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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	27
a89559	Alberto Leal Fernandes
a89521	Alexandra Dias Candeias
a89607	João Paulo Ribeiro Pereira
a89570	Tiago Carvalho Freitas

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,

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

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

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

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 $\mathit{inExpAr}$ e $\mathit{baseExpAr}$ para este tipo é a seguinte:

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

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

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

$$\begin{array}{l}
\mathit{prop_in_out_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{ExpAr}\ a \rightarrow \mathit{Bool} \\
\mathit{prop_in_out_idExpAr} = \mathit{inExpAr} \cdot \mathit{outExpAr} \equiv \mathit{id} \\
\mathit{prop_out_in_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{OutExpAr}\ a \rightarrow \mathit{Bool} \\
\mathit{prop_out_in_idExpAr} = \mathit{outExpAr} \cdot \mathit{inExpAr} \equiv \mathit{id}
\end{array}$$

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.*

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

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

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

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 $eval$.*

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

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.

$$f\ 0 = 1$$
$$f\ (n+1) = fib\ n + f\ n$$

Obter-se-á de imediato

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

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.[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:

$$f\ 0 = c$$
$$f\ (n+1) = f\ n + k\ n$$
$$k\ 0 = a + b$$
$$k\ (n+1) = k\ n + 2\ a$$

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

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

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

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

que implemente esta função.

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

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

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.

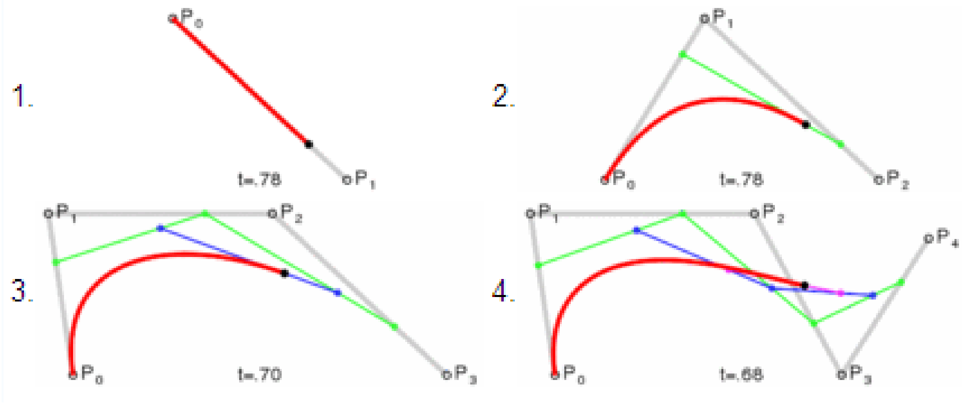

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

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

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.

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

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 $Float$).

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

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]

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

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:

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

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

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

[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 (4):

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

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

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

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

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

[10]Fonte: Wikipedia.

```
tick :: Float → World → World
tick dt world = world {time = (time world) + dt}
actions :: Event → World → World
actions (EventKey (MouseButton LeftButton) Down _ p) world =
  world {points = (points world) ++ [(λ(x, y) → map to_Q [x, y]) p]}
actions (EventKey (SpecialKey KeyDelete) Down _ _) world =
  world {points = cond (≡ [ ]) id init (points world)}
actions _ world = world
scaleTime :: World → Float
scaleTime w = (1 + cos (time w)) / 2
bezier2dAtTime :: World → (Float, Float)
bezier2dAtTime w = (bezier2dAt w) (scaleTime w)
bezier2dAt :: World → OverTime (Float, Float)
bezier2dAt w = bezier2d (points w)
thicCirc :: Picture
thicCirc = ThickCircle 4 10
ps :: [Float]
ps = map from_Q ps' where
  ps' :: [Q]
  ps' = [0, 0.01 .. 1]   -- interval
```

