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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	13
a81047	Catarina Machado
a82339	João Vilaça

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.

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

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

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

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

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

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

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

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

$$
\begin{array}{l}
\mathbf{type}\ MagicNo = String \\
\mathbf{type}\ Time = Int \quad \text{-- em milisegundos} \\
\mathbf{type}\ Entity = String \\
\mathbf{type}\ Value = Int
\end{array}
$$

Neste contexto, implemente as seguintes funções:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Problema 2

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

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

```
( 0 0 0 0 0 0 0 0 )
( 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:

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

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

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 5.

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:

$$
\begin{array}{l}
whitePx = PixelRGBA8\ 255\ 255\ 255\ 255 \\
blackPx\ = PixelRGBA8\ 0\ 0\ 0\ 255 \\
redPx\quad = PixelRGBA8\ 255\ 0\ 0\ 255
\end{array}
$$

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.

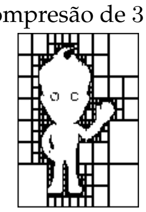
(d) Inversão de cores.

(e) Compresão de 1 nível.

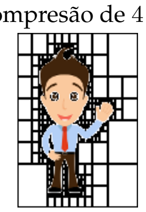
(f) Compresão de 2 níveis.

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

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

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

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

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

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

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

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

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

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

Problema 3

O cálculo das combinações de n k-a-k,

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

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

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

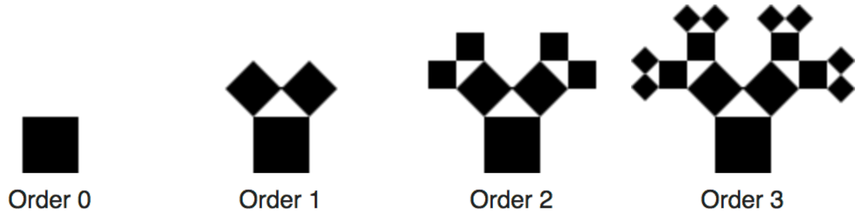


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

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

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:

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

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):*

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

Problema 4

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

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

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

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

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

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

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

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

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

