Cálculo de Programas
Trabalho Prático
LEI+MiEI — 2021/22

Departamento de Informática
Universidade do Minho

Fevereiro de 2022

Grupo nr. 20	
a93300	Bohdan Malanka
a93316	Henrique Alvelos
a93326	João Moreira

1 Preâmbulo

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 cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). 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, a validá-los e a produzir textos técnico-científicos de qualidade.

Antes de abodarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao sofware a instalar, etc.

Problema 1

Num sistema de informação distribuído, uma lista não vazia de transações é vista como um *blockchain* sempre que possui um valor de *hash* que é dado pela raiz de uma Merkle tree que lhe está associada. Isto significa que cada *blockchain* está estruturado numa Merkle tree. Mas, o que é uma Merkle tree?

Uma Merkle tree é uma *FTree* com as seguintes propriedades:

1. as folhas são pares ($hash$, transação) ou simplesmente o $hash$ de uma transação;
2. os nodos são $hashes$ que correspondem à concatenação dos $hashes$ dos filhos;
3. o $hash$ que se encontra na raiz da árvore é designado $Merkle\ \ Root$; como se disse acima, corresponde ao valor de $hash$ de todo o bloco de transações.

(1)

Assumindo uma lista não vazia de transações, o algoritmo clássico de construção de uma $Merkle\ \ Tree$ é o que está dado na Figura 1. Contudo, este algoritmo (que se pode mostrar ser um hilomorfismo de listas não vazias) é demasiadamente complexo. Uma forma bem mais simples de produzir uma $Merkle\ \ Tree$ é através de um hilomorfismo de *LTree*s. Começa-se por, a partir da lista de transações, construir uma *LTree* cujas folhas são as transações:

$$list2LTree :: [\,a\,] \rightarrow \mathit{LTree}\ a$$

Depois, o objetivo é etiquetar essa árvore com os $hashes$,

- *Se a lista for singular, calcular o* $hash$ *da transação.*
- *Caso contrário,*
  1. *Mapear a lista com a função* $hash$.
  2. *Se o comprimento da lista for ímpar, concatenar a lista com o seu último valor (que fica duplicado). Caso contrário, a lista não sofre alterações.*
  3. *Agrupar a lista em pares.*
  4. *Concatenar os* $hashes$ *do par produzindo uma lista de (sub-)árvores nas quais a cabeça terá a respetiva concatenação.*
  5. *Se a lista de (sub-)árvores não for singular, voltar ao passo 2 com a lista das cabeças como argumento, preservando a lista de (sub-)árvores. Se a lista for singular, chegamos à* $Merkle\ Root$. *Contudo, falta compor a* $Merkle\ Tree$ *final. Para tal, tendo como resultado uma lista de listas de (sub-)árvores agrupada pelos níveis da árvore final, é necessário encaixar sucessivamente os tais níveis formando a* $Merkle\ Tree$ *completa.*

Figura 1: Algoritmo clássico de construção de uma Merkle tree [4] .

$$lTree2MTree :: Hashable\ a \Rightarrow LTree\ a \rightarrow \underbrace{FTree\ \mathbb{Z}\ (\mathbb{Z}, a)}_{Merkle\ tree}$$

formando uma Merkle tree que satisfaça os três requisitos em (1). Em suma, a construção de um blockchain é um hilomorfismo de *LTree*s

$$\begin{array}{l} computeMerkleTree :: Hashable\ a \Rightarrow [\,a\,] \rightarrow FTree\ \mathbb{Z}\ (\mathbb{Z}, a) \\ computeMerkleTree = lTree2MTree \cdot list2LTree \end{array}$$

1. Comece por definir o gene do anamorfismo que constrói *LTree*s a partir de listas não vazias:

$$\begin{array}{l} list2LTree :: [\,a\,] \rightarrow LTree\ a \\ list2LTree = [\!(g\_list2LTree)\!] \end{array}$$

NB: para garantir que $list2LTree$ não aceita listas vazias deverá usar em $g\_list2LTree$ o inverso $outNEList$ do isomorfismo

$$inNEList = [singl, cons]$$

2. Assumindo as seguintes funções $hash$ e $concHash$:[1]

$$\begin{array}{l} hash :: Hashable\ a \Rightarrow a \rightarrow \mathbb{Z} \\ hash = toInteger \cdot (Data.Hashable.hash) \\ concHash :: (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z} \\ concHash = add \end{array}$$

defina o gene do catamorfismo que consome a *LTree* e produz a correspondente Merkle tree etiquetada com todos os $hashes$:

$$\begin{array}{l} lTree2MTree :: Hashable\ a \Rightarrow LTree\ a \rightarrow FTree\ \mathbb{Z}\ (\mathbb{Z}, a) \\ lTree2MTree = (\!|\, g\_lTree2MTree \,|\!) \end{array}$$

3. Defina g_mroot por forma a

$$\begin{array}{l} mroot :: Hashable\ b \Rightarrow [\,b\,] \rightarrow \mathbb{Z} \\ mroot = (\!|\, g\_mroot \,|\!) \cdot computeMerkleTree \end{array}$$

nos dar a Merkle *root* de um qualquer bloco $[\,b\,]$ de transações.

[1]Para invocar a função $hash$, escreva $Main.hash$.

4. Calcule $mroot\ trs$ da sequência de transações trs da no anexo e verifique que, sempre que se modifica (e.g. fraudulentamente) uma transação passada em trs, $mroot\ trs$ altera-se necessariamente. Porquê? (Esse é exactamente o princípio de funcionamento da tecnologia blockchain.)

Valorização (não obrigatória): implemente o algoritmo clássico de construção de Merkle trees

$$classicMerkleTree :: Hashable\ a \Rightarrow [a] \rightarrow FTree\ \mathbb{Z}\ \mathbb{Z}$$

sob a forma de um hilomorfismo de listas não vazias. Para isso deverá definir esse combinador primeiro, da forma habitual:

$$hyloNEList\ h\ g = cataNEList\ h \cdot anaNEList\ g$$

etc. Depois passe à definição do gene $g\_pairsList$ do anamorfismo de listas

$$pairsList :: [a] \rightarrow [(a, a)]$$
$$pairsList = [\![(g\_pairsList)]\!]$$

que agrupa a lista argumento por pares, duplicando o último valor caso seja necessário. Para tal, poderá usar a função (já definida)

$$getEvenBlock :: [a] \rightarrow [a]$$

que, dada uma lista, se o seu comprimento for ímpar, duplica o último valor.

Por fim, defina os genes $divide$ e $conquer$ dos respetivos anamorfismo e catamorfimo por forma a

$$classicMerkleTree = (hyloNEList\ conquer\ divide) \cdot (\mathsf{map}\ \ Main.hash)$$

Para facilitar a definição do $conquer$, terá apenas de definir o gene $g\_mergeMerkleTree$ do catamorfismo de ordem superior

$$mergeMerkleTree :: FTree\ a\ p \rightarrow [FTree\ a\ c] \rightarrow FTree\ a\ c$$
$$mergeMerkleTree = (\!| g\_mergeMerkleTree |\!)$$

que compõe a $FTree$ (à cabeça) com a lista de $FTree$s (como filhos), fazendo um "merge" dos valores intermédios. Veja o seguinte exemplo de aplicação da função $mergeMerkleTree$:

```
 > l = [Comp 3 (Unit 1, Unit 2), Comp 7 (Unit 3, Unit 4)]
 >
 > m = Comp 10 (Unit 3, Unit 7)
 >
 > mergeMerkleTree m l
Comp 10 (Comp 3 (Unit 1,Unit 2),Comp 7 (Unit 3,Unit 4))
```

NB: o *classicMerkleTree* retorna uma Merkle Tree cujas folhas são apenas o $hash$ da transação e não o par ($hash$, transação).

Problema 2

Se se digitar $man\ \ wc$ na shell do Unix (Linux) obtém-se:

```
NAME
     wc -- word, line, character, and byte count

SYNOPSIS
     wc [-clmw] [file ...]

DESCRIPTION
    The wc utility displays the number of lines, words, and bytes contained in
    each input file,  or standard input (if no file is specified) to the stan-
    dard  output.  A line is defined as  a string of characters delimited by a
    <newline> character.  Characters beyond the final <newline> character will
    not be included in the line count.
    (...)
```

```
The following options are available:
(...)
    -w   The number of words in each input file is written to the standard
         output.
(...)
```

Se olharmos para o código da função que, em C, implementa esta funcionalidade [1] e nos focarmos apenas na parte que implementa a opção `-w`, verificamos que a poderíamos escrever, em Haskell, da forma seguinte:

$$
\begin{array}{l}
wc\_w :: [\,Char\,] \rightarrow Int \\
wc\_w\ [\,] = 0 \\
wc\_w\ (c : l) = \\
\quad \textbf{if}\ \neg\ (sep\ c) \wedge lookahead\_sep\ l\ \textbf{then}\ wc\_w\ l + 1\ \textbf{else}\ wc\_w\ l \\
\quad\quad \textbf{where} \\
\quad\quad\quad sep\ c = (c \equiv \texttt{' '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'}) \\
\quad\quad\quad lookahead\_sep\ [\,] = True \\
\quad\quad\quad lookahead\_sep\ (c : l) = sep\ c
\end{array}
$$

Por aplicação da lei de recursividade mútua

$$
\left\{ \begin{array}{l} f \cdot \mathsf{in} = h \cdot \mathsf{F}\ \langle f, g \rangle \\ g \cdot \mathsf{in} = k \cdot \mathsf{F}\ \langle f, g \rangle \end{array} \right. \quad \equiv \quad \langle f, g \rangle = (\!|\langle h, k \rangle|\!) \tag{2}
$$

às funções wc_w e $lookahead\_sep$, re-implemente a primeira segundo o modelo *worker/wrapper* onde *worker* deverá ser um catamorfismo de listas:

$$
\begin{array}{l}
wc\_w\_final :: [\,Char\,] \rightarrow Int \\
wc\_w\_final = wrapper \cdot \underbrace{(\!|[g1\,, g2]|\!)}_{worker}
\end{array}
$$

Apresente os cálculos que fez para chegar à versão wc_w_final de wc_w, com indicação dos genes h, k e $g = [g1\,, g2]$.

