Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2020/21

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	77
a93241	Francisco Reis Izquierdo
a93273	José Pedro Martins Magalhães
a89526	Duarte Augusto Rodrigues Lucas
a93185	Carlos Filipe Almeida Dias

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 dois 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.

2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita "literária" [1], 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 `cp2021t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2021t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2021t.zip` e executando:

```
$ lhs2TeX cp2021t.lhs > cp2021t.tex
$ pdflatex cp2021t
```

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
```

Por outro lado, o mesmo ficheiro `cp2021t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2021t.lhs
```

[1] O suffixo 'lhs' quer dizer *literate Haskell*.

Abra o ficheiro `cp2021t.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.

3 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 de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo D 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 cp2021t.aux
$ makeindex cp2021t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck, que ajuda a validar programas em Haskell e a biblioteca Gloss para geração de gráficos 2D:

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

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

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

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

```
> 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 C disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

3.1 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 módulos principal encontra-se na pasta *app*.
- A lista de depêndencias 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*.

Problema 1

Os *tipos de dados algébricos* estudados ao longo desta disciplina oferecem uma grande capacidade expressiva ao programador. Graças à sua flexibilidade, torna-se trivial implementar DSLs e até mesmo linguagens de programação.

Paralelamente, um tópico bastante estudado no âmbito de Deep Learning é a derivação automática de expressões matemáticas, por exemplo, de derivadas. Duas técnicas que podem ser utilizadas para o cálculo de derivadas são:

- *Symbolic differentiation*
- *Automatic differentiation*

Symbolic differentiation consiste na aplicação sucessiva de transformações (leia-se: funções) que sejam congruentes com as regras de derivação. O resultado final será a expressão da derivada.

O leitor atento poderá notar um problema desta técnica: a expressão inicial pode crescer de forma descontrolada, levando a um cálculo pouco eficiente. *Automatic differentiation* tenta resolver este problema, calculando **o valor** da derivada da expressão em todos os passos. Para tal, é necessário calcular o valor da expressão **e** o valor da sua derivada.

Vamos de seguida definir uma linguagem de expressões matemáticas simples e implementar as duas técnicas de derivação automática. Para isso, seja dado o seguinte tipo de dados,

$$
\begin{array}{l}
\mathbf{data}\ \mathit{ExpAr}\ a = X \\
\quad \mid N\ a \\
\quad \mid \mathit{Bin}\ \mathit{BinOp}\ (\mathit{ExpAr}\ a)\ (\mathit{ExpAr}\ a) \\
\quad \mid \mathit{Un}\ \mathit{UnOp}\ (\mathit{ExpAr}\ a) \\
\ \ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

onde BinOp e UnOp representam operações binárias e unárias, respectivamente:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{BinOp} = \mathit{Sum} \\
\quad \mid \mathit{Product} \\
\ \ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\
\mathbf{data}\ \mathit{UnOp} = \mathit{Negate} \\
\quad \mid E \\
\ \ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

O construtor E simboliza o exponencial de base e.

Assim, cada expressão pode ser uma variável, um número, uma operação binária aplicada às devidas expressões, ou uma operação unária aplicada a uma expressão. Por exemplo,

$$\mathit{Bin}\ \mathit{Sum}\ X\ (N\ 10)$$

designa $x + 10$ na notação matemática habitual.

1. A definição das funções in e $\mathit{baseExpAr}$ para este tipo é a seguinte:

$$
\begin{array}{l}
\mathsf{in} = [\underline{X}, \mathit{num\_ops}]\ \mathbf{where} \\
\quad \mathit{num\_ops} = [N, \mathit{ops}] \\
\quad \mathit{ops} = [\mathit{bin}, \widehat{\mathit{Un}}] \\
\quad \mathit{bin}\ (\mathit{op}, (a, b)) = \mathit{Bin}\ \mathit{op}\ a\ b \\
\mathit{baseExpAr}\ f\ g\ h\ j\ k\ l\ z = f + (g + (h \times (j \times k) + l \times z))
\end{array}
$$

Defina as funções $\mathit{outExpAr}$ e $\mathit{recExpAr}$, e teste as propriedades que se seguem.

Propriedade [QuickCheck] 1 *in e $\mathit{outExpAr}$ são testemunhas de um isomorfismo, isto é, $\mathsf{in} \cdot \mathit{outExpAr} = \mathit{id}$ e $\mathit{outExpAr} \cdot \mathit{idExpAr} = \mathit{id}$:*

$$
\begin{array}{l}
\mathit{prop\_in\_out\_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{ExpAr}\ a \rightarrow \mathit{Bool} \\
\mathit{prop\_in\_out\_idExpAr} = \mathsf{in} \cdot \mathit{outExpAr} \equiv \mathit{id} \\
\mathit{prop\_out\_in\_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{OutExpAr}\ a \rightarrow \mathit{Bool} \\
\mathit{prop\_out\_in\_idExpAr} = \mathit{outExpAr} \cdot \mathsf{in} \equiv \mathit{id}
\end{array}
$$

2. Dada uma expressão aritmética e um escalar para substituir o X, a função

$$eval\_exp :: Floating\ a \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$$

calcula o resultado da expressão. Na página 12 esta função está expressa como um catamorfismo. Defina o respectivo gene e, de seguida, teste as propriedades:

Propriedade [QuickCheck] 2 *A função* $eval\_exp$ *respeita os elementos neutros das operações.*

$$\begin{array}{l} prop\_sum\_idr :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\ prop\_sum\_idr\ a\ exp = eval\_exp\ a\ exp \stackrel{?}{=} sum\_idr\ \mathbf{where} \\ \quad sum\_idr = eval\_exp\ a\ (Bin\ Sum\ exp\ (N\ 0)) \\ prop\_sum\_idl :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\ prop\_sum\_idl\ a\ exp = eval\_exp\ a\ exp \stackrel{?}{=} sum\_idl\ \mathbf{where} \\ \quad sum\_idl = eval\_exp\ a\ (Bin\ Sum\ (N\ 0)\ exp) \\ prop\_product\_idr :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\ prop\_product\_idr\ a\ exp = eval\_exp\ a\ exp \stackrel{?}{=} prod\_idr\ \mathbf{where} \\ \quad prod\_idr = eval\_exp\ a\ (Bin\ Product\ exp\ (N\ 1)) \\ prop\_product\_idl :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\ prop\_product\_idl\ a\ exp = eval\_exp\ a\ exp \stackrel{?}{=} prod\_idl\ \mathbf{where} \\ \quad prod\_idl = eval\_exp\ a\ (Bin\ Product\ (N\ 1)\ exp) \\ prop\_e\_id :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow Bool \\ prop\_e\_id\ a = eval\_exp\ a\ (Un\ E\ (N\ 1)) \equiv expd\ 1 \\ prop\_negate\_id :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow Bool \\ prop\_negate\_id\ a = eval\_exp\ a\ (Un\ Negate\ (N\ 0)) \equiv 0 \end{array}$$

Propriedade [QuickCheck] 3 *Negar duas vezes uma expressão tem o mesmo valor que não fazer nada.*

$$\begin{array}{l} prop\_double\_negate :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\ prop\_double\_negate\ a\ exp = eval\_exp\ a\ exp \stackrel{?}{=} eval\_exp\ a\ (Un\ Negate\ (Un\ Negate\ exp)) \end{array}$$

3. É possível otimizar o cálculo do valor de uma expressão aritmética tirando proveito dos elementos absorventes de cada operação. Implemente os genes da função

$$optmize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$$

que se encontra na página 12 expressa como um hilomorfismo[2] e teste as propriedades:

Propriedade [QuickCheck] 4 *A função* $optimize\_eval$ *respeita a semântica da função* $eval$.

$$\begin{array}{l} prop\_optimize\_respects\_semantics :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\ prop\_optimize\_respects\_semantics\ a\ exp = eval\_exp\ a\ exp \stackrel{?}{=} optmize\_eval\ a\ exp \end{array}$$