```
> animatePTree 3
```

Problema 5

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

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

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

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

que habita o tipo genérico dos "bags":

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

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

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

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

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

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

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

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

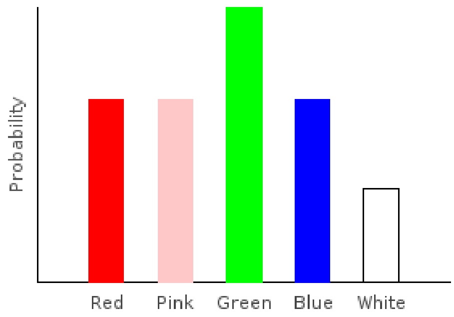


Figura 4: Distribuição de berlindes num saco.

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

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

$$marblesDist = dist\ bagOfMarbles$$

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

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

cf. Figura 4.

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

$$\begin{array}{l} \textbf{instance}\ Functor\ Bag\ \textbf{where} \\ \quad \mathsf{fmap}\ f = B \cdot \mathsf{map}\ (f \times id) \cdot unB \end{array}$$

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

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

2. Verifique-as com os seguintes testes unitários:

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

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

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

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

onde $b3$ é um saco dado em anexo.

Anexos

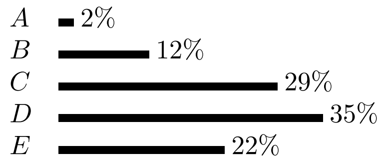
A Mónade para probabilidades e estatística

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

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

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

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



será representada pela distribuição

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

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*:

$$\begin{array}{l}
\textbf{instance}\ (Show\ a, Ord\ a, Eq\ a) \Rightarrow Show\ (Bag\ a)\ \textbf{where} \\
\quad show = showbag \cdot consol \cdot unB\ \textbf{where} \\
\quad\quad showbag = concat \cdot \\
\quad\quad\quad (+\!\!+ [\texttt{" \}"}]) \cdot (\texttt{"\{ "}:) \cdot \\
\quad\quad\quad (intersperse\ \texttt{" , "}) \cdot \\
\quad\quad\quad sort \cdot \\
\quad\quad\quad (\mathsf{map}\ \ f)\ \textbf{where}\ f\ (a, b) = (show\ a) +\!\!+ \texttt{" |-> "} +\!\!+ (show\ b) \\
unB\ (B\ x) = x
\end{array}$$

Igualdade de *bags*:

$$\begin{array}{l}
\textbf{instance}\ (Eq\ a) \Rightarrow Eq\ (Bag\ a)\ \textbf{where} \\
\quad b \equiv b' = (unB\ b)\ \text{`}lequal\text{`}\ (unB\ b') \\
\quad\quad \textbf{where}\ lequal\ a\ b = isempty\ (a \ominus b) \\
\quad\quad\quad ominus\ a\ b = a +\!\!+ neg\ b \\
\quad\quad\quad neg\ x = [(k, -i) \mid (k, i) \leftarrow x]
\end{array}$$

Ainda sobre o mónade Bag:

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

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):

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

Outras funções auxiliares:

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

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

O primeiro problema tem como tema uma block chain, ou seja, uma coleção de blocos que registam movimentos da moeda.

De modo a resolvê-lo, antes de procedermos ao desenvolvimento das suas 3 alíneas, tivemos que definir algumas funções que nos ajudarão a implementar as soluções requeridas.

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

Estas funções, nomeadamente $\mathit{inBlockchain}$, $\mathit{outBlockchain}$, $\mathit{recBlockchain}$, $\mathit{cataBlockchain}$, $\mathit{anaBlockchain}$ e $\mathit{hyloBlockchain}$, podem ser deduzidas tendo em consideração o Tipo de Dados do problema, a matéria de Cálculo de Programas e com a ajuda de alguns diagramas.

Uma vez que o tipo de Blockchain é $\mathit{Bc}\ \{\mathit{bc} :: \mathit{Block}\}$ ou $\mathit{Bcs}\ \{\mathit{bcs} :: (\mathit{Block}, \mathit{Blockchain})\}$, sabemos que o $\mathit{inBlockchain}$ e o $\mathit{outBlockchain}$ deverão "fechar"e "abrir"a Blockchain, respetivamente, logo, conseguimos representar os diagramas:

$$\mathit{Blockchain} \xleftarrow[\mathit{inBlockchain}]{} \mathit{Block} + (\mathit{Block} \times \mathit{Blockchain})$$

$$\mathit{Blockchain} \xrightarrow[\mathit{outBlockchain}]{} \mathit{Block} + (\mathit{Block} \times \mathit{Blockchain})$$

Assim, conseguimos perceber de imediato a definição de ambas as funções:

$$inBlockchain = [Bc, Bcs]$$

$$outBlockchain\ (Bc\ a) = i_1\ (a)$$
$$outBlockchain\ (Bcs\ (a, b)) = i_2\ (a, b)$$

Relativamente às funções *recBlockchain*, *cataBlockchain*, *anaBlockchain* e *hyloBlockchain*, os seus tipos já estavam presentes no enunciado e o significado e intuito de cada uma delas também já era sabido.

A título de exemplo, através do diagrama em seguida conseguimos ter uma melhor perceção de qual deverá ser a definição de cada uma destas funções. Assumimos que as funções g e h mencionadas são funções que devolvem a identidade.

$$\begin{tikzcd}
Blockchain \arrow[r, "g"] \arrow[d, "anaBlockchain\ g"'] & Block + (Block \times Blockchain) \arrow[d, "recBlockchain\ (anaBlockchain\ g)"] \\
Blockchain \arrow[r, shift left, "outBlockchain"] \arrow[d, "cataBlockchain\ h"'] & Block + (Block \times Blockchain) \arrow[l, shift left, "inBlockchain"] \arrow[d, "recBlockchain\ (cataBlockchain\ h)"] \\
Blockchain & Block + (Block \times Blockchain) \arrow[l, "h"]
\end{tikzcd}$$

Assim, intuitivamente conseguimos perceber a definição de cada uma delas.

$$recBlockchain\ g = id + (id \times g)$$
$$cataBlockchain\ h = h \cdot (recBlockchain\ (cataBlockchain\ h)) \cdot outBlockchain$$
$$anaBlockchain\ g = inBlockchain \cdot (recBlockchain\ (anaBlockchain\ g)) \cdot g$$
$$hyloBlockchain\ h\ g = cataBlockchain\ h \cdot anaBlockchain\ g$$

Na resolução das alíneas recorremos a alguns diagramas onde também fica implícito o porquê da definição de cada uma destas funções.

1. Função *allTransactions*

   O objetivo da função *allTransactions* é calcular a lista com todas as transações numa dada block chain, utilizando um catamorfismo.

   O diagrama desta função será:

$$\begin{tikzcd}
Blockchain \arrow[d, "cataBlockchain\ g"'] & Block + (Block \times Blockchain) \arrow[l, "inBlockchain"'] \arrow[d, "recBlockchain\ (cataBlockchain\ g)"] \\
Transactions & Block + (Block \times Transactions) \arrow[l, "g"]
\end{tikzcd}$$

   O objetivo é descobrir o gene g, para assim termos a definição final com algo do género $allTransactions = cataBlockchain\ g$.

   Assim, tendo em atenção o tipo de Block:

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

   E o tipo de Blockchain já apresentado, deduzimos que o g terá que ser um "either", $g = [b0, joint]$, onde de um lado irá tratar o *Block* e do outro o $Block \times Transactions$. Isso deve-se ao facto de g receber como parâmetro: $Block + (Block \times Transactions)$, ou seja, uma "soma", e devolver: *Transactions*, logo, obrigatoriamente a função terá que ser "either" para abolir o $+$.

(a) Descobrir $b0$

Para tratar o lado em que o domínio é $Block$, e sabendo que o resultado terá que ser $Transactions$, o objetivo desta função será "retirar"do $Block$ as $Transactions$.

Assim, teremos que definir $b0$ com projeções π_2 como podemos verificar no diagrama em seguida:

$$\begin{array}{c} MagicNo \times (Time \times Transactions) \\ \downarrow{\scriptstyle \pi_2 \cdot \pi_2} \\ Transactions \end{array}$$

Deste modo, fica definido $b0$ como:

$$b0 = \pi_2 \cdot \pi_2$$

(b) Descobrir $joint$

Tendo em conta o domínio da função $joint$, ou seja, $Block \times Transactions$, percebemos que o objetivo desta função será retirar de $Block$ as suas $Transactions$ e juntá-las às $Transactions$ já acumuladas (passadas como parâmetro).

Assim, uma definição de $joint$ $pointwise$ é:

$$joint\ (block, transac) = (\pi_2\ (\pi_2\ block)) +\!\!+\ transac$$

Esta função cumpre os requisitos a que a proposemos, uma vez que retira de $Block$ as $Transactions$, e concatena-as às $Transactions$ passadas como parâmetro.

Temos então a definição de $b0$ e $joint$, pelo que ficamos a saber qual é o gene g:

