Cálculo de Programas
Trabalho Prático
LCC — 2021/22 (2º semestre)

Departamento de Informática
Universidade do Minho

Junho de 2022

Grupo nr.	53
a94956	André Lucena Ribas Ferreira
a96936	Carlos Eduardo Da Silva Machado
a97485	Gonçalo Manuel Maia de Sousa

1 Preâmbulo

Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso, baseia-se num repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usa 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

O algorítmo da *divisão Euclidiana*,

$$\begin{array}{l} ed\ (n, 0) = Nothing \\ ed\ (n, d+1) = (Just \cdot \pi_1)\ (aux\ d\ n) \end{array}$$

dá erro quando o denominador d é zero, recorrendo à função auxiliar seguinte nos outros casos, paramétrica em d:

$$aux\ d = \langle q\ d, \langle r\ d, c\ d \rangle \rangle$$

Esta, por sua vez, é o emparelhamento das seguintes funções mutuamente recursivas,

$$\begin{array}{l} q\ d\ 0 = 0 \\ q\ d\ (n+1) = q\ d\ n + (\mathbf{if}\ x \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \mathbf{where}\ x = c\ d\ n \\ r\ d\ 0 = 0 \\ r\ d\ (n+1) = \mathbf{if}\ x \equiv 0\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + r\ d\ n\ \mathbf{where}\ x = c\ d\ n \\ c\ d\ 0 = d \\ c\ d\ (n+1) = \mathbf{if}\ x \equiv 0\ \mathbf{then}\ d\ \mathbf{else}\ x - 1\ \mathbf{where}\ x = c\ d\ n \end{array}$$

onde q colabora na produção do quociente, r na produção do resto, e c é uma função de controlo — todas paramétricas no denominador d.

Mostre, por aplicação da lei de recursividade mútua, que $aux\ d$ é a mesma função que o ciclo-for seguinte:

$$
\begin{array}{l}
loop\ d = \mathsf{for}\ (g\ d)\ (0,(0,d))\ \mathbf{where} \\
\quad g\ d\ (q,(r,0)) = (q+1,(0,d)) \\
\quad g\ d\ (q,(r,c+1)) = (q,(r+1,c))
\end{array}
$$

Sugestão: consultar o anexo B.

Problema 2

Considere o seguinte desafio, extraído de O Bebras - Castor Informático (edição 2020):

> ***11 — Robôs e Pedras Preciosas*** *A Alice e o Bob estão a controlar um robô num labirinto com pedras preciosas. O robô começa na localização indicada na figura abaixo [Fig. 1]. O robô segue um caminho até encontrar uma bifurcação. Um dos jogadores decide qual dos caminhos (esquerda ou direita) o robô deve tomar. Depois, o robô segue esse caminho até encontrar outra bifurcação, e assim consecutivamente (o robô nunca volta para trás no seu caminho).*
>
> *A Alice e o Bob decidem à vez qual a direção a seguir, com a Alice a começar, o Bob decidindo a 2ª bifurcação, a Alice a 3ª e por aí adiante. O jogo termina quando o robô chegar ao final de um caminho sem saída, com o robô a recolher todas as pedras preciosas que aí encontrar. A Alice quer que o robô acabe o jogo com o maior número possível de pedras preciosas, enquanto que o Bob quer que o robô acabe o jogo com o menor número possível de pedras preciosas.*
>
> *A Alice e o Bob sabem que cada um vai tentar ser mais esperto que o outro. Por isso se, por exemplo, o Bob redirecionar o robô para uma bifurcação onde é possível recolher 3 ou 7 pedras preciosas, ele sabe que a Alice vai comandar o robô escolhendo o caminho que leva às 7 pedras preciosas.*

O labirinto deste desafio (Fig. 1) configura uma árvore binária de tipo *LTree* cujas folhas têm o número de pedras preciosas do correspondente caminho:[1]

$$
\begin{array}{l}
t = \mathit{Fork}\ (\\
\quad \mathit{Fork}\ (\\
\quad\quad \mathit{Fork}\ (\mathit{Leaf}\ 2, \mathit{Leaf}\ 7), \\
\quad\quad \mathit{Fork}\ (\mathit{Leaf}\ 5, \mathit{Leaf}\ 4)), \\
\quad \mathit{Fork}\ (\\
\quad\quad \mathit{Fork}\ (\mathit{Leaf}\ 8, \mathit{Leaf}\ 6), \\
\quad\quad \mathit{Fork}\ (\mathit{Leaf}\ 1, \mathit{Leaf}\ 3)) \\
\quad\quad)
\end{array}
$$

1. Defina como catamorfismo de *LTree*'s a função $both :: \mathit{LTree}\ \mathit{Int} \to \mathit{Int} \times \mathit{Int}$ tal que

   $$(a, b) = both\ t$$

   dê,

   - em a: o resultado mais favorável à Alice, quando esta é a primeira a jogar, tendo em conta as jogadas do Bob e as suas;
   - em b: o resultado mais favorável ao Bob, quando este é o primeiro a jogar, tendo em conta as jogadas da Alice e as suas.

2. De seguida, extraia (por recursividade mútua) as funções (recursivas) $alice$ e bob tais que

   $$both = \langle alice, bob \rangle$$

   (Alternativamente, poderá codificar $alice$ e bob em primeiro lugar e depois juntá-las num catamorfismo recorrendo às leis da recursividade mútua.)

[1] Abstracção: as diferentes pedras preciosas são irrelevantes, basta o seu número.

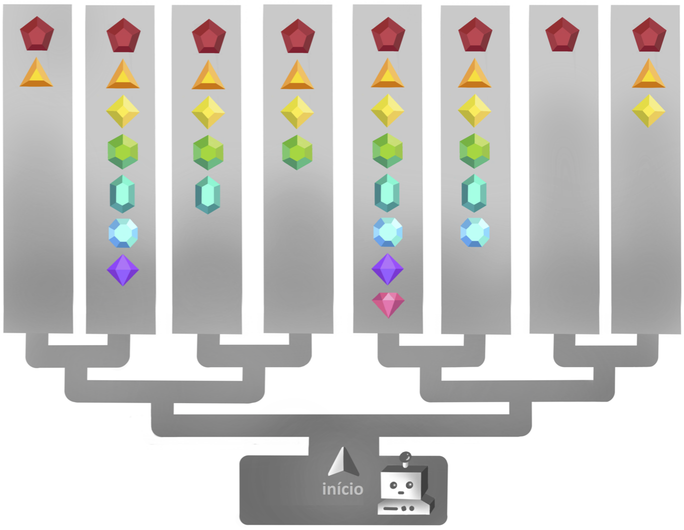


Figura 1: Labirinto de “Robôs e Pedras Preciosas”.