Problema 3

Neste problema pretende-se gerar o HTML de uma página de um jornal descrita como uma agregação estruturada de blocos de texto ou imagens:

$$\textbf{data}\ Unit\ a\ b = Image\ a \mid Text\ b\ \textbf{deriving}\ Show$$

O tipo $Sheet$ (="página de jornal")

$$\textbf{data}\ Sheet\ a\ b\ i = Rect\ (Frame\ i)\ (X\ (Unit\ a\ b)\ (Mode\ i))\ \textbf{deriving}\ Show$$

é baseado num tipo indutivo X que, dado em anexo (pág. 10), exprime a partição de um rectângulo (a página tipográfica) em vários subrectângulos (as caixas tipográficas a encher com texto ou imagens), segundo um processo de partição binária, na horizontal ou na vertical. Para isso, o tipo

$$\textbf{data}\ Mode\ i = Hr\ i \mid Hl\ i \mid Vt\ i \mid Vb\ i\ \textbf{deriving}\ Show$$

especifica quatro variantes de partição. O seu argumento deverá ser um número de 0 a 1, indicando a fracção da altura (ou da largura) em que o rectângulo é dividido, a saber:

- `Hr i` — partição horizontal, medindo i a partir da direita
- `Hl i` — partição horizontal, medindo i a partir da esquerda
- `Vt i` — partição vertical, medindo i a partir do topo
- `Vb i` — partição vertical, medindo i a partir da base

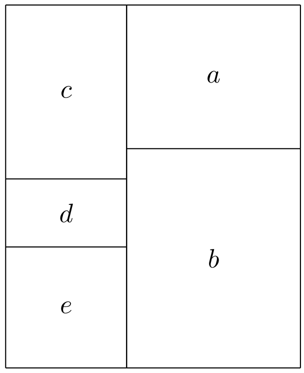
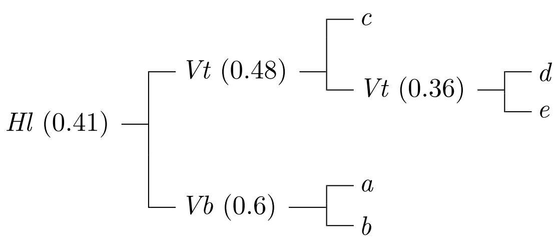


Figura 2: Layout de página de jornal.

Por exemplo, a partição dada na figura 2 corresponde à partição de um rectângulo de acordo com a seguinte árvore de partições:



As caixas delineadas por uma partição (como a dada acima) correspondem a folhas da árvore de partição e podem conter texto ou imagens. É o que se verifica no objecto $example$ da secção B que, processado por $sheet2html$ (secção B) vem a produzir o ficheiro `jornal.html`.

O que se pretende O código em Haskell fornecido no anexo B como "kit" para arranque deste trabalho não está estruturado em termos dos combinadores *cata-ana-hylo* estudados nesta disciplina. O que se pretende é, então:

1. A construção de uma biblioteca "pointfree" [2] com base na qual o processamento ("pointwise") já disponível possa ser redefinido.
2. A evolução da biblioteca anterior para uma outra que permita partições n-árias (para *qualquer* n finito) e não apenas binárias. [3]

Problema 4

Este exercício tem como objectivo determinar todos os caminhos possíveis de um ponto A para um ponto B. Para tal, iremos utilizar técnicas de *brute force* e *backtracking*, que podem ser codificadas no mónade das listas (estudado na aulas). Comece por implementar a seguinte função auxiliar:

1. $pairL :: [a] \to [(a,a)]$ que dada uma lista l de tamanho maior que 1 produz uma nova lista cujos elementos são os pares (x,y) de elementos de l tal que x precede imediatamente y. Por exemplo:

   $pairL\ [1,2] \equiv [(1,2)]$,
   $pairL\ [1,2,3] \equiv [(1,2),(2,3)]$ e
   $pairL\ [1,2,3,4] \equiv [(1,2),(2,3),(3,4)]$

   Para o caso em que $l = [x]$, i.e. o tamanho de l é 1, assuma que $pairL\ [x] \equiv [(x,x)]$. Implemente esta função como um *anamorfismo de listas*, atentando na sua propriedade:

[2] A desenvolver de forma análoga a outras bibliotecas que conhece (*eg.* *LTree*, etc).
[3] Repare que é a falta desta capacidade expressiva que origina, no "kit" actual, a definição das funções auxiliares da secção B, por exemplo.

- *Para todas as listas l de tamanho maior que 1, a lista* `map` π_1 $(pairL\ l)$ *é a lista original l a menos do último elemento. Analogamente, a lista* `map` π_2 $(pairL\ l)$ *é a lista original l a menos do primeiro elemento.*

De seguida necessitamos de uma estrutura de dados representativa da noção de espaço, para que seja possível formular a noção de *caminho* de um ponto A para um ponto B, por exemplo, num papel quadriculado. No nosso caso vamos ter:

$$\mathbf{data}\ Cell = Free \mid Blocked \mid Lft \mid Rght \mid Up \mid Down\ \mathbf{deriving}\ (Eq, Show)$$
$$\mathbf{type}\ Map = [[Cell]]$$

O terreno onde iremos navegar é codificado então numa *matriz* de células. Os valores *Free* and *Blocked* denotam uma célula como livre ou bloqueada, respectivamente (a navegação entre dois pontos terá que ser realizada *exclusivamente* através de células livres). Ao correr, por exemplo, $putStr\ \$\ showM\ \$\ map_1$ no interpretador irá obter a seguinte apresentação de um mapa:

```
_ _ _
_ X _
_ X _
```

Para facilitar o teste das implementações pedidas abaixo, disponibilizamos no anexo B a função $testWithRndMap$. Por exemplo, ao correr $testWithRndMap$ obtivemos o seguinte mapa aleatoriamente:

```
_ _ _ _ _ _ X _ _ X
_ X _ _ _ X _ _ _ _
_ _ _ _ _ X _ _ _ _
_ X _ _ _ _ _ _ _ X
_ _ _ _ _ _ X _ X _
_ _ _ _ _ _ _ _ _ _
_ X X _ _ X _ _ _ _
_ _ _ _ _ _ _ _ X _
_ _ _ _ _ X _ _ X _
_ _ X _ _ _ _ _ _ X
Map of dimension 10x10.
```

De seguida, os valores Lft, $Rght$, Up e $Down$ em $Cell$ denotam o facto de uma célula ter sido alcançada através da célula à esquerda, direita, de cima, ou de baixo, respectivamente. Tais valores irão ser usados na representação de caminhos num mapa.

2. Implemente agora a função $markMap :: [Pos] \to Map \to Map$, que dada uma lista de posições (representante de um *caminho* de um ponto A para um ponto B) e um mapa retorna um novo mapa com o caminho lá marcado. Por exemplo, ao correr no interpretador,

$$putStr\ \$\ showM\ \$\ markMap\ [(0,0),(0,1),(0,2),(1,2)]\ map_1$$

deverá obter a seguinte apresentação de um mapa e respectivo caminho:

```
> _ _
^ X _
^ X _
```

representante do caso em que subimos duas vezes no mapa e depois viramos à direita. Para implementar a função $markMap$ deverá recorrer à função $toCell$ (disponibilizada no anexo B) e a uma função auxiliar com o tipo $[(Pos, Pos)] \to Map \to Map$ definida como um *catamorfismo de listas*. Tal como anteriormente, anote as propriedades seguintes sobre $markMap$:[4]

- *Para qualquer lista l a função $markMap\ l$ é idempotente.*
- *Todas as posições presentes na lista dada como argumento irão fazer com que as células correspondentes no mapa deixem de ser Free.*

[4] Ao implementar a função $markMap$, estude também a função $subst$ (disponibilizada no anexo B) pois as duas funções tem algumas semelhanças.

Finalmente há que implementar a função $scout :: Map \to Pos \to Pos \to Int \to [[Pos]]$, que dado um mapa m, uma posição inicial s, uma posição alvo t, e um número inteiro n, retorna uma lista de caminhos que começam em s e que têm tamanho máximo $n + 1$. Nenhum destes caminhos pode conter t como elemento que não seja o último na lista (i.e. um caminho deve terminar logo que se alcança a posição t). Para além disso, não é permitido voltar a posições previamente visitadas e se ao alcançar uma posição diferente de t é impossivel sair dela então todo o caminho que levou a esta posição deve ser removido (*backtracking*). Por exemplo:

$scout\ map_1\ (0,0)\ (2,0)\ 0 \equiv [[(0,0)]]$
$scout\ map_1\ (0,0)\ (2,0)\ 1 \equiv [[(0,0),(0,1)]]$
$scout\ map_1\ (0,0)\ (2,0)\ 4 \equiv [[(0,0),(0,1),(0,2),(1,2),(2,2)]]$
$scout\ map_2\ (0,0)\ (2,2)\ 2 \equiv [[(0,0),(0,1),(1,1)],[(0,0),(0,1),(0,2)]]$
$scout\ map_2\ (0,0)\ (2,2)\ 4 \equiv [[(0,0),(0,1),(1,1),(2,1),(2,2)],[(0,0),(0,1),(1,1),(2,1),(2,0)]]$

3. Implemente a função

$$scout :: Map \to Pos \to Pos \to Int \to [[Pos]]$$

recorrendo à função $checkAround$ (disponibilizada no anexo B) e de tal forma a que $scout\ m\ s\ t$ seja um catamorfismos de naturais *monádico*. Anote a seguinte propriedade desta função:

- *Quanto maior for o tamanho máximo permitido aos caminhos, mais caminhos que alcançam a posição alvo iremos encontrar.*

Anexos

A Documentação para realizar o trabalho

Para cumprir de forma integrada os objectivos Rdo trabalho vamos recorrer a uma técnica de programação dita "literária" [2], 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 `cp2122t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2122t.lhs`[5] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2122t.zip` e executando:

```
$ lhs2TeX cp2122t.lhs > cp2122t.tex
$ pdflatex cp2122t
```

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 --lib
$ cabal install --ghc-option=-dynamic lhs2tex
```

NB: utilizadores do macOS poderão instalar o *cabal* com o seguinte comando:

```
$ brew install cabal-install
```

Por outro lado, o mesmo ficheiro `cp2122t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2122t.lhs
```

Abra o ficheiro `cp2122t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

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

é seleccionado pelo GHCi para ser executado.

