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

Departamento de Informática
Universidade do Minho

Fevereiro de 2022

Grupo nr.	14
a93179	Rui Monteiro
a93201	Rodrigo Rodrigues
a93324	Daniel Azevedo
a93283	Marco Costa

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:

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

- *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] .

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

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

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

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

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

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

   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 $hashes$:

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

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

$$\begin{array}{l} pairsList :: [a] \rightarrow [(a, a)] \\ pairsList = [\!(g\_pairsList)\!] \end{array}$$

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

$$\begin{array}{l} mergeMerkleTree :: FTree\ a\ p \rightarrow [FTree\ a\ c] \rightarrow FTree\ a\ c \\ mergeMerkleTree = (\!|\ g\_mergeMerkleTree\ |\!) \end{array}$$

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:

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

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:

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

O tipo $Sheet$ (="página de jornal")

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

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

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

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

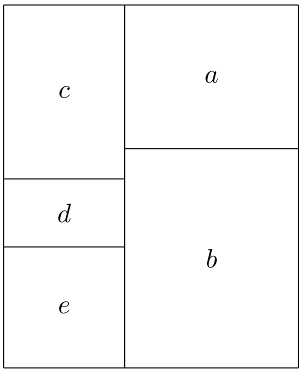
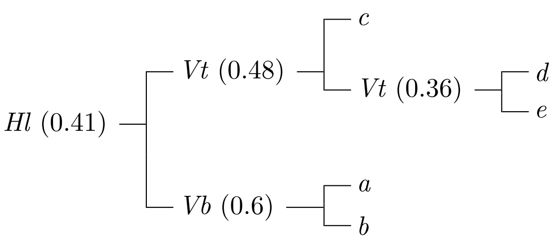


Figura 2: Layout de página de jornal.

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. $pairL :: [a] \rightarrow [(a, a)]$ 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:

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

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 $markMap :: [Pos] \rightarrow Map \rightarrow Map$, 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 $[(Pos, Pos)] \rightarrow Map \rightarrow Map$ 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]

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

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

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

[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:

$$
\begin{array}{l}
\mathit{trs} = [(\texttt{"compra"}, \texttt{"20211102"}, -50), \\
\quad (\texttt{"venda"}, \quad \texttt{"20211103"}, 100), \\
\quad (\texttt{"despesa"}, \texttt{"20212103"}, -20), \\
\quad (\texttt{"venda"}, \quad \texttt{"20211205"}, 250), \\
\quad (\texttt{"venda"}, \quad \texttt{"20211205"}, 120)]
\end{array}
$$

$$
\begin{array}{l}
\mathit{getEvenBlock} :: [a] \rightarrow [a] \\
\mathit{getEvenBlock}\ l = \mathbf{if}\ (\mathit{even}\ (\mathit{length}\ l))\ \mathbf{then}\ l\ \mathbf{else}\ l +\!\!+ [\mathit{last}\ l] \\
\mathit{firsts} = [\pi_1, \pi_1]
\end{array}
$$

Problema 2

$$
\begin{array}{l}
\mathit{wc\_test} = \texttt{"Here is a sentence, for testing.\textbackslash nA short one."} \\
\mathit{sp}\ c = (c \equiv \texttt{' '} \lor c \equiv \texttt{'\textbackslash n'} \lor c \equiv \texttt{'\textbackslash t'})
\end{array}
$$

Problema 3

Tipos:

$$
\begin{array}{l}
\mathbf{data}\ X\ u\ i = \mathit{XLeaf}\ u \mid \mathit{Node}\ i\ (X\ u\ i)\ (X\ u\ i)\ \mathbf{deriving}\ \mathit{Show} \\
\mathbf{data}\ \mathit{Frame}\ i = \mathit{Frame}\ i\ i\ \mathbf{deriving}\ \mathit{Show}
\end{array}
$$