Problema 3

O triângulo de Sierpinski (Fig. 2) é uma figura geométrica fractal em que um triângulo se subdivide recursivamente em sub-triângulos, da seguinte forma: considere-se um triângulo rectângulo e isósceles A cujos catetos têm comprimento s. A estrutura fractal é criada desenhando-se três triângulos no interior de A, todos eles rectângulos e isósceles e com catetos de comprimento $s \div 2$. Este passo é depois repetido para cada um dos triângulos desenhados e assim sucessivamente (Fig. 3).

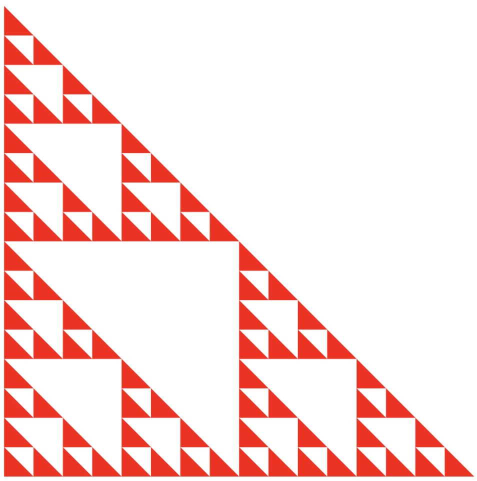


Figura 2: Um triângulo de Sierpinski com profundidade 4.

Um triângulo de Sierpinski é gerado repetindo-se infinitamente o processo acima descrito; no entanto para efeitos de visualização é conveniente parar o processo recursivo a um determinado nível.

A figura a desenhar é constituída por triângulos todos da mesma dimensão (por exemplo, no quarto triângulo da Fig. 3 desenharam-se 27 triângulos). Seja cada triângulo geometricamente descrito pelas coordenadas do seu vértice inferior esquerdo e o comprimento dos seus catetos:

$$\textbf{type}\ \mathit{Tri} = (\mathit{Point}, \mathit{Side})$$

onde

$$\textbf{type}\ \mathit{Side} = \mathit{Int}$$
$$\textbf{type}\ \mathit{Point} = (\mathit{Int}, \mathit{Int})$$

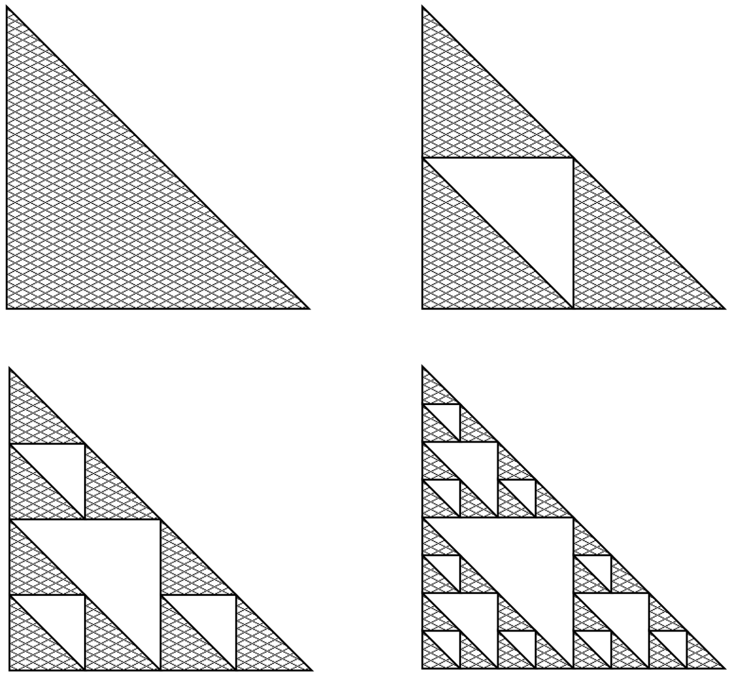


Figura 3: Construção de um triângulo de Sierpinski

A estrutura recursiva que suporta a criação de triângulos de Sierpinski é captada por uma árvore ternária,

$$\mathbf{data}\ \mathsf{LTree3}\ a = \mathit{Tri}\ a \mid \mathit{Nodo}\ (\mathsf{LTree3}\ a)\ (\mathsf{LTree3}\ a)\ (\mathsf{LTree3}\ a)\ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

em cujas folhas se irão encontrar os triângulos mais pequenos, todos da mesma dimensão, que deverão ser desenhados. Apenas estes conterão informação de carácter geométrico, tendo os nós da árvore um papel exclusivamente estrutural. Portanto, a informação geométrica guardada em cada folha consiste nas coordenadas do vértice inferior esquerdo e no cateto do respectivo triângulo. A função

$$\begin{aligned}&\mathit{sierpinski} :: (\mathit{Tri}, \mathit{Int}) \rightarrow [\mathit{Tri}]\\&\mathit{sierpinski} = \mathit{folhasSierp} \cdot \mathit{geraSierp}\end{aligned}$$

recebe a informação do triângulo exterior e a profundidade pretendida, que funciona como critério de paragem do processo de construção do fractal. O seu resultado é a lista de triângulos a desenhar.