A.1 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de 3 (ou 4) 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 em todos os exercícios do trabalho, para assim poderem responder a qualquer questão colocada 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 cp2122t.aux
$ makeindex cp2122t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck, que ajuda a validar programas em Haskell:

```
$ cabal install QuickCheck --lib
```

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

[5] O sufixo 'lhs' quer dizer *literate Haskell*.

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

Pode-se ainda controlar o número de casos de teste e sua complexidade, como o seguinte exemplo mostra:[6]

```
> quickCheckWith stdArgs { maxSuccess = 200, maxSize = 10 } prop
+++ OK, passed 200 tests.
```

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo B disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Stack O Stack é um programa útil para criar, gerir e manter projetos em Haskell. Um projeto criado com o Stack possui uma estrutura de pastas muito específica:

- Os módulos auxiliares encontram-se na pasta *src*.
- O módulo principal encontra-se na pasta *app*.
- A lista de dependências externas encontra-se no ficheiro *package.yaml*.

Pode aceder ao GHCi utilizando o comando:

```
stack ghci
```

Garanta que se encontra na pasta mais externa **do projeto**. A primeira vez que correr este comando as depêndencias externas serão instaladas automaticamente. Para gerar o PDF, garanta que se encontra na diretoria *app*.

A.2 Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:[7]

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

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

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

[6] Como já sabe, os testes normalmente não provam a ausência de erros no código, apenas a sua presença (cf. arquivo online). Portanto não deve ver o facto de o seu código passar nos testes abaixo como uma garantia que este está livre de erros.

[7] Exemplos tirados de [3].

B Código fornecido

Problema 1

Sequência de transações para teste:

```
trs = [("compra", "20211102", -50),
  ("venda",      "20211103", 100),
  ("despesa", "20212103", -20),
  ("venda",      "20211205", 250),
  ("venda",      "20211205", 120)]
```

```
getEvenBlock :: [a] -> [a]
getEvenBlock l = if (even (length l)) then l else l ++ [last l]
firsts = [π1, π1]
```

Problema 2

```
wc_test = "Here is a sentence, for testing.\nA short one."
sp c = (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t')
```

Problema 3

Tipos:

```
data X u i = XLeaf u | Node i (X u i) (X u i) deriving Show
data Frame i = Frame i i deriving Show
```

Funções da API[8]

```
printJournal :: Sheet String String Double -> IO ()
printJournal = write · sheet2html
write :: String -> IO ()
write s = do writeFile "jornal.html" s
  putStrLn "Output HTML written into file `jornal.html'"
```

Geração de HTML:

```
sheet2html (Rect (Frame w h) y) = htmlwrap (new_x2html y (w, h))
x2html :: X (Unit String String) (Mode Double) -> (Double, Double) -> String
x2html (XLeaf (Image i)) (w, h) = img w h i
x2html (XLeaf (Text txt)) _ = txt
x2html (Node (Vt i) x1 x2) (w, h) = htab w h (
  tr (td w (h * i) (x2html x1 (w, h * i))) ++
  tr (td w (h * (1 - i)) (x2html x2 (w, h * (1 - i))))
  )
x2html (Node (Hl i) x1 x2) (w, h) = htab w h (
  tr (td (w * i) h (x2html x1 (w * i, h)) ++
    td (w * (1 - i)) h (x2html x2 (w * (1 - i), h)))
  )
x2html (Node (Vb i) x1 x2) m = x2html (Node (Vt (1 - i)) x1 x2) m
x2html (Node (Hr i) x1 x2) m = x2html (Node (Hl (1 - i)) x1 x2) m
```

Funções auxiliares:

[8] API (="Application Program Interface").

```
twoVtImg a b = Node (Vt 0.5) (XLeaf (Image a)) (XLeaf (Image b))
fourInArow a b c d =
  Node (Hl 0.5)
    (Node (Hl 0.5) (XLeaf (Text a)) (XLeaf (Text b)))
    (Node (Hl 0.5) (XLeaf (Text c)) (XLeaf (Text d)))
```

HTML:

```
htmlwrap = html · hd · (title "CP/2122 - sheet2html") · body · divt
html = tag "html" [] · ("<meta charset=\"utf-8\" />"++)
title t = (tag "title" [] t++)
body = tag "body" ["BGCOLOR" ↦ show "#F4EFD8"]
hd = tag "head" []
htab w h = tag "table" [
  "width" ↦ show2 w, "height" ↦ show2 h,
  "cellpadding" ↦ show2 0, "border" ↦ show "1px"]
tr = tag "tr" []
td w h = tag "td" ["width" ↦ show2 w, "height" ↦ show2 h]
divt = tag "div" ["align" ↦ show "center"]
img w h i = tag "img" ["width" ↦ show2 w, "src" ↦ show i] ""
tag t l x = "<" ++ t ++ " " ++ ps ++ ">" ++ x ++ "</" ++ t ++ ">\n"
  where ps = unwords [concat [t, "=", v] | (t, v) ← l]
a ↦ b = (a, b)
show2 :: Show a ⇒ a → String
show2 = show · show
```

Exemplo para teste:

```
example :: (Fractional i) ⇒ Sheet String String i
example =
  Rect (Frame 650 450)
    (Node (Vt 0.01)
      (Node (Hl 0.15)
        (XLeaf (Image "cp2122t_media/publico.jpg"))
        (fourInArow "Jornal Público" "Domingo, 5 de Dezembro 2021" "Simulação para efe
      (Node (Vt 0.55)
        (Node (Hl 0.55)
          (Node (Vt 0.1)
            (XLeaf (Text
            "Universidade do Algarve estuda planta capaz de eliminar a doença do sol
            (XLeaf (Text
              "Organismo (semelhante a um fungo) ataca de forma galopante os montado
          (XLeaf (Image
              "cp2122t_media/1647472.jpg")))
        (Node (Hl 0.25)
          (twoVtImg
              "cp2122t_media/1647981.jpg"
              "cp2122t_media/1647982.jpg")
          (Node (Vt 0.1)
              (XLeaf (Text "Manchester United vence na estreia de Rangnick"))
              (XLeaf (Text "O Manchester United venceu, este domingo, em Old Trafford,
```

Problema 4

Exemplos de mapas:

$$map_1 = [[Free, Blocked, Free], [Free, Blocked, Free], [Free, Free, Free]]$$
$$map_2 = [[Free, Blocked, Free], [Free, Free, Free], [Free, Blocked, Free]]$$
$$map_3 = [[Free, Free, Free], [Free, Blocked, Free], [Free, Blocked, Free]]$$

Código para impressões de mapas e caminhos:

$$\begin{array}{l}
showM :: Map \to String \\
showM = unlines \cdot (\mathsf{map}\ \ showL) \cdot reverse \\
showL :: [Cell] \to String \\
showL = (\![f_1, f_2]\!) \ \mathbf{where} \\
\quad f_1 = \underline{\texttt{""}} \\
\quad f_2 = \widehat{(+\!\!+)} \cdot (fromCell \times id) \\
fromCell\ Lft = \texttt{" > "} \\
fromCell\ Rght = \texttt{" < "} \\
fromCell\ Up = \texttt{" \^{} "} \\
fromCell\ Down = \texttt{" v "} \\
fromCell\ Free = \texttt{" _ "} \\
fromCell\ Blocked = \texttt{" X "} \\
toCell\ (x, y)\ (w, z) \mid x < w = Lft \\
toCell\ (x, y)\ (w, z) \mid x > w = Rght \\
toCell\ (x, y)\ (w, z) \mid y < z = Up \\
toCell\ (x, y)\ (w, z) \mid y > z = Down
\end{array}$$

Código para validação de mapas (útil, por exemplo, para testes QuickCheck):

$$\begin{array}{l}
ncols :: Map \to Int \\
ncols = [\underline{0}, length \cdot \pi_1] \cdot outList \\
nlines :: Map \to Int \\
nlines = length \\
isValidMap :: Map \to Bool \\
isValidMap = \widehat{(\wedge)} \cdot \langle isSquare, sameLength \rangle \ \mathbf{where} \\
\quad isSquare = \widehat{(\equiv)} \cdot \langle nlines, ncols \rangle \\
\quad sameLength\ [\,] = True \\
\quad sameLength\ [x] = True \\
\quad sameLength\ (x1 : x2 : y) = length\ x1 \equiv length\ x2 \wedge sameLength\ (x2 : y)
\end{array}$$