Gloss:

```
picture :: World → Picture
picture world = Pictures
  [animateBezier (scaleTime world) (points world)
  , Color white · Line · map (bezier2dAt world) $ ps
  , Color blue · Pictures $ [Translate (from_Q x) (from_Q y) thicCirc | [x, y] ← points world]
  , Color green $ Translate cx cy thicCirc
  ] where
  (cx, cy) = bezier2dAtTime world
```

Animação:

```
animateBezier :: Float → [NPoint] → Picture
animateBezier _ [ ] = Blank
animateBezier _ [_] = Blank
animateBezier t l = Pictures
  [animateBezier t (init l)
  , animateBezier t (tail l)
  , Color red · Line $ [a, b]
  , Color orange $ Translate ax ay thicCirc
  , Color orange $ Translate bx by thicCirc
  ] where
  a@(ax, ay) = bezier2d (init l) t
  b@(bx, by) = bezier2d (tail l) t
```

Propriedades e *main*:

```
runBezier :: IO ()
runBezier = play (InWindow "Bézier" (600, 600) (0, 0))
  black 50 initW picture actions tick
runBezierSym :: IO ()
runBezierSym = quickCheckWith (stdArgs {maxSize = 20, maxSuccess = 200}) prop_bezier_sym
```

Compilação e execução dentro do interpretador:[11]

```
main = runBezier
run = do {system "ghc cp2021t"; system "./cp2021t"}
```

[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:

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

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

Outras funções auxiliares

Lógicas:

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

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:

$$
\begin{array}{l}
\mathit{cataExpAr}\ g = g \cdot \mathit{recExpAr}\ (\mathit{cataExpAr}\ g) \cdot \mathit{outExpAr} \\
\mathit{anaExpAr}\ g = \mathit{inExpAr} \cdot \mathit{recExpAr}\ (\mathit{anaExpAr}\ g) \cdot g \\
\mathit{hyloExpAr}\ h\ g = \mathit{cataExpAr}\ h \cdot \mathit{anaExpAr}\ g
\end{array}
$$

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

Tendo como base as funções definidas em cima, conhecendo o comportamento das expressões, e também a função *inExpAr* e o functor base *baseExpAr*, começamos por definir as funções que faltam para completar o catamorfismo e o anamorfismo essenciais para transformar este tipo de dados.

outExpAr

Inicialmente definimos o *outExpAr*, com o auxílio da definição do *inExpAr*, já que os dois formam um isomorfismo. A partir disto aplicamos algumas leis de forma a chegar à função referente ao *outExpAr*.

$$outExpAr\ \cdot\ inExpAr\ =\ id$$
$$\equiv \quad \{\text{ inExpAr }\}$$
$$outExpAr\ \cdot\ [\underline{X}, num\_ops]\ =\ id$$
$$\equiv \quad \{\text{ Fusão-+ (20) }\}$$
$$[outExpAr\ \cdot\ \underline{X},\ outExpAr\ \cdot\ num\_ops]\ =\ id$$
$$\equiv \quad \{\text{ Universal-+ (17) e Natural-id (1) (aplicada duas vezes) }\}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot num\_ops = i_2 \end{cases}$$
$$\equiv \quad \{\text{ Substituição de num\_ops, Igualdade extensional (71) e Def-comp (72) }\}$$
$$\begin{cases} outExpAr\ X = i_1\ () \\ outExpAr\ (N\ a) = i_2\ (i_1\ a) \\ outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b)))) \\ outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a))) \end{cases}$$

Além de termos utilizado estas leis em cima para deduzir a definição do *outExpAr*, também nos baseamos no tipo do *out* enquanto deduzíamos as leis, tornou-se mais intuitivo chegar à definição final, porque através dos tipos soubemos onde injetar cada parte da expressão, e basicamente só tivemos de injetar no sítio correto do tipo, cada representação da expressão.