Esta função é um hilomorfismo do tipo LTree3, i.e. a composição de duas funções: uma que gera LTree3s,

$$\begin{aligned}&\mathit{geraSierp} :: (\mathit{Tri}, \mathit{Int}) \rightarrow \mathsf{LTree3}\ \mathit{Tri}\\&\mathit{geraSierp} = \mathit{anaLTree3}\ g_2\end{aligned}$$

e outra que as consome:

$$\begin{aligned}&\mathit{folhasSierp} :: \mathsf{LTree3}\ \mathit{Tri} \rightarrow [\mathit{Tri}]\\&\mathit{folhasSierp} = \mathit{cataLTree3}\ g_1\end{aligned}$$

Trabalho a realizar:

1. Desenvolver a biblioteca *pointfree* para o tipo LTree3 de forma análoga a outras bibliotecas que conhece (*eg.* BTree, *LTree*, etc).
2. Definir os genes g_1 e g_2 do hilomorfismo $\mathit{sierpinski}$.
3. Correr

$$\mathit{teste} = \mathit{desenha}\ (\mathit{sierpinski}\ (\mathit{base}, 3))$$

para verificar a correcta geração de triângulos de Sierpinski em SVG[2], onde $desenha$ é uma função dada no anexo D que, para o argumento $sierpinski\ (base, 3)$, deverá produzir o triângulo colorido da Fig. 2.[3]

Problema 4

Os computadores digitais baseiam-se na representação Booleana da informação, isto é, sob a forma de *bits* que podem ter dois valores, vulg. 0 e 1. Um problema muito frequente é o de os bits se alterarem, devido a ruído ao nível electrónico. Essas alterações espúrias designam-se *bit-flips* e podem acontecer a qualquer nível: na transmissão de informação, na gravação em disco, etc, etc.

Em contraste com essas perturbações, o utilizador de serviços informáticos raramente dá pela sua presença. Porquê? Porque existe muito trabalho teórico em correcção dos erros gerados por *bit-flips*, que os permite esconder do utilizador.

O objectivo desta questão é conseguirmos avaliar experimentalmente o funcionamento de uma dessas técnicas de correcção de erros, a mais elementar de todas, chamada *código de repetição*, escrevendo tão pouco código (Haskell) quanto possível. Para isso vamos recorrer ao mónade das *distribuições probabilísticas* (detalhes no apêndice C).

Vamos supor que queremos medir a eficácia de um tal código na situação seguinte: queremos transmitir mensagens que constam exclusivamente de letras maiúsculas, representadas por 5 bits cada uma segundo o esquema seguinte de codificação,

$$
\begin{array}{l}
enc :: Char \to Bin \\
enc\ c = tobin\ (ord\ c - ord\ \texttt{'A'})
\end{array}
$$

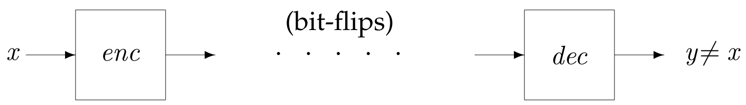
e descodificação,

$$
\begin{array}{l}
dec :: Bin \to Char \\
dec\ b = chr\ (frombin\ b + ord\ \texttt{'A'})
\end{array}
$$

onde $tobin$ e $frombin$ são funções dadas no anexo D. Por exemplo,

$$
\begin{array}{l}
enc\ \texttt{'A'}\ = [0, 0, 0, 0, 0] \\
enc\ \texttt{'B'}\ = [0, 0, 0, 0, 1] \\
\vdots \\
enc\ \texttt{'Z'}\ = [1, 1, 0, 0, 1]
\end{array}
$$

Embora dec e enc sejam inversas uma da outra, para o intervalo de $\texttt{'A'}$ a $\texttt{'Z'}$, deixam de o ser quando, a meio da transmissão, acontecem bit-flips:



Vejamos com quantificar "os estragos". Sabendo-se, por exemplo e por observação estatística, que a probabilidade de um 0 virar 1 é de 4% e a de 1 virar 0 é de 10% [4], simula-se essa informação sobre a forma de uma função probabilística, em Haskell:

$$
\begin{array}{l}
bflip :: Bit \to \mathsf{Dist}\ Bit \\
bflip\ 0 = D\ [(0, 0.96), (1, 0.04)] \\
bflip\ 1 = D\ [(1, 0.90), (0, 0.10)]
\end{array}
$$

Agora vamos simular o envio de caracteres. O que devia ser $transmit = dec \cdot enc$ vai ter agora que prever a existência de possíveis bit-flips no meio da transmissão:

$$
transmit = dec' \cdot propagate\ bflip \cdot enc
$$

Por exemplo, $transmit\ \texttt{'H'}$ irá dar a seguinte distribuição:

[2] SVG, abreviatura de *Scalable Vector Graphics*, é um dialecto de XML para computação gráfica. A biblioteca Svg.hs (fornecida) faz uma interface rudimentar entre Haskell e SVG.

[3] O resultado é gravado no ficheiro `_.html`, que pode ser visualizado num browser. Poderão ser feitos testes com outros níveis de produndidade.

[4] Estas probabilidades, na prática muito mais baixas, estão inflacionadas para mais fácil observação.

```
'H'  67.2%
'D'   7.5%
'F'   7.5%
'G'   7.5%
'P'   2.8%
'X'   2.8%
'E'   0.8%
'B'   0.8%
'C'   0.8%
'L'   0.3%
'N'   0.3%
'O'   0.3%
'T'   0.3%
'V'   0.3%
'W'   0.3%
'`'   0.1%
'A'   0.1%
```

A saída 'H', que se esperava com 100% de certeza, agora só ocorrerá, estatísticamente, com a probabilidade de 67.2%, consequência dos bit-flips, havendo um âmbito bastante grande de respostas erradas, mas com probabilidades mais baixas.

1. **Trabalho a fazer:** completar a definição do catamorfismo de listas $propagate$.

O que se pode fazer quanto a estes erros de transmissão? Os chamados códigos de repetição enviam cada bit um número impar de vezes, tipicamente 3 vezes. Cada um desses três bits (que na origem são todos iguais) está sujeito a bit-flips. O que se faz é *votar* no mais frequente — ver função v_3 no anexo. Se agora a transmissão do `'H'` for feita em triplicado, usando

$$transmit3 = dec' \cdot propagate3\ bflip3 \cdot enc$$

ter-se-á:

```
Main> transmit3 'H'
 'H'  91.0%
 'F'   2.6%
 'G'   2.6%
 'D'   2.6%
 'P'   0.4%
 'X'   0.4%
 'B'   0.1%
 'C'   0.1%
 'E'   0.1%
```

Vê-se que a probabilidade da resposta certa aumentou muito, para 91%, com redução também do espectro de respostas erradas.

2. **Trabalho a fazer:** completar a definição do catamorfismo de listas $propagate3$ e da função $bflip3$.