Código para geração aleatória de mapas e automatização de testes (envolve o mónade IO):

$$\begin{array}{l}
randomRIOL :: (Random\ a) \Rightarrow (a, a) \to Int \to \mathsf{IO}\ [a] \\
randomRIOL\ x = (\![f_1, f_2]\!) \ \mathbf{where} \\
\quad f_1 = \underline{return\ [\,]} \\
\quad f_2\ l = \mathbf{do}\ r1 \leftarrow randomRIO\ x \\
\qquad r2 \leftarrow l \\
\qquad return\ \$\ r1 : r2 \\
buildMat :: Int \to Int \to \mathsf{IO}\ [[Int]] \\
buildMat\ n = (\![f_1, f_2]\!) \ \mathbf{where} \\
\quad f_1 = \underline{return\ [\,]} \\
\quad f_2\ l = \mathbf{do}\ x \leftarrow randomRIOL\ (0 :: Int, 3 :: Int)\ n \\
\qquad y \leftarrow l \\
\qquad return\ \$\ x : y \\
testWithRndMap :: \mathsf{IO}\ () \\
testWithRndMap = \mathbf{do} \\
\quad dim \leftarrow randomRIO\ (2, 10) :: \mathsf{IO}\ Int \\
\quad out \leftarrow buildMat\ dim\ dim \\
\quad \mathsf{map}\ \leftarrow return\ \$\ \mathsf{map}\ \ (\mathsf{map}\ \ table)\ out \\
\quad putStr\ \$\ showM\ \mathsf{map} \\
\quad putStrLn\ \$\ \texttt{"Map of dimension "} +\!\!+ (show\ dim) +\!\!+ \texttt{"x"} +\!\!+ (show\ dim) +\!\!+ \texttt{"."}
\end{array}$$

```
putStr "Please provide a target position (must be different from (0,0)): "
```
$$t \leftarrow readLn :: \mathsf{IO}\ (Int, Int)$$
```
putStr "Please provide the number of steps to compute: "
```
$$n \leftarrow readLn :: \mathsf{IO}\ Int$$
$$\mathbf{let}\ paths = hasTarget\ t\ (scout\ \mathsf{map}\ (0,0)\ t\ n)\ \mathbf{in}$$
$$\mathbf{if}\ length\ paths \equiv 0$$
$$\mathbf{then}\ putStrLn\ \texttt{"No paths found."}$$
$$\mathbf{else}\ putStrLn\ \$\ \texttt{"There are at least "} +\!\!+ (show\ \$\ length\ paths) +\!\!+$$
$$\texttt{" possible paths. Here is one case: \textbackslash n"} +\!\!+ (showM\ \$\ markMap\ (head\ paths)\ \mathsf{map})$$

$$table\ 0 = Free$$
$$table\ 1 = Free$$
$$table\ 2 = Free$$
$$table\ 3 = Blocked$$
$$hasTarget\ y = filter\ (\lambda l \rightarrow elem\ y\ l)$$

Funções auxiliares $subst :: a \rightarrow Int \rightarrow [a] \rightarrow [a]$, que dado um valor x e um inteiro n, produz uma função $f : [a] \rightarrow [a]$ que dada uma lista l substitui o valor na posição n dessa lista pelo valor x:

$$subst :: a \rightarrow Int \rightarrow [a] \rightarrow [a]$$
$$subst\ x = (\![f_1, f_2]\!)\ \mathbf{where}$$
$$f_1 = \underline{\lambda l \rightarrow x : tail\ l}$$
$$f_2\ f\ (h : t) = h : f\ t$$

$checkAround :: Map \rightarrow Pos \rightarrow [Pos]$, que verifica se as células adjacentes estão livres:

$$\mathbf{type}\ Pos = (Int, Int)$$
$$checkAround :: Map \rightarrow Pos \rightarrow [Pos]$$
$$checkAround\ m\ p = concat\ \$\ \mathsf{map}\ (\lambda f \rightarrow f\ m\ p)$$
$$[checkLeft, checkRight, checkUp, checkDown]$$
$$checkLeft :: Map \rightarrow Pos \rightarrow [Pos]$$
$$checkLeft\ m\ (x,y) = \mathbf{if}\ x \equiv 0 \vee (m\ !!\ y)\ !!\ (x-1) \equiv Blocked$$
$$\mathbf{then}\ [\,]\ \mathbf{else}\ [(x-1, y)]$$
$$checkRight :: Map \rightarrow Pos \rightarrow [Pos]$$
$$checkRight\ m\ (x,y) = \mathbf{if}\ x \equiv (ncols\ m - 1) \vee (m\ !!\ y)\ !!\ (x+1) \equiv Blocked$$
$$\mathbf{then}\ [\,]\ \mathbf{else}\ [(x+1, y)]$$
$$checkUp :: Map \rightarrow Pos \rightarrow [Pos]$$
$$checkUp\ m\ (x,y) = \mathbf{if}\ y \equiv (nlines\ m - 1) \vee (m\ !!\ (y+1))\ !!\ x \equiv Blocked$$
$$\mathbf{then}\ [\,]\ \mathbf{else}\ [(x, y+1)]$$
$$checkDown :: Map \rightarrow Pos \rightarrow [Pos]$$
$$checkDown\ m\ (x,y) = \mathbf{if}\ y \equiv 0 \vee (m\ !!\ (y-1))\ !!\ x \equiv Blocked$$
$$\mathbf{then}\ [\,]\ \mathbf{else}\ [(x, y-1)]$$

QuickCheck

Lógicas:

$$\mathbf{infixr}\ 0 \Rightarrow$$
$$(\Rightarrow) :: (Testable\ prop) \Rightarrow (a \rightarrow Bool) \rightarrow (a \rightarrow prop) \rightarrow a \rightarrow Property$$
$$p \Rightarrow f = \lambda a \rightarrow p\ a \Rightarrow f\ a$$
$$\mathbf{infixr}\ 0 \Leftrightarrow$$
$$(\Leftrightarrow) :: (a \rightarrow Bool) \rightarrow (a \rightarrow Bool) \rightarrow a \rightarrow Property$$
$$p \Leftrightarrow f = \lambda a \rightarrow (p\ a \Rightarrow property\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow property\ (p\ a))$$
$$\mathbf{infixr}\ 4 \equiv$$
$$(\equiv) :: Eq\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow Bool)$$
$$f \equiv g = \lambda a \rightarrow f\ a \equiv g\ a$$

$\textbf{infixr } 4 \leqslant$
$(\leqslant) :: Ord\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)$
$f \leqslant g = \lambda a \to f\ a \leqslant g\ a$
$\textbf{infixr } 4 \wedge$
$(\wedge) :: (a \to Bool) \to (a \to Bool) \to (a \to Bool)$
$f \wedge g = \lambda a \to ((f\ a) \wedge (g\ a))$
instance *Arbitrary Cell* **where**
  -- 1/4 chance of generating a cell 'Block'.
  $arbitrary = \textbf{do } x \leftarrow chooseInt\ (0, 3)$
    $return\ \$\ f\ x\ \textbf{where}$
      $f\ x = \textbf{if } x < 3 \textbf{ then } Free \textbf{ else } Blocked$

C Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes.

Problema 1

inNEList

$$
\begin{aligned}
& inNEList = [singl,\ cons] \\
\equiv \quad & \{ \text{ Universal-+ (17) } \} \\
& \left\{ \begin{array}{l} inNEList \cdot i_1 = singl \\ inNEList \cdot i_2 = cons \end{array} \right. \\
\equiv \quad & \{ \text{ Igualdade Extensional (71), Def-comp (72) } \} \\
& \left\{ \begin{array}{l} inNEList\ (i_1\ a) = singl\ a \\ inNEList\ (i_2\ (h, t)) = cons\ (h, t) \end{array} \right.
\end{aligned}
$$

outNEList

$$
\begin{aligned}
& outNEList\ .\ inNEList = id \\
\equiv \quad & \{ \text{ Definição inNEList } \} \\
& outNEList\ .\ [singl,\ cons] = id \\
\equiv \quad & \{ \text{ Fusão-+ (20) } \} \\
& [outNEList\ .\ singl,\ outNEList\ .\ cons] = id \\
\equiv \quad & \{ \text{ Universal-+ (17) } \} \\
& \left\{ \begin{array}{l} id \cdot i_1 = outNEList \cdot singl \\ id \cdot i_2 = outNEList \cdot cons \end{array} \right. \\
\equiv \quad & \{ \text{ Natural-id (1), Igualdade Extensional (71), Def-comp (72) } \} \\
& \left\{ \begin{array}{l} outNEList\ (single\ a) = i_1\ a \\ outNEList\ (cons\ (h, t)) = i_2\ (h, t) \end{array} \right. \\
\equiv \quad & \{ \text{ singl a = [a], cons(h,t) = h:t } \} \\
& \left\{ \begin{array}{l} outNEList\ [a] = i_1\ a \\ outNEList\ (h : t) = i_2\ (h, t) \end{array} \right.
\end{aligned}
$$

recNEList

Como a única diferença de NEList das List normais é o caso base, a estrutura das listas não vazias é identica ás das listas comums.

$$\begin{tikzcd} NEList A \arrow[d, "f"'] \arrow[r, bend left, "outNEList"] \arrow[r, phantom, "\cong"] & A + (A \times NEList A) \arrow[l, bend left, "inNEList"] \arrow[d, "id + id \times f"] \\ C & A + (A \times C) \arrow[l] \end{tikzcd}$$