Definição do out:

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

recExpAr

A seguir apresenta-se uma restrição ao functor base de $ExpAr$. Ao elemento (*N a*) deste data type, vamos aplicar uma função **g** ao executar o processo recursivo da estrutura. À variável X constante, e aos operadores, vamos sempre preservar a identidade num processo recursivo, pois são sempre os mesmos qualquer que seja a transformação a aplicar, e não devem ser afetados. Às expressões ($ExpAr$) associadas a cada operador, vamos aplicar a mesma função (diferente da função g porque aqui estamos a trabalhar com expressões) para as transformar, e essa será a função **f**. Deste modo obtemos um novo functor base em função de **g** e **f**. O functor $recExpAr$, por sua vez, aplica uma única função às expressões associadas aos operadores binários ou unários, e por isso, pode assumir a definição do functor base mas preservando os números (*N a*).

$$
\begin{array}{ll}
& baseExpAr' \; g \; f \; = \; baseExpAr \; id \; g \; id \; f \; f \; id \; f \\
\equiv & \qquad \{ \text{ De acordo com o raciocínio explicado anteriormente } \} \\
& recExpr \; f = baseExpAr' \; id \; f
\end{array}
$$

Representando graficamente:

$$
\begin{array}{c}
1 + (a + (BinOp \times (ExpAr \; a \times ExpAr \; a) + (UnOp \times ExpAr \; a))) \\
\Big\downarrow {\scriptstyle id+(id+(id \times (f \times f)+(id \times f)))} \\
1 + (a + (BinOp \times (A \times A) + (UnOp \times A)))
\end{array}
$$

Este functor permite-nos transformar a estrutura das expressões recursivamente, e pode ser utilizado em qualquer catamorfismo deste tipo. Assumimos que a função **f** recebe o tipo $ExpAr$ e tem como saída um tipo abstrato A.

$baseExpAr' \; g \; f = baseExpAr \; id \; g \; id \; f \; f \; id \; f$

--

$recExpAr \; f = baseExpAr' \; id \; f$

g_eval_exp

A função $eval\_exp$, que calcula o valor de uma expressão, dado um valor para substituir na variável X, é nos dada como um catamorfismo. Com isto, resta-nos descobrir o seu gene. Começamos por representar o problema num diagrama que representa este catamorfismo, com o intuito de deduzir a definição do gene para obter o resultado pretendido.

$$
\begin{tikzcd}
ExpAr \; a \arrow[d, "(\!| g\_eval\_exp |\!)"'] \arrow[r, bend left, "outExpAr"] \arrow[r, phantom, "\cong"] & 1 + (a + ((BinOp \times (ExpAr \; a \times ExpAr \; a)) + (UnOp \times ExpAr \; a))) \arrow[l, bend left, "inExpAr"] \arrow[d, "recExpAr \; (\!| g\_eval\_exp |\!)"] \arrow[r, phantom, "\cong"] & {} \\
a & 1 + (a + ((BinOp \times (a \times a)) + (UnOp \times a))) \arrow[l, "g\_eval\_exp"]
\end{tikzcd}
$$

Depois de construir o diagrama, conseguimos perceber que o conteúdo do gene seria baseado em *eithers*, porque o tipo de entrada do gene contém várias somas, e o tipo de saída é apenas um **a**. Deste modo para transformar todas aquelas somas num único **a**, recorremos a duas funções auxiliares que nos simplificaram a transformação dos operadores no tipo de saída. Estas funções basicamente calculam o valor da expressão. Nos casos em que o tipo de entrada só contém um **a** ou então é do tipo 1, são aplicadas as funções id e $\underline{a}$, respetivamente.