$$\begin{array}{ll} & g = [b0, joint] \\ \equiv & \quad \{ \text{ Definição de b0 ; Definição de joint } \} \\ & g = [\pi_2 \cdot \pi_2, joint] \\ & \textbf{where}\ joint\ (x, y) = (\pi_2\ (\pi_2\ x)) +\!\!+\ y \end{array}$$

Deste modo, está definida a função $allTransactions$ pedida:

$$\begin{array}{l} allTransactions\ a = cataBlockchain\ [\pi_2 \cdot \pi_2, joint]\ a \\ \quad \textbf{where}\ joint\ (x, y) = (\pi_2\ (\pi_2\ x)) +\!\!+\ y \end{array}$$

2. Função $ledger$

O objetivo desta função é calcular o valor disponível de cada entidade numa dada block chain.

O tipo de retorno de $ledger$ deverá ser o seguinte:

$$[(Entity, Value)]$$

Para o desenvolvimento desta função aproveitamos a função $allTransactions$ definida na alínea anterior, que nos dá a lista de todas as transações efetuadas.

O tipo de uma Transação é o seguinte:

$$(Entity, (Value, Entity))$$

Sendo o primeiro $Entity$ a entidade que envia o valor a ser transacionado e o segundo o que diz respeito à entidade que recebe.

Assim, após termos a $[Transaction]$ aplicamos-lhe um $cataList$. Este catamorfismo tem um gene $g = [nil, insert]$ que é responsável por olhar para cada $Transaction$ e construir $[(Entity, Value)]$. Para cada $Transaction$ são adicionados dois novos pares à lista: o primeiro com (Entidade que enviou, - Valor que enviou) e o segundo com (Entidade que recebeu, Valor que recebeu).

Esse catamorfismo pode ser representado pelo seguinte diagrama:

$$\begin{tikzcd}
[\,Transaction\,] & 1 + (\,Transaction \times [\,Transaction\,]) \arrow[l, "{[nil,cons]}"'] \arrow[d, "id+(id\times(cataList\ g))"] \\
[(Entity,\ Value)] \arrow[from=u, "cataList\ g"'] & 1 + (\,Transaction \times [(Entity,\ Value)] \arrow[l, "{g=[nil,insert]}"]
\end{tikzcd}$$

Depois de termos a lista com todos os valores transacionados (tanto recebidos como enviados) e respetivas entidades aplicamos-lhe uma função chamada *groupL*.

A função *groupL* ordena a $[(Entity, Value)]$, agrupa devidamente os seus elementos e por fim devolve $[(Entity, Value)]$ mas sem repetições no que diz respeito às entidades, ou seja, a função soma todas as transações da entidade (quando se trata de um valor enviado o mesmo é negativo), ficando assim com os pares (Entidade, Valor Disponível).

Consequentemente, temos a função *ledger* definida:

$$\begin{array}{l}
ledger\ a = groupL\ (cataList\ [nil, insert]\ (allTransactions\ a)) \\
\quad \textbf{where}\ insert\ (x, y) = (\pi_1\ x, -\pi_1\ (\pi_2\ x)) : (\pi_2\ (\pi_2\ x), \pi_1\ (\pi_2\ x)) : y \\
\qquad groupL\ t = (sums \cdot \mathsf{map}\ \ (mapFst\ head \cdot unzip) \cdot groupBy\ (\lambda x\ y \rightarrow \pi_1\ x \equiv \pi_1\ y) \cdot sort)\ t \\
\qquad mapFst\ f\ \ (a, b) = (f\ \ a, b) \\
\qquad sums\ [\,] = [\,]; \\
\qquad sums\ ((a, b) : t) = (a, sum\ \ b) : sums\ t
\end{array}$$

3. Função *isValidMagicNr*

   Esta função verifica se todos os números mágicos numa dada block chain são únicos, tendo portanto o seguinte tipo de dados:

   $$isValidMagicNr :: Blockchain \rightarrow Bool$$

   Um número mágico é representado por: $MagicNo :: String$.

   E relembramos também o tipo de *Block*:

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

   Assim, numa primeira fase o nosso intuito foi que a função *isValidMagicNr*, através de um cataBlockchain com um gene:

   $$\begin{array}{l}
   g = [list, insert] \\
   list\ x = [\pi_1\ x] \\
   insert\ (x, y) = (\pi_1\ x) : y
   \end{array}$$

   obtivesse uma com lista de todos os números mágicos utilizados na *Blockchain*.

   Este *cataBlockchain* pode ser definido pelo seguinte diagrama:

   $$\begin{tikzcd}
   Blockchain \arrow[d, "cataBlockchain\ g"'] & Block + (Block \times Blockchain) \arrow[l, "inBlockchain"'] \arrow[d, "recBlockchain\ (cataBlockchain\ g)"] \\
   {[MagicNo]} & Block + (Block \times [MagicNo]) \arrow[l, "g"]
   \end{tikzcd}$$

   Depois disso, ordenamos esta lista utilizando a função *sort* e aplicamos a função *group* que transforma a lista de números mágicos numa lista de listas de números mágicos. Ou seja, se um número mágico aparecer mais do que uma vez o mesmo fica agrupado na lista dentro da lista com os restantes números mágicos iguais a si.

Por fim, aplicamos $all\ ((\equiv)\ 1 \cdot \mathsf{length}\ )$. O que acontece é que calculamos o comprimento de todas as listas dentro da lista maior e verificamos se os seus comprimentos são iguais a 1. Assim, caso haja uma lista com vários números mágicos a função retornará $False$, tal como é esperado.

A função $isValidMagicNr$ pode ser então definida como:

$$
\begin{array}{l}
isValidMagicNr\ a = all\ ((\equiv)\ 1 \cdot \mathsf{length}\ ) \cdot group \cdot sort\ \$\ cataBlockchain\ [list, insert]\ a \\
\quad \textbf{where}\ list\ x = [\pi_1\ x] \\
\qquad\qquad insert\ (x, y) = (\pi_1\ x) : y
\end{array}
$$

Problema 2

O segundo problema tem como tema uma estrutura de dados que é muito utilizada para representação e processamento de imagens- quadtrees. Tal como é referido no enunciado do problema, uma quadtree é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

Antes de procedermos ao desenvolvimento das funções propostas neste problema definimos algumas funções que nos serão muito úteis.

Uma $QTree$ poderá ser:

$$Cell\ a\ Int\ Int$$

Ou

$$Block\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)$$

Em consequência, as definições de $inQTree$ e $outQTree$ terão que ser as seguintes:

$$
\begin{array}{l}
inQTree = [uncurryCell, uncurryBlock] \\
\quad \textbf{where}\ uncurryCell\ (e, (n1, n2)) = Cell\ e\ n1\ n2 \\
uncurryBlock :: (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \rightarrow QTree\ a \\
uncurryBlock\ (q1, (q2, (q3, q4))) = Block\ q1\ q2\ q3\ q4 \\
outQTree\ (Cell\ e\ n1\ n2) = i_1\ (e, (n1, n2)) \\
outQTree\ (Block\ q1\ q2\ q3\ q4) = i_2\ (q1, (q2, (q3, q4)))
\end{array}
$$

O diagrama da função $inQTree$ é o seguinte:

$$QTree\ a \xleftarrow[inQTree]{} (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))$$

E o diagrama da função $outQTree$ é o seguinte:

$$QTree\ a \xrightarrow[outQTree]{} (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))$$

No caso da função $inQTree$, que "fecha"a $QTree$, o retorno deverá ser uma $QTree$, logo, no caso da $Cell$ (lado esquerdo do $+$) temos que ajustar o parâmetro recebido $(a, (Int, Int))$ devolvendo $Cell\ a\ Int\ Int$, para assim a função devolver a informação no tipo de dados correto. No caso do $Block$, recebendo $(q1, (q2, (q3, q4)))$ teremos que retornar $Block\ q1\ q2\ q3\ q4$. Esta função é definida como um "either" porque temos estas duas "hipóteses"de tipo de dados dentro do tipo de dados $QTree$.

No caso da função $outQTree$ o raciocínio é o inverso. Uma vez que esta função recebe uma $QTree$ podemos definir a função com dois casos diferentes, tal como se pode ver na solução por nós proposta.

Os dois diagramas em seguida ajudam-nos a perceber melhor como tratar os dois casos de $QTree$ na função $outQTree$:

$$
\begin{array}{c}
(a, (Int, Int)) \\
\downarrow{\scriptstyle i_1} \\
(a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a)))
\end{array}
$$

e

$$\begin{array}{c} (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \\ \downarrow{\scriptstyle i_2} \\ (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \end{array}$$

Assim, quando recebemos uma $Cell\ e\ n1\ n2$ o objetivo será injetá-la à esquerda de modo a respeitar o tipo de dados, devolvendo então: $i_1\ (e, (n1, n2))$. No caso de $Block\ q1\ q2\ q3\ q4$, injetamos à direita retornando: $i_2\ (q1, (q2, (q3, q4)))$.

Outras funções cruciais para a resolução deste problema são as seguintes:

$$\begin{aligned} &baseQTree\ f\ g = (f \times id) + (g \times (g \times (g \times g))) \\ &recQTree\ g = baseQTree\ id\ g \\ &cataQTree\ g = g \cdot (recQTree\ (cataQTree\ g)) \cdot outQTree \\ &anaQTree\ g = inQTree \cdot (recQTree\ (anaQTree\ g)) \cdot g \\ &hyloQTree\ h\ g = cataQTree\ h \cdot anaQTree\ g \end{aligned}$$

Para melhor compreensão do intuito de cada uma delas desenhamos dois diagramas:

O primeiro, mostra a função $baseQTree$. Atendendo ao seu tipo, já contido no enunciado, conseguimos perceber qual é o objetivo desta função. A título de exemplo, vamos considerar que f tem um tipo: $f :: A \rightarrow E$ e que g tem um tipo: $g :: C \rightarrow D$:

$$\begin{array}{c} (a, b) + (c, (c, (c, c))) \\ \downarrow{\scriptstyle baseQTreefg} \\ (e, b) + (d, (d, (d, d))) \end{array}$$

Assim, percebemos de imediata que a função $baseQTree$ terá que ser definida como $(f \times id) + (g \times (g \times (g \times g)))$:

$$\begin{array}{c} (a, b) + (c, (c, (c, c))) \\ \downarrow{\scriptstyle (f \times id) + (g \times (g \times (g \times g)))} \\ (e, b) + (d, (d, (d, d))) \end{array}$$

O segundo, que é um pouco mais complexo, é apenas um exemplo do que se pode fazer com a combinação destas funções, nomedamente $inQTree$, $outQTree$, $recQTree$, $cataQTree$, $anaQTree$ e $hyloQTree$. Vamos assumir que neste nosso diagrama as funções g e h mencionadas são funções que devolvem a identidade, ou seja, não alteram o conteúdo da $QTree$, mas respeitam os tipos de dados ideais:

$$\begin{tikzcd} QTree\ a \arrow[r, "g"] \arrow[d, "anaQTree\ g"'] & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[d, "recQTree\ (anaQTree\ g)"] \\ QTree\ a \arrow[r, shift left, "outQTree"] \arrow[d, "cataQTree\ h"'] & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[l, shift left, "inQTree"] \arrow[d, "recQTree\ (cataQTree\ h)"] \\ QTree\ a & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[l, "h"] \end{tikzcd}$$

A função $hyloQTree$ é definida como sendo $hyloQTree\ h\ g = cataQTree\ h \cdot anaQTree\ g$, ou seja, no diagrama anterior pode ser identificado por uma seta vertical que vai desde o argumento da função $anaQTree$ até ao retorno da função $cataQTree$.

Assim, com a ajuda destes diagramas, encontramos as definições procuradas.

Por fim, temos fmap, que tem como objetivo aplicar a função f a todas as $Cell$ da $QTree$, mais especificamente ao conteúdo da $Cell$ que diz respeito ao valor/ objeto da matriz (não à dimensão). A função simplesmente aplica a todos esses elementos a função f.

Assim, decidimos definir a nossa função fmap da seguinte forma:

instance $Functor\ QTree$ **where**
  fmap $f = cataQTree\ (inQTree \cdot baseQTree\ f\ id)$

No seguinte diagrama conseguimos perceber o que é que as funções $inQTree \cdot baseQTree\ f\ id$ fazem, considerando que f é uma função do tipo: $f :: A \rightarrow E$:

$$\begin{array}{c} (a, b) + (c, (c, (c, c))) \\ \downarrow {\scriptstyle baseQTree\ f\ id} \\ (e, b) + (c, (c, (c, c))) \\ \downarrow {\scriptstyle inQTree} \\ QTree\ e \end{array}$$

Ou seja, numa primeira fase a função $baseQTree\ f\ id$ aplica a função f ao conteúdo da $Cell$ e numa segunda fase a função $inQTree$ junta o resultado da aplicação da função $baseQTree\ f\ id$ numa $QTree$.

Assim, aplicando um $cataQTree$ a esta composição de funções construímos:

$$\begin{tikzcd} QTree\ a \arrow[d, "cataQTree\ g"'] & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[l, "inQTree"'] \arrow[d, "recQTree\ (cataQTree\ g)"] \\ QTree\ e & (a, (Int, Int)) + (QTree\ e, (QTree\ e, (QTree\ e, QTree\ e))) \arrow[l, "g"] \end{tikzcd}$$

Sendo $g\ f = inQTree \cdot baseQTree\ f\ id$.

Em suma, tal como nos diz a própria definição de catamorfismo, na seta vertical mais à direita o mesmo é aplicado recursivamente à parte direita do $+$ (o $Functor$, ou seja, $recQTree$ encarrega-se disso) e, depois disso, temos então a "cauda"processada, tal como podemos ver no diagrama. O nosso gene g responsabiliza-se pelo último passo de transformar na $Cell$ (lado esquerdo do $+$ inferior) o seu conteúdo (através da função f) e de juntar tudo numa só $QTree$.

Agora reunimos todas as condições para nos concentrarmos no desenvolvimento das alíneas deste problema.

1. Função $rotateQTree$

   O objetivo desta função é rodar uma $QTree$.

   Optamos por utilizar um catamorfismo para definir esta função. Assim, temos que $rotateQTree = cataQTree\ g$. Tendo em conta a definição de $cataQTree$ podemos definir g como um "either", onde um dos lados irá tratar a $Cell$ e o outro o $Block$.

   Consequentemente, para perceber que impacto esta função $rotateQTree$ terá na $QTree$ analisamos a matriz de bits (Figura 1a) e a respetiva codificação em quadtrees (Figura 5).

   Vamos agora exemplificar o nosso raciocínio com alguns exemplos:

   Numa primeira fase iremos analisar o que acontecerá numa $Cell$. Por exemplo, se tivermos a $Cell$:

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

   Que é representada por $Cell\ 0\ 4\ 2$, rodando-a 90º fica:

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

Que é representada por $Cell\ 0\ 2\ 4$.

Logo, fica implícito que no que diz respeito a rodar uma $Cell$ o que acontece é que as suas dimensões verticais e horizontais trocam.

Definimos então a função $rotateCell$ que roda uma $Cell$:

$$rotateCell\ (e, (n1, n2)) = Cell\ e\ n2\ n1$$

O tipo desta função terá que ser o acima apresentado tendo em consideração os tipos da função $cataQTree$ e $inQTree$.

Numa segunda fase analisamos o que acontece num $Block$.

Procedendo da mesma forma, definimos um $Block$ de exemplo:

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

Que é definida por $Block\ (Cell\ 0\ 2\ 2)\ (Cell\ 0\ 2\ 2)\ (Cell\ 1\ 2\ 2)\ (Cell\ 1\ 2\ 2)$.

Rodando o $Block$ 90º:

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

Ficamos com uma $Block\ (Cell\ 1\ 2\ 2)\ (Cell\ 0\ 2\ 2)\ (Cell\ 1\ 2\ 2)\ (Cell\ 0\ 2\ 2)$.

Assim, percebemos que os quatro parâmetros de $Block$ rodam entre si. O primeiro parâmetro passará a ser o segundo parâmetro, o segundo passará a ser o último, o terceiro fica em primeiro e, por fim, o último parâmetro fica a ser o terceiro.

Conseguimos então perceber a definição que trata o $Block$:

$$rotateBlock\ (q1, (q2, (q3, q4))) = Block\ q3\ q1\ q4\ q2$$

Assim, temos todas as condições necessárias para definir a função pedida, $rotateQTree$:

$$\begin{array}{l} rotateQTree = cataQTree\ [rotateCell, rotateBlock] \\ \quad \textbf{where}\ rotateCell\ (e, (n1, n2)) = Cell\ e\ n2\ n1 \\ \quad\quad rotateBlock\ (q1, (q2, (q3, q4))) = Block\ q3\ q1\ q4\ q2 \end{array}$$

O diagrama desta função é o seguinte:

$$\begin{tikzcd} QTree\ a \arrow[d, "cataQTree\ g"'] & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[l, "inQTree"'] \arrow[d, "recQTree\ (cataQTree\ g)"] \\ QTree\ a & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[l, "g"] \end{tikzcd} \quad (3)$$

Onde:

$$\begin{array}{l} g = [rotateCell, rotateBlock] \\ \textbf{where}\ rotateCell\ (e, (n1, n2)) = Cell\ e\ n2\ n1 \\ rotateBlock\ (q1, (q2, (q3, q4))) = Block\ q3\ q1\ q4\ q2 \end{array}$$

2. Função $scaleQTree$

   Esta função redimensiona uma $Qtree$ tendo em consideração o Int passado como parâmetro.

   Assim, intuitivamente percebemos que teremos que multiplicar o fator passado como parâmetro pela dimensão da matriz.

   Mais uma vez recorremos a um $cataQTree$ para definir a função $scaleQTree$. Deste modo, utilizando o mesmo raciocínio da função anterior, precisamos de definir o gene g como um "either" onde a $Cell$ e o $Block$ serão tratados individualmente.

   O gene g terá a seguinte definição:

   $$g\ n = [scaleCell\ n, uncurryBlock]$$

   Uma vez que as dimensões da matriz são somente responsabilidade da $Cell$ não teremos que alterar nada no $Block$.

   De forma a respeitar os tipos, no que diz respeito a tratar o $Block$, utilizamos a função $uncurryBlock$ já definida por nós, que apenas altera a forma como o $Block$ é mostrado, não alterando o seu conteúdo.

   Para tratar a $Cell$ vamos recorrer a uma função auxiliar, $scaleCell$, cujo único objetivo será multiplicar as dimensões da $Cell$ pelo fator e devolvê-la no tipo de dados correto:

   $$scaleCell\ n\ (e, (n1, n2)) = Cell\ e\ (n1 * n)\ (n2 * n)$$

   O diagrama desta função é o mesmo da função anterior, diagrama (3), variando somente o gene g, que neste caso é o anteriormente referido.

   Assim, temos definida a função $scaleQTree$:

   $$\begin{array}{l} scaleQTree\ n = cataQTree\ [scaleCell\ n, uncurryBlock] \\ \quad \mathbf{where}\ scaleCell\ n\ (e, (n1, n2)) = Cell\ e\ (n1 * n)\ (n2 * n) \end{array}$$

3. Função $invertQTree$

   O intuito da função $invertQTree$ é inverter as cores de uma quadtree. Assim, terá obrigatoriamente de se tratar de uma matriz de píxeis, neste caso de $PixelRGBA8$. É nos também dito que o pixel pode ser invertido calculando (255 - w), sendo w a componente de cor RGB.

   Logo, utilizando o mesmo raciocínio da função $scaleQTree$, vamos definir a função $invertQTree$ como um $cataQTree$ onde também somente a $Cell$ precisa de ser modificada, tendo em conta que é a $Cell$ que possui o conteúdo da $QTree$, ou seja, o parâmetro que nos interessa alterar.

   Logo, teremos um gene g com a seguinte definição:

   $$g = [invertCell, uncurryBlock]$$

   Precisamos então somente de definir a função $invertCell$, que poderá ser definida como:

   $$\begin{array}{l} invertCell\ ((PixelRGBA8\ a\ b\ c\ d), (n1, n2)) = \\ Cell\ (PixelRGBA8\ (255 - a)\ (255 - b)\ (255 - c)\ (255 - d))\ n1\ n2 \end{array}$$