4. Para calcular a derivada de uma expressão, é necessário aplicar transformações à expressão original que respeitem as regras das derivadas:[3]

   - Regra da soma:

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$$

[2]Qual é a vantagem de implementar a função $optimize\_eval$ utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

[3]Apesar da adição e multiplicação gozarem da propriedade comutativa, há que ter em atenção a ordem das operações por causa dos testes.

- Regra do produto:

$$\frac{d}{dx}(f(x)g(x)) = f(x) \cdot \frac{d}{dx}(g(x)) + \frac{d}{dx}(f(x)) \cdot g(x)$$

Defina o gene do catamorfismo que ocorre na função

$$sd :: Floating\ a \Rightarrow ExpAr\ a \rightarrow ExpAr\ a$$

que, dada uma expressão aritmética, calcula a sua derivada. Testes a fazer, de seguida:

Propriedade [QuickCheck] 5 *A função sd respeita as regras de derivação.*

$$\begin{array}{l}
prop_const_rule :: (Real\ a, Floating\ a) \Rightarrow a \rightarrow Bool \\
prop_const_rule\ a = sd\ (N\ a) \equiv N\ 0 \\
prop_var_rule :: Bool \\
prop_var_rule = sd\ X \equiv N\ 1 \\
prop_sum_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_sum_rule\ exp1\ exp2 = sd\ (Bin\ Sum\ exp1\ exp2) \equiv sum_rule\ \mathbf{where} \\
\quad sum_rule = Bin\ Sum\ (sd\ exp1)\ (sd\ exp2) \\
prop_product_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_product_rule\ exp1\ exp2 = sd\ (Bin\ Product\ exp1\ exp2) \equiv prod_rule\ \mathbf{where} \\
\quad prod_rule = Bin\ Sum\ (Bin\ Product\ exp1\ (sd\ exp2))\ (Bin\ Product\ (sd\ exp1)\ exp2) \\
prop_e_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop_e_rule\ exp = sd\ (Un\ E\ exp) \equiv Bin\ Product\ (Un\ E\ exp)\ (sd\ exp) \\
prop_negate_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop_negate_rule\ exp = sd\ (Un\ Negate\ exp) \equiv Un\ Negate\ (sd\ exp)
\end{array}$$

5. Como foi visto, *Symbolic differentiation* não é a técnica mais eficaz para o cálculo do valor da derivada de uma expressão. *Automatic differentiation* resolve este problema cálculando o valor da derivada em vez de manipular a expressão original.

   Defina o gene do catamorfismo que ocorre na função

   $$ad :: Floating\ a \Rightarrow a \rightarrow ExpAr\ a \rightarrow a$$

   que, dada uma expressão aritmética e um ponto, calcula o valor da sua derivada nesse ponto, sem transformar manipular a expressão original. Testes a fazer, de seguida:

   Propriedade [QuickCheck] 6 *Calcular o valor da derivada num ponto r via ad é equivalente a calcular a derivada da expressão e avalia-la no ponto r.*

   $$\begin{array}{l}
prop_congruent :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_congruent\ a\ exp = ad\ a\ exp \stackrel{?}{=} eval_exp\ a\ (sd\ exp)
\end{array}$$

Problema 2

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[4]

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

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

[4] Lei (3.94) em [2], página 98.

$$f\ 0 = 1$$
$$f\ (n+1) = \mathit{fib}\ n + f\ n$$

Obter-se-á de imediato

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

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.[5]
- 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[6], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

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

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

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

O que se pede então, nesta pergunta? Dada a fórmula que dá o n-ésimo número de Catalan,

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \tag{1}$$

derivar uma implementação de C_n que não calcule factoriais nenhuns. Isto é, derivar um ciclo-for

$$\mathit{cat} = \cdots \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where}\ \cdots$$

que implemente esta função.

Propriedade [QuickCheck] 7 *A função proposta coincidem com a definição dada:*

$$\mathit{prop\_cat} = (\geqslant 0) \Rightarrow (\mathit{catdef} \equiv \mathit{cat})$$

Sugestão: Começar por estudar muito bem o processo de cálculo dado no anexo B para o problema (semelhante) da função exponencial.

Problema 3

As curvas de Bézier, designação dada em honra ao engenheiro Pierre Bézier, são curvas ubíquas na área de computação gráfica, animação e modelação. Uma curva de Bézier é uma curva paramétrica, definida por um conjunto $\{P_0, ..., P_N\}$ de pontos de controlo, onde N é a ordem da curva.

O algoritmo de *De Casteljau* é um método recursivo capaz de calcular curvas de Bézier num ponto. Apesar de ser mais lento do que outras abordagens, este algoritmo é numericamente mais estável, trocando velocidade por correção.

[5] Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.

[6] Secção 3.17 de [2] e tópico Recursividade mútua nos vídeos das aulas teóricas.

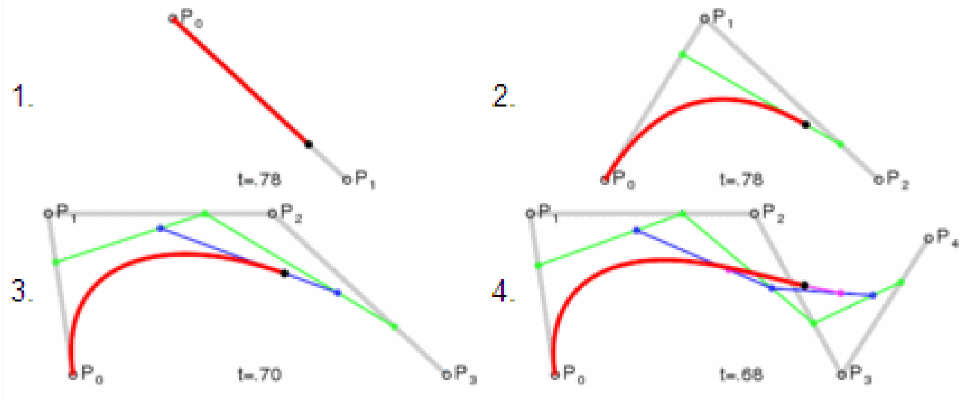

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

De forma sucinta, o valor de uma curva de Bézier de um só ponto $\{P_0\}$ (ordem 0) é o próprio ponto P_0. O valor de uma curva de Bézier de ordem N é calculado através da interpolação linear da curva de Bézier dos primeiros $N-1$ pontos e da curva de Bézier dos últimos $N-1$ pontos.

A interpolação linear entre 2 números, no intervalo $[0,1]$, é dada pela seguinte função:

$$\begin{array}{l} linear1d :: \mathbb{Q} \rightarrow \mathbb{Q} \rightarrow OverTime\ \mathbb{Q} \\ linear1d\ a\ b = formula\ a\ b\ \mathbf{where} \\ \quad formula :: \mathbb{Q} \rightarrow \mathbb{Q} \rightarrow Float \rightarrow \mathbb{Q} \\ \quad formula\ x\ y\ t = ((1.0 :: \mathbb{Q}) - (to_{\mathbb{Q}}\ t)) * x + (to_{\mathbb{Q}}\ t) * y \end{array}$$

A interpolação linear entre 2 pontos de dimensão N é calculada através da interpolação linear de cada dimensão.

O tipo de dados $NPoint$ representa um ponto com N dimensões.

$$\mathbf{type}\ NPoint = [\mathbb{Q}]$$

Por exemplo, um ponto de 2 dimensões e um ponto de 3 dimensões podem ser representados, respetivamente, por:

$$\begin{array}{l} p2d = [1.2, 3.4] \\ p3d = [0.2, 10.3, 2.4] \end{array}$$

O tipo de dados $OverTime\ a$ representa um termo do tipo a num dado instante (dado por um $Float$).

$$\mathbf{type}\ OverTime\ a = Float \rightarrow a$$

O anexo C tem definida a função

$$calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)$$

que calcula a interpolação linear entre 2 pontos, e a função

$$deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint$$

que implementa o algoritmo respectivo.

1. Implemente $calcLine$ como um catamorfismo de listas, testando a sua definição com a propriedade:

   Propriedade [QuickCheck] 8 *Definição alternativa.*

   $$\begin{array}{l} prop\_calcLine\_def :: NPoint \rightarrow NPoint \rightarrow Float \rightarrow Bool \\ prop\_calcLine\_def\ p\ q\ d = calcLine\ p\ q\ d \equiv zipWithM\ linear1d\ p\ q\ d \end{array}$$

2. Implemente a função $deCasteljau$ como um hilomorfismo, testando agora a propriedade:

Propriedade [QuickCheck] 9 *Curvas de Bézier são simétricas.*

$$
\begin{array}{l}
prop\_bezier\_sym :: [[\mathbb{Q}]] \rightarrow Gen\ Bool \\
prop\_bezier\_sym\ l = all\ (<\Delta) \cdot calc\_difs \cdot bezs\ \langle\$\rangle\ elements\ ps\ \mathbf{where} \\
\quad calc\_difs = (\lambda(x, y) \rightarrow zipWith\ (\lambda w\ v \rightarrow \mathbf{if}\ w \geqslant v\ \mathbf{then}\ w - v\ \mathbf{else}\ v - w)\ x\ y) \\
\quad bezs\ t = (deCasteljau\ l\ t, deCasteljau\ (reverse\ l)\ (from_{\mathbb{Q}}\ (1 - (to_{\mathbb{Q}}\ t)))) \\
\quad \Delta = 1\mathrm{e}{-2}
\end{array}
$$

3. Corra a função $runBezier$ e aprecie o seu trabalho[7] clicando na janela que é aberta (que contém, a verde, um ponto inicila) com o botão esquerdo do rato para adicionar mais pontos. A tecla $Delete$ apaga o ponto mais recente.

Problema 4