$unAction \; (Negate, a) = negate \; a$
$unAction \; (E, a) = expd \; a$

$binAction \; (Sum, (a, b)) = a + b$
$binAction \; (Product, (a, b)) = a * b$

$g\_eval\_exp\ a = [\underline{a}, [id, [binAction, unAction]]]$

clean e gopt

Nesta questão é nos fornecida uma função para otimizar as expressões, e para tal é utilizado um hilomorfismo $hyloExpAr$, que utiliza duas funções chamadas $gopt$ e $clean$. Segundo a definição deste hilomorfismo, podemos representá-lo como uma composição de um catamorfismo, cujo gene é $gopt$ com um anamorfismo, cujo gene é $clean$.

$$
\begin{aligned}
& optmize\_eval\ a = hyloExpAr\ (\ gopt\ a\ )\ clean \\
\equiv \quad & \{ \text{ Definição de hyloExpAr } \} \\
& optmize\_eval\ a = cataExpAr(gopt\ a)\ .\ anaExpAr(clean)
\end{aligned}
$$

Representando graficamente o hilomorfismo em função do catamorfismo e do anamorfismo, construímos o seguinte diagrama:

$$
\begin{tikzcd}
ExpAr\ a \arrow[r, bend left, "clean"] \arrow[d, "anaExpAr\ (clean)"'] & () + (a + ((BinOp \times (ExpAr\ a \times ExpAr\ a)) + (UnOp \times ExpAr\ a))) \arrow[d, "recExpAr\ (anaExpAr\ (clean))"] \\
ExpAr\ a \arrow[r, bend left, "outExpAr"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|gopt\ a|\!)"'] & () + (a + ((BinOp \times (ExpAr\ a \times ExpAr\ a)) + (UnOp \times ExpAr\ a))) \arrow[l, bend left, "inExpAr"] \arrow[d, "recExpAr\ (\!|gopt\ a|\!)"] \\
a & () + (a + ((BinOp \times (a \times a)) + (UnOp \times a))) \arrow[l, bend left, "gopt"]
\end{tikzcd}
$$

Como se pode ver no diagrama, a função clean não altera os tipos da estrutura, porque apenas simplifica as expressões. Foi por esta razão que nos baseamos no $outExpAr$, e sabíamos que era preciso injetar valores em sítios diferentes consoante o tipo de entrada. Essa injeção difere da do $outExpAr$ porque é necessário fazer as simplificações às expressões. Só em casos específicos é que precisamos de implementar essa diferença: quando recebemos um número ou um operador binário. Os casos que abordamos foram os elementos neutros da adição e multiplicação, e também o elemento absorvente da multiplicação. A partir deste raciocínio implementamos a função $clean$ da seguinte forma:

$$
\begin{aligned}
& clean\ (N\ a) = i_2\ (i_1\ a) \\
& clean\ (Bin\ Sum\ (N\ 0)\ a) = clean\ a \\
& clean\ (Bin\ Sum\ a\ (N\ 0)) = clean\ a \\
& clean\ (Bin\ Product\ (N\ 1)\ a) = clean\ a \\
& clean\ (Bin\ Product\ a\ (N\ 1)) = clean\ a \\
& clean\ (Bin\ Product\ (N\ 0)\ a) = clean\ (N\ 0) \\
& clean\ (Bin\ Product\ a\ (N\ 0)) = clean\ (N\ 0) \\
& clean\ resto = outExpAr\ resto
\end{aligned}
$$

Por outro lado, a função gopt recebe uma expressão e calcula o seu valor, ou seja, transforma uma $ExpAr$ num A. No cálculo realizamos manualmente as operações que envolviam elementos neutros e recorremos à função $binAction$, definida anteriormente, para o caso geral.

$$
\begin{aligned}
& gopt\ a = [\underline{a}, [id, [binAction', unAction]]]\ \textbf{where} \\
& \quad binAction'\ (Sum, (0, b)) = b \\
& \quad binAction'\ (Sum, (a, 0)) = a
\end{aligned}
$$

$binAction'\ (Product, (1, b)) = b$
$binAction'\ (Product, (a, 1)) = a$
$binAction'\ a = binAction\ a$

sd-gen e ad-gen

Finalmente, resta completar as duas funções principais deste problema, sd que tem a função de calcular a derivada de uma expressão através de sucessivais transformações (recursividade), e ad que, simplesmente procura calcular o valor da derivada de uma expressão no ponto, reduzindo uma expressão a um valor singular. Dados os catamorfimos destas duas funções, é necessário deduzir os genes respetivos e, ambos, vão produzir um par, sendo que o *sd-gen* tem como resultado (ExpAr a, ExpAr a), onde o primeiro elemento consiste na expressão a derivar e o segundo a derivada desta. Já o *ad-gen* produz um par (a, a), no qual o primeiro elemento é a solução da função no ponto e o segundo é a derivada no mesmo.