Funções da API[8]

$$
\begin{array}{l}
\mathit{printJournal} :: \mathit{Sheet}\ \mathit{String}\ \mathit{String}\ \mathit{Double} \rightarrow \mathsf{IO}\ () \\
\mathit{printJournal} = \mathit{write} \cdot \mathit{sheet2html} \\
\mathit{write} :: \mathit{String} \rightarrow \mathsf{IO}\ () \\
\mathit{write}\ s = \mathbf{do}\ \mathit{writeFile}\ \texttt{"jornal.html"}\ s \\
\quad \mathit{putStrLn}\ \texttt{"Output HTML written into file `jornal.html'"}
\end{array}
$$

Geração de HTML:

$$
\begin{array}{l}
\mathit{sheet2html}\ (\mathit{Rect}\ (\mathit{Frame}\ w\ h)\ y) = \mathit{htmlwrap}\ (\mathit{x2html}\ y\ (w, h)) \\
\mathit{x2html} :: X\ (\mathit{Unit}\ \mathit{String}\ \mathit{String})\ (\mathit{Mode}\ \mathit{Double}) \rightarrow (\mathit{Double}, \mathit{Double}) \rightarrow \mathit{String} \\
\mathit{x2html}\ (\mathit{XLeaf}\ (\mathit{Image}\ i))\ (w, h) = \mathit{img}\ w\ h\ i \\
\mathit{x2html}\ (\mathit{XLeaf}\ (\mathit{Text}\ \mathit{txt}))\ \_ = \mathit{txt} \\
\mathit{x2html}\ (\mathit{Node}\ (\mathit{Vt}\ i)\ \mathit{x1}\ \mathit{x2})\ (w, h) = \mathit{htab}\ w\ h\ (\\
\quad \mathit{tr}\ (\mathit{td}\ w\ (h * i)\ (\mathit{x2html}\ \mathit{x1}\ (w, h * i))) +\!\!+ \\
\quad \mathit{tr}\ (\mathit{td}\ w\ (h * (1 - i))\ (\mathit{x2html}\ \mathit{x2}\ (w, h * (1 - i)))) \\
\quad) \\
\mathit{x2html}\ (\mathit{Node}\ (\mathit{Hl}\ i)\ \mathit{x1}\ \mathit{x2})\ (w, h) = \mathit{htab}\ w\ h\ (\\
\quad \mathit{tr}\ (\mathit{td}\ (w * i)\ h\ (\mathit{x2html}\ \mathit{x1}\ (w * i, h)) +\!\!+ \\
\qquad \mathit{td}\ (w * (1 - i))\ h\ (\mathit{x2html}\ \mathit{x2}\ (w * (1 - i), h))) \\
\quad) \\
\mathit{x2html}\ (\mathit{Node}\ (\mathit{Vb}\ i)\ \mathit{x1}\ \mathit{x2})\ m = \mathit{x2html}\ (\mathit{Node}\ (\mathit{Vt}\ (1 - i))\ \mathit{x1}\ \mathit{x2})\ m \\
\mathit{x2html}\ (\mathit{Node}\ (\mathit{Hr}\ i)\ \mathit{x1}\ \mathit{x2})\ m = \mathit{x2html}\ (\mathit{Node}\ (\mathit{Hl}\ (1 - i))\ \mathit{x1}\ \mathit{x2})\ m
\end{array}
$$

Funções auxiliares:

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

```haskell
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:

```haskell
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:

```haskell
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:

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

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):

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

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

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

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

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

QuickCheck

Lógicas:

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

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

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

Para podermos determinar o gene de *lTree2MTree* temos primeiro que observar o seu tipo:

$$
\begin{tikzcd}
\mathit{LTree}\ A & & A^* \arrow[ll, "lTree2MTree"']
\end{tikzcd}
$$

Com isto, a solução formulada começa por converter a lista não vazia para o tipo $A + A \times A$. Assim, caso a lista tenha mais do que um elemento, dividimos a lista ao meio e passamos cada lista resultante a um dos ramos da árvore.

Desta solução obtemos o seguinte diagrama:

$$
\begin{tikzcd}
\mathit{LTree}\ A & & A + (\mathit{LTree}\ A, \mathit{LTree}\ A) \arrow[ll, "\mathsf{in}"'] \\
A^* \arrow[u, "lTree2MTree"] \arrow[r, "outNEList"] & A + (A \times A^*) \arrow[r, "{[g1, g2]}"] & A + (A^* \times A^*) \arrow[u, "id+lTree2MTree\times lT"']
\end{tikzcd}
$$

Devido ao facto de que a biblioteca das listas com sinal não foram disponibilizadas, tivemos antes que a definir:

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

Gene do anamorfismo:

Com tudo o resto definido só nos falta definir $g\_list2LTree = [g1, g2] \cdot outNEList$.

$$
\begin{array}{l}
\mathit{g\_list2LTree} = [g1, g2] \cdot \mathit{outNEList}\ \textbf{where} \\
\quad g1\ = i_1 \\
\quad g2\ (a, as) = (i_2 \cdot \mathit{splitAt}\ n)\ l\ \textbf{where} \\
\qquad l = a : as \\
\qquad n = \mathit{length}\ l\ \text{`}\mathit{div}\text{`}\ 2
\end{array}
$$

Gene do catamorfismo:

Para termos uma ideia mais clara do problema em questão vamos começar por observar o tipo de lTree2MTree:

$$\mathit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, A) \xleftarrow[\mathit{lTree2MTree}]{} A^*$$

Para descobrirmos o gene deste catamorfismo, temos que, como é habitual, dividir a *FTree* nos seus dois casos. Caso o nodo seja uma folha agrupamos o elemento com a sua hash. Caso contrário, etiquetamos o nodo com a soma das hashes das suas subárvores.

Assim, obtemos o seguinte diagrama:

$$\begin{tikzcd}
\mathit{LTree}\ A \arrow[r, "\mathsf{out}"] \arrow[d, "\mathit{lTree2MTree}"'] & A + (\mathit{LTree}\ A, \mathit{LTree}\ A) \arrow[d, "id + \mathit{lTree2MTree} \times \mathit{lTree2MTree}"] \\
\mathit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, B) & A + (\mathit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, B), \mathit{FTree}\ \mathbb{Z}\ (\mathbb{Z}, B)) \arrow[l, "{[g1, g2]}"]
\end{tikzcd}$$

Agora só nos falta definir $g = [g1, g2]$.

$$g1\ \ a = \mathit{Unit}\ \langle \mathit{Main.hash}\ a, a \rangle$$

$$\equiv \qquad \{\ \text{Nat-id (1); Fusão-}\times\text{ (9)}\ \}$$