Assim,

$$baseNEList\ f\ g = id + f \times g$$

$$\begin{aligned} & recNEList\ f = id + id \times f \\ \equiv & \quad \{ \text{Aplicando a definição dada de baseNEList} \} \\ & recNEList\ f = baseNEList\ id\ f \end{aligned}$$

cataNEList

$$\begin{aligned} \equiv & \quad \{ \text{Cancelamento-cata (46)} \} \\ & (\!|g|\!) \,.\, in = g \,.\, F(\!|g|\!) \\ \equiv & \quad \{ \text{in.out = id} \} \\ & (\!|g|\!) = g \,.\, F(\!|g|\!) \,.\, out \\ \equiv & \quad \{ \text{Aplicando as definições em Haskell já determinadas} \} \\ & cataNEList\ g = g \,.\, recNEList(cataNEList\ g) \,.\, outNEList \end{aligned}$$

anaNEList

$$\begin{aligned} \equiv & \quad \{ \text{Cancelamento-ana (55)} \} \\ & out \,.\, [\!(g)\!] = F[\!(g)\!] \,.\, g \\ \equiv & \quad \{ \text{in.out = id} \} \\ & [\!(g)\!] = in \,.\, F[\!(g)\!] \,.\, g \\ \equiv & \quad \{ \text{Aplicando as definições em Haskell já determinadas} \} \\ & anaNEList\ g = inNEList \,.\, recNEList(anaNEList\ g) \,.\, g \end{aligned}$$

hyloNEList

$$
\begin{array}{ll}
\equiv & \{ \text{ Definição de hilomorfismo } \} \\
& hyloNEList\ g\ h = (|g|).[(h)] \\
\equiv & \{ \text{ Cancelamento-cata (46); Cancelamento-ana(55) } \} \\
& hyloNEList\ g\ h = g.F(|g|).inNEList.outNEList.F[(h)].h \\
\equiv & \{ \text{ in.out = id; Aplicando as definições em Haskell já determinadas } \} \\
& hyloNEList\ g\ h = g.recNEList(cataNEListg).recNEList(anaNEListh).h
\end{array}
$$

list2LTree

$$
\begin{tikzcd}
LTreeA \arrow[rr, bend left, "\mathsf{out}"] \arrow[rr, phantom, "\cong"] & & A + (LTreeA \times LTreeA) \arrow[ll, bend left, "\mathsf{in}"] \\
NEList\ A \arrow[u, "list2LTree"] \arrow[r, "outNEList"] \arrow[rr, bend right, "g\_list2LTree"'] & A + (A \times NEListA) \arrow[r, "id+singl\times id"] & A + (NEListA \times NEListA) \arrow[u, "id+id\times(list2LTree)^2"']
\end{tikzcd}
$$

Se *g_list2LTree* recebe um [**a**] a função *outNEList* injeta o elemento **a** à esquerda. Seguidamente, esse elemento é preservado pelas funções *id* e por fim transformando numa *LTree*, neste caso através da função *inLTree* é gerado uma *Leaf a*.

Se *g_list2LTree* recebe um (**h**:**t**) a função *outNEList* injeta (**h**,**t**) à direita. O objetivo agora é aplicar recursivamente a função *list2LTree* a este valor. Porém, esta função só recebe listas como argumento, então ao par (**h**,**t**) aplica-se a função *singl* ao **h** e ao **t** não se mexe, formando assim, o par ([**h**], **t**). Deste modo, aplica-se agora a função *list2LTree* a cada lado do par, construindo outro par *(Leaf h, Fork(...,...))*, o qual é passado á função *inLTree* que constroí a árvore final.

Deste modo podemos deduzir que:

$$\boxed{g\_list2LTree = id + singl \times id \cdot outNEList}$$

No entanto, desta forma, ao correr esta função com a lista de teste disponibilizada reparamos que a árvore gerada não é balanceda, pois as folhas *(Leaf)* ficam sempre à esquerda e as ramificações *(Fork)* à direita.


```
*Main> list2LTree trs
Fork (Leaf ("compra","20211102",-50),Fork (Leaf ("venda","20211103",100),Fork (Leaf ("despesa","20212103",-20)
,Fork (Leaf ("venda","20211205",250),Leaf ("venda","20211205",120)))))
*Main>
```

Assim, reparamos que o problema estava no *outNEList*, porque vai sempre separar a cabeça da cauda e que posteriormente o *inLTree* vai inserir a cabeça numa folha e a cauda num *Fork*. Nesse sentido, a separação deve ser feita partindo a lista ao meio e assim sucessivamente e falando desta forma percebemos que este *divide* de assemelha com o do algoritmo *mergeSort*. Portanto, reparamos que na biblioteca *LTree.hs* encontra-se este algoritmo e o seu anamorfismo, que é o que pretendemos. Logo, alterando a definição do gene para:

$$\boxed{g\_list2LTree = lsplit}$$

a arvore fica balanceada:

```
*Main> list2LTree trs
Fork (Fork (Fork (Leaf ("compra","20211102",-50),Leaf ("venda","20211205",120)),Leaf ("despesa","20212103",-20))
,Fork (Leaf ("venda","20211103",100),Leaf ("venda","20211205",250)))
*Main> 
```

lTree2MTree

A partir do diagrama seguinte do catamorfismo de *LTree's* vamos tentar definir o gene *g_lTree2MTree*:

$$\begin{tikzcd}
LTree A \arrow[rr, bend left, "out"] \arrow[rr, phantom, "\cong"] \arrow[d, "lTree2MTree"'] & & A + (LTree A \times LTree A) \arrow[ll, bend left, "in"] \arrow[d, "id+(lTree2MTree \times lTree2MTree)"] \\
FTree\ \mathbb{Z}\ (\mathbb{Z}, A) & & A + (FTree\ \mathbb{Z}\ (\mathbb{Z}, A) \times FTree\ \mathbb{Z}\ (\mathbb{Z}, A)) \arrow[ll, "g_lTree2MTree"]
\end{tikzcd}$$

Notemos que *g_lTree2MTree* "sai"de uma soma e portanto usar-se-á o combinador *either*.
Vamos começar por definir *g_lTree2MTree* = $[g1, g2]$.
Ao executar o *outLTree*, o "caso da esquerda"é um elemento que corresponde a uma folha que neste momento apenas possui a tarnsação "a"e que é preservada pela função *id*. Assim, a função $g1$ recebe uma tarnsação "a"e precisa criar um par em que o primeiro elemento é *hash a* e o segundo é a própria tarnsação "a":

$$A \xrightarrow{\langle id, id\rangle} (A, A) \xrightarrow{hash \times id} (\mathbb{Z}, A) \xrightarrow{inFTree.i1} Unit\ (\mathbb{Z}, A)$$

$$
\begin{aligned}
& g1 = \mathsf{in} \cdot i_1 \cdot (hash \times id) \cdot \langle id, id\rangle \\
\equiv \quad & \{ \text{ Absorção-x (11), Natural-id (1) } \} \\
& g1 = \mathsf{in} \cdot i_1 \cdot \langle hash, id\rangle \\
\equiv \quad & \{ \text{ Definição inFTree } \} \\
& g1 = [\, Unit, \widehat{Comp}\,] \cdot \langle hash, id\rangle \\
\equiv \quad & \{ \text{ Cancelamento-+(18) } \} \\
& g1 = Unit \cdot \langle hash, id\rangle
\end{aligned}
$$

Finalizando, podemos definir a função $g1$ como:

$$\boxed{g1 = Unit \cdot \langle hash, id\rangle}$$

Relativamente ao $g2$, a função trata do caso em que a árvore não é uma folha mas uma ramificação *Fork*. Assim, aplicando recursivamente a função *lTree2MTree* ao par *(LTree A, LTree A)*, resultante da ramificação devolvida pela função *outLTree* obtém-se outro par $(FTree\ \mathbb{Z}\ (\mathbb{Z}, A), FTree\ \mathbb{Z}\ (\mathbb{Z}, A))$. Por fim, resta a função $g2$ juntar essas sub árvores numa só, concatenando a informação desses nodos e guardando no nodo do pai.

$$
\begin{tikzcd}
\textit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A) \arrow[d, "\langle auxConcHash, id \rangle"'] \\
(\mathbb{Z}, (\textit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A), \textit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A))) \arrow[d, "\mathsf{in} \cdot i_2 = \widehat{Comp}"'] \\
\textit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A)
\end{tikzcd}
$$

Finalizando, podemos definir a função $g2$ como:

$$\boxed{g2 = \widehat{Comp} \cdot \langle auxConcHash, id \rangle}$$

Em que a função auxiliar *auxConcHash* definida mais abaixo, recebe duas árvores *FTree* e aplica a função *concHash* à informação contida nos nodos da árvore.

```
*Main> computeMerkleTree trs
Comp 7233215638271874210 (Comp 1335611519691781255 (Comp 8405034121674606773 (Unit (3621895334249387219,("compra","20211102",-50))
,Unit (4783138787425219554,("venda","20211205",120))),Unit (-3534711300991412759,("despesa","20212103",-20))),Comp 589760411858009
2955 (Unit (1114465331154873531,("venda","20211103",100)),Unit (4783138787425219424,("venda","20211205",250))))
*Main> 
```

mroot

$$
\begin{tikzcd}
\textit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A) \arrow[r, phantom, "\cong"] \arrow[r, bend left, "\mathsf{out}"] \arrow[d, "mroot"'] & (\mathbb{Z}, A) + (\mathbb{Z} \times (\textit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A))^2) \arrow[l, bend left, "\mathsf{in}"] \arrow[d, "id + id \times mroot^2"] \\
\mathbb{Z} & (\mathbb{Z}, A) + (\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})) \arrow[l, "g\_mroot"]
\end{tikzcd}
$$

Nesta função quando o bloco [b] tem apenas uma transação a *Merkle root* esta no primeiro elemento do par $(\mathbb{Z}, A)$ gerado à esquerda.

$$\boxed{Assim, g\_mroot = [\pi_1, ?]}$$

Quando a árvore não é apenas uma folha a *Merkle root* está no nodo da raiz que é o primeiro elemento do par $(\mathbb{Z}, (\mathbb{Z}, A))$ gerado à direita, pelo que

$$\boxed{g\_mroot = [\pi_1, \pi_1]}$$

Ou seja, esta função já se encontra definida como:

$$\boxed{g\_mroot = firsts}$$

```
*Main> trs
[("compra","20211102",-50),("venda","20211103",100),("despesa","20212103",-20),("venda","20211205",250),("venda","20211205",120)]
*Main> trs2
[("compra","20211102",-50),("venda","20211103",100),("despesa","20212103",-20),("venda","20211205",250),("venda","20211205",500)]
*Main> mroot trs
7233215638271874210
*Main> mroot trs2
7233215638271873838
*Main>
```

4.