Apesar da sua eficácia, esta técnica de correcção de erros é dispendiosa, obrigando o envio do triplo dos bits. Isso levou a comunidade científica a encontrar formas mais sofisticadas para resolver o mesmo problema sem tal "overhead". Quem estiver interessado em saber mais sobre este fascinante tópico poderá começar por visualizar este vídeo no YouTube.

Anexos

A Documentação para realizar o trabalho

Para cumprir de forma integrada os objectivos do 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 utilizando o utiliário cabal disponível em haskell.org.

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 E 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.

No anexo D disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

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

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:[6]

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

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

[6] Exemplos tirados de [3].

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

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

B Regra prática para a recursividade mútua em $\mathbb{N}_0$

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[7]

Para o caso de funções sobre os números naturais ($\mathbb{N}_0$, com functor $\mathsf{F}\ X = 1 + X$) é fácil derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado Cálculo de Programas. Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema:

$$
\begin{array}{l}
\mathit{fib}\ 0 = 1 \\
\mathit{fib}\ (n+1) = f\ n \\
f\ 0 = 1 \\
f\ (n+1) = \mathit{fib}\ n + f\ n
\end{array}
$$

Obter-se-á de imediato

$$
\begin{array}{l}
\mathit{fib}' = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\
\quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f) \\
\quad \mathit{init} = (1, 1)
\end{array}
$$

usando as regras seguintes:

- O corpo do ciclo loop terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.[8]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n.
- Em init coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios do segundo grau $ax^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[9], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$
\begin{array}{l}
f\ 0 = c \\
f\ (n+1) = f\ n + k\ n \\
k\ 0 = a + b \\
k\ (n+1) = k\ n + 2\ a
\end{array}
$$

Seguindo a regra acima, calcula-se de imediato a seguinte implementação, em Haskell:

$$
\begin{array}{l}
f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\
\quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a) \\
\quad \mathit{init} = (c, a + b)
\end{array}
$$

[7] Lei (3.93) em [3], página 110.
[8] Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.
[9] Secção 3.17 de [3] e tópico Recursividade mútua nos vídeos de apoio às aulas teóricas.

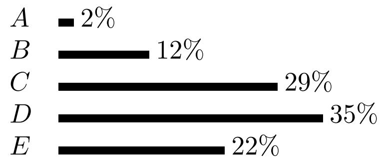
C O mónade das distribuições probabilísticas

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

$$\mathbf{newtype}\ \mathsf{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{1}$$

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

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



será representada pela distribuição

$$\begin{array}{l} d_1 :: \mathsf{Dist}\ Char \\ d_1 = D\ [(\texttt{'A'}, 0.02), (\texttt{'B'}, 0.12), (\texttt{'C'}, 0.29), (\texttt{'D'}, 0.35), (\texttt{'E'}, 0.22)] \end{array}$$

que o GHCi mostrará assim:

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

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

$$d_2 = uniform\ (words\ \texttt{"Uma frase de cinco palavras"})$$

isto é

```
     "Uma"  20.0%
   "cinco"  20.0%
      "de"  20.0%
   "frase"  20.0%
"palavras"  20.0%
```

distribuição *normais*, eg.

$$d_3 = normal\ [10 \mathinner{..} 20]$$

etc.[10] Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a, 1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q * p) \mid (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$$

em que $g : A \rightarrow \mathsf{Dist}\ B$ e $f : B \rightarrow \mathsf{Dist}\ C$ são funções **monádicas** que representam *computações probabilísticas*.

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

[10] Para mais detalhes ver o código fonte de Probability, que é uma adaptação da biblioteca PHP ("Probabilistic Functional Programming"). Para quem quiser saber mais recomenda-se a leitura do artigo [1].

D Código fornecido

Problema 3

Triângulo de base:

$$base = ((0,0), 32)$$

Desenho de triângulos em SVG:

$$desenha\ x = picd''\ [scale\ 0.44\ (0,0)\ (x \ggg tri2svg)]$$

Função que representa cada triângulo em SVG:

$$tri2svg :: Tri \to Svg$$
$$tri2svg\ (p, c) = (red \cdot polyg)\ [p, p\ .\!+ (0, c), p\ .\!+ (c, 0)]$$

NB: o tipo Svg é sinónimo de $String$:

$$\mathbf{type}\ Svg = String$$

Problema 4

Funções básicas:

$$\mathbf{type}\ Bit = Int$$
$$\mathbf{type}\ Bin = [Bit]$$
$$\mathbf{type}\ Bit3 = (Bit, Bit, Bit)$$
$$tobin = rtrim\ 5 \cdot pad\ 5 \cdot dec2bin$$
$$frombin = bin2dec \cdot rtrim\ 5$$
$$bin2dec :: Bin \to Int$$
$$bin2dec\ [a] = a$$
$$bin2dec\ b = bin2dec\ (init\ b) * 2 + last\ b$$
$$rtrim\ n\ a = drop\ (length\ a - n)\ a$$
$$dec2bin\ 0 = [\,]$$
$$dec2bin\ n = dec2bin\ m + [b]\ \mathbf{where}\ (m, b) = (n \div 2, mod\ n\ 2)$$
$$pad\ n\ x = take\ m\ zeros + x\ \mathbf{where}$$
$$\quad m = n - length\ x$$
$$\quad zeros = 0 : zeros$$
$$bflips = propagate\ bflip$$

Função que vota no bit mais frequente:

$$v_3\ (0,0,0) = 0$$
$$v_3\ (0,0,1) = 0$$
$$v_3\ (0,1,0) = 0$$
$$v_3\ (0,1,1) = 1$$
$$v_3\ (1,0,0) = 0$$
$$v_3\ (1,0,1) = 1$$
$$v_3\ (1,1,0) = 1$$
$$v_3\ (1,1,1) = 1$$

Descodificação monádica:

$$dec' = \mathsf{fmap}\ dec$$

Para visualização abreviada de distribuições:

$$consolidate :: Eq\ a \Rightarrow \mathsf{Dist}\ a \to \mathsf{Dist}\ a$$
$$consolidate = D \cdot filter\ q \cdot \mathsf{map}\ (id \times sum) \cdot collect \cdot unD\ \mathbf{where}\ q\ (a, p) = p > 0.001$$
$$collect\ x = nub\ [k \mapsto nub\ [d' \mid (k', d') \leftarrow x, k' \equiv k] \mid (k, d) \leftarrow x]$$

E Soluções dos alunos

Os alunos devem colocar neste anexo[11] 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 podem ser adicionadas outras funções auxiliares que sejam necessárias, bem como textos, inc. diagramas que expliquem como se chegou às soluções encontradas.