$$g1\ \ a = \mathit{Unit}\ \langle \mathit{Main.hash}, id \rangle\ a$$

$$\equiv \qquad \{\ \text{Igualdade Extensional (71); Def-comp (72)}\ \}$$

$$g1\ \ a = \mathit{Unit} \cdot \langle \mathit{Main.hash}, id \rangle$$

$\square$

Para $g2$ é necessário criar uma função $\mathit{getHash}$ que obtém a hash das subárvores.

$$\begin{cases} \mathit{getHash}\ (\mathit{Unit}\ (a, b)) = a \\ \mathit{getHash}\ (\mathit{Comp}\ a\ (l, r)) = a \end{cases}$$

$$\equiv \qquad \{\ \text{Natural-}\pi_1\text{ (12)}\ \}$$

$$\begin{cases} \mathit{getHash}\ (\mathit{Unit}\ (a, b)) = \pi_1\ (a, b) \\ \mathit{getHash}\ (\mathit{Comp}\ a\ (l, r)) = \pi_1\ (a, (l, r)) \end{cases}$$

$$\equiv \qquad \{\ \text{Definição inFTree}\ \}$$

$$\begin{cases} \mathit{getHash}\ (\mathsf{in}\ (i_1\ (a, b))) = \pi_1\ (a, b) \\ \mathit{getHash}\ (\mathsf{in}\ (i_2\ (a, (l, r)))) = \pi_1\ (a, (l, r)) \end{cases}$$

$$\equiv \qquad \{\ \text{Def-comp (72)}\ \}$$

$$\begin{cases} \mathit{getHash} \cdot \mathsf{in} \cdot i_1\ (a, b) = \pi_1\ (a, b) \\ \mathit{getHash} \cdot \mathsf{in} \cdot i_2\ (a, (l, r)) = \pi_1\ (a, (l, r)) \end{cases}$$

$$\equiv \qquad \{\ \text{Igualdade Extensional (71)}\ \}$$

$$\begin{cases} \mathit{getHash} \cdot \mathsf{in} \cdot i_1 = \pi_1 \\ \mathit{getHash} \cdot \mathsf{in} \cdot i_2 = \pi_1 \end{cases}$$

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

$$g \cdot \mathsf{in} = [\pi_1, \pi_1]$$

$$\equiv \qquad \{\ \text{Shunt-left (33)}\ \}$$

$$g = [\pi_1, \pi_1] \cdot \mathsf{out}$$

$\square$

$$g2\ (l, r) = \mathit{Comp}\ (\mathit{concHash}\ \langle \mathit{getHash}\ l, \mathit{getHash}\ r \rangle)\ (l, r)$$

$$\equiv \quad \{ \text{ Fusão-}\times\text{ (9); Natural-}\pi_1\text{; Natural-}\pi_2 \ \}$$

$$g2\ (l, r) = Comp\ (concHash\ \langle getHash \cdot \pi_1, getHash \cdot \pi_2 \rangle\ (l, r))\ (l, r)$$

$$\equiv \quad \{ \text{ Uncurry (84); Natural-id (1); Fusão-}\times\text{ (9) } \}$$

$$g2\ (l, r) = \widehat{Comp}\ \langle concHash \cdot \langle getHash \cdot \pi_1, getHash \cdot \pi_2 \rangle, id \rangle\ (l, r)$$

$$\equiv \quad \{ \text{ Igualdade Extensional (71) } \}$$

$$g2 = \widehat{Comp}\ \langle concHash \cdot \langle getHash \cdot \pi_1, getHash \cdot \pi_2 \rangle, id \rangle$$

$$\equiv \quad \{ \text{ Def-}\times\text{ (9) } \}$$

$$g2 = \widehat{Comp}\ \langle concHash \cdot (getHash \times getHash), id \rangle$$

$$\equiv \quad \{ \text{ Def-comp (72) } \}$$

$$g2 = \widehat{Comp} \cdot \langle concHash \cdot (getHash \times getHash), id \rangle$$

□

$getHash = [\pi_1, \pi_1] \cdot \mathsf{out}$
$g\_lTree2MTree :: Hashable\ c \Rightarrow c + (FTree\ \mathbb{Z}\ (\mathbb{Z}, c), FTree\ \mathbb{Z}\ (\mathbb{Z}, c)) \rightarrow FTree\ \mathbb{Z}\ (\mathbb{Z}, c)$
$g\_lTree2MTree = [g1, g2]$ **where**
  $g1 = Unit \cdot \langle Main.hash, id \rangle$
  $g2 = \widehat{Comp} \cdot \langle concHash \cdot (getHash \times getHash), id \rangle$

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

Para obtermos a Merkel root, temos que somar todos os hashes da Merkel tree. Assim, ficamos com o seguinte código:

$$\begin{cases} g\_mroot\ (i_1\ (h, a)) = h \\ g\_mroot\ (i_2\ (h, (l, r))) = h + l + r \end{cases}$$