Seja dada a fórmula que calcula a média de uma lista não vazia x,

$$avg\ x = \frac{1}{k} \sum_{i=1}^{k} x_i \tag{2}$$

onde $k = length\ x$. Isto é, para sabermos a média de uma lista precisamos de dois catamorfismos: o que faz o somatório e o que calcula o comprimento a lista. Contudo, é facil de ver que

$$
\begin{array}{l}
avg\ [a] = a \\
avg(a : x) = \frac{1}{k+1}(a + \sum_{i=1}^{k} x_i) = \frac{a+k(avg\ x)}{k+1}\ \text{para}\ k = length\ x
\end{array}
$$

Logo avg está em recursividade mútua com $length$ e o par de funções pode ser expresso por um único catamorfismo, significando que a lista apenas é percorrida uma vez.

1. Recorra à lei de recursividade mútua para derivar a função $avg\_aux = (\![[b, q]]\!)$ tal que $avg\_aux = \langle avg, length \rangle$ em listas não vazias.
2. Generalize o raciocínio anterior para o cálculo da média de todos os elementos de uma LTree recorrendo a uma única travessia da árvore (i.e. catamorfismo).

Verifique as suas funções testando a propriedade seguinte:

Propriedade [QuickCheck] 10 *A média de uma lista não vazia e de uma LTree com os mesmos elementos coincide, a menos de um erro de 0.1 milésimas:*

$$
\begin{array}{l}
prop\_avg :: [Double] \rightarrow Property \\
prop\_avg = nonempty \Rightarrow diff \leqslant \underline{0.000001}\ \mathbf{where} \\
\quad diff\ l = avg\ l - (avgLTree \cdot genLTree)\ l \\
\quad genLTree = [\![(lsplit)]\!] \\
\quad nonempty = (>[\,])
\end{array}
$$

Problema 5

(**NB**: Esta questão é **opcional** e funciona como **valorização** apenas para os alunos que desejarem fazê-la.)

Existem muitas linguagens funcionais para além do Haskell, que é a linguagem usada neste trabalho prático. Uma delas é o F# da Microsoft. Na directoria `fsharp` encontram-se os módulos Cp, Nat e LTree codificados em F#. O que se pede é a biblioteca BTree escrita na mesma linguagem.

Modo de execução: o código que tiverem produzido nesta pergunta deve ser colocado entre o `\begin{verbatim}` e o `\end{verbatim}` da correspondente parte do anexo D. Para além disso, os grupos podem demonstrar o código na oral.

[7] A representação em Gloss é uma adaptação de um projeto de Harold Cooper.

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:[8]

$$
\begin{aligned}
& \quad id = \langle f, g \rangle \\
\equiv & \qquad \{ \text{ universal property } \} \\
& \quad \begin{cases} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{cases} \\
\equiv & \qquad \{ \text{ identity } \} \\
& \quad \begin{cases} \pi_1 = f \\ \pi_2 = g \end{cases} \\
\Box &
\end{aligned}
$$

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 Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina[9], onde se pretende implementar um ciclo que implemente o cálculo da aproximação até $i = n$ da função exponencial $exp\ x = e^x$, via série de Taylor:

$$
exp\ x \quad = \quad \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{3}
$$

Seja $e\ x\ n = \sum_{i=0}^{n} \frac{x^i}{i!}$ a função que dá essa aproximação. É fácil de ver que $e\ x\ 0 = 1$ e que $e\ x\ (n+1) = e\ x\ n + \frac{x^{n+1}}{(n+1)!}$. Se definirmos $h\ x\ n = \frac{x^{n+1}}{(n+1)!}$ teremos $e\ x$ e $h\ x$ em recursividade mútua. Se repetirmos o processo para $h\ x\ n$ etc obteremos no total três funções nessa mesma situação:

$$
\begin{aligned}
& e\ x\ 0 = 1 \\
& e\ x\ (n+1) = h\ x\ n + e\ x\ n \\
& h\ x\ 0 = x \\
& h\ x\ (n+1) = x\ /\ (s\ n) * h\ x\ n \\
& s\ 0 = 2 \\
& s\ (n+1) = 1 + s\ n
\end{aligned}
$$

Segundo a *regra de algibeira* descrita na página 3.1 deste enunciado, ter-se-á, de imediato:

$$
\begin{aligned}
& e'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\
& \quad init = (1, x, 2) \\
& \quad loop\ (e, h, s) = (h + e, x\ /\ s * h, 1 + s) \\
& \quad prj\ (e, h, s) = e
\end{aligned}
$$

[8] Exemplos tirados de [2].

[9] Cf. [2], página 102.

C Código fornecido

Problema 1

$expd :: Floating\ a \Rightarrow a \rightarrow a$
$expd = Prelude.exp$
$\mathbf{type}\ OutExpAr\ a = () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))$

Problema 2

Definição da série de Catalan usando factoriais (1):

$$catdef\ n = (2 * n)! \div ((n + 1)! * n!)$$

Oráculo para inspecção dos primeiros 26 números de Catalan[10]:

$oracle = [$
$\quad 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,$
$\quad 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020,$
$\quad 91482563640, 343059613650, 1289904147324, 4861946401452$
$\quad]$

Problema 3

Algoritmo:

$deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint$
$deCasteljau\ [\,] = nil$
$deCasteljau\ [p] = \underline{p}$
$deCasteljau\ l = \lambda pt \rightarrow (calcLine\ (p\ pt)\ (q\ pt))\ pt\ \mathbf{where}$
$\quad p = deCasteljau\ (init\ l)$
$\quad q = deCasteljau\ (tail\ l)$

Função auxiliar:

$calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)$
$calcLine\ [\,] = \underline{nil}$
$calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\ \mathbf{where}$
$\quad g :: (\mathbb{Q}, NPoint \rightarrow OverTime\ NPoint) \rightarrow (NPoint \rightarrow OverTime\ NPoint)$
$\quad g\ (d, f)\ l = \mathbf{case}\ l\ \mathbf{of}$
$\qquad [\,] \rightarrow nil$
$\qquad (x : xs) \rightarrow \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z$

2D:

$bezier2d :: [NPoint] \rightarrow OverTime\ (Float, Float)$
$bezier2d\ [\,] = \underline{(0, 0)}$
$bezier2d\ l = \lambda z \rightarrow (from_{\mathbb{Q}} \times from_{\mathbb{Q}}) \cdot (\lambda[x, y] \rightarrow (x, y))\ \$\ ((deCasteljau\ l)\ z)$

Modelo:

$\mathbf{data}\ World = World\ \{points :: [NPoint]$
$\quad , time :: Float$
$\quad \}$
$initW :: World$
$initW = World\ [\,]\ 0$

[10]Fonte: Wikipedia.

$$
\begin{aligned}
&\mathit{tick} :: \mathit{Float} \to \mathit{World} \to \mathit{World}\\
&\mathit{tick}\ \mathit{dt}\ \mathit{world} = \mathit{world}\ \{\mathit{time} = (\mathit{time}\ \mathit{world}) + \mathit{dt}\}\\
&\mathit{actions} :: \mathit{Event} \to \mathit{World} \to \mathit{World}\\
&\mathit{actions}\ (\mathit{EventKey}\ (\mathit{MouseButton}\ \mathit{LeftButton})\ \mathit{Down}\ \_\ p)\ \mathit{world} =\\
&\quad \mathit{world}\ \{\mathit{points} = (\mathit{points}\ \mathit{world}) +\!\!+ [(\lambda(x,y) \to \mathsf{map}\ \ \mathit{to}_{\mathbb{Q}}\ [x,y])\ p]\}\\
&\mathit{actions}\ (\mathit{EventKey}\ (\mathit{SpecialKey}\ \mathit{KeyDelete})\ \mathit{Down}\ \_\ \_)\ \mathit{world} =\\
&\quad \mathit{world}\ \{\mathit{points} = \mathit{cond}\ (\equiv [\,])\ \mathit{id}\ \mathit{init}\ (\mathit{points}\ \mathit{world})\}\\
&\mathit{actions}\ \_\ \mathit{world} = \mathit{world}\\
&\mathit{scaleTime} :: \mathit{World} \to \mathit{Float}\\
&\mathit{scaleTime}\ w = (1 + \mathit{cos}\ (\mathit{time}\ w))\ /\ 2\\
&\mathit{bezier2dAtTime} :: \mathit{World} \to (\mathit{Float}, \mathit{Float})\\
&\mathit{bezier2dAtTime}\ w = (\mathit{bezier2dAt}\ w)\ (\mathit{scaleTime}\ w)\\
&\mathit{bezier2dAt} :: \mathit{World} \to \mathit{OverTime}\ (\mathit{Float}, \mathit{Float})\\
&\mathit{bezier2dAt}\ w = \mathit{bezier2d}\ (\mathit{points}\ w)\\
&\mathit{thicCirc} :: \mathit{Picture}\\
&\mathit{thicCirc} = \mathit{ThickCircle}\ 4\ 10\\
&\mathit{ps} :: [\mathit{Float}]\\
&\mathit{ps} = \mathsf{map}\ \ \mathit{from}_{\mathbb{Q}}\ \mathit{ps}'\ \mathbf{where}\\
&\quad \mathit{ps}' :: [\mathbb{Q}]\\
&\quad \mathit{ps}' = [0, 0.01\,..\,1] \quad \text{-- interval}
\end{aligned}
$$

Gloss:

$$
\begin{aligned}
&\mathit{picture} :: \mathit{World} \to \mathit{Picture}\\
&\mathit{picture}\ \mathit{world} = \mathit{Pictures}\\
&\quad [\mathit{animateBezier}\ (\mathit{scaleTime}\ \mathit{world})\ (\mathit{points}\ \mathit{world})\\
&\quad , \mathit{Color}\ \mathit{white} \cdot \mathit{Line} \cdot \mathsf{map}\ \ (\mathit{bezier2dAt}\ \mathit{world})\ \$\ \mathit{ps}\\
&\quad , \mathit{Color}\ \mathit{blue} \cdot \mathit{Pictures}\ \$\ [\mathit{Translate}\ (\mathit{from}_{\mathbb{Q}}\ x)\ (\mathit{from}_{\mathbb{Q}}\ y)\ \mathit{thicCirc} \mid [x,y] \leftarrow \mathit{points}\ \mathit{world}]\\