$sd\_gen :: Floating\ a \Rightarrow$
$\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow$
$\quad\quad (ExpAr\ a, ExpAr\ a)$
$sd\_gen = [derivX, [derivA, [derivBin, derivUn]]]$ **where**
$\quad derivX\ () = (X, N\ 1)$
$\quad derivA\ a = (N\ a, N\ 0)$
$\quad derivBin\ (Sum, ((a, a'), (b, b'))) = (Bin\ Sum\ a\ b, Bin\ Sum\ a'\ b')$
$\quad derivBin\ (Product, ((a, a'), (b, b'))) = (Bin\ Product\ a\ b, Bin\ Sum\ (Bin\ Product\ a\ b')\ (Bin\ Product\ a'\ b))$
$\quad derivUn\ (Negate, (a, a')) = (Un\ Negate\ a, Un\ Negate\ a')$
$\quad derivUn\ (E, (a, a')) = (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ a')$

$ad\_gen\ v = [auxX\ v, [auxN, [auxBin, auxUn]]]$ **where**
$\quad auxX\ v\ () = (v, 1)$
$\quad auxN\ a = (a, 0)$
$\quad auxBin\ (Sum, ((a, b), (c, d))) = ((+)\ a\ c, (+)\ b\ d)$
$\quad auxBin\ (Product, ((a, b), (c, d))) = ((*)\ a\ c, (+)\ ((*)\ a\ d)\ ((*)\ b\ c))$
$\quad auxUn\ (Negate, (a, b)) = (-a, -b)$
$\quad auxUn\ (E, (a, b)) = (expd\ a, (*)\ (expd\ a)\ b)$

Problema 2

Para resolver esta questão, e baseando-nos no exemplo dado no enunciado, temos de começar por encontrar funções mutuamente recursivas, tal como na função exponencial. Para tal desdobramos a equação, até encontrarmos três destas funções, tal como se segue:

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \tag{4}$$

$c\ 0 = 1$
$c\ (n+1) = \frac{(2\ (n+1))!}{((n+1)+1)!(n+1)!} = \frac{(2\ n+2)!}{(n+2)!(n+1)!} =$
$= \frac{(2\ n+2)\ (2\ n+1)\ (2\ n)!}{(n+2)\ (n+1)\ (n)!(n+1)!} = \frac{2\ (n+1)\ (2\ n+1)\ (2\ n)!}{(n+2)\ (n+1)\ (n!)\ (n+1)!} =$
$= \frac{2\ (2\ n+1)\ (2\ n)!}{(n+2)\ (n!)\ (n+1)!} = \frac{(2\ n)!}{(n+1)!(n)!} \cdot \frac{4\ n+2}{n+2} = c\ n \cdot \frac{d\ n}{e\ n}$

Já tendo a função principal dividida em três que se complementam, resta-nos definir os casos base de cada uma delas, e também o caso do fator (n + 1) para aplicar a recursividade em cada uma delas.

$$
\begin{array}{l}
d\ 0 = 2 \\
d\ n = 4\ n + 2 \\
d\ (n+1) = (4\ (n+1) + 2) = 4\ n + 6 = d\ n + 4 \\
e\ 0 = 2 \\
e\ n = n + 2 \\
e\ (n+1) = n + 1 + 2 = e\ n + 1
\end{array}
$$

Por fim, resta-nos passar tudo para código, e já conseguimos completar o *loop*, que consiste em utilizar 3 funções mutuamente recursivas como corpo do ciclo, porque é nesta parte que aplicamos a recursividade. O *init* contém todos os casos de partida, quando n é 0. O *prj* é o que nos dá aquilo que realmente queremos, que neste caso é a função principal dos números de *Catalan*, e portanto, das três funções devolve-nos o resultado da primeira.