$$\equiv \quad \{ \text{ Uncurry (84) } \}$$

$$\begin{cases} g\_mroot\ (i_1\ (h, a)) = h \\ g\_mroot\ (i_2\ (h, (l, r))) = \widehat{(+)}\ (h, \widehat{(+)}\ (l, r)) \end{cases}$$

$$\equiv \quad \{ \text{ Definição } add = \widehat{(+)} \ \}$$

$$\begin{cases} g\_mroot\ (i_1\ (h, a)) = h \\ g\_mroot\ (i_2\ (h, (l, r))) = add\ (h, (add\ (l, r))) \end{cases}$$

$$\equiv \quad \{ \text{ Def-comp (72); Definições } \pi_1\ (a, b) = a;\ \pi_2\ (a, b) = b \ \}$$

$$\begin{cases} g\_mroot \cdot i_1\ (h, a) = \pi_1\ (h, a) \\ g\_mroot \cdot i_2\ (h, (l, r)) = add \cdot \langle \pi_1, add \cdot \pi_2 \rangle\ (h, (l, r)) \end{cases}$$

$$\equiv \quad \{ \text{ Igualdade Extensional (71) } \}$$

$$\begin{cases} g\_mroot \cdot i_1 = \pi_1 \\ g\_mroot \cdot i_2 = add \cdot \langle \pi_1, add \cdot \pi_2 \rangle \end{cases}$$

$$\equiv \quad \{ \text{ Natural-id (1); Def-}\times\text{ (10) } \}$$

$$\begin{cases} g\_mroot \cdot i_1 = \pi_1 \\ g\_mroot \cdot i_2 = add \cdot (id \times add) \end{cases}$$

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

$$g\_mroot = [\pi_1, add \cdot (id \times add)]$$

□

$g\_mroot = [g1, g2]$ **where**

$g1 = \pi_1$
$g2 = add \cdot (id \times add)$

Exercício 1.4:

Como a etiqueta da Merkel root é a soma de todas as hashes dos nodos das subárvores, uma alteração num dos elementos alterará as etiquetas de todos os nodos superiores e, obviamente, da Merkel root também.

Valorização:

$$\left\{\begin{array}{l} g\_pairsList\ [\,] = i_1\ ()\\ g\_pairsList\ (h:t) = i_2\ ((h, head\ t), tail\ t)\end{array}\right.$$

$\equiv$ $\{$ Definições $\pi_1\ (a,b) = a$; $\pi_2\ (a,b) = b$ $\}$

$$\left\{\begin{array}{l} g\_pairsList \cdot nil = i_1 \cdot outList \cdot getEvenBlock\ ()\\ g\_pairsList \cdot cons\ (h,t) = i_2 \cdot \langle\langle\pi_1, \cdot\rangle, head \cdot \pi_2, tail \cdot \pi_2\rangle \cdot outList \cdot getEvenBlock\ (h,t)\end{array}\right.$$

$\equiv$ $\{$ Igualdade Extensional (71) $\}$

$$\left\{\begin{array}{l} g\_pairsList \cdot nil = i_1\\ g\_pairsList \cdot cons = i_2 \cdot \langle\langle\pi_1, \cdot\rangle, head \cdot \pi_2, tail \cdot \pi_2\rangle\end{array}\right.$$

$\equiv$ $\{$ Natural-id (1); Def-$\times$ (10) $\}$

$$\left\{\begin{array}{l} g\_pairsList \cdot nil = i_1\ g\_pairsList \cdot cons = i_2 \cdot \langle id \times head, tail \cdot \pi_2\rangle\\ \cdot\end{array}\right.$$

$\equiv$ $\{$ Eq-+ (27) $\}$

$$g\_pairsList \cdot [nil, cons] = [i_1, i_2 \cdot \langle id \times head, tail \cdot \pi_2\rangle]$$

$\equiv$ $\{$ Definição **in** $= [nil, cons]$; Shunt-Left (33) $\}$

$$g\_pairsList = [i_1, i_2 \cdot \langle i_1 \times head, tail \cdot \pi_2\rangle] \cdot out$$

$\Box$

Como forma de garantirmos que há sempre um número par de elementos, antes de executarmos o anamorfismo executamos a função $getEvenBlock$.

$pairsList :: [a] \rightarrow [(a,a)]$
$pairsList = [\!(g\_pairsList)\!] \cdot getEvenBlock$
$g\_pairsList = [g1, g2] \cdot outList$ **where**
  $g1 = i_1$
  $g2 = i_2 \cdot \langle id \times head, tail \cdot \pi_2\rangle$
$classicMerkleTree :: Hashable\ a \Rightarrow [a] \rightarrow FTree\ \mathbb{Z}\ \mathbb{Z}$
$classicMerkleTree = (hyloNEList\ conquer\ divide) \cdot (\mathtt{map}\ \ Main.hash)$