   Esta função pode também ser ilustrada através do diagrama (3), mas com um gene g definido como o mencionado anteriormente.

   Consequentemente, temos todas as condições necessárias para definir a função $invertQTree$:

   $$\begin{array}{l} invertQTree = cataQTree\ [invertCell, uncurryBlock] \\ \quad \mathbf{where}\ invertCell\ ((PixelRGBA8\ a\ b\ c\ d), (n1, n2)) = \\ \qquad Cell\ (PixelRGBA8\ (255 - a)\ (255 - b)\ (255 - c)\ (255 - d))\ n1\ n2 \end{array}$$

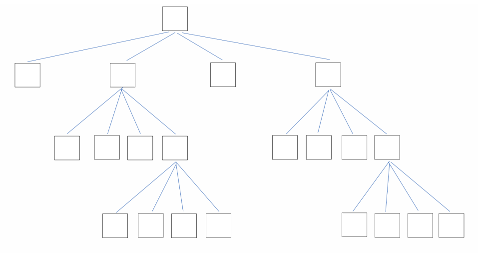


Figura 5: Esquema da $QTree$ qt.

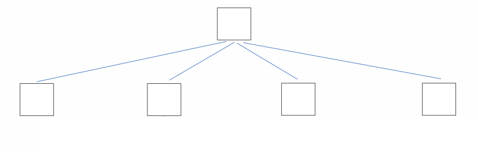


Figura 6: Esquema da $QTree$ qt comprimida em 2 níveis.

4. Função $compressQTree$

   O objetivo desta função é comprimir a $QTree$ cortando folhas da árvore de modo a reduzir a sua profundidade num dado número de níveis, passado como parâmetro na função. Basicamente, o que irá acontecer à imagem é perder informação e por isso "desfocar".

   Para o desenvolvimento desta função recorremos a uma $anaQTree$ uma vez que consideramos que seria mais simples de tratar o problema usando esse conceito/ definição.

   De uma forma mais concreta, pensamos em utilizar uma $anaQTree$ uma vez que sabiamos que ela se iria concentrar em cada nível da árvore gradualmente, começando da raiz até às folhas. Assim, de uma forma geral, pensamos que a nossa $anaQTree$ deveria ter um gene que averigua-se se estamos no nível desejado e em caso afirmativa eliminasse toda a $QTree$ a partir desse nível.

   Assim, passamos para o chamado desenvolvimento do problema:

   Temos por exemplo $QTree$ da Figura 5, que é representada pela $QTree$ qt, que já vinha também no enunciado:

$$qt = Block$$
$$(Cell\ 0\ 4\ 4)$$
$$(Block\ (Cell\ 0\ 2\ 2)\ (Cell\ 0\ 2\ 2)\ (Cell\ 1\ 2\ 2)\ (Block\ (Cell\ 1\ 1\ 1)\ (Cell\ 0\ 1\ 1)\ (Cell\ 0\ 1\ 1)\ (Cell\ 0\ 1\ 1)))$$
$$(Cell\ 1\ 4\ 4)$$
$$(Block\ (Cell\ 1\ 2\ 2)\ (Cell\ 0\ 2\ 2)\ (Cell\ 0\ 2\ 2)\ (Block\ (Cell\ 0\ 1\ 1)\ (Cell\ 0\ 1\ 1)\ (Cell\ 0\ 1\ 1)\ (Cell\ 1\ 1\ 1)))$$

   Se quisermos aplicar a função $compressQTree$ 2, ou seja, comprimir dois níveis, o esquema da $QTree$ de retorno deverá ser o seguinte o esquema presenta na Figura 6, representado por:

$$qt\_compress2 = Block$$
$$(Cell\ 0\ 4\ 4)$$
$$(Cell\ 0\ 4\ 4)$$
$$(Cell\ 1\ 4\ 4)$$
$$(Cell\ 1\ 4\ 4)$$

   Deste modo, apercebemo-nos que teriamos que ter uma função que nos "cortasse" todos os ramos da $QTree$ a abolir. Percebemos através da Figura 5 e da Figura 6 que se essa função, que poderá chamar-se $corta$, receber como parâmetro a parte da $Qtree$ a eliminar deveremos transforma-la numa única $Cell$.

Nesta fase, tivemos que perceber quais as dimensões da $Cell$ resultado pelo que, por exemplo, se tivermos:

$$Block\ (Cell\ 0\ 3\ 2)\ (Cell\ 1\ 2\ 2)\ (Cell\ 1\ 4\ 2)\ (Cell\ 0\ 1\ 2)$$

Representado por:

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

A $Cell$ resultado deverá ser:

$$Cell\ 0\ 5\ 4$$

E em modo matriz: Representado por:

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

Assim, percebemos que se o $Block\ a\ b\ c\ d$ for então constítuido apenas com $Cell$, ou seja, $a, b, c, d ::$ $Cell$, as dimensões resultado deverão ser calculadas da seguinte forma:

$$corta\ (Block\ (Cell\ c1\ n1\ n2)\ (Cell\ c2\ m1\ m2)\ (Cell\ c3\ k1\ k2)\ (Cell\ c4\ o1\ o2)) =$$
$$Cell\ (c1)\ (n1 + m1)\ (n2 + k2)$$

Optamos por dar como resultado o valor da primeira $Cell$, que no nosso caso é c1.

No caso de o $Block$ ainda não ser constituído por apenas $Cell$, tal como referimos anteriormente, aplicamos recursivamente a função $corta$ a cada um dos "argumentos"de $Block$, e ao próprio $Block$, até conseguirmos chegar a um $Block$ só com $Cell$ e o convertermos para $Cell$ da forma já referida. A recursividade irá tratar de nos fornecer a solução que desejamos. Para esta hipótese a função $corta$ será então:

$$corta\ (Block\ q1\ q2\ q3\ q4) = corta\ (Block\ (corta\ q1)\ (corta\ q2)\ (corta\ q3)\ (corta\ q4))$$

Pensando no caso de quando esta função é aplicada a somente uma $Cell$ isolada a função deverá retornar a $Cell$ argumento exatamente igual:

$$corta\ (Cell\ a\ b\ c) = Cell\ a\ b\ c$$

Juntando estas 3 hipóteses, temos então a função $corta$ definida:

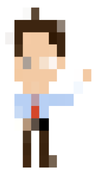
$$corta\ (Cell\ a\ b\ c) = Cell\ a\ b\ c$$
$$corta\ (Block\ (Cell\ c1\ n1\ n2)\ (Cell\ c2\ m1\ m2)\ (Cell\ c3\ k1\ k2)\ (Cell\ c4\ o1\ o2)) =$$
$$Cell\ (c1)\ (n1 + m1)\ (n2 + k2)$$
$$corta\ (Block\ q1\ q2\ q3\ q4) = corta\ (Block\ (corta\ q1)\ (corta\ q2)\ (corta\ q3)\ (corta\ q4))$$

Assim, precisamos apenas de definir concretamente, e através de código, em que momento esta função será aplicada.

Aproveitando a função $depthQTree$, que calcula a profundidade de uma $QTree$, podemos então contruir uma função auxiliar, por exemplo chamada $compress$, que irá receber o número de níveis a eliminar. Como sabemos que o nosso $anaQTree$ irá olhar individualmente para cada nível, a

(a) Figura Person comprimida em 1 nível.

(b) Figura Person comprimida em 2 níveis.



(a) Figura Person comprimida em 3 níveis.

(b) Figura Person comprimida em 4 níveis.

nossa $compress\ \ n$ averiguará se a profundidade do ramo atual é igual ao número de níveis a cortar.

Podemos utilizar esta técnica pois o anamorfismo percorre cada ramo de cima (raiz) para baixo e não volta para cima, por isso, quando o sub-ramo tiver uma profundidade igual ao número de níveis a cortar sabemos que é esse o nível que procuramos, para podermos aplicar a nossa função $corta$. Se ainda não tivermos chegado ao nível que procuramos, a função $compress$ deverá devolver a $QTree$, ou seja, o ramo, intacto.

Tivemos que ter em consideração o caso em que o número de níveis a cortar é maior que toda a profundidade da árvore, sendo que neste caso toda a $QTree$ deverá ser cortada (é aplicada à função $cortar$) ficando somente a $Cell$ comprimida.

É ainda mencionar que, tendo em conta o tipo de $anaQTree$, a nossa $compress$ deverá retornar sempre uma $QTree$ "aberta" e, assim, aplicamos a função $outQTree$.

Finalmente, juntando todos estes passos, temos a função $compressQTree$ definida:

$$
\begin{array}{l}
corta :: QTree\ a \rightarrow QTree\ a \\
corta\ (Cell\ a\ b\ c) = Cell\ a\ b\ c \\
corta\ (Block\ (Cell\ c1\ \ n1\ \ n2)\ (Cell\ c2\ \ m1\ \ m2)\ (Cell\ c3\ \ k1\ \ k2)\ (Cell\ c4\ \ o1\ \ o2)) = \\
\quad Cell\ (c1)\ (n1 + m1)\ (n2 + k2) \\
corta\ (Block\ q1\ \ q2\ \ q3\ \ q4) = corta\ (Block\ (corta\ q1)\ (corta\ q2)\ (corta\ q3)\ (corta\ q4)) \\
compressQTree\ n = anaQTree\ (compress\ n) \\
\quad \textbf{where}\ compress\ n\ a \mid (n \equiv (tam\ a) \lor n > (tam\ a)) = outQTree\ (corta\ a) \\
\qquad \mid otherwise = outQTree\ a \\
\qquad tam\ a = depthQTree\ a
\end{array}
$$

O anamorfismo que representa esta função é o seguinte:

$$
\begin{tikzcd}
QTree\ a \arrow[r, "g"] \arrow[d, "anaQTree\ g"'] & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[d, "recQTree\ (anaQTree\ g)"] \\
QTree\ a & (a, (Int, Int)) + (QTree\ a, (QTree\ a, (QTree\ a, QTree\ a))) \arrow[l, "inQTree"]
\end{tikzcd} \tag{4}
$$

Tal como dizia no enunciado, contruímos as $QTrees$ com compressões 1, 2, 3 e 4, obtendo então, respetivamente, a Figura 7a, a Figura 7b, a Figura 8a e a Figura 8b.

5. Função $outlineQTree$

A função $outlineQTree$ 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.

Após analisar o problema e as funções que o enunciado já fornece percebemos que podemos aproveitar uma função já existente e adaptá-la ao nosso problema. Esta função é a função $qt2bm$, que converte uma $QTree\ a$ em $Matrix\ a$.

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

Assim, olhamos para o gene do $cataQTree\ [f, g]$ e pensamos no que teremos que alterar para a nossa função $outlineQTree$ retornar uma $Matrix\ Bool$. Sabemos que teremos que aplicar a função passada como parâmetro às $Cell$ de modo a saber se a mesma se trata de um píxel (conjunto de píxeis) de fundo.

Uma vez que é a $Cell$ que contém os píxeis, no que diz respeito ao $Block$ nada precisará de ser feito. Logo, a função g será exatamente a mesma.

No caso das $Cell$, teremos então que dividir a função f em dois casos, um deles quando após aplicada a função dada como parâmetro ao conteúdo da $Cell$ dá $True$ e o segundo quando dá $False$:

(a) $True$- contéudo da $Cell$ ser um píxel de fundo:

Neste caso, sabemos que teremos que devolver uma $Matrix\ Bool$ onde o interior será $False$ e o contorno $True$, por exemplo, para uma Matriz de 4x4:

```
( True True  True  True )
( True False False True )
( True False False True )
( True True  True  True )
```

Logo, aproveitando a função $matrix$, usamos uma expressão lambda onde se as coordenadas (x, y) da matriz pertencerem à borda da mesma, ou seja, isso acontece quando $x \equiv 1$ ou $y \equiv 1$ ou $x \equiv largura\ maxima$ ou $y \equiv altura\ maxima$, o valor é $True$, caso contrário o valor será $False$.

(b) $False$- O contéudo da $Cell$ não ser um píxel de fundo:

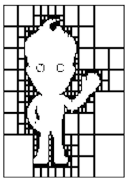
Neste caso, basta aplicar a função $matrix$ onde o conteúdo dos elementos é $False$.

Consequentemente, temos todos os dados para definir a nossa função $outlineQTree$:

$$
\begin{array}{l}
outlineQTree\ magic\ a = cataQTree\ [f\ magic, g]\ a \\
\quad \mathbf{where}\ f\ magic\ (k, (i, j)) \\
\qquad \mid (magic\ k) = matrix\ j\ i\ (\lambda(x, y) \rightarrow \mathbf{if}\ (x \equiv 1 \lor y \equiv 1 \lor x \equiv j \lor y \equiv i)\ \mathbf{then}\ True\ \mathbf{else}\ False) \\
\qquad \mid otherwise = matrix\ j\ i\ \underline{False} \\
\qquad g\ (a, (b, (c, d))) = (a \updownarrow b) \leftrightarrow (c \updownarrow d)
\end{array}
$$

No enunciado desta função era ainda sugerido que produzissemos imagens com o auxílio de duas funções que utilizam a $outlineQTree$. As duas imagens que obtivemos estão presentes na Figura 9a e a Figura 9b.

É ainda de salientar que todas as funções do problema passaram em todos os testes do QuickCheck e nos Testes Unitários.



(a) Figura Person produzida com a função $outlineBMP$.

(b) Figura Person produzida com a função $addOutlineBMP$.

Problema 3

O objetivo deste problema é derivar as funções $base\ k$ e $loop$ de modo a podermos calcular as combinações de n k-a-k,

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

recorrendo a um ciclo-for onde apenas se fazem multiplicações e somas:

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

Tendo em conta o **where** $h\ k\ n$ e o **in** $a\ /\ b$ da função acima apresentada e comparando com a definição de $h\ k$ do enunciado,

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

constatamos que o nosso a em **in** $a\ /\ b$ terá que ser a função $f\ k$ e o b será a função g.

Assim, para podermos descobrir a definição das funções pedidas ($base\ k$ e $loop$) teremos que, a partir da definição de $f\ k$ (e consequentemente de $l\ k$) e da definição de g (e consequentemente de s), descobrirmos a definição de $\langle f\ k, l\ k\rangle$ e de $\langle g, s\rangle$ (com recurso à lei da recursividade múltipla) e posteriormente combinarmos os seus resultados com a lei de banana-split.

Para tal, dividimos o nosso problema em diferentes partes:

1. Determinar $\langle f\ k, l\ k\rangle$

   Para isso, tirando partido da indicação do enunciado, ou seja, com o intuito de aplicar a lei da recursividade múltipla, temos:

$$\begin{cases} f\ k \cdot \mathbf{in} = o \cdot F\ \langle f\ k, l\ k\rangle \\ l\ k \cdot \mathbf{in} = p \cdot F\ \langle f\ k, l\ k\rangle \end{cases}$$
$$\equiv \qquad \{\ \text{Fokkinga: (50)}\ \}$$
$$\langle f\ k, l\ k\rangle = (\!|\langle o, p\rangle|\!)$$

   O objetivo será então determinar o e p e, para isso, teremos que olhar individualmente para cada uma das funções $f\ k$ e $l\ k$, apresentadas no enunciado:

   (a) Descobrir o (com a ajuda da função $f\ k$)

$$\begin{cases} f\ k\ 0 = 1 \\ f\ k\ (d+1) = (l\ k\ d) * (f\ k\ d) \end{cases}$$
$$\equiv \qquad \{\ \text{Def comp: (74) ; (*) escrita como função prefixo}\ \}$$

$$\begin{cases} f\ k \cdot 0 = 1 \\ f\ k \cdot (d+1) = (*)\ (l\ k\ d)\ (f\ k\ d) \end{cases}$$

$\equiv$ { Igualdade extencional: (73) ; Def const: (76) }

$$\begin{cases} f\ k \cdot \underline{0} = \underline{1} \\ f\ k \cdot (d+1) = (*)\ (l\ k\ d)\ (f\ k\ d) \end{cases}$$

$\equiv$ { Definição de mul: mul (a, b) = (*) a b ; }

$$\begin{cases} f\ k \cdot \underline{0} = \underline{1} \\ f\ k \cdot (d+1) = mul\ (f\ k\ d, l\ k\ d) \end{cases}$$

$\equiv$ { Def split: (78) ; Def comp: (74) ; Definição de succ: succ d = d + 1 ; Igualdade extencional: (73) }

$$\begin{cases} f\ k \cdot \underline{0} = \underline{1} \\ f\ k \cdot (\mathsf{succ}\ ) = mul \cdot \langle f\ k, l\ k \rangle \end{cases}$$

$\equiv$ { Eq + : (27) ; Natural id : (1) }

$$[f\ k \cdot \underline{0}, f\ k \cdot (\mathsf{succ}\ )] = [\underline{1} \cdot id, mul \cdot \langle f\ k, l\ k \rangle]$$

$\equiv$ { Fusão + : (20) ; Absorção x : (11) }

$$f\ k \cdot [\underline{0}, (\mathsf{succ}\ )] = [\underline{1}, mul] \cdot (id + \langle f\ k, l\ k \rangle)$$

$\equiv$ { Definição de in e functor (dos naturais): $\mathbf{in} = [\underline{0}, (\mathsf{succ}\ )]$, $F\ \langle f\ k, l\ k \rangle = (id + \langle f\ k, l\ k \rangle)$ }

$$f\ k \cdot \mathbf{in} = [\underline{1}, mul] \cdot F\ \langle f\ k, l\ k \rangle$$

Logo, $o = [\underline{1}, mul]$.

(b) Descobrir p (com a ajuda da função $l\ k$)

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

$\equiv$ { Def comp: (74) }

$$\begin{cases} l\ k \cdot 0 = k+1 \\ l\ k \cdot (d+1) = l\ k\ d+1 \end{cases}$$

$\equiv$ { Definição de succ: succ d = d + 1 ; Def const: (76) ; Igualdade extencional: (73) }

$$\begin{cases} l\ k \cdot \underline{0} = \underline{(\mathsf{succ}\ \ k)} \\ l\ k \cdot (\mathsf{succ}\ ) = \mathsf{succ}\ \cdot (l\ k) \end{cases}$$

$\equiv$ { Eq + : (27) ; Natural id : (1) }

$$[l\ k \cdot \underline{0}, l\ k \cdot (\mathsf{succ}\ )] = [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot l\ k]$$

$\equiv$ { Natural id : (1) ; Cancelamento x : (7) }

$$[l\ k \cdot \underline{0}, l\ k \cdot (\mathsf{succ}\ )] = [\underline{(\mathsf{succ}\ \ k)} \cdot id, \mathsf{succ}\ \cdot \pi_2 \cdot \langle f\ k, l\ k \rangle]$$

$\equiv$ { Fusão + : (20) ; Absorção x : (11) }

$$l\ k \cdot [\underline{0}, (\mathsf{succ}\ )] = [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle f\ k, l\ k \rangle)$$

$\equiv$ { Definição de in e functor (dos naturais): $\mathbf{in} = [\underline{0}, (\mathsf{succ}\ )]$, $F\ \langle f\ k, l\ k \rangle = (id + \langle f\ k, l\ k \rangle)$ }

$$l\ k \cdot \mathbf{in} = [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \cdot F\ \langle f\ k, l\ k \rangle$$

Logo, $p = [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2]$.

Deste modo, após encontrarmos a definição de o e de p conseguimos determinar a definição de $\langle f\ k, l\ k \rangle$ uma vez que já haviamos concluido que $\langle f\ k, l\ k \rangle = (\!| \langle o, p \rangle |\!)$.
Logo, $\langle f\ k, l\ k \rangle = (\!| \langle [\underline{1}, mul], [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \rangle |\!)$.

2. Determinar $\langle g, s \rangle$

Para descobrir $\langle g, s\rangle$ seguimos o mesmo raciocínio, isto é, tentamos descobrir um v e um j de modo a podermos aplicar a lei da recursividade múltipla:

$$\left\{\begin{array}{l} g \cdot \mathbf{in} = v \cdot F\ \langle g, s\rangle \\ s \cdot \mathbf{in} = j \cdot F\ \langle g, s\rangle \end{array}\right.$$

$\equiv$ { Fokkinga: (50) }

$$\langle g, s\rangle = (\!|\langle v, j\rangle|\!)$$

(a) Descobrir v (com a ajuda da função g)

$$\left\{\begin{array}{l} g\ 0 = 1 \\ g\ (d+1) = (g\ d) * (s\ d) \end{array}\right.$$

$\equiv$ { Def comp: (74) ; (*) escrita como função prefixo }

$$\left\{\begin{array}{l} g \cdot 0 = 1 \\ g \cdot (d+1) = (*)\ (g\ d)\ (s\ d) \end{array}\right.$$

$\equiv$ { Igualdade extencional: (73) ; Def const: (76) }

$$\left\{\begin{array}{l} g \cdot \underline{0} = \underline{1} \\ g \cdot (d+1) = (*)\ (g\ d)\ (s\ d) \end{array}\right.$$

$\equiv$ { Definição de mul: mul (a, b) = (*) a b ; }

$$\left\{\begin{array}{l} g \cdot \underline{0} = \underline{1} \\ g \cdot (d+1) = mul\ (g\ d, s\ d) \end{array}\right.$$

$\equiv$ { Def split: (78) ; Def comp: (74) ; Definição de succ: succ d = d + 1 ; Igualdade extencional: (73) }

$$\left\{\begin{array}{l} g \cdot \underline{0} = \underline{1} \\ g \cdot (\mathsf{succ}\ ) = mul \cdot \langle g, s\rangle \end{array}\right.$$

$\equiv$ { Eq + : (27) ; Natural id : (1) }

$$[g \cdot \underline{0}, g \cdot (\mathsf{succ}\ )] = [\underline{1} \cdot id, mul \cdot \langle g, s\rangle]$$

$\equiv$ { Fusão + : (20) ; Absorção x : (11) }

$$g \cdot [\underline{0}, (\mathsf{succ}\ )] = [\underline{1}, mul] \cdot (id + \langle g, s\rangle)$$

$\equiv$ { Definição de in e functor (dos naturais): $\mathbf{in} = [\underline{0}, (\mathsf{succ}\ )]$, $F\ \langle g, s\rangle = (id + \langle g, s\rangle$ }

$$g \cdot \mathbf{in} = [\underline{1}, mul] \cdot F\ \langle g, s\rangle$$

Logo, $v = [\underline{1}, mul]$.

(b) Descobrir j (com a ajuda da função s)

$$\left\{\begin{array}{l} s\ 0 = 1 \\ s\ (d+1) = s\ d + 1 \end{array}\right.$$

$\equiv$ { Def comp: (74) }

$$\left\{\begin{array}{l} s \cdot 0 = 1 \\ s \cdot (d+1) = s\ d + 1 \end{array}\right.$$

$\equiv$ { Definição de succ: succ d = d + 1 ; Def const: (76) ; Igualdade extencional: (73) }

$$\left\{\begin{array}{l} s \cdot \underline{0} = \underline{1} \\ s \cdot (\mathsf{succ}\ ) = \mathsf{succ}\ \cdot s \end{array}\right.$$

$\equiv$ { Eq + : (27) ; Natural id : (1) }

$$[s \cdot \underline{0}, s \cdot (\mathsf{succ}\ )] = [\underline{1}, \mathsf{succ}\ \cdot s]$$

$$\equiv \quad \{ \text{ Natural id : (1) ; Cancelamento x : (7) } \}$$

$$[s \cdot \underline{0}, s \cdot (\mathsf{succ}\ )] = [\underline{1} \cdot id, \mathsf{succ}\ \cdot \pi_2 \cdot \langle g, s \rangle]$$

$$\equiv \quad \{ \text{ Fusão + : (20) ; Absorção x : (11) } \}$$

$$s \cdot [\underline{0}, (\mathsf{succ}\ )] = [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \cdot (id + \langle s, g \rangle)$$

$$\equiv \quad \{ \text{ Definição de in e functor (dos naturais): } \mathbf{in} = [\underline{0}, (\mathsf{succ}\ )] \text{ , } F\ (\langle s, g \rangle = (id + \langle s, g \rangle) \ \}$$

$$s \cdot \mathbf{in} = [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \cdot F\ \langle s, g \rangle$$

Logo, $j = [\underline{1}, \mathsf{succ}\ \cdot \pi_2]$.

Deste modo, após encontrarmos a definição de v e de j conseguimos determinar a definição de $\langle g, s \rangle$ uma vez que já haviamos constatado que $\langle g, s \rangle = (\!|\langle v, j \rangle|\!)$.
Logo, $\langle g, s \rangle = (\!|\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!)$.

3. Aplicar a lei de banana split à definição de $\langle f\ \ k, l\ \ k \rangle$ e $\langle g, s \rangle$
   A lei de banana split (51) é a seguinte:

$$\langle (\!|i|\!), (\!|j|\!) \rangle$$

$$\equiv \quad \{ \text{ Banana-split : (51) } \}$$

$$(\!|(i \times j) \cdot \langle F\ \pi_1, F\ \pi_2 \rangle|\!)$$