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

Problema 1

Por estudo e aplicação da regra prática no anexo B, entendemos que, para chegarmos à definição apresentada, devemos ter em conta os condicionais em cada uma das funções.

Para determinar $aux\ d = \langle q\ d, \langle r\ d, c\ d \rangle\rangle$ é necessário determinar cada uma das funções de tal forma que se possa utilizar a regra de *Fokkinga*. O functor aplicável neste caso é o functor de naturais, isto é, $F\ f = id + f$. Nesse sentido, $F\ \langle q\ d, \langle r\ d, c\ d \rangle\rangle = id + \langle q\ d, \langle r\ d, c\ d \rangle\rangle$.

$$\left\{\begin{array}{l} q\ d\ 0 = 0 \\ q\ d\ (n+1) = q\ d\ n + (x \equiv 0) \rightarrow 1, 0 \ \mathbf{where}\ x = c\ d\ n \end{array}\right.$$

$\equiv \qquad \{$ (72), (74), (71); (72), (78), substituir x, def succ $\}$

$$\left\{\begin{array}{l} (q\ d) \cdot \underline{0} = \underline{0} \\ ((q\ d) \cdot \mathsf{succ}\ )\ n = (q\ d)\ n + ((\equiv 0)\ (c\ d\ n) \rightarrow 1, 0) \end{array}\right.$$

$\equiv \qquad \{$ (84), uncurry(+) = add, (76), (72), (71) $\}$

$$\left\{\begin{array}{l} (q\ d) \cdot \underline{0} = \underline{0} \\ (q\ d) \cdot \mathsf{succ}\ = add \cdot \langle q\ d, (\equiv 0) \cdot (cd) \rightarrow \underline{1}, \underline{0} \rangle \end{array}\right.$$

$\equiv \qquad \{$ (27), def inNat $\}$

$$(q\ d) \cdot \mathsf{in} = [\underline{0}, add \cdot \langle q\ d, (\equiv 0) \cdot (cd) \rightarrow \underline{1}, \underline{0} \rangle]$$

$\equiv \qquad \{$ (3), (32), (7), (1) $\}$

$$(q\ d) \cdot \mathsf{in} = [\underline{0}, add \cdot \langle id \cdot (qd), ((\equiv 0) \rightarrow \underline{1}, \underline{0}) \cdot \pi_2 \cdot \langle r\ d, c\ d \rangle\rangle]$$

$\equiv \qquad \{$ (11), (22) $\}$

$$(q\ d) \cdot \mathsf{in} = [\underline{0}, add \cdot (id \times (((\equiv 0) \rightarrow \underline{1}, \underline{0}) \cdot \pi_2))] \cdot (id + \langle q\ d, \langle r\ d, c\ d \rangle\rangle)$$

$\square$

Analogamente, consegue-se chegar às definições de $r\ d$ e de $c\ d$:

$$\left\{\begin{array}{l} (r\ d) \cdot \mathsf{in} = [\underline{0}, ((\equiv 0) \cdot \pi_2 \rightarrow \underline{0}, \mathsf{succ}\ \cdot \pi_1) \cdot \pi_2] \cdot (id + \langle q\ d, \langle r\ d, c\ d \rangle\rangle) \\ (c\ d) \cdot \mathsf{in} = [\underline{d}, ((\equiv 0) \rightarrow \underline{d}, (-1)) \cdot \pi_2 \cdot \pi_2] \cdot (id + \langle q\ d, \langle r\ d, c\ d \rangle\rangle) \end{array}\right.$$

$\square$

Aplicando a regra de *Fokkinga*, para $aux\ d = \langle q\ d, \langle r\ d, c\ d \rangle\rangle$ e com:

$$\begin{array}{l} q'\ d = (add \cdot (id \times ((Cp.cond\ (\equiv 0)\ \underline{1}\ \underline{0}) \cdot \pi_2))) \\ r'\ d = ((Cp.cond\ ((\equiv 0) \cdot \pi_2)\ \underline{0}\ (\mathsf{succ}\ \cdot \pi_1)) \cdot \pi_2) \\ c'\ d = ((Cp.cond\ (\equiv 0)\ \underline{d}\ (-1)) \cdot \pi_2 \cdot \pi_2) \end{array}$$

$$aux\ d = (\!|\langle [\underline{0}, q'\ d], [\underline{0}, r'\ d]\rangle\ [\underline{d}, c'\ d]|\!)$$

$\equiv \qquad \{$ (28) duas vezes $\}$

$$aux\ d = (\!|[\langle \underline{0}, \langle \underline{0}, \underline{d} \rangle\rangle, \langle q'\ d, \langle r'\ d, c'\ d \rangle\rangle]|\!)$$

$\square$

[11] E apenas neste anexo, i.e, não podem alterar o resto do documento.

Seja $g\ d = \langle q'\ d, \langle r'\ d, c'\ d\rangle\rangle$. Pelo tipo de $aux\ d = \langle q\ d, \langle r\ d, c\ d\rangle\rangle$:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "aux\ d"'] & 1 + \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0) \arrow[l, "\mathsf{in}"'] \arrow[d, "id + \langle q\ d, \langle r\ d, c\ d\rangle\rangle"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0) \arrow[l, "{[\langle \underline{0}, \langle \underline{0}, \underline{d}\rangle\rangle, g\ d]}"]
\end{tikzcd}$$

Deduzimos o tipo de $g\ d :: \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0) \rightarrow \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0)$. Assim, podemos definir $g\ d$ de forma $point - wise$, com pares $(q, (r, c))$ pertencentes a $\mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0)$. Vemos pelas definições de cada função que faz parte de $g\ d$ que todas elas têm comportamentos distintos para o caso de c ser igual a 0 ou de ser diferente. Desse modo, podemos definir $g\ d$ por esses dois casos, ou seja, se $g\ d$ receber $(q, (r, 0))$ ou se receber $(q, (r, c + 1))$, o que determina completamente o domínio da função, já que c pertence a $\mathbb{N}_0$. Assim, tendo em conta que $g\ d\ (q, (r, c)) = (q'\ d\ (q, (r, c)), (r'\ d\ (q, (r, c)), c'\ d\ (q, (r, c))))$ pela regra (76):

$$\begin{cases} g\ d\ (q, (r, 0)) = (q + 1, (0, d)) \\ g\ d\ (q, (r, c + 1)) = (q, (r + 1, c)) \end{cases}$$