$divide = \bot$
$conquer = [head, joinMerkleTree]$ **where**
  $joinMerkleTree\ (l,m) = mergeMerkleTree\ m\ (evenMerkleTreeList\ l)$
  $mergeMerkleTree = (\![\,[h_1, h_2]\,]\!)$
  $h_1\ c\ l = \bot$
  $h_2\ (c,(f,g))\ l = \bot$
  $evenMerkleTreeList = \bot$

Problema 2

$wc\_w\_final :: [Char] \rightarrow Int$
$wc\_w\_final = wrapper \cdot worker$
$worker = (\![g1, g2]\!)$
$wrapper = \pi_2$

Para resolvermos o problema, recorremos à recursividade mútua em que no primeiro elemento calculamos $lookahead\_sep$ e no segundo usamos o resultado da recursividade para saber se se deve, ou não, incrementar o acumulador.

$$\begin{tikzcd}
Char^* \arrow[r, "outList"] \arrow[d, "worker"'] & 1 + (Char \times Char^*) \arrow[d, "id + id \times worker"] \\
Bool \times \mathbb{N} & 1 + (Char \times (Bool \times \mathbb{N})) \arrow[l, "{[g1, g2]}"]
\end{tikzcd}$$

Gene de $worker$:

Como worker é o catamorfismo de duas funções h e k em recursividade mútua, concluímos que $[g1, g2] = [\langle h_1, k_1\rangle, \langle h_2, k_2\rangle]$.

Assim, através da lei Eq-+ (17) temos:

$$g1 = \langle h_1, k_1\rangle$$
$$g2 = \langle h_2, k_2\rangle$$

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

$$h_1 = \underline{True}$$
$$h_2\ (c, (a, b)) = c \equiv \texttt{' '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'}$$
$$k_1 = \underline{0}$$
$$k_2\ (c, (a, b)) = \textbf{if}\ (\neg\ (h_2\ (c, (a, b))) \wedge a)\ \textbf{then}\ b + 1\ \textbf{else}\ b$$

Problema 3

O isomorfismo inX/outX é representado pelo seguinte diagrama:

$$\begin{tikzcd}
A + (B \times X\ A\ B^2) \arrow[r, shift left, "inX"] & X\ A\ B \arrow[l, shift left, "outX"]
\end{tikzcd}$$

A partir deste concluímos que as funções in e out de X são as seguintes:

$$inX :: u + (i, (X\ u\ i, X\ u\ i)) \to X\ u\ i$$
$$inX\ (i_1\ u) = XLeaf\ u$$
$$inX\ (i_2\ (i, (l, r))) = Node\ i\ l\ r$$
$$outX\ (XLeaf\ u) = i_1\ u$$
$$outX\ (Node\ i\ l\ r) = i_2\ (i, (l, r))$$

A partir do tipo $A + (B \times X\ A\ B^2)$ criamos ambos bifuntor e funtor.

$$baseX\ f\ h\ g = f + (h \times (g \times g))$$
$$recX\ f = baseX\ id\ id\ f$$

As funções cataX, anaX e hyloX são criadas a partir dos diagramas respetivos:

$$\begin{tikzcd}
X\ A\ B \arrow[r, "out"] \arrow[d, "(\!|g|\!)"'] & A + (B \times ((X\ A\ B) \times (X\ A\ B))) \arrow[d, "id + id \times (\!|g|\!)"] \\
C & A + (B \times (C \times C)) \arrow[l, "g"]
\end{tikzcd}$$

Assim concluímos que $(\!|g|\!) = g \cdot (id + id \times (\!|g|\!)) \cdot out$

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

$$\begin{tikzcd}
X\ B\ C & B + (C \times ((X\ B\ C) \times (X\ B\ C))) \arrow[l, "\mathbf{in}"'] \\
A \arrow[u, "ana\ g"] \arrow[r, "g"'] & B + (C \times (A \times A)) \arrow[u, "id + id \times (ana\ g)"']
\end{tikzcd}$$

Logo, $ana\ g = \mathbf{in} \cdot (id + id \times (ana\ g)) \cdot g$

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

$$\begin{tikzcd}
C & & A + (B \times (C \times C)) \arrow[ll, "g"'] \\
X\ \ B\ \ C \arrow[u, "(\!|g|\!)"] \arrow[rr, shift left, "\mathbf{in}"] & & B + (C \times ((X\ \ B\ \ C) \times (X\ \ B\ \ C))) \arrow[u, "id+id\times(\!|g|\!)"'] \arrow[ll, shift left, "out"] \\
