quatro passos diferentes, *1, *2, *3 e *4 como pode ser visto em comentário da função.

$$
\begin{array}{l}
wandh :: QTree\ a \rightarrow (a, (Int, Int))\\
wandh = lastStep \cdot (\!|\cdot|\!)\\
\quad \langle [singl \cdot (id \times (fromIntegral \cdot mul \cdot \langle toInteger \cdot \pi_1, toInteger \cdot \pi_2 \rangle)), \quad \text{-- *1}\\
\qquad \mathsf{conc} \cdot \langle \mathsf{conc} \cdot \langle \pi_1 \cdot \pi_1, \pi_1 \cdot \pi_1 \cdot \pi_2 \rangle, \mathsf{conc} \cdot \langle \pi_1 \cdot \pi_1 \cdot \pi_2 \cdot \pi_2, \pi_1 \cdot \pi_2 \cdot \pi_2 \cdot \pi_2 \rangle \rangle], \quad \text{-- *2}\\
\qquad \langle [toInteger \cdot \pi_1 \cdot \pi_2, add \cdot \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_2 \rangle], \quad \text{-- *3}\\
\qquad\quad [toInteger \cdot \pi_2 \cdot \pi_2, add \cdot \langle \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2 \cdot \pi_1 \cdot \pi_2 \cdot \pi_2 \rangle] \rangle \rangle \quad \text{-- *4}
\end{array}
$$

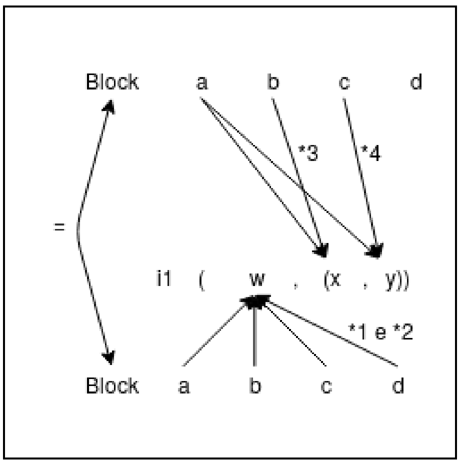


Figura 8: Figura representativa do funcionamento do wandh

O *1 constroi uma lista de pares em que o primeiro elemento é o tipo da árvore (a) e o segundo é a área do bloco, o intuito do *2 é construir uma só lista com os pares todos concatenados da seguinte maneira:

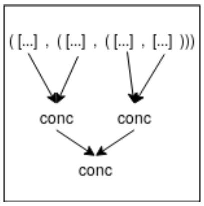


Figura 9: Figura representativa do funcionamento do *2

O *3 calcula o comprimento do bloco e por fim o *4 calcula a altura do bloco, de seguida na lista resultante é retirado o par que ocupa maior área ignorando todos os outros pares.

$$
\begin{array}{l}
minPair :: [(a, Int)] \to (a, Int) \\
minPair\ [a] = a \\
minPair\ (h : t) = \mathbf{let}\ x = minPair\ t\ \mathbf{in}\ (miniPair\ h\ x) \\
miniPair :: (t, Int) \to (t, Int) \to (t, Int) \\
miniPair\ (a, b)\ (c, d) = \mathbf{if}\ (b > d)\ \mathbf{then}\ (a, b)\ \mathbf{else}\ (c, d) \\
lastStep :: ([(a, Int)], (Integer, Integer)) \to (a, (Int, Int)) \\
lastStep = (\pi_1 \cdot minPair) \times (fromIntegral \times fromIntegral)
\end{array}
$$

No fim é chegada a seguinte solução da compress:

$$compressQTree\ x\ qt = (\!| compressGen |\!)\ (minusNat\ (depthQTree\ qt)\ x, qt)$$

Foram geradas segundo esta solução as seguintes imagens:

(a) Compressão nível 0

(b) Compressão nível 1

(c) Compressão nível 2

(d) Compressão nível 3

(e) Compresão de nível 4

Resolução da função *outlineQTree*

Função cujo intuito é converter uma quad tree numa matriz monocromática de modo a desenhar o contorno de uma malha poligonal para tal foi alterada uma função disponibilizada pelos docentes de modo a:

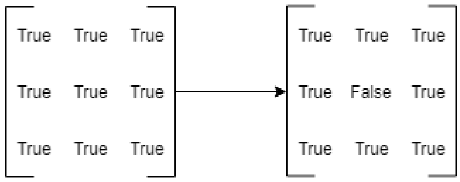
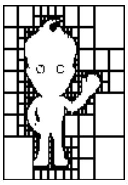


Figura 11: Figura representativa do funcionamento da função qtB2bmB

$$
\begin{array}{l}
qtB2bmB :: QTree\ Bool \rightarrow Matrix\ Bool \\
qtB2bmB = (\![[f, g]]\!)\ \mathbf{where} \\
\quad f\ (k, (i, j)) = \mathbf{if}\ (k \wedge (i \geqslant 3) \wedge (j \geqslant 3))\ \mathbf{then}\ matriz\ i\ j\ \mathbf{else}\ matrix\ j\ i\ \underline{k} \\
\quad g\ (a, (b, (c, d))) = (a \updownarrow b) \leftrightarrow (c \updownarrow d) \\
\quad matriz\ i\ j = (matrix\ 1\ i\ true) \leftrightarrow \qquad \text{-- linha de cima} \\
\qquad (((matrix\ (j-2)\ 1\ true) \updownarrow \qquad \text{-- coluna esquerda} \\
\qquad ((matrix\ (j-2)\ (i-2)\ false) \qquad \text{-- matriz central} \\
\qquad \updownarrow (matrix\ (j-2)\ 1\ true))) \qquad \text{-- coluna direita} \\
\qquad \leftrightarrow (matrix\ 1\ i\ true)) \qquad \text{-- linha de baixo} \\
outlineQTree\ f = qtB2bmB \cdot (\!| (inQTree \cdot (baseQTree\ f\ id)) |\!)
\end{array}
$$

Os resultados obtidos pelo grupo foram os seguintes:



(a) Resultado da outlineBMP

(b) Resultado da addOutlineBMP

Problema 3

Resolução

De modo a resolver este problema, o grupo recorreu as leis do cálculo funcional de modo a derivar as funções de *base k* e *loop*, começou-se por aplicar a lei de recursividade múltipla a $\langle f\ \ k, l\ \ k\rangle$ do seguinte modo:

A partir da definição de l e de f dadas no enunciado deduziu-se:

$$
\left\{ \begin{array}{l}
\left\{ \begin{array}{l} f\ k\ \underline{0} = \underline{1} \\ f\ k\ succ = mul\ .\ < f\ k, l\ k > \end{array} \right. \\
\left\{ \begin{array}{l} l\ k\ \underline{0} = \underline{k+1} \\ l\ k\ succ = (+1)\ .\ l\ k \end{array} \right.
\end{array} \right.
$$

$$\equiv \{Eq+;\ def\ de\ inNat\}$$

$$
\left\{ \begin{array}{l}
(f\ k)\ .\ inNat = [\underline{1}\ ,\ mul\ .\ < f\ k, l\ k >] \\
(l\ k)\ .\ inNat = [\underline{k+1}\ ,\ (+1)\ .\ l\ k]
\end{array} \right.
$$

$$\equiv \{Fokkinga\}$$

$$
\left\{ \begin{array}{l}
f\ k\ .\ inNat = [\underline{1}\ ,\ mul]\ .\ (id + \ < f\ k\ ,\ l\ k >) \\
l\ k\ .\ inNat = [\underline{k+1}\ ,\ (+1)\ .\ \pi 2]\ .\ (id + \ < f\ k,\ l\ k >)
\end{array} \right.
$$

$$\equiv \{Fokkinga\}$$

$$(\!| < [1\ , mul]\ ,\ [\underline{k+1}\ ,\ (+1)\ .\ \pi 2] > |\!)$$

A partir da definição de g e de s dadas no enunciado deduziu-se:

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

$\equiv \{Eq+\ ;\ Recursividade\ multipla\ ;\ def\ inNat\}$

$$\begin{cases} g\ .\ inNat = [\underline{1}\ ,\ mul]\ .\ (id +\ < g\ ,\ s >) \\ s\ .\ inNat = [\underline{1}\ ,\ (+1)\ .\ \pi 2]\ .\ (id +\ < g\ ,\ s >) \end{cases}$$

$\equiv \{Fokkinga\}$

$(| < [\underline{1}\ ,\ mul]\ ,\ [\underline{1}\ ,\ (+1)\ .\ \pi 2] > |)$

Aplicando a lei "Banana-split"aos resultados dos dois últimos cálculos, obteve-se a seguinte demonstração:

$< (| < [\underline{1}, mul], [\underline{k+1}, (+1) . \pi 2] > |), (| < [\underline{1}, mul], [\underline{1}, (+1) . \pi 2] > |) >$

$\equiv \{"Banana - Split"\}$

$(|(< [\underline{1}\ ,\ mul]\ ,\ [\underline{k+1}\ ,\ (+1)\ .\ \pi 2] > |) \times (| < [\underline{1}\ ,\ mul]\ ,\ [\underline{1}\ ,\ (+1)\ .\ \pi 2] >)\ .\ < (id\ +\ \pi 1)\ ,\ (id\ +\ \pi 2) > |)$

$\equiv \{Absorcao \times\}$

$(| << [\underline{1}, mul], [\underline{k+1}, (+1) . \pi 2] >\ . (id + \pi 1), < [\underline{1}, mul], [\underline{1}, (+1) . \pi 2] > . (id\ +\ \pi 2) > |)$

$\equiv \{Lei\ da\ troca\}$

$(|\ < [< \underline{1}\ ,\ \underline{k+1} >\ ,\ < mul\ ,\ (+1)\ .\ \pi 2 >]\ .\ (id\ +\ \pi 1)\ ,\ [< \underline{1}\ ,\ \underline{1} >\ ,\ < mul\ ,\ (+1)\ .\ \pi 2 >]\ .\ (id\ +\ \pi) > |)$

$\equiv \left\{ Absorcao + ;\ < \underline{1}\ ,\ \underline{k+1} >\ =\ \underline{(1\ ,\ k+1)}\ ;\ < \underline{1}\ ,\ \underline{1} >\ =\ \underline{(1\ ,\ 1)} \right\}$

$(| < [\underline{(1\ ,\ k+1)}, < mul.\pi 1, (+1).\pi 2.\pi 1 >], [\underline{(1\ ,\ 1)}, < mul.\pi 1, (+1).\pi 2.\pi 2 >\ ] > |)$

$\equiv \{Lei\ da\ troca\}$

$(|[< \underline{(1\ ,\ k+1)}, \underline{(1\ ,\ 1)} >, << mul.\pi 1, (+1).\pi 2.\pi 1 >, < mul.\pi 2, (+1).\pi 2.\pi 2 >>\ ]|)$

De forma a adaptar à função dada pelos docentes, chegou-se à seguinte definição de base k:

$base\ k = (1, k+1, 1, 1)$

Criaram-se as seguintes funções *flatRandL*, *unFlatReL*, de modo a chegar à seguinte definição de loop:

$flatRandL :: ((t, t1), (t2, t3)) \rightarrow (t, t1, t2, t3)$
$flatRandL\ ((a, b), (c, d)) = (a, b, c, d)$
$unFlatReL :: (t, t1, t2, t3) \rightarrow ((t, t1), (t2, t3))$
$unFlatReL\ (a, b, c, d) = ((a, b), (c, d))$
$loop = flatRandL \cdot \langle \langle mul \cdot \pi_1, (+1) \cdot \pi_2 \cdot \pi_1 \rangle, \langle mul \cdot \pi_2, (+1) \cdot \pi_2 \cdot \pi_2 \rangle \rangle \cdot unFlatReL$