$$
\begin{array}{l}
c\ 0 = 1 \\
c\ (n+1) = c\ n * d\ n \div e\ (n+1) \\
d\ 0 = 2 \\
d\ (n+1) = 4 + d\ n \\
e\ 0 = 2 \\
e\ (n+1) = 1 + e\ n \\
cat = prj \cdot \mathsf{for}\ loop\ inic\ \mathbf{where} \\
\quad loop\ (c, d, e) = (c * d \div e, 4 + d, 1 + e) \\
\quad inic = (1, 2, 2) \\
\quad prj\ (c, d, e) = c
\end{array}
$$

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

Problema 3

$$
\begin{array}{l}
h1 = \bot \\
h2\ ((p : ps), f) = \bot \\
calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\
calcLine = cataList\ h\ \mathbf{where} \\
\quad h = \bot \\
deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint \\
deCasteljau = hyloAlgForm\ alg\ coalg\ \mathbf{where} \\
\quad coalg = \bot \\
\quad alg = \bot \\
hyloAlgForm = \bot
\end{array}
$$

Problema 4

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

Listas não vazias

De forma a poder trabalhar sobre listas não vazias (que são utilizadas nesta questão) tivemos que definir uma função $outList'$ e um catamorfismo $cataList'$ específicos, para usarmos na construção da solução deste problema. O *out* será o destrutor deste tipo, e deste modo podemos injetar à esquerda um único elemento ou então à direita a cabeça da lista, concatenada com a sua cauda.

$$
\begin{aligned}
&outList'\ [h] = i_1\ h \\
&outList'\ (h : t) = i_2\ (h, t) \\
&cataList'\ g = g \cdot recList\ (cataList'\ g) \cdot outList'
\end{aligned}
$$

avg_aux

Para descobrir a definição desta função, já temos o novo catamorfismo definido e só falta descobrir o gene a aplicar para obter o resultado pretendido. Sabemos do enunciado que a função avg está em recursividade mútua com $length$, ou seja, estamos a reduzir a informação da estrutura para obter a média, e ao mesmo tempo, o tamanho da lista. Também nos é dito que o gene é um *either*, e por isso, só nos resta definir as funções que o constituem. Primeiro é necessário definir um diagrama para auxiliar o raciocínio:

$$
\begin{tikzcd}
A^{+} \arrow[r, "outList'"] \arrow[d, "(\![b,q]\!)"'] & A + (A \times A^{+}) \arrow[d, "recList\ (\![b,q]\!)"] \\
A \times N_0 & A + A \times (A \times N_0) \arrow[l, "{[b,q]}"]
\end{tikzcd}
$$

Partindo deste catamorfismo, surge com mais facilidade o comportamento da função avg_aux. Para descobrir o seu gene, assumimos que já temos a média e o tamanho da cauda da lista, e só precisamos de aplicar a transformação final ao elemento que está à cabeça. Tanto o componente b como o q do *either* serão um split, visto que a partir do tipo do habitante da lista, que neste caso é um A, vamos gerar um par. O b é aplicado ao elemento singular da lista, e por isso colocamos à esquerda do par o próprio valor, e à direita o número 1. O q é mais complexo porque temos de lidar com o elemento que está à cabeça da lista e o par que já vem calculado previamente. Como à esquerda fica a média, utilizamos à esquerda a função $avgCalc$ para a calcular, e à direita a função $succAux$, que simplesmenta adiciona um ao valor que vem calculado anteriormente. Com isto, definimos as funções desta forma:

$$
\begin{aligned}
&succAux = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \\
&avgCalc\ (a, (avg, len)) = (a + (avg * len))\ /\ (len + 1) \\
&avg\_aux = cataList'\ [b, q]\ \textbf{where} \\
&\quad b = \langle id, \underline{1} \rangle \\
&\quad q = \langle avgCalc, succAux \rangle
\end{aligned}
$$

avgLTree

Solução para árvores de tipo LTree:

Nesta questão, começamos novamente pela elaboração do diagrama, e como já tinhamos percebido bem o problema com a questão anterior, bastou agora adaptar ao novo tipo de dados LTree.

$$
\begin{tikzcd}
LTree\ A \arrow[r, "outLTree"] \arrow[d, "(\![b,q]\!)"'] & A + (LTree\ A \times LTree\ A) \arrow[d, "recLTree\ (\![b,q]\!)"] \\
A \times N_0 & A + ((A \times N_0) \times (A \times N_0)) \arrow[l, "{[b,q]}"]
\end{tikzcd}
$$

Com isto torna-se simples a aplicação do gene, porque já temos a média e o tamanho das sub-árvores, agora temos de os calcular para a árvore completa. No caso do elemento singular, podemos projetar a definição anterior para aqui, com o $\langle id, \underline{1} \rangle$. Para descobrir o q, temos de adaptar os cálculos ao novo tipo de dados, e funciona da seguinte maneira: para a média multiplicamos a média de cada sub-árvore pelo seu tamanho, depois somamos ambos os resultados e dividimos pelo tamanho conjunto das sub-árvores, ou seja, a soma do comprimento de cada uma; para o comprimento, basta somar os valores das duas sub-árvores e obtemos o comprimento total.

Como tanto nesta questão como na anterior só nos interessa conhecer a média, aplicamos π_1 depois de executar o catamorfismo, em ambos os casos, para nos devolver o elemento que se encontra à esquerda no par (a média).

$$
\begin{array}{l}
avgLTree = \pi_1 \cdot (\!|\, g \,|\!) \ \mathbf{where} \\
\quad g = [\langle id, \underline{1} \rangle, \langle auxAvgLTree, auxLenLTree \rangle] \\
\quad auxAvgLTree\ ((a1, l1), (a2, l2)) = (((a1 * l1) + (a2 * l2)) \,/\, (l1 + l2)) \\
\quad auxLenLTree\ ((\_, l1), (\_, l2)) = l1 + l2
\end{array}
$$

Problema 5

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

```
module BTree

open Cp
open List
open Seq

// (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 -------------------------------------------------------

let baseBTree f g = id -|- (f >< (g >< g))
let recBTree g = baseBTree id g
let rec cataBTree g = g << (recBTree (cataBtree g)) << outBTree
let rec anaBTree g = inBTree << (recBTree (anaBTree g)) << g
let hyloBTree h g = cataBTree h << anaBTree g

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

let fmap f = cataBTree ( inBTree << baseBTree f id)

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

let inordt x = cataBTree inord x

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

let preordt x = cataBTree preord x

let postordt x =
    let f (a,(l,r)) = l @ r @ [a]
    in cataBTree (either nil f) x

// (4.4) Quicksort

let rec part x y =
    match x y with
    | p [] -> ([],[])
    | p (h::t) -> if p h then let (s,l) = part p t in (h::s,l) else let (s,l) = part

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

let qSort x = hyloBTree inord qsep x

// (4.5) Traces

let union left right =
    List.append left right |> Seq.distinct |> List.ofSeq

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

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

// (4.6)

let present x = inord x

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

let hanoi x = hyloBTree present strategy x

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

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

```
    in cataBTree g x

let balBTree x = p1 << baldepth x

let depthBTree x = p2 << baldepth x

// (6) Going polytipic

let tnat f x =
    let theta = uncurry mappend
    in either (konst mempty) (theta << (f >< theta)) x

let monBTree f x = cataBTree (tnat f) x

let preordt' x = monBTree singl x

let countBTree' x = monBTree (konst (Sum 1)) x

// (7) Zipper

type Deriv<'a> = Dr bool of 'a (BTree of 'a)
type Zipper<'a> = List<Deriv<'a>>

let rec plug x y =
    match x y 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))
```

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.