Assim, podemos constatar que, no nosso caso:

$$(\!|i|\!) = (\!|\langle [\underline{1}, mul], [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!)$$

e

$$(\!|j|\!) = (\!|\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!)$$

Assim, retomando o resultado anterior e aplicando a lei temos:

$$\langle (\!|\langle [\underline{1}, mul], [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!), (\!|\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!) \rangle$$

$$\equiv \quad \{ \text{ Banana-split : (51) } \}$$

$$(\!|((\!|\langle [\underline{1}, mul], [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!) \times (\!|\langle [\underline{1}, mul], [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \rangle|\!)) \cdot \langle F\ \pi_1, F\ \pi_2 \rangle|\!)$$

Agora podemos então continuar a resolver o problema, sempre com o intuito de chegar a um $(\!|[b, i]|\!)$ para podermos aplicar a definição de ciclo for:

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

Retomando o resultado anterior e continuando a aplicar as leis do cálculo de programas, temos:

$$(\!|(\langle [\underline{1}, mul], [\underline{(\mathsf{succ}\ \ k)}, \mathsf{succ}\ \cdot \pi_2] \rangle \times \langle [\underline{1}, mul], [\underline{1}, \mathsf{succ}\ \cdot \pi_2] \rangle) \cdot \langle F\ \pi_1, F\ \pi_2 \rangle|\!)$$

$$\equiv \quad \{ \text{ Absorção x : (11) ; Lei da troca : (28) } \}$$

$$(\!|\langle [\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)} \rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2 \rangle] \cdot F\ \pi_1, [\langle \underline{1}, \underline{1} \rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2 \rangle] \cdot F\ \pi_2 \rangle|\!)$$

$$\equiv \quad \{ \text{ Definição do Functor (dos naturais): } F\ \pi_1 = id + \pi_1, F\ \pi_2 = id + \pi_2 \ \}$$

$$(\!|\langle [\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)} \rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2 \rangle] \cdot (id + \pi_1), [\langle \underline{1}, \underline{1} \rangle, \langle mul, \mathsf{succ}\ \cdot \pi_2 \rangle] \cdot (id + \pi_2) \rangle|\!)$$

$$\equiv \quad \{ \text{ Absorção + : (22) x2 ; Fusão x : (10) x2 ; Natural id : (1) x2 } \}$$

$$(\!|\langle [\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)} \rangle, \langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1 \rangle], [\langle \underline{1}, \underline{1} \rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle] \rangle|\!)$$

$$
\begin{aligned}
\equiv\quad & \{\ \text{Lei da troca : (28)}\ \} \\
& \llparenthesis [\langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle, \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle] \rrparenthesis \\
\equiv\quad & \{\ \text{Definição de for: } \mathsf{for}\ b\ \underline{i} = \llparenthesis [i, b] \rrparenthesis\ \} \\
& \mathsf{for}\ \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle\ \langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle
\end{aligned}
$$

Assim, tendo em conta a definição do enunciado:

$$\mathsf{for}\ loop\ (base\ k)$$

E comparando com o nosso resultado:

$$\mathsf{for}\ \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle\ \langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle$$

Temos que:

$$
\begin{aligned}
& loop = \langle\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1\rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle\rangle \\
& base\ k = \langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle
\end{aligned}
$$

4. Derivar a definição de $base\ k$

   Focando em $base\ k$:

$$base\ k = \langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle$$

Vamos agora introduzir variáveis à definição *pointfree* para podermos ter o resultado da função $base\ k$ em haskell.

Uma vez que o tipo de split, por exemplo, $\langle id, id\rangle$ é:

$$
\begin{tikzcd}
A \arrow[d, "\langle id, id\rangle"'] \\
A \times A
\end{tikzcd}
$$

O tipo de $\langle\langle id, id\rangle, \langle id, id\rangle\rangle$ será:

$$
\begin{tikzcd}
A \arrow[d, "\langle\langle id, id\rangle, \langle id, id\rangle\rangle"'] \\
(A \times A) \times (A \times A)
\end{tikzcd}
$$

Assim, conseguimos perceber qual é o tipo da variável que temos que adicionar uma vez que no nosso caso também se trata de um split de split.

Logo, continuando a derivação:

$$
\begin{aligned}
& base\ k = \langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle \\
\equiv\quad & \{\ \text{Igualdade extensional : (73)}\ \} \\
& base\ k = \langle\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle, \langle \underline{1}, \underline{1}\rangle\rangle\ a \\
\equiv\quad & \{\ \text{Def split : (78)}\ \} \\
& base\ k = (\langle \underline{1}, \underline{(\mathsf{succ}\ \ k)}\rangle\ a, \langle \underline{1}, \underline{1}\rangle\ a) \\
\equiv\quad & \{\ \text{Def split : (78) x2}\ \} \\
& base\ k = ((\underline{1}\ a, \underline{(\mathsf{succ}\ \ k)}\ a), (\underline{1}\ a, \underline{1}\ a)) \\
\equiv\quad & \{\ \text{Def const : (76) x4 ; Definição de succ: succ k = k + 1}\ \} \\
& base\ k = ((1, k + 1), (1, 1))
\end{aligned}
$$

Porém, tendo em consideração a implementação desejada:

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

Vemos que em **let** $(a, \_, b, \_)$ o tipo de dados é um quádruplo o que implica que os tipos de $loop$ e $base\ k$, mais especificamente o tipo de retorno, terão que ser também um quádruplo (o $\mathsf{for}\ loop\ (base\ k)\ n$, por sua vez, também deverá retornar um quádruplo).

Deste modo, através da aplicação de uma simples função que nos altere o tipo de dados, por exemplo:

$$altera :: ((a, b), (c, d)) \rightarrow (a, b, c, d)$$
$$altera\ ((a, b), (c, d)) = (a, b, c, d)$$

Conseguimos ter a derivação desejada da função $base\ k$:

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

5. Derivar a definição de $loop$

   Por último, para derivar a definição de $loop$ teremos que pensar da mesma forma, ou seja, inserir variáveis. Porém, ao contrário da função $base\ k$, as variáveis a inserir terão que ser do tipo ((a, b), (c, d)) uma vez que temos um split com projeções π_1 e π_2. Nos diagramas abaixo pode-se verificar o motivo desta diferença de domínio:

   Tendo em conta o primeiro split do split maior, temos:

$$\begin{tikzcd} (A \times B) \times (C \times D) \arrow[d, "\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1 \rangle"'] \\ Z \times B \end{tikzcd}$$

   É de salientar que o tipo mul (a, b) = a * b, ou seja, o tipo desta função é:

$$\begin{tikzcd} (A \times B) \arrow[d, "mul"'] \\ Z \end{tikzcd}$$

   Tendo agora em consideração o segundo split do split maior, temos:

$$\begin{tikzcd} (A \times B) \times (C \times D) \arrow[d, "\langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle"'] \\ (Z \times D) \end{tikzcd}$$

   Temos então,

$$\begin{aligned}
& loop = \langle \langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1 \rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle \rangle \\
\equiv \quad & \{ \text{ Igualdade extensional : (73) } \} \\
& loop = \langle \langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1 \rangle, \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle \rangle\ ((a, b), (c, d)) \\
\equiv \quad & \{ \text{ Def split : (78) } \} \\
& loop = (\langle mul \cdot \pi_1, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_1 \rangle\ ((a, b), (c, d)), \langle mul \cdot \pi_2, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle\ ((a, b), (c, d))) \\
\equiv \quad & \{ \text{ Def split : (78) x2 ; Def comp : (74) x6 ; Def proj : (81) x6 } \} \\
& loop = ((mul\ (a, b), \mathsf{succ}\ \ b), (mul\ (c, d), \mathsf{succ}\ \ d)) \\
\equiv \quad & \{ \text{ Definição de succ : succ k = k + 1 x2 ; Definição de mul : mul (m,n) = m * n x2 } \} \\
& loop = ((a * b, b + 1), (c * d, d + 1))
\end{aligned}$$

Deparámo-nos com o mesmo problema da função $base\ k$ e por isso vamos aplicar novamente a função $altera$ para reparar o tipo de dados de retorno. Temos então:

$$loop\ (a, b, c, d) = (a * b, b + 1, c * d, d + 1)$$

Deste modo, obtemos os dois resultados pretendidos:

$$base\ k = (1, k + 1, 1, 1)$$
$$loop\ (a, b, c, d) = (a * b, b + 1, c * d, d + 1)$$

Problema 4

O quarto problema aborda *Fractais*. Fractais são formas geométricas que podem ser construídas recursivamente de acordo com um conjunto de equações matemáticas. Iremos concentrarmos no exemplo clássico de árvores de Pitágoras.

A par do problema 1 e 2, começamos por definir algumas funções que serão utilizadas na resolução deste problema.

Assim, analisando o tipo de uma $FTree$ sabemos que uma:

$$FTree\ a\ b$$

poderá ser

$$Unit\ b$$

ou

$$Comp\ a\ (FTree\ a\ b)\ (FTree\ a\ b)$$

Deste modo, as funções $inFTree$ e $outFTree$ são as seguintes:

$$inFTree = [Unit, uncurryB\ Comp]$$
$$\quad \mathbf{where}\ uncurryB\ f\ (a, (t1, t2)) = f\ a\ t1\ t2$$
$$outFTree\ (Unit\ c) = i_1\ c$$
$$outFTree\ (Comp\ a\ t1\ t2) = i_2\ (a, (t1, t2))$$

Diagrama de $inFTree$:

$$FTree\ a1\ a2 \xleftarrow[inFTree]{} a2 + (a1, (FTree\ a1\ a2, FTree\ a1\ a2))$$

Diagrama de $outFTree$:

$$FTree\ a1\ a2 \xrightarrow[outFTree]{} a2 + (a1, (FTree\ a1\ a2, FTree\ a1\ a2))$$

Ambas as funções foram explicadas com mais pormenor no Problema 2, que apesar de dizer respeito a $QTree$, o pensamento para as definir foi muito semelhante.

Temos também as seguintes funções:

$$recFTree\ f = baseFTree\ id\ id\ f$$
$$cataFTree\ a = a \cdot (recFTree\ (cataFTree\ a)) \cdot outFTree$$
$$anaFTree\ f = inFTree \cdot (recFTree\ (anaFTree\ f)) \cdot f$$
$$hyloFTree\ a\ c = cataFTree\ a \cdot anaFTree\ c$$

Tal como nas funções $inFTree$ e $outFTree$, estas quatro funções também são muito semelhantes às desenvolvidas no Problema 2 pelo que não iremos voltar a entrar em pormenor porque a justificação é praticamente a mesma.

O diagrama que agrega todas estas funções é o seguinte, partindo do mesmo princípio que, a título de exemplo, os genes g e h não alteram a $FTree$ (são id):

$$\begin{tikzcd}
FTree\ a1\ a2 \arrow[r, "g"] \arrow[d, "anaFTree\ g"'] & a2 + (a1, (FTree\ a1\ a2, FTree\ a1\ a2)) \arrow[d, "recFTree\ (anaFTree\ g)"] \\
FTree\ a1\ a2 \arrow[r, "outFTree"] \arrow[d, "cataFTree\ h"'] & a2 + (a1, (FTree\ a1\ a2, FTree\ a1\ a2)) \arrow[l, "inFTree"] \arrow[d, "recFTree\ (cataFTree\ h)"] \\
FTree\ a1\ a2 & a2 + (a1, (FTree\ a1\ a2, FTree\ a1\ a2)) \arrow[l, "h"]
\end{tikzcd}$$

A função $hyloFTree$ é definida como sendo $hyloFTree\ h\ g = cataFTree\ h \cdot anaFTree\ g$, ou seja, no diagrama anterior pode ser identificado por uma seta vertical que vai desde o argumento da função $anaFTree$ até ao retorno da função $cataFTree$.