Quando calculamos a *Merkle root* do bloco de transações **trs**, obtemos a respetiva chave do bloco, mas se alterarmos alguma transação desse mesmo bloco e voltarmos a calcular a *mroot trs*, reparamos que o valor agora é diferente do anterior. Nesse sentido, isto se deve porque esta chave final é resultado de concatenações de pares de transações e se uma é fraudulentamente alterada o valor na raiz também muda. Assim, este mecanismo permite a integridade, validade e segurança dos dados numa *Blockchain*, pois cada bloco possui a chave do bloco anterior e do seguinte e se um for alterado os seus vizinhos não vai fazer correspondência e vai ser indentificada fraude no final.

Valorização

pairsList

Este problema é muito semelhante ao Problema 4 relativo à função *pairL*, pelo que os diagramas dos anamorfismos são praticamente idênticos:

$$\begin{tikzcd}
(A,A)^* \arrow[rrr, bend left, "outList"] \arrow[rrr, phantom, "\cong"] & & & 1+((A,A)\times(A,A)^*) \arrow[lll, bend left, "inList"] \\
A^* \arrow[u, "pairsList"] \arrow[r, "getEvenBlock"] \arrow[rrr, bend right, "g_pairsList"'] & A^* \arrow[r, "outList"] & 1+A+A^* \arrow[r, "id+?"] & 1+((A,A)\times A^*) \arrow[u, "id+id\times pairsList"']
\end{tikzcd}$$

A lista que recebemos como parâmetro começamos por aplicar a função *getEvenBlock* para o caso de o tamanho da lista for impar o último elemento é duplicado. De seguida, decompomos a lista pela respetiva cabeça e cauda (h,t) e aplicamos $dux \times id$ que tranforma a cabeça (h) no par (h,h) e preserva a cauda. Por fim, com a função auxiliar *pair_aux* modificamos o segundo elemento do par(h,h) por (h, *head* t) e à cauda retiramos a cabeça pois esta já foi inserida num par.

Assim, o gene fica definido da seguinte forma:

$$\boxed{g\_pairsList = (id + pair\_aux \cdot (dup \times id)) \cdot outList \cdot getEvenBlock}$$

```
*Main> pairsList trs
[(("compra","20211102",-50),("venda","20211103",100)),(("despesa","20212103",-20),("venda","20211205",250)),(("venda","20211205",120),("venda","20211205",120))]
*Main>
```

Soluções

Listas não vazias:

$$
\begin{array}{l}
outNEList\ [a] = i_1\ a \\
outNEList\ (h : t) = i_2\ (h, t) \\
baseNEList\ g\ f = id + g \times f \\
recNEList\ f = baseNEList\ id\ f \\
cataNEList\ g = g \cdot recNEList\ (cataNEList\ g) \cdot outNEList \\
anaNEList\ g\ = inNEList \cdot recNEList\ (anaNEList\ g) \cdot g \\
hyloNEList\ h\ g = cataNEList\ h \cdot anaNEList\ g
\end{array}
$$

Gene do anamorfismo:

$$g\_list2LTree = lsplit$$

Gene do catamorfismo:

$$
\begin{array}{l}
g\_lTree2MTree :: Hashable\ c \Rightarrow c + (FTree\ \mathbb{Z}\ (\mathbb{Z}, c), FTree\ \mathbb{Z}\ (\mathbb{Z}, c)) \rightarrow FTree\ \mathbb{Z}\ (\mathbb{Z}, c) \\
g\_lTree2MTree = [g1, g2]\ \mathbf{where} \\
\quad g1\ = Unit \cdot \langle Main.hash, id \rangle \\
\quad g2\ = \widehat{Comp} \cdot \langle auxConcHash, id \rangle
\end{array}
$$

$$
\begin{array}{l}
auxConcHash :: (FTree\ \mathbb{Z}\ (\mathbb{Z}, b1), FTree\ \mathbb{Z}\ (\mathbb{Z}, b2)) \rightarrow \mathbb{Z} \\
auxConcHash\ (Unit\ (a, \_), Unit\ (b, \_)) = concHash\ (a, b) \\
auxConcHash\ (Unit\ (a, \_), Comp\ b\ (\_, \_)) = concHash\ (concHash\ (a, a), b) \\
auxConcHash\ (Comp\ a\ (\_, \_), Unit\ (b, \_)) = concHash\ (a, concHash\ (b, b)) \\
auxConcHash\ (Comp\ a\ (\_, \_), Comp\ b\ (\_, \_)) = concHash\ (a, b)
\end{array}
$$

Gene de $mroot$ ("get Merkle root"):

$$g\_mroot = firsts$$

Valorização:

$$
\begin{array}{l}
pairsList :: [a] \rightarrow [(a, a)] \\
pairsList = [\![g\_pairsList]\!] \\
g\_pairsList = (id + pair\_aux \cdot (dup \times id)) \cdot outList \cdot getEvenBlock\ \mathbf{where} \\
\quad pair\_aux\ ((h_1, h_2), t) = ((h_1, head\ t), tail\ t) \\
classicMerkleTree :: Hashable\ a \Rightarrow [a] \rightarrow FTree\ \mathbb{Z}\ \mathbb{Z} \\
classicMerkleTree = hyloNEList\ conquer\ divide \cdot \mathsf{map}\ \ Main.hash \\
divide = ((singl \cdot Unit) + createMtree) \cdot outNEList\ \mathbf{where} \\
\quad createMtree = singl \cdot Unit \times id \\
conquer = [head, joinMerkleTree]\ \mathbf{where} \\
\quad joinMerkleTree\ (l, m) = mergeMerkleTree\ m\ (evenMerkleTreeList\ l) \\
\quad mergeMerkleTree = (\![[h_1, h_2]]\!) \\
\quad h_1\ c\ l = \bot \\
\quad h_2\ (c, (f, g))\ l = \bot \\
\quad evenMerkleTreeList = getEvenBlock
\end{array}
$$

Problema 2

Por aplicações da lei da recursividade mútua às funções *wc_c*, *lookAhead_sep* temos:

$$
\begin{aligned}
& \langle wc\_c, lookahead\_sep \rangle = (\!|\langle h, k \rangle|\!) \\
\equiv \quad & \quad \{ \text{ Definições de h e k } \} \\
& \langle wc\_c, lookahead\_sep \rangle = (\!|\langle [h_1, h_2], [k_1, k_2] \rangle|\!) \\
\equiv \quad & \quad \{ \text{ Lei da Troca (28) } \} \\
& \langle wc\_c, lookahead\_sep \rangle = (\!|[\langle h_1, k_1 \rangle, \langle h_2, k_2 \rangle]|\!)
\end{aligned}
$$

Logo,

$$\boxed{worker = (\!|[\langle h_1, k_1 \rangle, \langle h_2, k_2 \rangle]|\!)}$$

que por sua vez:

$$\begin{cases} g1 = \langle h_1, k_1 \rangle \\ g2 = \langle h_2, k_2 \rangle \end{cases}$$

Falta descobrir os genes h e k:

$$
\begin{tikzcd}
Char^* \arrow[rr, bend left, "out"] \arrow[d, "lookahead\_sep"'] & & 1 + Char \times Char^* \arrow[ll, bend left, "\mathsf{in}"] \arrow[d, "id+id\times\langle wc\_c, lookahead\_sep\rangle"] \\
Bool & & 1 + Char \times (Int \times Bool) \arrow[ll, "{[k_1, k_2]}"]
\end{tikzcd}
$$

Analisando o diagrama, facilmente se deduz o k_1, pois quando a função *lookahead_sep* recebe uma lista vazia, segundo a implementação em Haskell verificamos que:

$k_1 = \underline{True}$

Para o k_2 apenas nos interessa a cabeça do par *(h,(int,bool))* resultando da aplicação da função *out* das listas. Assim, para retirar a cabeça usamos a função π_1 e ao resultado aplicamos a função dada *sp* para verificar se o carácter é de separação ou não. Logo:

$k_2 = sp \cdot \pi_1$

$$
\begin{tikzcd}
Char^* \arrow[rr, bend left, "out"] \arrow[d, "wc\_c"'] & & 1 + Char \times Char^* \arrow[ll, bend left, "\mathsf{in}"] \arrow[d, "id+id\times\langle wc\_c, lookahead\_sep\rangle"] \\
Int & & 1 + Char \times (Int \times Bool) \arrow[ll, "{[h_1, h_2]}"]
\end{tikzcd}
$$

Agora olhando para o diagrama da função *wc_c* e para o código dado em Haskell, verificamos que o resultado, para a lista vazia, é de 0 palavras. Então:

$h_1 = \underline{0}$

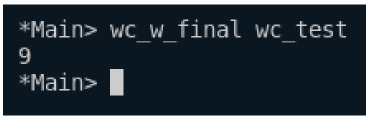
Para a função h_2 reparamos que esta recebe um *(char,(int, bool))* resultante da recursividade das funções $\langle wc\_c, lookahead\_c \rangle$ após o *out* de listas. Portanto, o primeiro elemento corresponde à cabeça da lista e o segundo elemeto, que é outro par, é o resultado da recursividade aplicado à cauda. Deste modo, analizando o código mais uma vez observamos que há uma condição, isto é, precisamos de verificar se a cabeça não é um elemento separador e se o próximo elemento da cauda é. Se sim, então incrementa-se o resultado que se tem no segundo par, se não, devolve-se o que temos.