$\square$

Concluindo:

$$aux\ d = (\![[\langle \underline{0}, \langle \underline{0}, \underline{d}\rangle\rangle, \langle q'\ d, \langle r'\ d, c'\ d\rangle\rangle]]\!)$$

$\equiv \quad \{$ def $(g\ d)$, ficha 3 $\}$

$$aux\ d = (\![[\underline{(0, (0, d))}, g\ d]]\!)$$

$\equiv \quad \{$ def for $\}$

$$aux\ d = \mathsf{for}\ g\ d\ 0, (0, d)$$

$\equiv \quad \{$ def $(loop\ d)$ $\}$

$$aux\ d = loop\ d$$

$\square$

Problema 2

Na resolução deste problema resolvemos definir inicialmente as funções $alice$, a que chamamos a e bob, a que chamamos b. Definimos $alice$ e bob inicialmente em modo pointwise tal que:

$alice :: Ord\ c \Rightarrow LTree\ c \rightarrow c$
$alice\ (Leaf\ x) = id\ x$
$alice\ (Fork\ (t1, t2)) = (\widehat{max} \cdot (bob \times bob))\ (t1, t2)$
$bob :: Ord\ c \Rightarrow LTree\ c \rightarrow c$
$bob\ (Leaf\ x) = id\ x$
$bob\ (Fork\ (t1, t2)) = (\widehat{min} \cdot (alice \times alice))\ (t1, t2)$

Procuramos agora manipular estas definições para que seja possivel aplicar a lei $Fokkinga$.

$$\begin{cases} a \cdot leaf\ x = id\ x \\ a \cdot Fork\ (t1, t2) = \widehat{max}\ (b \times b)\ (t1, t2) \end{cases}$$

$\equiv \quad \{$ (71) $\}$

$$\begin{cases} a \cdot leaf = id \\ a \cdot Fork = \widehat{max}\ (b \times b) \end{cases}$$

$\equiv \quad \{$ (27) $\}$

$$[a \cdot Leaf, a \cdot Fork] = [id, \widehat{max} \cdot (b \times b)]$$

$\equiv \quad \{ \text{ (20) } \}$

$$a \cdot [Leaf, Fork] = [id, \widehat{max} \cdot (b \times b)]$$

$\equiv \quad \{ \text{ (def in) } \}$

$$a \cdot \mathbf{in} = [id, \widehat{max} \cdot (b \times b)]$$

$\equiv \quad \{ \text{ (1),(7) } \}$

$$a \cdot \mathbf{in} = [id \cdot id, \widehat{max} \cdot (\pi_2 \cdot \langle a, b \rangle \times \pi_2 \cdot \langle a, b \rangle)]$$

$\equiv \quad \{ \text{ (14) } \}$

$$a \cdot \mathbf{in} = [id \cdot id, \widehat{max} \cdot (\pi_2 \times \pi_2) \cdot (\langle a, b \rangle \times \langle a, b \rangle)]$$

$\equiv \quad \{ \text{ (22) } \}$

$$a \cdot \mathbf{in} = [id, \widehat{max} \cdot (\pi_2 \times \pi_2)] \cdot (id + (\langle a, b \rangle \times \langle a, b \rangle))$$

$\equiv \quad \{ \text{ (def F(split a b)) } \}$

$$a \cdot \mathbf{in} = [id, (\widehat{max} \cdot (\pi_2 \times \pi_2))] \cdot F \ \langle a, b \rangle$$

$\Box$

O calculo para b é feito de modo inteiramente análogo. Ficamos então com:

$$\begin{cases} a \cdot \mathbf{in} = [id, \widehat{max} \cdot (\pi_2 \times \pi_2)] \cdot F \ \langle a, b \rangle \\ b \cdot \mathbf{in} = [id, \widehat{min} \cdot (\pi_1 \times \pi_1)] \cdot F \ \langle a, b \rangle \end{cases}$$

$\equiv \quad \{ \text{ (52) } \}$

$$\langle a, b \rangle = (\!| \langle [id, \widehat{max} \cdot (\pi_2 \times \pi_2)], [id, \widehat{min} \cdot (\pi_1 \times \pi_1)] \rangle |\!)$$

$\equiv \quad \{ \text{ both = split a b } \}$

$$\langle a, b \rangle = (\!| \langle [id, \widehat{max} \cdot (\pi_2 \times \pi_2)], [id, \widehat{min} \cdot (\pi_1 \times \pi_1)] \rangle |\!)$$

$\Box$

em Haskell:

$$both :: Ord \ d \Rightarrow LTree \ d \rightarrow (d, d)$$
$$both = (\!| \langle [id, \widehat{max} \cdot (\pi_2 \times \pi_2)], [id, \widehat{min} \cdot (\pi_1 \times \pi_1)] \rangle |\!)$$

Problema 3

Resolução do in: Como Nodo é curried e queremos passar para um par com um par e um elemento, temos de fazer uncurry após uncurry, uma LTree3 por definição ou é apenas um Tri ou é um Nodo Tri Tri Tri, logo usamos um either. Resolução do out: Como temos um either, vamos ter que usar as funções i1 e i2, no caso de ser um Tri basta inserir i1 t, no caso de ser um Nodo de 3 Tri , inserimos o par com um par e um elemento, basta inserir os elementos a b c na forma ((a,b),c). Resolução do restante: As recLTree3, cataLTree3, anaLTree3 e hyloLTree3 são funções iguais às funções de outras bibliotecas e o baseLTree3 foi feito de acordo com o tipo, como o bifuntor B(X,Y) = X -–$((Y \times Y) \times Y)$, *o baseLTree3* $f \ g = f$–- ((g ¿¡ g) ¿¡ g).