Problema 4

In e Out de FTree

$inFTree = [Unit, comp]$ **where** $comp\ (a, (b, c)) = Comp\ a\ b\ c$
$outFTree\ (Unit\ a) = i_1\ a$
$outFTree\ (Comp\ a\ b\ c) = i_2\ (a, (b, c))$

Anamorfismo, Catamorfismo, Hilomorfismo, functor base e map de uma FTree

$baseFTree\ f\ g\ h = g + f \times (h \times h)$
$recFTree\ f = id + id \times (f \times f)$

$$cataFTree\ g = g \cdot recFTree\ (cataFTree\ g) \cdot outFTree$$
$$anaFTree\ g = inFTree \cdot recFTree\ (anaFTree\ g) \cdot g$$
$$hyloFTree\ g\ h = cataFTree\ g \cdot anaFTree\ h$$
$$\textbf{instance}\ Bifunctor\ FTree\ \textbf{where}$$
$$\quad bimap\ f\ g = cataFTree\ (inFTree \cdot (baseFTree\ f\ g\ id))$$

Resoluções

Resolução da função *generatePTree*

De modo a proceder ao mecanismo de geração da árvore, foi usado um par (x,y) onde o y é a ordem pretendida e o x é a ordem atual, ao longo das iterações do programa o x é incrementado sendo também calculado o tamanho do lado, quando a ordem atual (x) é igual à ordem pretendida (y) é criada uma $Unit$.

$$fTreeGen :: (Int, Int) \rightarrow Float + (Float, ((Int, Int), (Int, Int)))$$
$$fTreeGen\ (x, y) = \textbf{if}\ (x \equiv y)\ \textbf{then}\ i_1\ tam\ \textbf{else}\ i_2\ (tam, ((x+1, y), (x+1, y)))\ \textbf{where}\ tam = size * (((sqrt\ 2)\ /\ 2$$
$$\quad size = 100 \quad \text{-- tamanho do lado do primeiro quadrado}$$
$$generatePTree = curry\ (anaFTree\ fTreeGen)\ 0$$

Resolução da função *drawPTree*

Nesta função foi criada uma lista por compreensão que possui a primeira figura, a direção e a árvore sendo aplicada as transformações à primeira figura ao longo da árvore da maneira a obter $FTree\ Picture\ Picture$, é de notar que a árvore de ordem zero nunca é calculada nem a árvore que é recebida esta irá ser reaproveitada, gera apenas as árvores intermédias de maneira a poupar cálculos.

$$draw :: FTree\ a\ b \rightarrow [FTree\ Picture\ Picture]$$
$$draw\ x = \textbf{if}\ (depth > 1)\ \textbf{then}\ [Unit\ sq] +\!\!+ \quad \text{-- árvore de ordem zero}$$
$$\quad (\mathsf{map}\ [\![\cdot]\!]\ [(sq, (3, gen\ y)) \mid y \leftarrow [1\mathinner{.\,.}(depth-1)]]) \quad \text{-- lista das árvores intermédias}$$
$$\quad +\!\!+ [[\![(sq, (3, x))]\!]] \quad \text{-- última árvore}$$
$$\quad \textbf{else if}\ (depth \equiv 1)\ \textbf{then}\ [Unit\ sq, [\![(sq, (3, gen\ 1))]\!]]$$
$$\quad\quad \textbf{else}\ [Unit\ sq]$$
$$\quad\quad\quad \textbf{where}\ depth = depthFTree\ x \quad \text{-- profundidade da árvore recebida}$$
$$\quad\quad\quad\quad size = 100 \quad \text{-- tamanho do primeiro quadrado}$$
$$\quad\quad\quad\quad sq = square\ size \quad \text{-- função que calcula as imagens}$$
$$\quad\quad\quad\quad [\![\cdot]\!] = anaFTree\ (pictureGen) \quad \text{-- gera a árvore de Pictures}$$
$$\quad\quad\quad\quad gen\ y = generatePTree\ y$$