Concluindo, assim, a seguinte definição de h_2:

$$\boxed{h_2 = aux}$$

Onde *aux* é:

$$\boxed{aux\ (c, (i, b)) = \textbf{if}\ \neg\ (sp\ c) \wedge b\ \textbf{then}\ i + 1\ \textbf{else}\ i}$$



```
*Main> wc_w_final wc_test
9
*Main>
```

Solução

$$
\begin{aligned}
&wc\_w\_final :: [\,Char\,] \to Int\\
&wc\_w\_final = wrapper \cdot worker\\
&worker = (\![g1, g2]\!)\\
&wrapper = \pi_1
\end{aligned}
$$

Gene de *worker*:

$$
\begin{aligned}
&g1 = \langle h_1, k_1 \rangle\\
&g2 = \langle h_2, k_2 \rangle
\end{aligned}
$$

Genes $h = [h_1, h_2]$ e $k = [k_1, k_2]$ identificados no cálculo:

$$
\begin{aligned}
&h_1 = \underline{0}\\
&h_2 = aux\ \textbf{where}\\
&\quad aux\ (c, (i, b)) = \textbf{if}\ \neg\ (sp\ c) \wedge b\ \textbf{then}\ i + 1\ \textbf{else}\ i\\
&k_1 = \underline{True}\\
&k_2 = sp \cdot \pi_1
\end{aligned}
$$

Problema 3

inX:

$$
\begin{tikzcd}
X\ \ u\ \ i \arrow[rr, bend left, "outX"] \arrow[rr, phantom, "\cong"] & & u + (i, (X\ \ u\ \ i, X\ \ u\ \ i)) \arrow[ll, bend left, "inx"]
\end{tikzcd}
$$

$$
\begin{aligned}
&inX = [XLeaf, Node]\\
\equiv&\qquad \{\ \text{Universal-+ (17)}\ \}\\
&\left\{\begin{array}{l} inX \cdot i_1 = XLeaf\\ inX \cdot i_2 = Node \end{array}\right.\\
\equiv&\qquad \{\ \text{Igualdade extensional (71), Def-comp (72)}\ \}\\
&\left\{\begin{array}{l} inX\ (i_1\ u) = XLeaf\ \ u\\ inX\ (i_2\ (i_1\ (a, b))) = Node\ \ i\ \ a\ \ b \end{array}\right.
\end{aligned}
$$

outX:

$$outX \cdot inX = id$$

$\equiv$ { definição de inX }

$$outX \cdot [XLeaf, Node] = id$$

$\equiv$ { Fusão-+ (20) }

$$[outX \cdot XLeaf, outX \cdot Node] = id$$

$\equiv$ { Universal-+ (17) }

$$\begin{cases} id \cdot i_1 = outX \cdot XLeaf \\ id \cdot i_2 = outX \cdot Node \end{cases}$$

$\equiv$ { Natural-ID (1), Igualdade extensional (71), Def-comp(72) }

$$\begin{cases} outX\ (XLeaf\ \ u) = i_1\ u \\ outX\ (Node\ \ i\ \ a\ \ b) = i_2\ (i, (a, b)) \end{cases}$$

baseX:

$$\begin{tikzcd}
X \arrow[rr, bend left, "outX"] \arrow[rr, phantom, "\cong"] \arrow[d, "g"'] & & A + B \times (X \times X) \arrow[ll, bend left, "inX"] \arrow[d, "f+g\times(g\times g)"] \\
C & & A\ +(B\ \times(C \times C)) \arrow[ll, "{[h_1,h_2]}"]
\end{tikzcd}$$

recX:

$$recX\ f = id + (id\ x\ (f\ \ x\ f))$$

$\equiv$ { Aplicando a definição de baseX }

$$recX\ f = baseX\ \ id\ id\ f$$

cataX:

$\equiv$ { Cancelamento-cata (46) }

$$(\!|g|\!)\ .\ in = g\ .\ F(\!|g|\!)$$

$\equiv$ { in . out = id }

$$(\!|g|\!) = g\ .\ F(\!|g|\!)\ .\ out$$

$$\equiv \qquad \{ \text{ Aplicando as definições em Haskell já definidas } \}$$

$$cataX\ g = g \cdot recX\ (cataX\ g) \cdot outX$$

anaX:

$$\equiv \qquad \{ \text{ Cancelamento-ana (55) } \}$$

$$out\ .\ \llbracket g \rrbracket = g\ .\ F\llbracket g \rrbracket\ .\ g$$

$$\equiv \qquad \{ \text{ in . out = id } \}$$

$$\llbracket g \rrbracket = in\ .\ F\llbracket g \rrbracket\ .\ g$$

$$\equiv \qquad \{ \text{ Aplicando as definições em Haskell já definidas } \}$$

$$anaX\ g = inX \cdot recX\ (anaX\ g) \cdot g$$

hyloX:

$$\equiv \qquad \{ \text{ Definição de hilomorfismo } \}$$

$$hyloX\ g\ h = (\!| g |\!)\ .\ \llbracket h \rrbracket$$

$$\equiv \qquad \{ \text{ Cancelamento-cata (46), Cancelamento-ana(55) } \}$$

$$hyloX\ g\ h = g \cdot F(\!| g |\!) \cdot inX \cdot outX \cdot F\llbracket h \rrbracket \cdot h$$

$$\equiv \qquad \{ \text{ in . out = id, Aplicando as definições em Haskell já definidas } \}$$

$$hyloX\ g\ h = g \cdot recX\ (cataX\ g) \cdot recX\ (anaX\ h) \cdot h$$

Segue-se o resto da resolução deste problema:

x2html

$$\begin{tikzcd}
X\ u\ i \arrow[rr, bend left, "outX"] \arrow[rr, phantom, "\cong"] \arrow[d, "new_x2html"'] & & u + (i, (x1, x2)) \arrow[ll, bend left, "inX"] \arrow[d, "id + id \times (new_x2html \times new_x2html)"] \\
Char^* & & u + (i, (s1, s2)) \arrow[ll, "g_x2html"]
\end{tikzcd}$$

Analizando o diagrama do catamorfismo e o código em Haskell da função *x2html* percebemos que o gene *g_x2html* recebe uma folha *XLeaf* que tanto pode conter uma imagem ou texto, por isso definimos o gene da seguinte forma:

$$\boxed{\begin{array}{c} g\_x2html = [g1, g2] \textbf{ where} \\ g1 = auxX1 \\ g2 = auxX2 \end{array}}$$

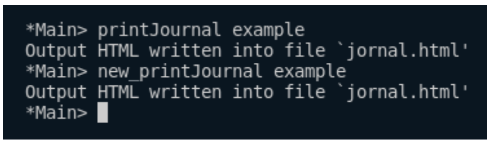
Nesse sentido, se a função auxX1 receber uma *Image* é usada a função dada *img* para devolver o formato *HTML* de uma imagem. Caso a função auxiliar em questão receber um *Text* é apenas devolvido a string que contém o texto.

Assim,

$$\boxed{\begin{array}{c} auxX1\ (Image\ a)\ (w, h) = img\ w\ h\ a \\ auxX1\ (Text\ b)\ \_ = b \end{array}}$$

Quanto à função auxX2 esta recebe um par em que o primeiro elemento é a informação que esta no nodo da estrutura X e que pode ser 4 tipos diferentes de *Mode* e o segundo elemento é um outro par que contém a recursividade aplicada à árvore esquerda e direita, respetivamente. Assim, a definição desta função é bastante parecida à versão original, apenas se reescrevendo as chamadas recursivas:

$$\boxed{\begin{array}{c} auxX2\ (Vt\ i, (s1, s2))\ (w, h) = \\ htab\ w\ h\ (tr\ (td\ w\ (h*i)\ (s1\ (w, h*i))) +\!\!+ tr\ (td\ w\ (h*(1-i))\ (s2\ (w, h*(1-i))))) \\ auxX2\ (Hl\ i, (s1, s2))\ (w, h) = \\ htab\ w\ h\ (tr\ (td\ (w*i)\ h\ (s1\ (w*i, h)) +\!\!+ td\ (w*(1-i))\ h\ (s2\ (w*(1-i), h)))) \\ auxX2\ (Vb\ i, (s1, s2))\ m = auxX2\ (Vt\ (1-i), (s1, s2))\ m \\ auxX2\ (Hr\ i, (s1, s2))\ m = auxX2\ (Hl\ (1-i), (s1, s2))\ m \end{array}}$$



Solução

$$
\begin{array}{l}
inX :: u + (i, (X\ u\ i, X\ u\ i)) \to X\ u\ i \\
inX\ (i_1\ u) = XLeaf\ u \\
inX\ (i_2\ (i, (l, r))) = Node\ i\ l\ r \\
outX :: X\ a1\ a2 \to a1 + (a2, (X\ a1\ a2, X\ a1\ a2)) \\
outX\ (XLeaf\ u) = i_1\ u \\
outX\ (Node\ i\ l\ r) = i_2\ (i, (l, r)) \\
baseX\ f\ h\ g = f + (h \times (g \times g)) \\
recX\ f = baseX\ id\ id\ f \\
cataX\ g = g \cdot recX\ (cataX\ g) \cdot outX \\
anaX\ g = inX \cdot recX\ (anaX\ g) \cdot g \\
hyloX\ c\ d = c \cdot recX\ (cataX\ c) \cdot recX\ (anaX\ d) \cdot d \\
new\_printJournal = write \cdot new\_sheet2html \\
new\_sheet2html\ (Rect\ (Frame\ w\ h)\ y) = htmlwrap\ (new\_x2html\ y\ (w, h)) \\
new\_x2html :: X\ (Unit\ String\ String)\ (Mode\ Double) \to (Double, Double) \to String
\end{array}
$$

```
new_x2html = cataX g_x2html
g_x2html = [g1, g2] where
  g1 = auxX1
  g2 = auxX2
auxX1 (Image a) (w, h) = img w h a
auxX1 (Text b) _ = b
auxX2 (Vt i, (s1, s2)) (w, h) = htab w h (
  tr (td w (h * i) (s1 (w, h * i))) ++
  tr (td w (h * (1 - i)) (s2 (w, h * (1 - i))))
  )
auxX2 (Hl i, (s1, s2)) (w, h) = htab w h (
  tr (td (w * i) h (s1 (w * i, h)) ++
    td (w * (1 - i)) h (s2 (w * (1 - i), h)))
  )
auxX2 (Vb i, (s1, s2)) m = auxX2 (Vt (1 - i), (s1, s2)) m
auxX2 (Hr i, (s1, s2)) m = auxX2 (Hl (1 - i), (s1, s2)) m
```

Problema 4

pairL:

$$
\begin{tikzcd}
(A, A)^* \arrow[rr, bend left, "outX"] \arrow[rr, phantom, "\cong"] & & 1 + (A \times A) \times (A \times A)^* \arrow[ll, bend left, "inX"] \\
A^* \arrow[u, "pairL"] \arrow[r, "outList"] \arrow[rr, bend right, "g"'] & 1 + A \times A^* \arrow[r, "id+?"] & 1 + (A \times A) \times A^* \arrow[u, "id+id\times pairL"']
\end{tikzcd}
$$

A ideia é decompor a lista original para poder depois construir numa nova lista de pares. Embora estas funções não recebam listas vazias é necessário cobrir o caso em que a lista é vazia. Assim, o resultado é a própria lista vazia, logo basta preservar através da função *id*.