Biblioteca LTree3:

$$inLTree3 :: a + ((\mathsf{LTree3} \ a, \mathsf{LTree3} \ a), \mathsf{LTree3} \ a) \rightarrow \mathsf{LTree3} \ a$$
$$inLTree3 = [Tri, \widehat{\widehat{Nodo}}]$$
$$outLTree3 :: \mathsf{LTree3} \ a \rightarrow a + ((\mathsf{LTree3} \ a, \mathsf{LTree3} \ a), \mathsf{LTree3} \ a)$$
$$outLTree3 \ (Tri \ t) = i_1 \ t$$
$$outLTree3 \ (Nodo \ a \ b \ c) = i_2 \ ((a, b), c)$$
$$baseLTree3 \ f \ g = f + ((g \times g) \times g)$$
$$recLTree3 \ f = baseLTree3 \ id \ f$$
$$cataLTree3 \ f = f \cdot (recLTree3 \ (cataLTree3 \ f)) \cdot outLTree3$$

$$\mathit{anaLTree3}\ f = \mathit{inLTree3} \cdot (\mathit{recLTree3}\ (\mathit{anaLTree3}\ f)) \cdot f$$
$$\mathit{hyloLTree3}\ f\ g = \mathit{cataLTree3}\ f \cdot \mathit{anaLTree3}\ g$$

Genes do hilomorfismo *sierpinski*:

O g1 é o gene do catamorfismo e o g2 é o gene do anamorfismo. Resolução do g1: A partir de um Tri ou ((Tri*, Tri*), Tri*), queremos obter uma lista de Tri (Tri*). Se for um tri, queremos colocar em uma lista singular, senão temos fazer a concatenação das três listas de Tri. Optamos por escreve a função em haskell pointwise e passar para pointfree.

$$g_1 = [\mathit{singl}, \widehat{(+\!\!+)} \cdot ((\widehat{(+\!\!+)} \times \mathit{id}))]$$
$$g_2\ (t, 0) = i_1\ t$$
$$g_2\ (((x, y), s), n) = i_2\ ((\mathit{t1}, \mathit{t2}), \mathit{t3})\ \textbf{where}$$
$$\quad \mathit{t1} = (((x, y), s\ `\mathit{div}`\ 2), n - 1)$$
$$\quad \mathit{t2} = (((x + s\ `\mathit{div}`\ 2, y), s\ `\mathit{div}`\ 2), n - 1)$$
$$\quad \mathit{t3} = (((x, y + s\ `\mathit{div}`\ 2), s\ `\mathit{div}`\ 2), n - 1)$$

Diagramas do catamorfismo e anamorfismo:

$$\begin{tikzcd}
\mathsf{LTree3}\ \mathit{Tri} \arrow[d, "\mathit{cataLTree3}\ g_1"'] & \mathit{Tri} + ((\mathit{Tri} \times \mathit{Tri}) \times \mathit{Tri}) \arrow[l, "\mathit{inLTree3}"'] \arrow[d, "\mathit{id}+((g_{1\,1})_1)"] \\
{[\mathit{Tri}]} & \mathit{Tri} + (([\mathit{Tri}] \times [\mathit{Tri}]), [\mathit{Tri}]) \arrow[l, "g_1"]
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{Tri} \times \mathit{Int} \arrow[r, "g_2"'] \arrow[d, "\mathit{anaLTree3}\ g_2"'] & \mathit{Tri} + ((\mathit{Tri} \times \mathit{Tri}) \times \mathit{Tri}) \arrow[d, "\mathit{id}+((g_{2\,2})_2)"] \\
\mathsf{LTree3}\ \mathit{Tri} & \mathit{Tri} + (([\mathit{Tri}] \times [\mathit{Tri}]), [\mathit{Tri}]) \arrow[l, "\mathit{inLTree3}"]
\end{tikzcd}$$

Explicação do gene g1: Queremos obter uma lista do tipo Tri a partir de um +, logo o g1 será um either. A primeira parte do either é simples, temos apenas um triângulo, logo basta colocar em uma lista através da função singl. Para a segunda parte do either, queremos as uma lista com todos os elementos das três listas l1, l2 e l3 que estão curried. Portanto, pensamos numa função simples em haskell e passamos para uma pointfree. Passagem para pointfree:

$$g_1\ ((x, y), z) = x +\!\!+ y +\!\!+ z$$

$\equiv$ { passar ++ de infix para prefix duas vezes }

$$g_1\ ((x, y), z) = (+\!\!+)\ ((+\!\!+)\ x\ y)\ z$$

$\equiv$ { (84) duas vezes }

$$g_1\ ((x, y), z) = \widehat{(+\!\!+)}\ (\widehat{(+\!\!+)}\ (x, y), z)$$

$\equiv$ { (73), (77), (72) }

$$g_1\ ((x, y), z) = \widehat{(+\!\!+)} \cdot (\widehat{(+\!\!+)} \times \mathit{id})\ ((x, y), z)$$

$\equiv$ { (71) }

$$g_1 = \widehat{(+\!\!+)} \cdot (\widehat{(+\!\!+)} \times \mathit{id})$$

□

Explicação do gene g2: Como se trata de um anamorfismoLTree3, mas precisar de usar as funções injetoras i1 e i2 por causa do either. O segundo elemento do par é a profundidade, se for 0, queremos que imprima o triângulo atual, logo i1 t, sendo t o primeiro elemento do par, correspondente à sua informação geométrica, ou seja, do tipo Tri. Caso contrário, ainda temos de descer pelo menos mais um nível, e por isso queremos decrementar o segundo elemento e gerar os três triângulos filhos, os seus catetos possuem metade do comprimento do pai, logo no segundo elemento do Tri, utilizamos a divisão inteira para obter metade do tamanho, o que difere entre eles é a sua posição, o primeiro fica na mesma posição que o pai, só com menor comprimento dos catetos. Sendo x o tamanho horizontal e y o tamanho vertical do pai, o segundo estará na posição (x,y+y/2) e o terceiro (x+x/2,y), ressaltando novamente que a operação de divisão é entre inteiros.