Aqui são aplicadas as transformações consoante a direção à picture anterior, a direção é distinguida consoante três números, $dir \equiv 3$ representa o início ou seja, não tem direção, $dir \equiv 1$ é a direção da esquerda ou seja encontra-se na esquerda, $dir \equiv 2$ encontra-se na direita.

$$\textbf{type}\ PicAndFTree\ a\ b = (Picture, (Int, FTree\ a\ b))$$
$$pictureGen :: PicAndFTree\ a\ b \rightarrow Picture + (Picture, (PicAndFTree\ a\ b, PicAndFTree\ a\ b))$$
$$pictureGen\ (pic, (dir, Unit\ a)) = \textbf{if}\ (dir \equiv 1)\ \textbf{then}\ i_1\ (esquerda\ pic)\ \textbf{else}\ i_1\ (direita\ pic)$$
$$pictureGen\ (pic, (dir, Comp\ a\ b\ c)) = \textbf{if}\ (dir \equiv 3)\ \textbf{then}\ i_2\ (pic, ((pic, (1, b)), (pic, (2, c))))$$
$$\quad \textbf{else if}\ (dir \equiv 1)\ \textbf{then}\ i_2\ esqi2\ \textbf{else}\ i_2\ diri2$$
$$\quad\quad \textbf{where}\ esq = esquerda\ pic$$
$$\quad\quad\quad dirp = direita\ pic$$
$$\quad\quad\quad esqi2 = (esq, ((esq, (1, b)), (esq, (2, c))))$$
$$\quad\quad\quad diri2 = (dirp, ((dirp, (1, b)), (dirp, (2, c))))$$

São aplicadas transformações às figuras da esquerda, transformações essas que são uma translação na oblíqua orientada à esquerda em que são as alterações correspondentes as coordenadas do polígono.

$$esquerda :: Picture \rightarrow Picture$$
$$esquerda = rotationLft \cdot Main.resize \cdot translationLft$$

$$
\begin{aligned}
&\mathit{translationLft} :: \mathit{Picture} \rightarrow \mathit{Picture}\\
&\mathit{translationLft}\ (\mathit{Polygon}\ [(a,b),(c,d),(e,f),(g,h)]) =\\
&\quad (\mathit{Polygon}\ [(a + \mathit{upx} + \mathit{leftx}, b + \mathit{upy} + \mathit{lefty}),\\
&\qquad (c + \mathit{upx} + \mathit{leftx}, d + \mathit{upy} + \mathit{lefty}),\\
&\qquad (e + \mathit{upx} + \mathit{leftx}, f + \mathit{upy} + \mathit{lefty}),\\
&\qquad (g + \mathit{upx} + \mathit{leftx}, h + \mathit{upy} + \mathit{lefty})])\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad \mathit{upx} = (c - (a)) \quad \text{-- translação orientada ao x}\\
&\qquad\qquad \mathit{upy} = (d - (b)) \quad \text{-- translação orientada ao y}\\
&\qquad\qquad \mathit{leftx} = ((c - e)\ /\ 2) \quad \text{-- translação à esquerda no x}\\
&\qquad\qquad \mathit{lefty} = ((d - f)\ /\ 2) \quad \text{-- translação à esquerda no y}
\end{aligned}
$$

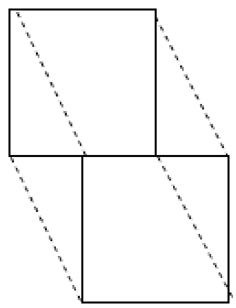


Figura 13: Figura representativa do funcionamento geral do *translationLft*

São aplicadas transformações às figuras da direita, transformações essas que são uma translação na oblíqua orientada à direita em que são as alterações correspondentes as coordenadas do polígono.