Para o caso em que é dada uma lista **l** de tamanho maior que 1, esta é decomposta no par **(h,t)** pela função outlist.

$$
(A, A)^* \xrightarrow[dup \times id]{} (A, A) \times A^* \xrightarrow[completePair]{} (A, A) \times A^*
$$

A seguir é preciso transformar a cabeça **h** num par **(h,h)** através da função dup e, de seguida, com a função auxiliar *completePair* completar o par de modo a que fique **(h,x)** em que **x** é a cabela da lista **t** isto se **t** não for lista vazia senão mantém-se o par **(h,h)**.

Deste modo o gene do anamorfismo g é definido como:

$$
\boxed{g = (id + completePair \cdot (dup \times id)) \cdot outList}
$$

No entanto, ao correr a função *pairL [1,2,3,4]* reparamos que o output é **[(1,2),(2,3),(3,4),(4,4)]** em que o último elemento está a mais. Porém se fizermos *init* dessa lista descartando o último elemento a função vai falhar para o caso *pairL [1]* retornando *[]* em vez de **[(1,1)]**.

Então modificamos a definição pair para o seguinte modo:

$$
\boxed{\begin{array}{l}
pairL = (p \rightarrow f, h) \\
\textbf{where } p\ l = length\ l > 1 \\
f = init \cdot [\![(g)]\!] \\
h = [\![(g)]\!]
\end{array}}
$$

```
*Main> pairL [1, 2]
[(1,2)]
*Main> pairL [1, 2, 3]
[(1,2),(2,3)]
*Main> pairL [1, 2, 3, 4]
[(1,2),(2,3),(3,4)]
*Main> quickCheck prop_reconst
+++ OK, passed 100 tests.
*Main>
```

markMap

Diagrama geral da função *markMap*:

$$\begin{tikzcd}
Pos^* \arrow[r, "pairL"'] \arrow[dr, "markMap"'] & (Pos, Pos)^* \arrow[d, "f=(\!|g|\!)"'] \\
& Map
\end{tikzcd}$$

Como podemos observar no diagrama, a função começa por transformar a lista inicial de posições numa lista de pares. Assim, essa lista será o input do catamorfismo de listas que a consome e transforma num mapa marcado com o respetivo caminho. Necessitamos definir a função *f2* do gene **g=[id,f2]** e para isso foi feito um diagrama para extrair o tipo dos seus argumentos:

$$\begin{tikzcd}
(Pos, Pos)^* \arrow[rr, bend left, "outX"] \arrow[rr, phantom, "\cong"] \arrow[d, "f"'] & & 1 + (Pos, Pos) \times (Pos, Pos)^* \arrow[ll, bend left, "inX"] \arrow[d, "id+id\times f"] \\
Map & & 1 + (Pos, Pos) \times Map \arrow[ll, "g=[id,f_2]"]
\end{tikzcd}$$

Com o diagrama concluiu-se que a função *f2* recebia um par **((Pos,Pos),h)**, em que o primeiro elemento é um par de posições que indicam o movimento no mapa e por isso será necessário usar a função toCell para o processar. Por outro lado, o segundo elemento refere-se a uma função que devolve o mapa marcado com as posições na cauda da lista original e que será aplicado no mapa original dado como argumento.

$$([Pos, Pos], h)^* \xrightarrow[\langle\pi_1, toCell\rangle\times id]{} (Pos, Cell), h \xrightarrow[substCell]{} Map$$

Como referido anteriormente, será necessário gerar o tipo *Cell* com as coordenadas do primeiro par guardando no lado direito o resultado e preservando o lado esquerdo pois é onde vai ser inserido no mapa. Finalmente, foi criada a função auxiliar *substCell* que recebe o par *((Pos, Cell),h)* mais o mapa dado e com o auxílio da função *subst* faz as substituições necessárias no mapa original. Esta função também leva em conta se uma *Call* está bloqueada ou não garantindo percorrer em apenas caminhos livres. Logo:

$$\boxed{f_2 = substCell \cdot (\langle \pi_1, \widehat{toCell} \rangle \times id)}$$

```
*Main> markMap [(0, 0), (0, 1), (0, 2), (1, 2)] map1
[[Up,Blocked,Free],[Up,Blocked,Free],[Lft,Free,Free]]
*Main> putStr $ showM $ markMap [(0, 0), (0, 1), (0, 2), (1, 2)] map1
 >  _  _
 ^  X  _
 ^  X  
*Main> 
```

Soluções

$$
\begin{array}{l}
pairL :: [a] \to [(a,a)] \\
pairL = cond\ p\ f\ h\ \textbf{where} \\
\quad p\ l = length\ l > 1 \\
\quad f\ = init \cdot (\!|g\_pairL|\!) \\
\quad h\ = (\!|g\_pairL|\!) \\
\quad g\_pairL = (id + completePair \cdot (dup \times id)) \cdot outList\ \textbf{where} \\
\qquad completePair\ ((x1,x2),l) = \textbf{if}\ length\ l > 0\ \textbf{then}\ ((x1, head\ l), l)\ \textbf{else}\ ((x1,x2),l)
\end{array}
$$

$$
\begin{array}{l}
markMap :: [Pos] \to Map \to Map \\
markMap\ l = (\![\underline{id}, f_2]\!)\ (pairL\ l)\ \textbf{where} \\
\quad f_2 = substCell \cdot (\langle \pi_1, \widehat{toCell} \rangle \times id) \\
\quad substCell\ (((r,c), move), h)\ m \\
\qquad \mid (m\ !!\ c)\ !!\ r \equiv Blocked = h\ m \\
\qquad \mid otherwise = subst\ (subst\ move\ r\ (h\ m\ !!\ c))\ c\ (h\ m)
\end{array}
$$

$$
\begin{array}{l}
scout :: Map \to Pos \to Pos \to Int \to [[Pos]] \\
scout\ m\ s\ t = (\![f_1, \ggg\!\!= f_2\ m\ s]\!)\ \textbf{where} \\
\quad f_1 = \bot \\
\quad f_2 = \bot
\end{array}
$$

Valorização (opcional) Completar as seguintes funções de teste no QuickCheck para verificação de propriedades das funções pedidas, a saber:

Propriedade [QuickCheck] 1 *A lista correspondente ao lado esquerdo dos pares em $(pairL\ l)$ é a lista original l a menos do último elemento. Analogamente, a lista correspondente ao lado direito dos pares em $(pairL\ l)$ é a lista original l a menos do primeiro elemento:*

$$
\begin{array}{l}
prop\_reconst\ l \\
\quad \mid length\ l \leqslant 1 = \mathsf{map}\ \ \pi_1\ (pairL\ l) \equiv \mathsf{map}\ \ \pi_2\ (pairL\ l) \\
\quad \mid otherwise = init\ l \equiv \mathsf{map}\ \ \pi_1\ (pairL\ l) \wedge tail\ l \equiv \mathsf{map}\ \ \pi_2\ (pairL\ l)
\end{array}
$$

Propriedade [QuickCheck] 2 *Assuma que uma linha (de um mapa) é prefixa de uma outra linha. Então a representação da primeira linha também prefixa a representação da segunda linha:*

$$prop\_prefix2\ l\ l' = \bot$$

Propriedade [QuickCheck] 3 *Para qualquer linha (de um mapa), a sua representação deve conter um número de símbolos correspondentes a um tipo célula igual ao número de vezes que esse tipo de célula aparece na linha em questão.*

$$prop\_nmbrs\ l\ c = \bot$$

$$
\begin{array}{l}
count :: (Eq\ a) \Rightarrow a \to [a] \to Int \\
count = \bot
\end{array}
$$

Propriedade [QuickCheck] 4 *Para qualquer lista l a função* $markMap\ l$ *é idempotente.*

$$inBounds\ m\ (x, y) = \bot$$
$$prop\_idemp2\ l\ m = \bot$$

Propriedade [QuickCheck] 5 *Todas as posições presentes na lista dada como argumento irão fazer com que as células correspondentes no mapa deixem de ser* $Free$*.*

$$prop\_extr2\ l\ m = \mathbf{if}\ (length\ l > 1 \wedge isValidMap\ m)\ \mathbf{then}\ checkCell\ (init\ l)\ (markMap\ l\ m)\ \mathbf{else}\ False$$
$$checkCell\ [\,]\ m = True$$
$$checkCell\ ((x, y) : t)\ m = ((m\ !!\ y)\ !!\ x) \not\equiv Free \wedge checkCell\ t\ m$$

Propriedade [QuickCheck] 6 *Quanto maior for o tamanho máximo dos caminhos mais caminhos que alcançam a posição alvo iremos encontrar:*

$$prop\_reach\ m\ t\ n\ n' = \bot$$

Índice

Referências

[1] B.W. Kernighan and D.M. Richtie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1978.

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

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

[4] SelfKey. What is a Merkle tree and how does it affect blockchain technology?, 2015. Blog: https://selfkey.org/what-is-a-merkle-tree-and-how-does-it-affect-blockchain-techno
Last read: 7 de Fevereiro de 2022.