A função $baseFTree\ f\ g\ h$, que possui o tipo:

$$\begin{aligned}
&(a1 \to b1) \to \\
&(a2 \to b2) \to \\
&(a3 \to d) \to \\
&a2 + (a1, (a3, a3)) \to \\
&b2 + (b1, (d, d))
\end{aligned}$$

Tem o objetivo de alterar os argumentos da FTree aplicando-lhe funções tal como podemos ver no diagrama em seguida, assumindo que $f : a1 \to b1$, $g : a2 \to b2$ e $h : a3 \to d$.

$$\begin{tikzcd}
a2 + (a1, (a3, a3)) \arrow[d, "g+(f\times(h\times h)"'] \\
b2 + (b1, (d, d))
\end{tikzcd}$$

Esta função é então definida como:

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

Temos ainda um $Bifuntor$:

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

O gene $g = inFTree \cdot baseFTree\ f\ h\ id$ do cataFTree pode ser demonstrado através do seguinte diagrama, onde assumimos que $f : a1 \to b1$ e $h : a2 \to b2$.

$$\begin{tikzcd}
a2 + (a1, (a3, a3)) \arrow[d, "baseFTree\ f\ h\ id"'] \\
b2 + (b1, (a3, a3)) \arrow[d, "inFTree"'] \\
FTree\ b1\ b2
\end{tikzcd}$$

Construindo o $cataFTree$ aplicado ao gene referido anteriormente temos:

$$\begin{tikzcd}
FTree\ a1\ a2 \arrow[d, "cataFTree\ g"'] & a2 + (a1, (FTree\ a1\ a2, FTree\ a1\ a2)) \arrow[l, "inFTree"'] \arrow[d, "recFTree\ (cataFTree\ g)"] \\
FTree\ b1\ a2 & a2 + (b1, (FTree\ a1\ a2, FTree\ a1\ a2)) \arrow[l, "g"]
\end{tikzcd}$$

Mais uma vez, as justificações do Problema 2 relaticamente à função fmap fazem um "match" com a função $bimap$, pelo que uma explicação mais detalhada acerca desta última função pode ser encontrada no Problema 2.

Temos ainda o tipo de dados $PTree$ composto por:

$PTree :: FTree\ Square\ Square$

Onde $Square$:

$Square :: Float$

Concentrando agora nas funções pedidas no enunciado:

1. Função $generatePTree$

   Esta função tem o objetivo de gerar uma árvore de Pitágoras para uma dada ordem.

   Tal como indica o enunciado, definimos esta função como uma $anaFTree$:

   $$\begin{array}{l} generatePTree\ a = anaFTree\ ((\underline{0} + \langle m, \langle id, id \rangle \rangle) \cdot outNat)\ a \\ \quad \textbf{where} \\ \qquad m\ x = 50 * (sqrt\ (2)\ /\ 2) \uparrow abs\ (x - a) \end{array}$$

   Assim, através do seguinte diagrama conseguimos ver os tipos do gene do $anaFTree$:

   $$\begin{array}{l} g = ((\underline{0} + \langle m, \langle id, id \rangle \rangle) \cdot outNat \\ m\ x = 50 * (sqrt\ (2)\ /\ 2) \uparrow abs\ (x - a) \end{array}$$

   Que podem ser representados pelo seguinte diagrama:

   $$\begin{tikzcd} Int \arrow[d, "outNat"'] \\ 1 + Int \arrow[d, "(\underline{0} + \langle m, \langle id, id \rangle \rangle)"'] \\ \underline{0} + (Square, (Int, Int)) \end{tikzcd}$$

   Com o anamorfismo obtemos o seguinte diagrama:

   $$\begin{tikzcd} Int \arrow[r, "g = \underline{0} + \langle m, \langle id, id \rangle \rangle"] \arrow[d, "anaFTree\ g"'] & \underline{0} + (Square, (Int, Int)) \arrow[d, "recFTree\ (anaFTree\ g)"] \\ PTree\ (FTree\ Square\ Square) & \underline{0} + (Square, (PT, PT)) \arrow[l, "inFTree"] \end{tikzcd} \tag{5}$$

   Onde $PT :: PTree\ (FTree\ Square\ Square)$.

   O que acontece é que o o $m\ \ x$ irá criar um $Square$, onde o $\langle m, \langle id, id \rangle \rangle)$ irá obter o resultado $Square \times (\mathbb{N}_0 \times \mathbb{N}_0)$. Para cada um dos $\mathbb{N}_0$, irá ser aplicado o anamorfismo de modo a obtermos a $PTree\ (FTree\ Square\ Square)$.

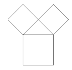
2. Função $drawPTree$

   O objetivo de $drawPTree$ é animar incrementalmente os passos de construção de uma função de Pitágoras recorrendo à biblioteca gloss.

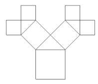
   ...

   $$\begin{array}{l} drawPTree\ a = anaList\ ((nil + \langle list, id \rangle) \cdot outNat)\ (depthFTree\ a) \\ \quad \textbf{where} \\ \qquad list\ n = (foldMap\ lineLoop\ (squares\ n)) \\ \qquad squares\ n = concat\ \$\ take\ (depthFTree\ a - n)\ \$\ iterateM\ mkBranches\ start \\ \qquad start = [(-100, 0), (0, 0), (0, -100), (-100, -100)] \\ \qquad iterateM\ f\ x = iterate\ (\gg\!= f)\ (pure\ x) \end{array}$$

(a) Árvore de Pitágoras em construção momento 1.



(b) Árvore de Pitágoras em construção momento 2.



(c) Árvore de Pitágoras em construção momento 3.

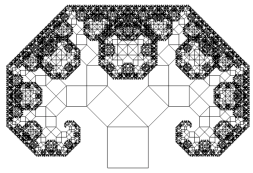


Figura 11: Árvore de Pitágoras de ordem 15.

$$
\begin{aligned}
&\mathit{mkBranches}\ [a, b, c, d] = \mathbf{let}\ d = 0.5 <*> (b <+> ((-1) <*> a))\\
&\qquad\qquad\qquad\qquad\qquad\quad l1 = d <+> \mathit{orth}\ d\\
&\qquad\qquad\qquad\qquad\qquad\quad l2 = \mathit{orth}\ l1\\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\quad [[a <+> l2, b <+> (2 <*> l2), a <+> l1, a]\\
&\qquad\qquad\qquad\qquad\qquad\quad ,[a <+> (2 <*> l1), b <+> l1, b, b <+> l2]]\\
&(a, b) <+> (c, d) = (a + c, b + d)\\
&n <*> (a, b) = (a * n, b * n)\\
&\mathit{orth}\ (a, b) = (-b, a)
\end{aligned}
$$

Tal como sugere no enunciado, se corremos $\mathit{animatePTree}\ 3$ os passos que vão aparecendo no ecrão encontram-se na Figura 10b, Figura **??** e Figura 10c, respetivamente.

Aplicando para um parâmetro maior, neste caso igual a 15, obtemos a árvore de Pitágoras representada na Figura 11.

Problema 5

O quinto e último problema diz respeito a $\mathit{monades}$. O mónade do problema é conhecido por *bag*, *saco* ou *multi-conjunto*, permitindo que os elementos de um conjunto tenham multiplicidades associadas.

Para este problema o objetivo é que completemos a definição de $\mathit{Monad\ Bag}$, nomeadamente no que diz respeito ao μ (multiplicação do mónade Bag) e a respetiva função auxiliar $\mathit{singletonbag}$. É também necessário definir a função dist.

Começando pela função $\mathit{singletonbag}$, tal como o próprio nome indica, o objetivo dela é ter um saco com apenas um elemento do valor passado como parâmetro, para ser o return do Monad.

Assim, a sua definição é intuitiva e é a seguinte:

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

É o construtor B que se encarrega de fazer com que o tipo de retorno de $\mathit{singletonbag}$ seja o desejado.

Quanto ao μ, sabemos que o seu tipo de dados deverá ser o seguinte:

$$\mu :: \mathit{Bag}\ (\mathit{Bag}\ a) \rightarrow \mathit{Bag}\ a$$

Deste modo, tendo ainda em consideração a definição de Monad sabemos que esta função recebe um Bag de Bags, como por exemplo:

$$b2 :: \mathit{Bag}\ (\mathit{Bag}\ \mathit{Marble})$$

$$b2 = B\ [(B\ [(Pink, 2), (Green, 3), (Red, 2), (Blue, 2), (White, 1)], 5)$$
$$, (B\ [(Pink, 1), (Green, 2), (Red, 1), (Blue, 1)], 2)]$$

E que o objetivo é que a função retorne um Bag somente, que, no caso de correr $\mu\ b2$ deverá ser:

$$b1 :: Bag\ Marble$$
$$b1 = B\ [(Pink, 12), (Green, 19), (Red, 12), (Blue, 12), (White, 5)]$$

Sucintamente, o objetivo do μ é que dado um Bag com Bags lá dentro todos os seus elementos se juntem num só Bag. Para tal, é necessário ter em atenção quantos Bags estão dentro do Bag maior e agrupar convenientemente os seus conteúdos, tal como vimos no exemplo anterior.

Assim, com recurso a algumas funções auxiliares produzimos o código em seguida:

$$\begin{array}{l} \mu\ b = B\ (concat\ (\mathsf{fmap}\ unB\ (junta\ (unB\ b)))) \\ \quad \mathbf{where}\ junta\ ((ba, int) : bas) = (fmapSpecial\ (*int)\ ba) : (junta\ bas) \\ \qquad junta\ [\,] = [\,] \\ \qquad fmapSpecial\ f = B \cdot \mathsf{map}\ \ (id \times f) \cdot unB \end{array}$$

O diagrama seguinte mostra as alterações nos tipos de dados que vão acontecendo no decorrer da definição de μ por nós proposta:

$$\begin{array}{c} Bag\ (Bag\ a) \\ \downarrow{\scriptstyle unB} \\ [(Bag\ a, Int)] \\ \downarrow{\scriptstyle junta} \\ [Bag\ a] \\ \downarrow{\scriptstyle \mathsf{fmap}\ unB} \\ [[(a, Int)]] \\ \downarrow{\scriptstyle concat} \\ [(a, Int)] \\ \downarrow{\scriptstyle B} \\ Bag\ a \end{array}$$

Passando a explicar por palavras passo a passo o que acontece na função μ, temos que:

1. $un\ B$

   Numa primeira fase "desembrulhamos" o Bag de Bags, passando agora a ter uma lista com $(Bag\ a, Int)$ onde poderemos iterar mais facilmente sobre os elementos.

2. $junta$

   Esta função, com a ajuda de uma função auxiliar chamada $fmapSpecial$, transforma a lista de $(Bag\ a) \times Int$, sendo que este Int representa o número de "sacos" que existem de $Bag\ a$, numa só lista de $Bag\ a$. A função multiplica o número de sacos que existem daquele tipo pelo conteúdo dentro do $Bag\ a$. Ou seja, se temos 3 sacos com 2 berlindes azuis dentro de cada um sabemos que no total temos 6 berlindes azuis, e é exatamente isso que a função faz dentro de cada $Bag\ a$ da lista.

3. fmap unB

   Este passo "abre" todos os $Bag\ a$ dentro de $[Bag\ a]$, ficando agora com o tipo $[[(a, Int)]]$. Mais uma vez, este passo é feito para facilitar o manuseamento dos elementos.

4. $concat$

   Esta função, já predefinida, é responsável por concatenar a $[[(a, Int)]]$ em $[(a, Int)]$. O que esta função faz é somente juntar os pares (a, Int) numa só lista.

5. B

   Neste último passo é aplicado o construtor de Bag à lista de (a, Int) e com isso fica garantido que os (a, Int) repetidos se juntam, somando os respetivos Int, e que o tipo de dados de retorno é o tipo desejado, nomeadamente $Bag\ a$.

Para a função $dist$, tendo em consideração o exemplo dado no enunciado, onde para o "saco":

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

O resultado da aplicação desta função deverá ser:

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

Percebemos que $dist$ divide o número de elementos de cada elemento pelo número total de elementos.
Assim, definimos a função como:

$$\begin{array}{l} dist\ (B\ a) = D\ ((\mathsf{map}\ \ (\lambda(x, y) \rightarrow (x, (/)\ (toFloat\ y)\ (toFloat\ (number\ a))))\ a) \\ \quad \mathbf{where}\ number\ [\,] = 0 \\ \qquad number\ ((\_, int) : cs) = int + number\ cs \end{array}$$

A função percorre todos os elementos de $B\ a$ ($[a, Int]$, mais especificamente, uma vez que é este o parâmetro que passamos ao map) e divide cada Int pelo número total de elementos do conjunto, que é calculado com a ajuda da função auxiliar $number$.

De modo a retornarmos o resultado da forma correta, nomeadamente Dist a, tivemos que fazer duas pequenas alterações no código: a primeira foi converter o y e o $number\ a$ (numero total de berlindes) de cada par para $Float$ antes de os dividirmos. A segunda é aplicar o construtor D, do módulo $Probability$, ao resultado do map para assim conseguirmos obter o tipo de dados desejado.

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:

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

[7] Exemplos tirados de [?].