$$
\begin{aligned}
&\mathit{direita} :: \mathit{Picture} \rightarrow \mathit{Picture}\\
&\mathit{direita} = \mathit{rotationRgt} \cdot \mathit{Main.resize} \cdot \mathit{translationRgt}\\
&\mathit{translationRgt} :: \mathit{Picture} \rightarrow \mathit{Picture}\\
&\mathit{translationRgt}\ (\mathit{Polygon}\ [(a,b),(c,d),(e,f),(g,h)]) =\\
&\quad (\mathit{Polygon}\ [(a + \mathit{upx} + \mathit{rigthx}, b + \mathit{upy} + \mathit{rigthy}),\\
&\qquad (c + \mathit{upx} + \mathit{rigthx}, d + \mathit{upy} + \mathit{rigthy}),\\
&\qquad (e + \mathit{upx} + \mathit{rigthx}, f + \mathit{upy} + \mathit{rigthy}),\\
&\qquad (g + \mathit{upx} + \mathit{rigthx}, h + \mathit{upy} + \mathit{rigthy})])\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad \mathit{upx} = (c - (a)) \quad \text{-- translação orientada ao x}\\
&\qquad\qquad \mathit{upy} = (d - (b)) \quad \text{-- translação orientada ao y}\\
&\qquad\qquad \mathit{rigthx} = ((e - c)\ /\ 2) \quad \text{-- translação à direita no x}\\
&\qquad\qquad \mathit{rigthy} = ((f - d)\ /\ 2) \quad \text{-- translação à direita no y}
\end{aligned}
$$

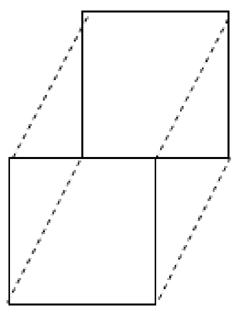


Figura 14: Figura representativa do funcionamento geral do *translationRgt*

Aplica a razão à picture de forma a diminuir o tamanho ao longo das várias iterações.

```
resize :: Picture → Picture
resize (Polygon [(a, b), (c, d), (e, f), (g, h)]) =
  (Polygon [(a + rsizeAx, b + rsizeAy),
    (c + rsizeBx, d + rsizeBy),
    (e − rsizeAx, f − rsizeAy),
    (g − rsizeBx, h − rsizeBy)])
      where r = 0.20
        rsizeAx = ((e − a) / 2) ∗ r
        rsizeAy = ((f − b) / 2) ∗ r
        rsizeBx = ((g − c) / 2) ∗ r
        rsizeBy = ((h − d) / 2) ∗ r
```

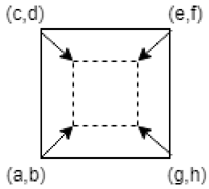


Figura 15: Diagrama representativo do funcionamento do *resize*

No resize são calculados os vetores diagonais com razão sendo depois aplicado aos pontos do Polígono, resultando num quadrado de dimensões inferiores ao originário.

Aplica a rotação à direita tendo sempre o cuidado de manter a ordem dos pontos.

```
rotationRgt :: Picture → Picture
rotationRgt (Polygon [(a, b), (c, d), (e, f), (g, h)]) =
  (Polygon [(centerx − horizx, centery − horizy),
    (centerx + verticalx, centery + verticaly),
    (centerx + horizx, centery + horizy),
    (centerx − verticalx, centery − verticaly)])
      where r = 2 − (sqrt 2 / 2)
        centerx = (a + (e − a) / 2)   -- abcissa do centro
        centery = (b + (f − b) / 2)   -- ordenada do centro
        verticalx = ((c − a) / 2) ∗ r
        verticaly = ((d − b) / 2) ∗ r
        horizx = ((g − a) / 2) ∗ r
        horizy = ((h − b) / 2) ∗ r
```

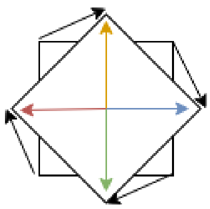


Figura 16: Figura representativa do funcionamento geral do *rotationRgt*

No *rotationRgt* primeiramente é calculado o centro e apartir daí são calculadas as posições dos novos pontos tudo sob a forma de vetores e pontos. De modo a gerar estes novos pontos, é calculado um vetor vertical e outro horizontal cada um com a dimensão de ele próprio aplicando a este a razão, depois conforme esta rotação são aplicadas novas direções a este gerando por fim os pontos resultantes da rotação, como pode ser visto conforme na figura supracitada.