&\quad , \mathit{Color}\ \mathit{green}\ \$\ \mathit{Translate}\ \mathit{cx}\ \mathit{cy}\ \mathit{thicCirc}\\
&\quad]\ \mathbf{where}\\
&\quad (\mathit{cx}, \mathit{cy}) = \mathit{bezier2dAtTime}\ \mathit{world}
\end{aligned}
$$

Animação:

$$
\begin{aligned}
&\mathit{animateBezier} :: \mathit{Float} \to [\mathit{NPoint}] \to \mathit{Picture}\\
&\mathit{animateBezier}\ \_\ [\,] = \mathit{Blank}\\
&\mathit{animateBezier}\ \_\ [\_] = \mathit{Blank}\\
&\mathit{animateBezier}\ t\ l = \mathit{Pictures}\\
&\quad [\mathit{animateBezier}\ t\ (\mathit{init}\ l)\\
&\quad , \mathit{animateBezier}\ t\ (\mathit{tail}\ l)\\
&\quad , \mathit{Color}\ \mathit{red} \cdot \mathit{Line}\ \$\ [a, b]\\
&\quad , \mathit{Color}\ \mathit{orange}\ \$\ \mathit{Translate}\ \mathit{ax}\ \mathit{ay}\ \mathit{thicCirc}\\
&\quad , \mathit{Color}\ \mathit{orange}\ \$\ \mathit{Translate}\ \mathit{bx}\ \mathit{by}\ \mathit{thicCirc}\\
&\quad]\ \mathbf{where}\\
&\quad a@(\mathit{ax}, \mathit{ay}) = \mathit{bezier2d}\ (\mathit{init}\ l)\ t\\
&\quad b@(\mathit{bx}, \mathit{by}) = \mathit{bezier2d}\ (\mathit{tail}\ l)\ t
\end{aligned}
$$

Propriedades e *main*:

$$
\begin{aligned}
&\mathit{runBezier} :: \mathsf{IO}\ ()\\
&\mathit{runBezier} = \mathit{play}\ (\mathit{InWindow}\ \texttt{"Bézier"}\ (600, 600)\ (0, 0))\\
&\quad \mathit{black}\ 50\ \mathit{initW}\ \mathit{picture}\ \mathit{actions}\ \mathit{tick}\\
&\mathit{runBezierSym} :: \mathsf{IO}\ ()\\
&\mathit{runBezierSym} = \mathit{quickCheckWith}\ (\mathit{stdArgs}\ \{\mathit{maxSize} = 20, \mathit{maxSuccess} = 200\})\ \mathit{prop\_bezier\_sym}
\end{aligned}
$$

Compilação e execução dentro do interpretador:[11]

$$
\begin{aligned}
&\mathit{main} = \mathit{runBezier}\\
&\mathit{run} = \mathbf{do}\ \{\mathit{system}\ \texttt{"ghc cp2021t"}; \mathit{system}\ \texttt{"./cp2021t"}\}
\end{aligned}
$$

[11]Pode ser útil em testes envolvendo Gloss. Nesse caso, o teste em causa deve fazer parte de uma função *main*.

QuickCheck

Código para geração de testes:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{UnOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Negate}, E] \\
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{BinOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Sum}, \mathit{Product}] \\
\textbf{instance}\ (\mathit{Arbitrary}\ a) \Rightarrow \mathit{Arbitrary}\ (\mathit{ExpAr}\ a)\ \textbf{where} \\
\quad \mathit{arbitrary} = \textbf{do} \\
\qquad \mathit{binop} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{unop} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{exp1} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{exp2} \leftarrow \mathit{arbitrary} \\
\qquad a \quad\ \leftarrow \mathit{arbitrary} \\
\qquad \mathit{frequency} \cdot \mathsf{map}\ \ (\mathit{id} \times \mathit{pure})\ \$\ [(20, X), (15, N\ a), (35, \mathit{Bin}\ \mathit{binop}\ \mathit{exp1}\ \mathit{exp2}), (30, \mathit{Un}\ \mathit{unop}\ \mathit{exp1})]
\end{array}
$$

$$
\begin{array}{l}
\textbf{infixr}\ 5 \stackrel{?}{=} \\
(\stackrel{?}{=}) :: \mathit{Real}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\
(\stackrel{?}{=})\ x\ y = (\mathit{to}_{\mathbb{Q}}\ x) \equiv (\mathit{to}_{\mathbb{Q}}\ y)
\end{array}
$$

Outras funções auxiliares

Lógicas:

$$
\begin{array}{l}
\textbf{infixr}\ 0 \Rightarrow \\
(\Rightarrow) :: (\mathit{Testable}\ \mathit{prop}) \Rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{prop}) \rightarrow a \rightarrow \mathit{Property} \\
p \Rightarrow f = \lambda a \rightarrow p\ a \Rightarrow f\ a \\
\textbf{infixr}\ 0 \Leftrightarrow \\
(\Leftrightarrow) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow a \rightarrow \mathit{Property} \\
p \Leftrightarrow f = \lambda a \rightarrow (p\ a \Rightarrow \mathit{property}\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow \mathit{property}\ (p\ a)) \\
\textbf{infixr}\ 4 \equiv \\
(\equiv) :: \mathit{Eq}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \equiv g = \lambda a \rightarrow f\ a \equiv g\ a \\
\textbf{infixr}\ 4 \leqslant \\
(\leqslant) :: \mathit{Ord}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \leqslant g = \lambda a \rightarrow f\ a \leqslant g\ a \\
\textbf{infixr}\ 4 \wedge \\
(\wedge) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \wedge g = \lambda a \rightarrow ((f\ a) \wedge (g\ a))
\end{array}
$$

D 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, disgramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

São dadas:

$$
\begin{array}{l}
(\!|g|\!) = g \cdot \mathit{recExpAr}\ (\!|g|\!) \cdot \mathit{outExpAr} \\
\mathit{anaExpAr}\ g = \mathsf{in} \cdot \mathit{recExpAr}\ (\mathit{anaExpAr}\ g) \cdot g \\
\mathit{hyloExpAr}\ h\ g = (\!|h|\!) \cdot \mathit{anaExpAr}\ g
\end{array}
$$

$eval\_exp :: Floating\ a \Rightarrow a \to (ExpAr\ a) \to a$
$eval\_exp\ a = (\!| g\_eval\_exp\ a |\!)$
$optmize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \to (ExpAr\ a) \to a$
$optmize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean$
$sd :: Floating\ a \Rightarrow ExpAr\ a \to ExpAr\ a$
$sd = \pi_2 \cdot (\!| sd\_gen |\!)$
$ad :: Floating\ a \Rightarrow a \to ExpAr\ a \to a$
$ad\ v = \pi_2 \cdot (\!| ad\_gen\ v |\!)$

D.1 Solução:

Uma vez que estamos a trabalhar com um tipo indutivo novo iremos representar o diagrama genérico de um catamorfismo que atua sobre o tipo indutivo ExprAr. Além disso, iremos representar o bifunctor de base bem como a função out associada a este tipo indutivo.

Pela análise da função $baseExprAr$ conseguimos perceber de um modo geral como o bifunctor de base atua. Assim temos o seguinte diagrama:

$$\begin{tikzcd}
ExpAr\ A \arrow[d, "(\!| f |\!)"'] & 1 + A + (BinOp \times (ExpAr\ A \times ExpAr\ A)) + (UnOp \times ExpAr\ A) \arrow[l, "\mathsf{in}"'] \arrow[d, "id + id + (id \times (\!| f |\!) + (\!| f |\!)) + (id \times (\!| f |\!))"] \\
A & 1 + A + (BinOp \times (A \times A)) + (UnOp \times A) \arrow[l, "gene"]
\end{tikzcd}$$

Como em qualquer catamorfismo, está presente o isomorfismo in- out e pelas leis de Cálculo de Programas, conseguimos obter a definição da função outExpAr.

NB: Por efeitos de simplificação, iremos referir a função $outExpAr$ como apenas out.

Temos:

$$
\begin{array}{ll}
& out \cdot \mathsf{in} = id \\
\equiv & \quad \{ \text{ Definição de in } \} \\
& out \cdot [\underline{X}, num\_ops] = id \\
\equiv & \quad \{ \text{ Definição de } num\_ops \ \} \\
& out \cdot [\underline{X}, [N, ops]] \\
\equiv & \quad \{ \text{ Definição de } ops \ \} \\
& out \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] \\
\equiv & \quad \{ \text{ Fusão + (20) } \} \\
& [out \cdot \underline{X}, [out \cdot N, [out \cdot bin, out \cdot \widehat{Un}]]] = id \\
\equiv & \quad \{ \text{ Universal + (17), Natural id (1) } \} \\
& \left\{ \begin{array}{l} out \cdot \underline{X} = i_1\ () \\ [out \cdot N, [out \cdot bin, out \cdot \widehat{Un}]] = i_2\ () \end{array} \right. \\
\equiv & \quad \{ \text{ Universal + (17) } \} \\
& \left\{ \begin{array}{l} out \cdot \underline{X} = i_1\ () \\ \left\{ \begin{array}{l} out \cdot N = i_2\ () \cdot i_1\ () \\ [out \cdot bin, out \cdot \widehat{Un} = i_2\ () \cdot i_2\ ()] \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \text{ Universal + (17) } \} \\
& \left\{ \begin{array}{l} out \cdot \underline{X} = i_1\ () \\ \left\{ \begin{array}{l} out \cdot N = i_2\ () \cdot i_1\ () \\ \left\{ \begin{array}{l} out \cdot bin = i_2\ () \cdot i_2\ () \cdot i_1\ () \\ out \cdot \widehat{Un} = i_2\ () \cdot i_2\ () \cdot i_2\ () \end{array} \right. \end{array} \right. \end{array} \right.
\end{array}
$$

$$\equiv \qquad \{ \text{ Ig Existencial (71), Def de Composição (72), Def de Const (74), Def de Uncurry (84), Def de } Un\text{, Def } Bin \}$$