A \arrow[u, "ana\ \ g"] \arrow[rr, "g"'] & & B + (C \times (A \times A)) \arrow[u, "id+id\times(ana\ \ g)"']
\end{tikzcd}$$

Como é possível observar no diagrama, o hilomorfismo é apenas a junção do catamorfismo e do anamorfismo que acabamos de criar.

$$hyloX\ \ h\ g = cataX\ \ h \cdot anaX\ \ g$$

Para criarmos a biblioteca para um número variável de partições, temos antes que escrever o seu tipo. Neste caso, em vez de ter um par de filhos, cada nodo tem uma lista destes.

Deste modo, o seu tipo pode ser escrito em Haskell do seguinte modo:

$$\mathbf{data}\ XNary\ u\ i = NLeaf\ \ u \mid NNode\ \ i\ [XNary\ \ u\ \ i]\ \mathbf{deriving}\ (Show, Eq)$$

Agora seguimos os mesmos passos que usamos para criar a biblioteca anterior.

$$\begin{tikzcd}
A + (B \times (XNary\ \ A\ \ B)^*) \arrow[r, shift left, "inX"] & XNary\ \ A\ \ B \arrow[l, shift left, "outX"]
\end{tikzcd}$$

$$\begin{aligned}
&inXNary\ (i_1\ \ u) = NLeaf\ \ u \\
&inXNary\ (i_2\ \ (i, l)) = NNode\ \ i\ \ l \\
&outXNary\ (NLeaf\ \ u) = i_1\ \ u \\
&outXNary\ (NNode\ \ i\ \ l) = i_2\ \ (i, l)
\end{aligned}$$

Como, neste tipo, um nodo tem uma lista de filhos, o bifuntor e o funtor fazem uso do map .

$$\begin{aligned}
&baseXNary\ f\ \ h\ \ g = f + (h \times \mathsf{map}\ \ g) \\
&recXNary\ f = baseXNary\ \ id\ \ id\ \ f
\end{aligned}$$

$$\begin{tikzcd}
XNary\ \ A\ \ B \arrow[rr, "out"] \arrow[d, "(\!|g|\!)"'] & & A + (B \times (XNary\ \ A\ \ B)^*) \arrow[d, "id+id\times(\mathsf{map}\ \ (\!|g|\!))"] \\
C & & A + (B \times C^*) \arrow[ll, "g"]
\end{tikzcd}$$

$$cataXNary\ \ g = g \cdot (recXNary\ \ (cataXNary\ \ g)) \cdot outXNary$$

$$\begin{tikzcd}
XNary\ \ B\ \ C & & B + (C \times (XNary\ \ B\ \ C)^*) \arrow[ll, "\mathbf{in}"'] \\
A \arrow[u, "ana\ \ g"] \arrow[rr, "g"'] & & B + (C \times A^*) \arrow[u, "id+id\times(\mathsf{map}\ \ (ana\ \ g))"']
\end{tikzcd}$$

$$anaXNary\ \ g = inXNary \cdot (recXNary\ \ (anaXNary\ \ g)) \cdot g$$

$$\begin{tikzcd}
C & & A + (B \times C^*) \arrow[ll, "g"'] \\
XNary\ B\ C \arrow[u, "(\!|g|\!)"] \arrow[rr, shift left, "\mathbf{in}"] & & B + (C \times (XNary\ B\ C)^*) \arrow[ll, shift left, "out"] \arrow[u, "id+id\times(\mathsf{map}\ \ (\!|g|\!))"'] \\
A \arrow[u, "ana\ g"] \arrow[rr, "g"'] & & B + (C \times A^*) \arrow[u, "id+id\times(\mathsf{map}\ \ (ana\ g))"']
\end{tikzcd}$$

$$hyloXNary\ h\ g = cataXNary\ h \cdot anaXNary\ g$$

Problema 4

$$\begin{array}{l}
pairL = [\![g]\!]\ \mathbf{where} \\
\quad g\ [x] = i_1\ (x, x) \\
\quad g\ (h : i : t) = i_2\ ((h, i), (i : t))
\end{array}$$

$$\begin{tikzcd}
(Pos \times Pos)^* \arrow[d, "(\!|g|\!)"'] & 1 + (Pos \times Pos) \times (Pos \times Pos)^* \arrow[l, "inList={[nil,cons]}"'] \arrow[d, "id+id\times(\!|g|\!)"] \\
Map^{Map} & 1 + (Pos \times Pos) \times Map^{Map} \arrow[l, "g={[\underline{id},f_2]}"]
\end{tikzcd}$$

O diagrama acima foi criado com o objetivo de visualizar o funcionamento de f_2. Esta recebe um $tuple$ cujo primeiro elemento é um $tuple$ de posições e o segundo uma função que, dado um mapa, gera um mapa. O diagram facilita o entendimento de que f_2 gera uma função que gera um mapa quando receber um mapa. Assim, deduzimos que o comportamento funcional de f_2 passa por (A): processar o primeiro elemento do tuplo e gerar uma função que dado um mapa gera um mapa e de seguida (B): compor a nova função com o 2º elemento do tuplo, gerando uma nova função capaz de gerar um mapa quando receber um mapa.