Aplica a rotação à equerda tendo também aqui sempre o cuidado manter a ordem dos pontos.

$$
\begin{array}{l}
rotationLft :: Picture \rightarrow Picture \\
rotationLft\ (Polygon\ [(a,b),(c,d),(e,f),(g,h)]) = \\
\quad (Polygon\ [(centerx - verticalx, centery - verticaly), \\
\quad\quad (centerx - horizx, centery - horizy), \\
\quad\quad (centerx + verticalx, centery + verticaly), \\
\quad\quad (centerx + horizx, centery + horizy)]) \\
\quad\quad\quad \mathbf{where}\ r = 2 - (sqrt\ 2\ /\ 2) \\
\quad\quad\quad\quad centerx = (a + (e - a)\ /\ 2) \quad \text{-- abcissa do centro} \\
\quad\quad\quad\quad centery = (b + (f - b)\ /\ 2) \quad \text{-- ordenada do centro} \\
\quad\quad\quad\quad verticalx = ((c - a)\ /\ 2) * r \\
\quad\quad\quad\quad verticaly = ((d - b)\ /\ 2) * r \\
\quad\quad\quad\quad horizx = ((g - a)\ /\ 2) * r \\
\quad\quad\quad\quad horizy = ((h - b)\ /\ 2) * r
\end{array}
$$

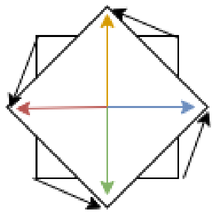


Figura 17: Figura representativa do funcionamento geral do $rotationLft$

No $rotationLft$ tal como no $rotationRgt$ é calculado o centro e apartir daí são calculadas as posições dos novos pontos. De modo a gerar estes novos pontos, é calculado um vetor vertical e outro horizontal cada um com a dimensão de ele próprio aplicando a este a razão, depois conforme esta rotação são aplicadas novas direções a este gerando por fim os pontos resultantes da rotação, como pode ser visto conforme na figura aludida.

A solução final obtida pelo grupo foi esta em que que o $extraiPic$ extrai da árvore todos os polígonos, e o $drawPTree$ concatena todos os polígonos na figura transformando numa imagem única.

$$extraiPic = cataFTree\ [singl, cons \cdot (id \times \mathsf{conc})]$$
$$drawPTree = \mathsf{map}\ \ (pictures \cdot extraiPic) \cdot draw$$

Problema 5

Resoluções

Resolução da função *singletonbag*

A função $singletonbag$ retorna um Bag de A's, visto que o número de ocorrências do A é apenas um, por isso, basta devolver a lista com o par (A,1). Como se pode ver na seguinte definição:

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

Dado o objetivo da função, concatenou-se as listas do Bag interior e, de seguida, aplicou-se o construtor Bag de modo a devolver um único Bag com todas as listas.

$$\mu = (B \cdot concat \cdot \mathsf{map}\ \ (unB \cdot \pi_1) \cdot unB)$$

De modo a elaborar a função de distribuição de probabilides, foram usadas várias funções auxiliares. Em primeiro lugar, definiu-se a função $total$ que irá na lista dentro do Bag e retira-se a frequência de cada e somam-se todas a seguir (retornando esse inteiro). Depois, a definição *divi* recebe um inteiro e devolve a divisão desse mesmo inteiro com o total (probabilidade de retirar aquele elemento). Por último, devolve um par com elemento/probabilidade. Posto isto, aplica-se o construtor de uma distribuição.

$$
\begin{array}{l}
dist\ x = (D \cdot \mathsf{map}\ \ (id \times divi) \cdot unB)\ x\ \mathbf{where} \\
\quad total = (fromIntegral \cdot sum \cdot (\!|g|\!) \cdot unB)\ x \\
\quad divi\ y = (fromIntegral\ y)\ /\ total \\
\quad g = [nil, cons \cdot (\pi_2 \times id)]
\end{array}
$$

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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 [**?**].