$$\begin{cases} out\ (X) = i_1\ () \\ \begin{cases} out\ (N\ x) = i_2\ (i_1\ (x)) \\ \begin{cases} out\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b)))) \\ out\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a))) \end{cases} \end{cases} \end{cases}$$

$\square$

$$outExpAr\ (X) = i_1\ ()$$
$$outExpAr\ (N\ x) = i_2\ (i_1\ x)$$
$$outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b))))$$
$$outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a)))$$

D.2 Solução:

Pela análise do diagrama, percebemos como a recursividade, isto é, como um catamorfismo associado a este tipo indutivo ”consome”a estrutura de dados. Temos também o functor do tipo indutivo dado pela função $baseExpAr$.

Assim conseguimos chegar de forma indutiva à definição da função recExpAr:

$$recExpAr\ f = baseExpAr\ id\ id\ id\ f\ f\ id\ f$$

D.3 Solução:

Dada a situação em que nos é dado um escalar e uma expressão, ao termos de calcular o valor da mesma, substituindo o valor do escalar de forma apropriada à expressão, conseguimos mais uma vez através da análise do diagrama perceber como a recursividade está explícita neste caso. Primeiramente iremos definir o diagrama associado ao catamorfismo $eval\_exp$:

$$\begin{tikzcd} ExpAr\ A \arrow[d, "eval\_exp"'] & 1 + A + (BinOp \times (ExpAr\ A \times ExpAr\ A)) + (UnOp \times ExpAr\ A) \arrow[l, "\mathsf{in}"'] \arrow[d, "id + id + (id \times (eval\_exp) + (eval\_exp)) + (id \times (eval\_exp))"] \\ A & 1 + A + (BinOp \times (A \times A)) + (UnOp \times A) \arrow[l, "g\_eval\_exp"] \end{tikzcd}$$

É de salientar que o ponto fulcral do problema é induzir o gene do catamorfismo $eval\_exp$, ou seja ”descobrir”como definir a função g_eval_exp. Ora consoante o tipo de $ExpAr$ em causa e um escalar, de forma a calcular o valor da expressão para esse escalar, temos uma das seguintes possibilidades ou até mesmo várias das seguintes possibilidades:

- Caso 1: Uma expressão ser uma incógnita x e dado um escalar, o valor da expressão é o próprio escalar;

$$g\_eval\_exp\ escalar\ (i_1\ ()) = escalar$$

- Caso 2: Uma expressão ser um escalar $valor$ e dado um escalar, o valor da expressão é o próprio $valor$;

$$g\_eval\_exp\ escalar\ (i_2\ (i_1\ valor)) = valor$$

- Caso 3: Uma expressão ser uma soma/produto entre dois $ExpAr$ e dado um escalar, o valor da expressão é somar/multiplicar os dois $ExpAr$, substituindo as incógnitas pelo escalar;

NB: É de salientar que ambos os $ExpAr$ supramencionados foram já processados pelo catamorfismo e as incógnitas substituidas pelo valor do escalar, na recursividade quando esta chega aos casos de base (caso 1 e caso 2).

$$g\_eval\_exp\ escalar\ (i_2\ (i_2\ (i_1\ (Sum, (a, b))))) = a + b$$
$$g\_eval\_exp\ escalar\ (i_2\ (i_2\ (i_1\ (Product, (a, b))))) = a * b$$

- Caso 4: Uma expressão ser uma negação de um $ExpAr$ e dado um escalar, o valor da expressão é negar o $ExpAr$, substituindo as incógnitas pelo escalar;

NB: É de salientar que o $ExpAr$ supramencionado foi já processado pelo catamorfismo e as incógnitas substituidas pelo valor do escalar, na recursividade quando esta chega aos casos de base (caso 1 e caso 2).

$$g\_eval\_exp\ escalar\ (i_2\ (i_2\ (i_2\ (Negate, a)))) = (-1) * a$$

- Caso 5: Uma expressão ser uma base de e cujo expoente é um $ExpAr$ e dado um escalar, o valor da expressão é elevar a base e ao expoente $ExpAr$, substituindo as incógnitas pelo escalar;

NB: É de salientar que o $ExpAr$ supramencionado foi já processado pelo catamorfismo e as incógnitas substituidas pelo valor do escalar, na recursividade quando esta chega aos casos de base (caso 1 e caso 2).

$$g\_eval\_exp\ escalar\ (i_2\ (i_2\ (i_2\ (E, a)))) = Prelude.exp\ a$$

D.4 Solução:

De forma a tirar proveito das propriedades dos elementos absorventes e neutros das operações matemáticas impostas no tipo indutivo em causa, teremos de analisar os vários casos em que conseguimos "limpar"uma expressão. Além disso, a maneira que iremos trabalhar com estes casos é a mesma para a função $outExpAr$ associada a este tipo indutivo, uma vez que iremos apenas receber uma $ExpAr$ e a iremos "limpar". Assim, consoante o tipo $ExpAr$ em causa, de forma a tirar proveito das propriedades dos elementos neutro e absorventes temos uma das seguintes possibilidades ou até mesmo várias das seguintes possibilidades:

- Caso 1: Uma expressão ser um produto de uma $ExpAr$ com 0 é o mesmo que apenas ter 0, tirando proveito da propriedade do elemento absorvente da multiplicação;

$$clean\ (Bin\ Product\ a\ (N\ 0)) = outExpAr\ (N\ 0)$$
$$clean\ (Bin\ Product\ (N\ 0)\ a) = outExpAr\ (N\ 0)$$

- Caso 2: Uma expressão ser a base de e cujo expoente é 0 é o mesmo que apenas ter 1, tirando proveito da propriedade do elemento absorvente da exponeciação;

$$clean\ (Un\ E\ (N\ 0)) = outExpAr\ (N\ 1)$$

- Caso 3: Uma expressão ser a negação de 0 é o mesmo que apenas ter 0, tirando proveito da propriedade do elemento neutro da negação;

$$clean\ (Un\ Negate\ (N\ 0)) = outExpAr\ (N\ 0)$$

- Caso 4: Uma expressão que à partida não tira proveito de nenhuma das propriedades supramencionadas, terá de ser analisada nas suas partes, sendo esta análise já efetuada aquando da recursividade;

Caso 5: Uma expressão que à partida não tira proveito de nenhuma das propriedades supramencionadas, terá de ser analisada nas suas partes, sendo esta análise já efetuada aquando da recursividade;

$$clean\ x = outExpAr\ x$$

A função gopt "consome", isto é, calcula apenas a expressão, fazendo uso da função g_eval_exp acima definida.

$$gopt\ exp = g\_eval\_exp\ exp$$

Ora é de ressalvar, que pela a análise da definição do hilomorfismo associado a este tipo indutivo, $hyloExpAr$, vemos que a função que "constroi"a estrutura de dados, que desempenha o papel de anamorfismo, é a função $clean$ e a função que consome esta estrutura intermédia criada pela função $clean$ é a função $gopt$ que desempenha o papel de catamorfismo.

D.5 Solução:

Uma vez que queremos calcular a derivada de uma expressão, teremos de ter em conta os vários casos possíveis e que se adequam ao tipo indutivo em causa e que estão presentes na matemática que aprendemos no ensino básico. Além disso, pelas regras que nos são apresentadas como ponto de partida, conseguimos perceber que iremos lidar com um catamorfismo, que terá casos de base e casos recursivos, sendo que estes últimos já serão processados pelo próprio catamorfismo sd. Primeiramente iremos analisar a tipagem da função sd_gen que irá tratar do cálculo da derivada de uma expressão do tipo $ExpAr$.

$$\begin{array}{l} sd\_gen :: Floating\ a \Rightarrow \\ \quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow (ExpAr\ a \end{array}$$

Ao analisar a tipagem da função sd_gen ficamos com uma dúvida: "O que são os pares de $ExpAr$ nos operadores Bin e Un?".

Sabemos que estamos perante um gene de catamorfismo e que teremos casos em que, dado o operador em causa, temos argumentos por processar, isto é, por calcular a sua derivada. Pela tipagem percebemos que o catamorfismo em causa pede os tais dois pares e pela a análise das regras de derivação, percebemos que a primeira componente de cada par é a expressão que ainda não foi derivada e a segunda componente é a expressão que já foi derivada. Assim, consoante o tipo $ExpAr$ em causa, seguindo as regras de derivação temos uma das seguintes possibilidades ou até mesmo várias das seguintes possibilidades:

- Caso 1: Regra da derivada de uma incógnita:

$$\frac{d}{dx}(x) = 1$$

$$sd\_gen\ (i_1\ ()) = (X, N\ 1)$$

- Caso 2: Regra da derivada de uma constante:

$$\frac{d}{dx}(n) = 0$$

$$sd\_gen\ (i_2\ (i_1\ a)) = (N\ a, N\ 0)$$

- Caso 3: Regra da derivada de uma soma:

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$$

$$sd\_gen\ (i_2\ (i_2\ (i_1\ (Sum, ((a, b), (c, d)))))) = (Bin\ Sum\ a\ c, Bin\ Sum\ b\ d)$$

- Caso 4: Regra da derivada de uma produto:

$$\frac{d}{dx}(f(x)g(x)) = f(x) \cdot \frac{d}{dx}(g(x)) + \frac{d}{dx}(f(x)) \cdot g(x)$$

$$sd\_gen\ (i_2\ (i_2\ (i_1\ (Product, ((a, b), (c, d)))))) = (Bin\ Product\ a\ c, (Bin\ Sum\ (Bin\ Product\ a\ d)\ (Bin\ Product\ b\ c$$

- Caso 5: Regra da derivada de uma negação:

$$sd\_gen\ (i_2\ (i_2\ (i_2\ (Negate, (a, b))))) = (Un\ Negate\ a, Un\ Negate\ b)$$

- Caso 6: Regra da derivada de uma expressão cuja base é e:

$$\frac{d}{dx}e^a = e^a \cdot \frac{d}{dx}(a)$$

$$sd\_gen\ (i_2\ (i_2\ (i_2\ (E, (a, b))))) = (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ b)$$

D.6 Solução:

Sabemos que estamos perante um gene de catamorfismo e que teremos casos em que, dado o operador em causa, temos argumentos por processar, isto é, por calcular a sua derivada. Além disso, agora é nos dados um escalar de forma a calcularmos a derivada no ponto (escalar) dado, sem manipular ou transformar a expressão em causa. Assim, consoante o tipo $ExpAr$ em causa, seguindo as regras de derivação temos uma das seguintes possibilidades ou até mesmo várias das seguintes possibilidades:

- Caso 1: Regra da derivada de uma incógnita:

$$\frac{d}{dx}(x) = 1$$

$$ad\_gen\ x\ (i_1\ ()) = (x, 1)$$

- Caso 2: Regra da derivada de uma constante:

$$\frac{d}{dx}(n) = 0$$

$$ad\_gen\ x\ (i_2\ (i_1\ a)) = (a, 0)$$

- Caso 3: Regra da derivada de uma soma:

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$$

$$ad\_gen\ x\ (i_2\ (i_2\ (i_1\ (Sum, ((a, b), (c, d)))))) = (a + c, b + d)$$

- Caso 4: Regra da derivada de uma produto:

$$\frac{d}{dx}(f(x)g(x)) = f(x) \cdot \frac{d}{dx}(g(x)) + \frac{d}{dx}(f(x)) \cdot g(x)$$

$$ad\_gen\ x\ (i_2\ (i_2\ (i_1\ (Product, ((a, b), (c, d)))))) = (a * c, (a * d) + (b * c))$$

- Caso 5: Regra da derivada de uma negação:

$$ad\_gen\ x\ (i_2\ (i_2\ (i_2\ (Negate, (a, b))))) = (a * (-1), b * (-1))$$

- Caso 6: Regra da derivada de uma expressão cuja base é e:

$$\frac{d}{dx}e^a = e^a \cdot \frac{d}{dx}(a)$$

$$ad\_gen\ x\ (i_2\ (i_2\ (i_2\ (E, (a, b))))) = (Prelude.exp\ a, b * (Prelude.exp\ a))$$

Problema 2

Solução:

Uma vez que queremos derivar uma implementação da fórmula de Catalan, (função Cn) sem fatoriais, teremos de descobrir alguma "propriedade"da mesma. Ora a base desta "propriedade"já nos é que é a recursividade mútua. Assim, teremos de chegar a uma fórmula que expresse recursividade mútua,tendo como ponto de partida a fórmula que dá o n-ésimo número de Catalan:

$$C\ n = (2\ n)\ !/\ ((n+1)\ ! \times (n!))$$

□

Ao analisarmos a fórmula e ao estarmos a trabalhar nos $\mathbb{N}_0$, conseguimos inferir o caso base e o caso recursivo da fórmula de Catalan. Assim, temos o caso base e o caso recursivo abaixo representados respetivamente:

$$
\begin{array}{ll}
\equiv & \{ \text{ Definição do caso base, Definição do caso recursivo (para n + 1) } \} \\
& \left\{ \begin{array}{l} C\ 0 = 0\ !/\ (1\ ! \times 0!) \\ C\ (n+1) = Cn \times (2\ n+2)\ !/\ ((n+2)\ !\ (n+1)!) \end{array} \right. \\
\equiv & \{ \text{ Simplificação do caso base, Propriedade do factorial } (x! = x \times (x-1)!) \ \} \\
& \left\{ \begin{array}{l} C\ 0 = 1 \\ C\ (n+1) = ((2\ n+2) \times (2\ n+1) \times (2\ n)!)\ /\ ((n+2) \times (n+1) \times (n!) \times (n+1)!) \end{array} \right. \\
\equiv & \{ \text{ Propriedade do factorial } (x! = x \times (x-1)!) \ \} \\
& \left\{ \begin{array}{l} C\ 0 = 1 \\ C\ (n+1) = ((2\ n+2) \times (2\ n+1) \times (2\ n)!)\ /\ ((n+2) \times (n+1) \times n\ ! \times (n+1)!) \end{array} \right. \\
\equiv & \{ \text{ Definição de } C\ n = (2\ n)\ !/\ ((n!) \times (n+1)!) \ \} \\
& \left\{ \begin{array}{l} C\ 0 = 1 \\ C\ (n+1) = Cn \times ((2\ n+2) \times (2\ n+1))\ /\ ((n+2) \times (n+1)) \end{array} \right. \\
\equiv & \{ \text{ Colocar o 2 em evidência } \} \\
& \left\{ \begin{array}{l} C\ 0 = 1 \\ C\ (n+1) = Cn \times (2\ (n+1) \times (2\ n+1))\ /\ ((n+2) \times (n+1)) \end{array} \right.
\end{array}
$$

$\equiv$ { Cortar o denomidor e numerador $n+1$ }

$$\begin{cases} C\ 0 = 1 \\ C\ (n+1) = Cn \times (2\ (2\ n+1))\ /\ (n+2) \end{cases}$$

$\equiv$ { Intrudução das funções f e g }

$$\begin{cases} C\ 0 = 1 \\ C\ (n+1) = Cn \times f\ (n)\ /\ g\ (n) \end{cases}$$

$\square$

NB: Criamos duas funções de forma a simplificar o cálculo, em que a função f é o numerador e a função g é o denominador. Tendo:
$f\ (n) = 2\ (2\ n+1)$
$g\ (n) = n+2$

Uma vez definidas as funções "auxiliares", conseguimos também definir os casos base das mesma. Tal como referido anteriormente, como estamos nos $\mathbb{N}_0$, conseguimos induzir facilmente os casos base das funções e além disso, simplificar as mesmas, tendo-se:

$\equiv$ { Definição de f, Definição de g }

$$\begin{cases} f\ (n) = 2\ (2\ n+1) \\ g\ (n) = n+2 \end{cases}$$

$\equiv$ { Definição do caso base e recursivo de f, Definição do caso base e recursivo de g }

$$\begin{cases} \begin{cases} f\ (0) = 2 \\ f\ (n+1) = 2\ (2\ (n+1)+1) \end{cases} \begin{cases} g\ (0) = 2 \\ g\ (n+1) = n+2+1 \end{cases} \end{cases}$$

$\equiv$ { Simplificação do caso recursivo de f }

$$\begin{cases} \begin{cases} f\ (0) = 2 \\ f\ (n+1) = 2\ (2\ n+1)+4 \end{cases} \begin{cases} g\ (0) = 2 \\ g\ (n+1) = n+2+1 \end{cases} \end{cases}$$

$\equiv$ { Definição de $f\ n = 2\ (2\ n+1)$, Definição de $g\ n = n+2$ }

$$\begin{cases} \begin{cases} f\ (0) = 2 \\ f\ (n+1) = f\ (n)+4 \end{cases} \begin{cases} g\ (0) = 2 \\ g\ (n+1) = g\ (n)+1 \end{cases} \end{cases}$$

$\square$

Após a simplificação da fórmula de Catalan dada, conseguimos chegar a uma expressão que não depende de factoriais. Ao analisar em detalhe, percebemos algo fulcral, que é a existência de recursividade mútua nesta nova expressão.

Estamos prontos para usar a dica que nos foi dado no enunciado, usando a função *loop*. Segundo o enunciado, na primeira etapa "O corpo do ciclo loop terá tantos argumentos quanto o número de funções mutuamente recursivas.", ora as funções mutuamente recursivas serão 3 sendo que 2 delas formarão um par (as funções f e g). Seguindo a segunda etapa "Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente", sendo que os argumentos da função *loop* serão as funções mutuamente recursivas f, g e C. Com estas etapas juntamente com a terceira etapa, temos a definição da função *loop*:

$$loop\ (C, (f, g)) = ((C * f)\ /\ g, (f+4, g+1))$$

$\square$

NB: É de salientar que a função *loop* é quem vai tratar da recursividade mútua das várias funções em causa (f, g, C);

Ao chegarmos à etapa final, a função init vai recolher os casos bases pela mesma ordem que as funções argumentos aparecem na função *loop*. Assim e analisando os cálculos acima, temos a definição da função *init*:

$$init = (1, 2, 2)$$

$\square$

Baseando-nos no exemplo aplicamos também a função π_1 após o for $loop\ init$, obtendo:

$$cat = \pi_1 \cdot \mathsf{for}\ loop\ init$$

$\square$

Assim, obtemos as definições das funções supramencionadas:

$$\begin{array}{l} f\ 0 = 2 \\ f\ (n+1) = f\ (n) + 4 \\ g\ 0 = 2 \\ g\ (n+1) = g\ (n) + 1 \\ c\ 0 = 1 \\ c\ (n+1) = c\ (n) * (f\ (n)\ /\ g\ (n)) \\ loop\ (c, (f, g)) = (\widehat{\cdot \div \cdot}\ ((c * f), g), (f + 4, g + 1)) \\ inic = (1, (2, 2)) \\ prj = \pi_1 \end{array}$$

por forma a que

$$cat = prj \cdot \mathsf{for}\ loop\ inic$$

seja a função pretendida. **NB**: usar divisão inteira. Apresentar de seguida a justificação da solução encontrada.

Problema 3

D.7 Solução:

Uma vez que a função $calcLine$ calcula a interpolação linear entre dois pontos e cada um destes pontos é do tipo $NPoint$ que por sua vez é uma lista de $\mathbb{Q}$ de tamanho N, sendo N o número de dimensões de cada ponto, temos que a função $calcLine$ é um catamorfismo tal é referido no enunciado, que irá "consumir"as dimensões em causa. Dado que o tipo em causa "consome"listas de $\mathbb{Q}$ conseguimos perceber e inferir que o bifunctor de base associado a este tipo indutivo é o mesmo que é usado para o tipo indutivo $List$. Assim temos:

$T\ A = NPoint$
$B\ (X, Y) = X + X \times Y$
$B\ (id, f) = id + id \times f$

De seguida, iremos definir o diagrama genérico associado ao tipo indutivo $NPoint$:

$$\begin{tikzcd} Npoint \arrow[d, "(\!|f|\!)"'] & & 1 + Q * Npoint \arrow[ll, "\mathbf{in}"'] \arrow[d, "id + id \times (\!|f|\!)"] \\ (Npoint \to Overtime\ Npoint) & & 1 + Q \times (Npoint \to Overtime\ Npoint) \arrow[ll, "gene"] \end{tikzcd}$$

Com a informação do diagrama acima, conseguimos inferir o diagrama associado ao catamorfismo $calcLine$:

$$\begin{tikzcd} Npoint \arrow[d, "calcLine"'] & & 1 + Q * Npoint \arrow[ll, "\mathbf{in}"'] \arrow[d, "id + id \times calcLine"] \\ (Npoint \to Overtime\ Npoint) & & 1 + Q \times (Npoint \to Overtime\ Npoint) \arrow[ll, "h"] \end{tikzcd}$$

Como foi referido anteriormente, uma vez que a função $calcLine$ é um catamorfismo, falta-nos induzir o gene h associado a este catamorfismo. Conseguimos perceber através da dica dada no anexo C, que o gene irá fazer uso da função g disponibilizada para tratar do que vem da recursividade. Assim, conseguimos chegar à definição da função $calcLine$ através do seu gene. Com isto tem-se:

$calcLine :: NPoint \to (NPoint \to OverTime\ NPoint)$
$calcLine = cataList\ h$ **where**

$h = [\underline{\underline{nil}}, g]$
$g :: (\mathbb{Q}, NPoint \to OverTime\ NPoint) \to (NPoint \to OverTime\ NPoint)$
$g\ (d, f)\ l = \textbf{case}\ l\ \textbf{of}$
$\quad [\,] \to nil$
$\quad (x : xs) \to \lambda z \to concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z$

D.8 Solução:

$deCasteljau :: [NPoint] \to OverTime\ NPoint$
$deCasteljau = hyloAlgForm\ alg\ coalg\ \textbf{where}$
$\quad coalg = \bot$
$\quad alg = \bot$
$hyloAlgForm = \bot$

Problema 4

D.9 Solução para listas não vazias:

Uma vez que estamos a trabalhar com listas não vazias teremos que definir um tipo indutivo para tal.Ora este irá ser muito semelhante ao já conhecido tipo indutivo List. Assim temos:

$T\ A = NotEmptyList\ A$
$B\ (X, Y) = X + X \times Y$
$B\ (id, f) = id + id \times f$

Diagrama genérico de um catamorfismo sobre o tipo indutivo *NotEmptyList*:

$$\begin{tikzcd} NotEmptyList\ A \arrow[d, "(\!|f|\!)"'] & A + A * NotEmptyList\ A \arrow[l, "\mathsf{in}"'] \arrow[d, "id + id \times (\!|f|\!)"] \\ B & A + A \times B \arrow[l, "f"] \end{tikzcd}$$

Além disso, teremos de definir a função out assim como a função que caracteriza o comportamento recursivo de um catamorfismo sobre o tipo indutivo supramencionado. Deste modo temos:

$outNotEmptyList\ [a] = i_1\ (a)$
$outNotEmptyList\ (a : x) = i_2\ (a, x)$
$recNotEmptyList\ f = id + id \times f$
$(\!|g|\!) = g \cdot recNotEmptyList\ (\!|g|\!) \cdot outNotEmptyList$

Uma vez que temos na alínea 1 do enunciado do Problema: $avg\_aux = (\!|[b, q]|\!)$ tal que $avg\_aux = \langle avg, length \rangle$ para listas não vazias, teremos primeiro de definir o gene da função length e da função avg para o tipo indutivo em causa. Assim temos, para a função length:

$$\begin{tikzcd} NotEmptyList\ A \arrow[d, "length"'] & A + A \times NotEmptyList\ A \arrow[l, "\mathsf{in}"'] \arrow[d, "id + id \times (length)"] \\ A & A + A \times A \arrow[l, "{[one, \mathsf{succ} \cdot \pi_2]}"] \end{tikzcd}$$

E para a a função avg, temos:

$$\begin{tikzcd} NotEmptyList\ A \arrow[d, "avg"'] & A + A \times NotEmptyList\ A \arrow[l, "\mathsf{in}"'] \arrow[d, "id + id \times (avg)"] \\ A & A + A \times A \arrow[l, "{[g1, g2]}"] \end{tikzcd}$$

Onde:
$g1 = id$
$g2 = \widehat{\cdot \div \cdot}\ (\widehat{add}\ (\pi_1, \widehat{mul}\ (\pi_2, k)), \mathsf{succ}\ \ k)$
$k = length\ x$
Em que x é a cauda da lista.
NB: Para efeito de simplificação, usaremos o gene do catamorfismo avg como $[g1, g2]$.
Uma vez que queremos usar a lei de recursividade mútua, temos como ponto de partida a seguinte expressão:
$avg\_aux = \langle avg, length \rangle$
Onde o diagrama da função avg_aux é o seguinte:

$$
\begin{tikzcd}
NotEmptyList\ A \arrow[d, "avg\_aux"'] & & A + A \times NotEmptyList\ A \arrow[ll, "\mathsf{in}"'] \arrow[d, "id+(id\times avg_a ux)"] \\
A \times A & & A + (A \times (A \times A)) \arrow[ll, "{[b, q]}"]
\end{tikzcd}
$$

Assim, teremos de aplicar as leis do Cálculo de Programas:

$$
\begin{array}{ll}
& avg\_aux = \langle avg, length \rangle \\
\equiv & \quad \{ \text{ Definição de avg como catamorfismo, definição de length como catamorfismo } \} \\
& avg\_aux = \langle [g1, g2], [one, \mathsf{succ}\ \cdot \pi_2] \rangle \\
\equiv & \quad \{ \text{ Lei Banana-Split (53) } \} \\
& avg\_aux = (\!|([g1, g2] \times [one, \mathsf{succ}\ \cdot \pi_2]) \cdot \langle F\ \pi_1, F\ \pi_2 \rangle |\!) \\
\equiv & \quad \{ \text{ Definição de Functor para } NotEmptyList \ \} \\
& avg\_aux = (\![g1, g2] \times [one, \mathsf{succ}\ \cdot \pi_2] \cdot \langle id + (id \times \pi_1), id + (id \times \pi_2) \rangle |\!) \\
\equiv & \quad \{ \text{ Lei Absorção x (11) } \} \\
& avg\_aux = (\!|\langle [g1, g2] \cdot (id + id \times \pi_1), [one, \mathsf{succ}\ \cdot \pi_2] \cdot (id + id \times \pi_2) \rangle |\!) \\
\equiv & \quad \{ \text{ Lei Absorção + (22), Natural id (1), natural } \pi_2 \text{ (13) } \} \\
& avg\_aux = (\!|\langle [g1, g2 \cdot (id \times \pi_1)], [one, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2] \rangle |\!) \\
\equiv & \quad \{ \text{ Lei da Troca (28) } \} \\
& avg\_aux = (\!|[\langle g1, one \rangle, \langle g2 \cdot (id \times \pi_1), \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle]|\!) \\
\square &
\end{array}
$$

Desta forma chegámos ao pretendido: $avg\_aux = (\!|[b, q]|\!)$
Com
$b = \langle g1, one \rangle$
$q = \langle g2 \cdot (id \times \pi_1), \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle$
E substituindo pelas definições de g1 e g2, temos:
$b = \langle id, one \rangle$
$q = \langle av, len \rangle$
Onde:
$av = \widehat{\cdot \div \cdot}\ (\widehat{add}\ (\pi_1, \widehat{mul}\ (\pi_2, k)), \mathsf{succ}\ \ k) \cdot (id \times \pi_1)$
$len = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2$
Com isto, temos então definida a função avg_aux sobre a forma de um catamorfismo.

$avg\_aux = (\!|gene|\!)$ **where**
  $gene = [b, q]$
  $b = \langle id, \underline{1} \rangle$
  $q = \langle av, len \rangle$

$$
\begin{aligned}
&av\ (valor, (med, comp)) = ((med * comp) + valor)\ /\ (comp + 1)\\
&len\ (valor, (med, comp)) = comp + 1
\end{aligned}
$$

$$avg = \pi_1 \cdot avg\_aux$$

D.10 Solução para árvores de tipo LTree:

Ora, após o raciocínio para o tipo indutivo $NotEmptyList$, o raciocínio para o tipo indutivo LTree é o mesmo.

NB: A diferença é que para este tipo indutivo temos que a média só é calculada nas folhas e temos de ir somando o comprimento das sub-árvores (ramo da esquerda e ramo da direita).

Assim temos que a função $avgLTree$ é um split de catamorfismos. Logo, temos o seguinte diagrama:

$$
\begin{tikzcd}
\mathsf{LTree}\ A \arrow[d, "avgLTree"'] & A + \mathsf{LTree}\ A \times \mathsf{LTree}\ A \arrow[l, "\mathbf{in}"'] \arrow[d, "id+(avgLTree \times avgLTree)"] \\
(A \times A) & A + ((A \times A) \times (A \times A)) \arrow[l, "{[b,q]}"]
\end{tikzcd}
$$

Seguindo o mesmo raciocínio, mas aplicado ao tipo indutivo LTree temos agora dois pares obtidos através da recursividade mútua das funções length e avg para este tipo indutivo. Assim cada par é constituido pela média das folhas e comprimento das sub-árvores.

$$
\begin{aligned}
&avgLTree = \pi_1 \cdot (\!|\, gene \,|\!)\ \mathbf{where}\\
&\quad gene = [b, q]\\
&\quad b = \langle id, \underline{1} \rangle\\
&\quad q = \langle av, len \rangle\\
&\quad av\ ((med\_l, comp\_l), (med\_r, comp\_r)) = (med\_l * comp\_l + med\_r * comp\_r)\ /\ (comp\_l + comp\_r)\\
&\quad len\ ((med\_l, comp\_l), (med\_r, comp\_r)) = comp\_l + comp\_r
\end{aligned}
$$

Problema 5

Inserir em baixo o código F# desenvolvido, entre `\begin{verbatim}` e `\end{verbatim}`:

```
// {-# OPTIONS_GHC -XNPlusKPatterns #-}

// (c) MP-I (1998/9-2006/7) and CP (2005/6-2016/7)

module BTree

open Cp
//import Data.List
//import Data.Monoid

// (1) Datatype definition -----------------------------------------------------

type BTree<'a> = Empty | Node of 'a * (BTree<'a> * BTree<'a>)

let inBTree x = (either (konst Empty) Node) x

let outBTree x =
      match x with
      | Empty -> Left ()
      | Node (a,(t1,t2)) -> Right (a,(t1,t2))

// (2) Ana + cata + hylo -------------------------------------------------------
```

```
// recBTree g = id -|- (id >< (g >< g))

let baseBTree f g = id -|- (f >< (g >< g))

let recBTree g = baseBTree id g

let rec cataBTree g x = (g << (recBTree (cataBTree g)) << outBTree) x

let rec anaBTree g x = (inBTree << (recBTree (anaBTree g)) << g) x

let hyloBTree h g x = (cataBTree h << anaBTree g) x



// (3) Map -----------------------------------------------------------------------

// instance Functor BTree
//       where fmap f = cataBTree ( inBTree . baseBTree f id )
let fmap f x = cataBTree ( inBTree << baseBTree f id ) x

// equivalent to:
//       where fmap f = anaBTree ( baseBTree f id . outBTree )

// (4) Examples ------------------------------------------------------------------

// (4.1) Inversion (mirror) ------------------------------------------------------

let invBTree x = cataBTree (inBTree << (id -|- (id >< swap))) x

// (4.2) Counting ----------------------------------------------------------------

let countBTree x = cataBTree (either (konst 0) (succ << (uncurry (+)) << p2)) x

// (4.3) Serialization -----------------------------------------------------------

let preord x =
    let f(x,(l,r))=x::l@r
    in (either nil f) x

let preordt x = cataBTree preord x  // pre-order traversal

let postordt x =
    let f(x,(l,r))=l@r@[x]
    in cataBTree (either nil f) x   // post-order traversal

let inord x =
    let join(x,(l,r))=l@[x]@r
     in (either nil join) x

let inordt x = cataBTree inord x   // in-order traversal

// (4.4) Quicksort ---------------------------------------------------------------

let rec part r c =
  match c with
  | [] -> ([],[])
  | (x::xs) -> let (s,c) = part r xs
```

```
                 in if x < r then (x::s,c) else (s,x::c)

let qsep l =
  match l with
  | [] -> Left ()
  | (x::xs) -> let (s,l) = part x xs in Right (x,(s,l))

let qSort x = hyloBTree inord qsep x // the same as (cataBTree inord). (anaBTree qsep

(* pointwise versions:
qSort [] = []
qSort (h:t) = let (t1,t2) = part (<h) t
              in  qSort t1 ++ [h] ++ qSort t2

or, using list comprehensions:

qSort [] = []
qSort (h:t) = qSort [ a | a <- t , a < h ] ++ [h] ++
              qSort [ a | a <- t , a >= h ]

*)
// (4.5) Traces -----------------------------------------------------------------

let auxAdd c t = c :: t

let rec union c1 c2 =
     match c2 with
     | [] -> c1
     | (h::t) -> if (List.exists (fun c -> c = h) c1) then union c1 t
                                          else union (c1@[h]) t

let tunion(a,(l,r)) = union (List.map (auxAdd a) l) (List.map (auxAdd a) r)

let traces x = cataBTree (either (konst [[]]) tunion) x


// (4.6) Towers of Hanoi --------------------------------------------------------

// pointwise:
// hanoi(d,0) = []
// hanoi(d,n+1) = (hanoi (not d,n)) ++ [(n,d)] ++ (hanoi (not d, n))


let present x = inord x  // same as in qSort

let strategy x =
  match x with
  |(d,0) -> Left ()
  |(d,n) -> Right ((n-1,d),((not d,n-1),(not d,n-1)))

let hanoi x = hyloBTree present strategy x

(*
    The Towers of Hanoi problem comes from a puzzle marketed in 1883
    by the French mathematician Édouard Lucas, under the pseudonym
    Claus. The puzzle is based on a legend according to which
    there is a temple, apparently in Bramah rather than in Hanoi as
    one might expect, where there are three giant poles fixed in the
```

```
    ground. On the first of these poles, at the time of the world’s
    creation, God placed sixty four golden disks, each of different
    size, in decreasing order of size. The Bramin monks were given
    the task of moving the disks, one per day, from one pole to another
    subject to the rule that no disk may ever be above a smaller disk.
    The monks’ task would be complete when they had succeeded in moving
    all the disks from the first of the poles to the second and, on
    the day that they completed their task the world would come to
    an end!

    There is a wellknown inductive solution to the problem given
    by the pseudocode below. In this solution we make use of the fact
    that the given problem is symmetrical with respect to all three
    poles. Thus it is undesirable to name the individual poles. Instead
    we visualize the poles as being arranged in a circle; the problem
    is to move the tower of disks from one pole to the next pole in
    a specified direction around the circle. The code defines H n d
    to be a sequence of pairs (k,d’) where n is the number of disks,
    k is a disk number and d and d’ are directions. Disks are numbered
    from 0 onwards, disk 0 being the smallest. (Assigning number 0
    to the smallest rather than the largest disk has the advantage
    that the number of the disk that is moved on any day is independent
    of the total number of disks to be moved.) Directions are boolean
    values, true representing a clockwise movement and false an anticlockwise
    movement. The pair (k,d’) means move the disk numbered k from
    its current position in the direction d’. The semicolon operator
    concatenates sequences together, [] denotes an empty sequence
    and [x] is a sequence with exactly one element x. Taking the pairs
    in order from left to right, the complete sequence H n d prescribes
    how to move the n smallest disks onebyone from one pole to the
    next pole in the direction d following the rule of never placing
    a larger disk on top of a smaller disk.

    H 0     d = [],
    H (n+1) d = H n d ; [ (n, d) ] ; H n d.

    (excerpt from R. Backhouse, M. Fokkinga / Information Processing
    Letters 77 (2001) 71--76)

*)


// (5) Depth and balancing (using mutual recursion) --------------------------


let baldepth x =
  let h(a,((b1,b2),(d1,d2))) = (b1 && b2 && abs(d1-d2)<=1,1+max d1 d2)
  let f((b1,d1),(b2,d2)) = ((b1,b2),(d1,d2))
  let g x = (either (konst(true,1)) (h<<(id><f))) x
  in cataBTree g x

let balBTree x = (p1 << baldepth) x

let depthBTree x = (p2 << baldepth) x

(*
// (6) Going polytipic -------------------------------------------------------
```

```
// natural transformation from base functor to monoid

let tnat f = let theta = uncurry mappend
  in either (const mempty) (theta . (f >< theta))

// monoid reduction

let monBTree f = cataBTree (tnat f)

// alternative to (4.2) serialization ----------------------------------------

let preordt' = monBTree singl

// alternative to (4.1) counting ---------------------------------------------

let countBTree' = monBTree (const (Sum 1))

// (7) Zipper ----------------------------------------------------------------

type Deriv <'a> = Dr Bool of 'a * BTree <'a>

type Zipper <'a> = [ Deriv <'a> ]

let rec plug l =
  match l with
  | [] t -> t
  | ((Dr False a l):z) t -> Node (a,(plug z t,l))
  | ((Dr True  a r):z) t -> Node (a,(r,plug z t))

---------------------------- end of library ----------------------------------
*)
```

Índice

Referências

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

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