Passo (A): Primeiro, é necessário determinar a célula que vai substituir a célula existente com o auxílio da função $toCell$. Ao verificar o código fornecido para este trabalho prático, verificou-se que esta função recebe como entrada duas posições e não um $tuple$ de posições, portanto utilizou-se $uncurry$. Nestas condições, $\widehat{toCell}$ é capaz de gerar uma nova célula. Tendo a célula gerada, a substituição é feita na posição indicada pelo primeiro elemento do $tuple$ de posições. Por fim, a função $substM$ é um catamorfismo de naturais que trata do passo de gerar a função que é capaz de gerar um mapa quando receber um mapa. Esta função faz uso da função $subst$ já disponibilizada, recebendo um elemento da substituir, e a posição (coluna e linha) onde deve ser efetuada a substituição.

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "substM\ x\ coluna"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id+(substM\ x\ coluna)"] \\
Map^{Map} & 1 + Map^{Map} \arrow[l, "g={[g1,g2]}"]
\end{tikzcd}$$

Na função $substM$ temos dois casos:

1. $g1$: caso em que a linha é 0
2. $g2$: caso resursivo

Através da função $substM$ é possível gerar uma função que gera mapas a partir de mapas através do elemento a substituir e à posição onde substitui-lo. Temos $\langle \widehat{toCell}, \pi_1 \rangle$ que devolve um $tuple$ com o elemento a substituir no primeiro elemento e a posição onde substituilo no segundo. Utilizamos

$\widehat{substM} \cdot assocl$ para que a função $substM$ possa aceitar esse tipo à entrada, e possa gerar uma função que gera mapas a partir de mapas recebendo um *tuple* como o descrito à entrada.

Passo (B): O segundo e último passo consiste em compor duas funções presentes num *tuple* através da auxiliar *composicao*, obtendo $f_2 = composicao \cdot ((\widehat{substM} \cdot assocl) \cdot \langle \widehat{toCell}, \pi_1 \rangle \times id))$.

$$
\begin{array}{l}
markMap :: [Pos] \rightarrow Map \rightarrow Map \\
markMap\ l = (\![\underline{id}, f_2]\!)\ (pairL\ l)\ \textbf{where} \\
\quad f_2 = composicao \cdot ((\widehat{substM} \cdot assocl) \cdot \langle \widehat{toCell}, \pi_1 \rangle \times id) \\
composicao\ (f, g) = f \cdot g \\
substM :: b \rightarrow Int \rightarrow Int \rightarrow [[b]] \rightarrow [[b]] \\
substM\ x\ lin = (\![g1, g2]\!) \\
\quad \textbf{where}\ g1 = \underline{\lambda(c : cs) \rightarrow subst\ x\ lin\ c : cs} \\
\qquad g2\ f\ (c : cs) = c : f\ cs
\end{array}
$$

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

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:*

$$prop\_reconst\ l = \bot$$

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.*

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

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

$$
\begin{array}{l}
inBounds\ m\ (x, y) = \bot \\
prop\_idemp2\ l\ m = \bot
\end{array}
$$

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

$$prop\_extr2\ l\ m = \bot$$

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.