Problema 4

A função $propagate$ tem como objetivo aplicar $f :: Monad\ m \Rightarrow (t \rightarrow m\ a)$ a todos os elementos da lista de entrada, analogamente a um `map` . Ao contrário dessa função, esta deve coletar todos os resultados em estrutura monádica dessa aplicação numa só estrutura aplicada a uma lista, Dist $[Bit]$, ao invés de uma lista de estruturas do mesmo tipo de saída de f, [Dist Bit]. Então, para se definir a função $propagate$ é necessário definir o seu comportamento, depois de receber a função f. Renomeie-se os tipos das funções tais que $t \equiv A$ e $a \equiv B$. Assim, para definir corretamente a função $propagate\ f$ é necessário conhecer qual o gene do seu catamorfismo, uma função que obtenha $M\ [B]$ a partir de $1 + A \times M\ [B]$.

Por um lado, a função f obtém $M\ \ B$ a partir do elemento da cabeça da lista A, chegando então a $1 + M\ \ B \times M\ \ [B]$. Por outro lado, pretendemos concatenar o resultado B, retirando-lhe o mónade momentaneamente, ao resultado de aplicar a chamada recursiva à cauda, nomeadamente a função $monad\_cons$ que retira do mónade os elementos do par e devolve a concatenação no mónade, utilizando $return$.

$$
\begin{aligned}
&monad\_cons :: Monad\ m \Rightarrow (m\ a, m\ [a]) \rightarrow m\ [a]\\
&monad\_cons\ (a, b) = \textbf{do}\ \{x \leftarrow a; y \leftarrow b; return\ (x : y)\}
\end{aligned}
$$

Ambas estes passos podem ser compostos na função g_2, definida na solução, que aplica f ao elemento à cabeça antes de o concatenar, por absorção.

$$
\begin{aligned}
& g_2\, f = [return \cdot nil, monad\_cons] \cdot (id + f \times id)\\
\equiv \quad & \{\ (22); (1)\ \}\\
& g_2\, f = [return \cdot nil, monad\_cons \cdot (f \times id)]
\end{aligned}
$$

$\square$

$$
\begin{aligned}
& (monad\_cons \cdot (f \times id))\ (a, b)\\
= \quad & \{\ (72); (77); (1)\ \}\\
& monad\_cons\ (f\ \ a, b)
\end{aligned}
$$

$\square$

No caso do elemento pertencer ao tipo 1, o que corresponde à lista vazia, o gene deve criar uma estrutura monádica com a lista vazia, isto é, aplicar $return$ após nil ao elemento $()$.

```tikzcd
{[A]} \arrow[d, "propagate\ f"'] & 1 + A \times [A] \arrow[l, "inList"'] \arrow[d, "id + id \times (propagate\ f)"] \\
M\ [B] & 1 + A \times M\ [B] \arrow[l, "{[return \cdot nil, g_2\ f]}"] \arrow[d, "id + f \times id"] \\
 & 1 + M\ B \times M\ [B] \arrow[ul, "{[return \cdot nil, g_1]}"]
```

Definição de $propagate$:

$$
\begin{aligned}
&propagate :: Monad\ m \Rightarrow (t \rightarrow m\ a) \rightarrow [t] \rightarrow m\ [a]\\
&propagate\ f = (\!|g\ f|\!)\ \textbf{where}
\end{aligned}
$$

$g\ f = [return \cdot nil, g_2\ f]$
$g_2\ f\ (a, b) = \textbf{do}\ \{x \leftarrow (f\ \ a); y \leftarrow b; return\ (x : y)\}$

Ao contrário de da função $propagate$, para se definir $propagate3$, é necessário explicitar os tipos de $f :: (Monad\ \ m)\ \Rightarrow\ (Bit3\ \rightarrow\ m\ \ Bit3)$, já que se pretende que $propagate3$ triplique, especificamente, cada Bit da lista de entrada para uma estrutura do tipo $Bit3$, durante a sua execução. A sequência de aplicações do gene, que manipulam a cabeça da lista, é a seguinte:

1. Construir o triplo $Bit3$ a partir de Bit, replicando-o três vezes.

2. Aplicar f (por exemplo, $bflip3$) a $Bit3$, produzindo $m\ \ Bit3$.

3. Reduzir $m\ \ Bit3$ a $m\ \ Bit$, utilizando a função $v_3 :: Bit3\ \rightarrow\ Bit$ estendida para se aplicar a $m\ \ Bit3$ com o $functor$ desse mónade, através de fmap.

4. Reconstruir $m\ [Bit]$ de forma análoga a $propagate$, através de $[return \cdot nil, g_1\ f]$.

Tal como anteriormente, todos estes passos podem ser compostos com g_1 numa só função por sucessivas aplicações da regra de $absorção{-}{+}$.

$$\begin{tikzcd}
{[Bit]} \arrow[d, "propagate3\ f"'] & 1 + Bit \times M\ [Bit] \arrow[l, "inList"'] \arrow[d, "id + id \times (propagate3\ f)"] \\
M\ [Bit] & 1 + Bit \times M\ [Bit] \arrow[l, "{[return \cdot nil, g_2\ f]}"] \arrow[d, "id + (a,a,a) \times id"] \\
 & 1 + Bit3 \times M\ [Bit] \arrow[d, "id + bitflip3 \times id"] \\
 & 1 + M\ \ Bit3 \times M\ [Bit] \arrow[d, "id + (\mathsf{fmap}\ (v_3)) \times id"] \\
 & 1 + M\ \ Bit \times M\ [Bit] \arrow[uuul, "{[return \cdot nil, g_1]}"]
\end{tikzcd}$$

Definição de $propagate3$:

$propagate3 :: (Monad\ m) \Rightarrow (Bit3 \rightarrow m\ Bit3) \rightarrow [Bit] \rightarrow m\ [Bit]$
$propagate3\ f = (\!|g\ f|\!)\ \textbf{where}$
$\quad g\ f = [return \cdot nil, g_2\ f]$
$\quad g_2\ f\ (a, b) = \textbf{do}\ \{x \leftarrow ((\mathsf{fmap}\ v_3) \cdot f)\ (a, a, a); y \leftarrow b; return\ (x : y)\}$

A função $bflip3$, a programar a seguir, deverá estender $bflip$ aos três bits da entrada:

$bflip3 :: Bit3 \rightarrow \mathsf{Dist}\ Bit3$
$bflip3\ (a, b, c) = \textbf{do}\ \{x \leftarrow bflip\ \ a; y \leftarrow bflip\ \ b; z \leftarrow bflip\ \ c; return\ (x, y, z)\}$

Índice

Referências

[1] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[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.